WILEY

TIMELY. PRACTICAL. RELIABLE.

Professional Java User Interfaces

Mauro Marinilli

Professional Java User Interfaces

Mauro Marinilli

John Wiley & Sons, Ltd

Main Patterns and Design Strategies organized by functional layer

Adaptation 272

Interaction and Control
Scattered Control 260
Mediator Pattern 263
Explicit Control State 260
Observer Pattern 240
Event Arbitrator 245
Message-Based Communication 250
Command Pattern 258
Active Object 280

Presentation
Separate GUI Design Details from Other code 234

Business Domain
Model-View-Controller 252
Model-View-Presenter 254
Domain-Driven Patterns 354

DataIO
Data Transfer Object 234
Proxy 234
Broker 234
Value Model 282

Content
Composite Pattern 59
Explicit Navigation 231
Composite Context 277

Main principles

- GUI design and overall development
  - User-centered design (44)

    A design approach for building highly usable user interfaces, putting the emphasis on the user.
  - Cost-driven design (81)

    GUI design comes first, with an eye on development complexity. For example, avoid using ad-hoc components (81) in your GUI as far as possible.
  - Iterative GUI development (169)

    Iterate: GUI design and implementation, profiling, software and usability testing.
- Implementation
  - The principle of Single Functional Responsibility (227)

    Provide only one functional responsibility per class/method.
  - Object lifecycle management – a general mindset (281)

    Instantiate lazily and dispose eagerly, avoid garbage collector bottlenecks.
  - Don't go against the flow (284)

    GUI toolkits are complex beasts, so don't ignore them and implement fancy designs counter to the architecture or style of the underlying GUI toolkits and infrastructure (RCP).

Visual refactorings

Other refactorings are discussed in Chapter 5.

- Extract explicit panel (195), Extract stand-alone panel (196), and Composable units (292)

  Extract the code of an existing GUI panel into a separate implementation to enhance modularity and reusability.
- Merge panel (197)

  Merge different implementations representing the same panel into a common one.
- Add parameter to panel (197) and Remove parameter from panel (198)

  Add parameters to customize a panel and its opposite refactoring, Remove parameter from panel (198).
- Parameterize panel (199)

  Implement two slightly different panels with a unique code base.
- Replace parameter with panel (200)

  Instead of adding a parameter, separate the implementation of the two panels.
- Rename panel (201)

  Change the name of a panel.

Cheat Sheet

An extremely simplified and by no means exhaustive basic reference to some of the topics discussed in the book.

GUI Design

- How do I signal to the user my GUI is busy?

  Change mouse pointer to hour glass for any operation that lasts more than two seconds, always use progress indicators, and update progress every five seconds.

- How do I validate my GUI?

  Involve users in design, use prototyping, software testing, memory profiling (Chapter 5), questionnaire evaluation (Appendices A and B), and usability testing.

- How do I organize the GUI window area?

  Use the *Area Organization* design strategy (120).

- How do I allow the user to select or create information in a GUI?

  Use the *Chooser* design strategy, 126.

- How do I deploy my GUI?

  Use Java Web Start when the user population is confident with approving certificates, as for internal software. Use installers in other cases, and for large installation bundles. Consider also using applets!

Software Design

- How do I keep my GUI responsive to user interaction during long-running operations?

  Use the *Active Object* pattern, 280 (for Swing, the `SwingWorker` class) for any operation that might last more than one second.

- How do I implement control (reaction to user interaction) in my GUI?

  Depending on the number of items to be controlled by control rules, use:
  – Scattered control (260) – few items, reactive-only control rules.
  – Centralized control, the Mediator pattern (263) – many items, any kind of control rule.
  – Explicit Control State (260) – complex control rules, need for flexibility.

- How do I implement undo/redo in a GUI?

  Build a queue or stack of edits (587), use the *Command* pattern (258) for user actions.

- How do I implement role-based authorization/security in my GUI?

  Build a dedicated authorization manager class using Adaptation (272).

- How do I implement user customization and user profiles in my GUI?

  Build a dedicated profile manager class using Adaptation (272).

- How do I reuse existing panels in my GUI?

  Use visual refactorings (194).

- How do I organize implementation for modularity and extensibility in a large or complex GUI?

  Define and implement a *Composable Unit* strategy, 292.

- How do I implement content assembly – adding components to a screen or panel – in a GUI?

  Depending on the features you want use:

  – Static assembly (229), for simple layouts, no reusability.

  – Simple Builders (229), for ease of use, separation of concerns, limited flexibility.

  – Create and use *Domain-specific* or *Little languages*, 466 – good separation, maximum flexibility.

- How do I organize complex event-based interactions among objects in my GUI?

  Use an Event Arbitrator (245) to:

  – Avoid event loops and rationalize chains of observers-observables.

  – Shield client classes from low-level events.

  – Forward events to complex data structures based on the Composite pattern.

- How do I handle large data collections?

  Depending on the context of the problem, use:

  – Eager disposal (281) of objects that are no longer needed – simple references, extremely large trees.

  – Weak or soft references, for cached objects and data that can be created or fetched on the fly.

  – Paging (281) for loading a few pages at time, discarding old ones, such as large table models or large collections of expensive objects.

- How do I communicate data remotely in a modular way?

  Separate screen data state from domain objects using Data Transfer Objects (234).

- How do I handle data represented in widgets?

  Define Screen Data State (SDS, 330) and the widgets that will interface SDS to the user. For synchronization with domain objects data (if any) use:

  – Manual synchronization of SDS and data, for simple, small GUIs.

  – Data binding support, for medium to large, complex GUIs.

Professional Java User Interfaces

Professional Java User Interfaces

Mauro Marinilli

John Wiley & Sons, Ltd

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 33 Park Road, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada M9W 1L1

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data

Marinilli, Mauro.
Professional Java user interfaces / Mauro Marinilli.
p. cm.
Includes bibliographical references and index.
ISBN 0-471-48696-5 (pbk. : alk. paper)
1. Java (Computer program language) 2. User interfaces (Computer systems) I. Title.
QA76.73.J38M34954 2006
005.13'3--dc22
2006004498

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 13: 978-0-471-48696-5

ISBN 10: 0-471-48696-5

Typeset in 10/13.5pt Palatino by Laserwords Private Limited, Chennai, India

Printed and bound in Great Britain by Bell & Bain, Glasgow

This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

To the person who keeps alive in his daily work
the Spirit of Wonder of the early days.

Brief Contents

Contents

Acknowledgements

I have been working on this book for more than five years, in one way or another. It is by no means the result of a single person (the author). A very large number of people shaped it, so many that it will be impossible to name them all.

I should first thank Sally Tickner and the management at John Wiley and Sons, for allowing me to deliver the manuscript after such a huge delay, which was mostly caused – only a partial excuse – by my never-ceasing joy of working on and taming new and complex adventures, rather than lack of interest in the subject.

It’s easy to remember Steve Rickaby of WordMongers, whose expert hand made this book readable and sometimes even enjoyable. If it wasn’t for his help this gigantic work would have been very different.

A special thanks too to my students, who kept the fire of honest enthusiasm and the Spirit of Wonder high, and who were extremely patient with my shameful schedule. I hope I gave them back at least a small portion of what they gave to me.

I wish also to thank the reviewers for their careful work, the many clients and colleagues for the countless lessons that I tried to put together in the book, and all the people who spent precious time out of their lives putting into written form their hard-won experience, often without any economic return.

Last but not least, my biggest thanks go to my family, my Bella, and my closest friends. Without their presence, patience, and constant support, this book wouldn’t have been possible.

Introduction

This introduction is structured as follows:

The interactivity thrill talks about the magic of the first time and other things.

The organization of the book discusses the book's contents and organization.

Book readers and personas provides a more user-centered approach to the contents of the book.

The interactivity thrill

Current software technology allows developers to build graphical user interfaces (GUIs) for only the cost of the labor, and with greater simplicity than ever before. Despite that, GUIs, and Java GUIs among them, are often totally frustrating and disappointing. In the words of Alan C. Kay[1]:

> "A twentieth century problem is that technology has become too 'easy.' When it was hard to do anything, whether good or bad, enough time was taken so that the result was usually good. Now we can make things almost trivially, especially in software, but most of the designs are trivial as well. This is *inverse vandalism*: the making of things because you can. Couple this to even less sophisticated buyers and you have generated an exploitation marketplace similar to that set up for teenagers. A counter to this is to generate enormous dissatisfaction with one's designs using the entire history of human art as a standard and goal. Then the trick is to decouple the dissatisfaction from self worth – otherwise it is either too depressing or one stops too soon with trivial results."

Basically, inverse vandals don't care about their work and its impact on the lives of users and the many others affected by their work, which is a pity. Software has a sort of magic in itself, and interactive software provides a concrete, vivid example of such a magic. Whether you are a teenager playing a video game or an old guy fiddling with an early computer in your garage, there was probably a moment in your life when you were totally amazed by a piece of software – otherwise you would probably have chosen another career.

1. *The Early History of Smalltalk*, http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html

Such a feeling alone, and perhaps a rather selfish and self-gratifying one, is not enough to provide reliable, professional results. There is a need to study and apply a wide array of subjects in depth, filtering user's needs through experience and the relentless application of ambitious but sensible designs and solutions, both on the GUI side and in its implementation. Despite all this hard work – or possibly because of it – the fun still remains, and I hope you can see it between the lines of this book. Finally, some words about my professional background, that could help in providing a better understanding and a more critical view of the book's contents.

My long experience is mostly on internal projects, that is, building software for customers, and also spans a few products building shrink-wrapped software. As far as Java is concerned, I started working with Java GUIs in 1998, trying to focus on client-side aspects whenever possible. I worked on a couple of large and complex GUIs, and on other projects that ranged from the weather forecasting system for the Italian air force to large multinational corporate ERPs, various Web sites, a large GUI framework for advanced enterprise clients – on various aspects still unmatched on the market – and more recently have been hopping on and off planes throughout Europe and US as a consultant while trying to find the time for a number of EU and academic research projects.

Usable GUIs and usable books

Writing a book like this is in many ways similar to GUI development[1]. The author has a target audience, at least in his mind (end users), and many little daily hindrances. He needs to work to earn his keep, trying to maintain a private life and struggle with mundane things like mastery of the English language (GUI design guidelines), the wrong dpi settings in scanned pictures (API inconsistencies), ever-newer technologies, and all the rest. Luckily he is not alone. He is the less-experienced part of a great team of professionals (the development team), good-willed reviewers (user representatives), and wonderful private-life supporters. Nevertheless, new ideas and existing content that should be better addressed seem never-ending (feature creep), and the manuscript keeps growing (deadlines shifting). The author is constrained by deadlines and wants to deliver something useful (at least within *his* definition of usefulness). There are different kinds of GUI development. There are shrink-wrapped products, where there is competition and users can easily opt for your product or a competing one, as in the case of a shareware music player, and various forms of internal projects where users have no choice but to read the

1. The term *development* is meant to indicate the general process of building a GUI, including GUI design and implementation.

book/use the application. A documentation manual fits the latter category[1]: unfortunately for me, this book falls into the first category.

All the above has a common denominator: the end user. The ultimate objective is to write a book that *you* would like to read, in which the message comes across as smoothly and as richly as possible and as *you* expect, saving *you* time and effort, while possibly providing *you* with a pleasant experience. This book – and the next application *you* are going to create – will be effective and useful as long as its very inception, its design and writing, focuses on end users.

The organization of the book

The book is organized in three parts. The first part introduces HCI and GUI design, starting from general concepts and concluding with recurring GUI designs. The second part, from Chapter 5 to Chapter 12, discusses general implementation advice. The third part, from Chapter 13 to Chapter 16, discusses some examples applications, from analysis and GUI design to the software architecture and the implementation – something rather rare to find in literature. Finally, two appendices provide evaluation questionnaires specifically targeted at Java GUIs.

The following gives a brief description of the book's contents.

Part	Chapter	Title	Description
	1	*Putting GUI Development into Context*	Framing GUI development in the wider context of software development, introducing a general reference functional model for GUIs, and UML diagrams.
GUI Design	2	*Introduction to User Interface Design*	A basic introduction to some key themes of HCI and user-centered, general user interface design.
	3	*Java GUI Design*	Practical GUI design for the Java platform with some practical examples, introducing the Java Look and Feel design guidelines.
	4	*Recurring User Interface Designs*	Recurring design solutions in desktop applications, with reusable code.

1. In real-world situations users have another popular choice: skip reading the manual altogether.

Part	Chapter	Title	Description
Implementation	5	*Iterative GUI Development with Java*	Building GUIs iteratively using OOP. Introducing software testing, usability testing for Java GUIs, and GUI-specific refactorings.
	6	*Iterative GUI Development with Java*	Introduction to software design strategies and OOP design patterns for GUIs.
	7	*Code Organization*	Main software architectures for GUI applications and some reusable utility classes.
	8	*Form-Based Rich Clients*	An example iterative, test-driven GUI development.
	9	*Web-Based User Interfaces*	Web GUI design basics and related Java technologies.
	10	*J2ME User Interfaces*	An introduction to J2ME GUI technologies and GUI design for wireless devices, with some example code for MIDP.
	11	*Java Tools and Technologies*	A review of the main tools and technologies available for Java application development, with particular focus on open source software.
	12	*Advanced Issues*	Some topics of interest for complex GUIs: building custom frameworks, usability applied to API design, memory management, legacy GUI code, and domain-specific languages for GUIs.

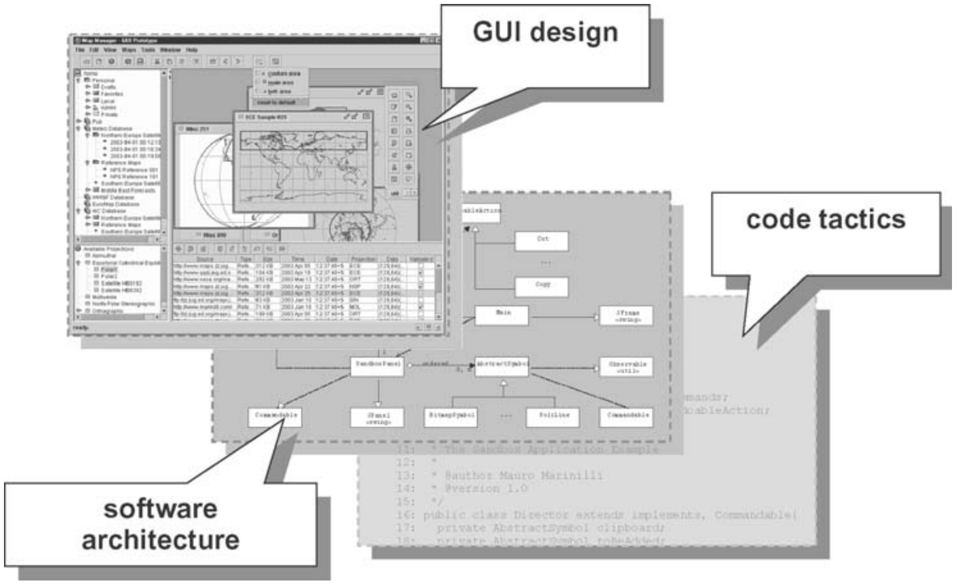
Part	Chapter	Title	Description
Examples	13	*Rich Client Platforms*	Introduction to Rich Client Platforms (RCP) and Eclipse RCP GUI design guidelines, with an example service-oriented GUI for the Eclipse RCP.
	14	*The Personal Portfolio Application*	Design and development of an example application using use cases. An alternative design using JDNC is also discussed.
	15	*An Example OO User Interface*	Using the OOUI approach to design and implement an example application, compared with the use of the Naked Objects framework.
	16	*An Example Ad-Hoc Component*	An example ad-hoc component and its comparison with the *JHotDraw* framework.
Appendixes	A	*A Questionnaire for Evaluating Java User Interfaces*	
	B	*A Questionnaire for Evaluating J2ME Applications*	

Three levels of advice

Building a usable, cost-effective, professional-quality GUI is a complex and multidisciplinary process that involves mastery of many different skills. In this book we will cover three different perspectives: the design of the user interface, the software architecture behind it, and the tactics related to the source code, as shown in the figure below.

The three level of advice in the book

Professional GUIs are carefully designed and implemented pieces of software. For this reason special attention is given in this book to implementation details, especially at the design and architectural level – in my experience the only way to absorb reliably the sort of complexity-by-accretion that real world GUIs exhibit. Source code listings and code-level tactics are mentioned only briefly, to save space and reduce the danger of sending my copy-editor to sleep.



Conventions used in the book

Throughout the book notes are represented using the graphical convention below.

> This is a note.

All references are gathered in a reference section and represented following the Chicago Manual of Style, fourteenth edition, University of Chicago Press, 1993.

Source code

Source code is provided on my Web site at:

`http://www.marinilli.com/books/b1/b1.html`

or you can start from the home page at `http://www.marinilli.com` and follow the links from there. It is organized into separated bundles for each chapter, and a single file containing the code for all the chapters is also available. Sources are provided with Ant build files and with Eclipse projects.

Some of the example applications can also be launched on line using JNLP links, available at:

`http://www.marinilli.com/books/b1/b1.html`

The JNLP client will ask for authorization prior to installing the application.

Reader feedback

A book is an inherently limited means of communication, at least when compared with computer-based interactive tools. In order to balance this unfair equilibrium, a public forum will be available on my Web site for readers to give feedback, pose questions, download the source code, or start a discussion.

Book readers and personas

You might have bought this book, and I do thank you for that. Unfortunately, it is more than 707 pages long and you could not have the time or will to read it all from end to end, neither would it be a time-efficient thing to do. The objective of this section is to help you save your valuable time getting quickly to *your* point, gaining also a first glimpse of techniques for focusing the design around end users.

The book can be used in a number of ways: it is useful for experienced developers that want to explore ideas on GUI development, and can be used in courses on practical GUI design and implementation. Intermediate developers can take advantage of the many examples provided to explore sample implementations.

The book has been designed with three types of reader in mind:

i. Those that have better things to do in life than fiddling with theoretical issues, and just need to put together something that works, now.
ii. Novice readers who want to explore the complexity of professional GUI development.
iii. Those that are experienced and critical about ready-made solutions, and would like a critical and wider discussion of the major issues for GUI development in Java.

The following sections describe a set of fictitious readers, built with the persona technique[1]. If you are lucky enough to recognize yourself as one of them, or somewhere in between, their approach to the book might suit you. Alternatively, if you are one of those brave, tough, developers who think all this Lars and Melinda stuff is a bit silly, skip the following section and jump directly to the first chapter.

1. These user representations are called *Personae* and were introduced by A. Cooper in (Cooper 1999). They are useful for defining the user population clearly to designers, even for a book.

Lars, a Java intermediate programmer

Lars, 24, doesn't have time to waste. In his first glance at the book he sees many interesting things, but he needs to deliver a small (twenty-some screens) form-based corporate rich client application within five weeks. He needs to interface with an existing J2EE application and with a third-party Web service. After a quick look on the Web, he remains a bit confused by the many technologies and options available: he wants to look at some working code and get a clear understanding of how it works, together with some advice to help him to build a bigger picture of the best choices available, without wasting time on other fancy details.

Lars will then…

- Take a bird's eye view of Chapter 6 and an even quicker glimpse at Chapter 7 to see about client tier architectures.
- Perhaps take a look at Chapter 5, to see if there is some useful technique he can take advantage of in his project.
- Read the discussion about SWT vs. Swing in Chapter 11, opting for SWT and the Eclipse RCP for his project.
- Consequently focus his attention on Chapter 13 (RCP), and Chapter 8 (Form-based GUIs).
- Use the quick references on the book's reverse covers as required.
- After his project is completed, get back to the rest of the book…

Keiichi, a tech lead

Keiichi is a technical leader in a medium-sized software company in Japan who wants to explore new ideas about GUI development. He is starting a new project and has some spare time that wants to spend on refreshing his knowledge about GUI design and development. He is particularly interested in the architecture of complex desktop GUIs that provide undo/redo support, complex validation, role-based fine-grained authorization, and more.

Keiichi will then…

- Read Chapter 1.
- Read parts of Chapter 2 and Chapter 3 to get an idea of GUI design.
- Browse Chapter 4 for a quick look at recurring GUI designs in desktop applications.
- Read Chapter 5 about iterative GUI development – a subject in which he is quite interested.

- Have a look at Chapter 6 for some implementation strategies and common issues related to complex GUI development.
- Take a quick look at Chapter 7 and Chapter 8 for completing his introduction to the implementation of complex rich client GUIs with Java.
- Read Chapter 12 about advanced issues and ideas for implementing non-trivial GUIs.
- Browse the example applications in the third part of the book.
- After adding notes and bookmarks, put back the book on his shelf, promising himself to get back to it when the new project has started…

Shridhar, a professor in computer science

Shridhar is an assistant professor in a university in Kanpur, India. He is 35, married with two children. He is preparing a course on the practical development of complex GUIs. He wants to include the essentials of user interface design, advanced software design patterns, and many case studies that will form the backbone of the course. He bought the book on line to evaluate its adoption as a reference textbook for the course, integrating the parts in which he is more interested with other material. Shridhar finds some companion material for the book on line and plans to use it for his course. In particular, he is interested in using SWT for an interesting research project.

Shridhar will then…

- Organize his course content around the functional model introduced in Chapter 1.
- Plan to devote the first part of the course to GUI design issues, based on Chapters 2 and 3.
- Use Chapter 6 as the theoretical base for the second part of his course, about implementation of complex GUIs in Java.
- Use the examples in Chapters 13–16 for the case studies. He plans to extract software design patterns and architecture contents from these chapters to use in the hands-on part of his course.
- Think about creating assignments based on the ideas provided in the various chapters he has read. Because he is interested in the Eclipse RCP and SWT, he will focus on the ideas discussed in Chapter 8.

Melinda (Mellie), a manager

Mellie has a technical background and a basic overview of object-oriented technology. She wants to have an overview of current technology for GUI development with OOP, and feels that she needs to refresh her knowledge of current state-of-the-art development of client-side software. She worked in software testing back in the 1980s, and is now a senior group manager in the IT department of a medium-sized insurance company that does some in-house development. She wants to get a basic, high-level understanding of the latest trends in GUI design and development.

Mellie will then…

- Read Chapter 1 about the development context of GUI design and implementation.
- Interested by the topic of GUI design, move to study Chapter 2 for an introduction to basic user interface design.
- Read Chapter 3 for an example of an OOP GUI technology stack, showing guidelines, practical GUI design examples, and other technology-oriented topics that can also be used outside the Java world.
- Take a look at the pictures in Chapter 4, to see the most commonly-used GUI designs in real world applications, and try to match them with her daily practice of software applications (a mix of Microsoft Project, the Microsoft Office suite, and some corporate intranet applications).
- Have a look at Chapter 5 to get an idea of iterative GUI development.
- Note a few useful terms to be inserted in her next presentation, such as usability inspections, continuous profiling, and more.
- Eventually, put the book in her ‘favorites’ pile, hoping to have more time for it another day.

William, a first year student in a Master in CS course

William has just moved to Vancouver and is excited about starting his masters program in Computer Science and eager to become a proficient software developer. He already has some exposure to Java and the Swing toolkit, and he knows that he is going to have some courses about these topics. He wants to know more about software architectures and how complex desktop GUIs are built, possibly starting his own open source project.

William will then…

- Start reading the example applications, looking for interesting situations and trying to understand the proposed solutions.
- As he is interested in the Sandbox application discussed in Chapter 16, download and compile the source code, tweaking it to add new features.

- Jump to Chapter 6 for the theoretical background behind the implementations proposed in the example applications.
- Turn his interest to the qText application in Chapter 6, studying its simple architecture, downloading the code, and adding new commands to the editor.
- Browse the rest of the book as he needs to.

Karole, a business analyst

Karole has a degree in programming and works for a software company. While working as a full-time analyst on her current project, she discovers that she enjoys dealing with customers. She feels she would like to work more on the GUI side of software development and move into GUI design. She would like to get a wider picture of GUI development, using Java as a practical example, but also be exposed to more general concepts.

Karole will then…

- Read Chapter 1 about the development context for GUI design and implementation.
- Study Chapter 2 for an introduction to basic GUI design advice.
- Read Chapter 3 for a discussion of GUI guidelines, practical GUI design examples, and other technology-oriented topics that can also be used outside the Java world.
- Study Chapter 4, to understand the most commonly-used GUI designs and the rationale behind them.
- Perhaps read Chapters 5 and 8 to gain a better grasp of the latest iterative development techniques for client applications.
- Snoop around the rest of the book as required.

Juan, an experienced programmer

Juan is an experienced programmer in his late twenties living in Schaumburg, IL. He has just bought the book in a bookshop and is excited about it. He has some spare time, an hour or two, on a Saturday morning. He wants to browse the book for something fun, taking it easy, while sipping his favorite blend of Cappuccino in a café while waiting for his fiancé Francene, who is having her nails done. Juan is looking for cool new technologies, interesting application architectures, exciting techniques, or just a cartoon or fancy pictures before a long shopping session with Francene[1].

Juan will then…

- Browse Chapter 4 for a quick glimpse of common GUI design issues, such as choosers, area organization, and so on.
- Have a look at some of the pictures of the various look and feels in Chapter 11.
- Look at Chapter 10 for information about J2ME GUIs, and have a quick look at Chapter 9 for Web Java GUIs.
- Have a glimpse at some of the techniques discussed in Chapter 12.
- Take a look at the various pictures of the example applications in the third part of the book.
- Then, when he has more time, get back to this section to find another fictitious user who matches his needs, so that he can start seriously reading the book.

1. This is not a spurious use of a technical book, as it might seem at first. Establishing a positive emotional relationship with something we need in our work life is always a win-win situation. Working with something pleasant will make us feel better, being more productive, and perhaps sparing precious energy for something other than dull work.

1 Putting GUI Development into Context

This chapter provides a comprehensive introduction to the design and development of Java applications with non-trivial user interfaces. After introducing a general-purpose reference model that will guide our discussion in the remainder of the book, we introduce the organizational aspects related to UI development, discussing the role of people in the entire software lifecycle process for GUI software. We then consider the issue of early design, where we briefly introduce the delicate and often overlooked transition from analysis to UI design. A section is devoted to some interesting lifecycle models and the way they support the process of building professional user interfaces. The chapter concludes with a minimal introduction to some useful UML notation that will be used throughout the book.

The chapter is structured as follows:

1.1, Introduction briefly discusses the current state of GUI technologies and the use made of them by developers.

1.2, Focusing on users discusses user-centered design and development throughout the software lifecycle.

1.3, A functional decomposition for user interfaces introduces an abstract model for GUIs that is used throughout the book.

1.4, Tool selection: the Java singularity discusses the selection of a set of ingredient libraries technologies, many of them open source, to speed up GUI development.

1.5, Organizational aspects introduces some of the issues related to the management of the multidisciplinary teams that are common in GUI development.

1.6, Early design introduces requirements and use cases for professional GUIs.

1.7, Lifecycle models, processes and approaches briefly introduces some software lifecycle models: Rational Unified Process, Extreme Programming and other Agile approaches, the LUCID methodology, and evolutionary prototyping, focusing on GUI design and development.

1.8, UML notation introduces some UML diagrams of interest that are used throughout the book.

1.1 Introduction

The wealth of GUI design options provided by rich client GUI technologies is still poorly mastered by developers struggling to provide remarkable designs in a cost-effective way. This is what happens when powerful media and technologies lack widespread, deep expertise and practical support.

The same thing used to happen two thousand years ago. Pliny the Elder, an ancient Roman scholar and encyclopedist, despised his compatriots' paintings and preferred Greek classic art. His complaint was about the use of newer techniques that exploited a much wider number of colors, while Greek classic paintings used only four colors. Today we have a chance to see these much despised 'excessive' paintings, thanks to the catastrophe that buried Pompeii in 79 AD, freezing a moment of history in one of the most rich and developed areas of the age – similar to what California represents to Western civilization today – and, unfortunately, accidentally killing Pliny. Surprisingly, these miraculous survivals show a realistic and powerful use of the newer – and much harder – techniques, together with some unskillful art works.

Moving from Roman paintings to user interfaces, in the 1980s the computer industry experienced a similar mass-market technology shift in visual technologies with the introduction of powerful raster graphics with millions of colors, large dedicated memory spaces and new, ad-hoc input devices. Today, after another twenty years or more, we are in a situation no different from the Roman paintings of the 60's AD. These new technologies provided a steep increase in complexity, and developers (like the Roman painters of Pliny's age) are still struggling to tame such power for building cost-effective, usable and enjoyable GUIs.

1.2 Focusing on users

The most striking difference between designing and building a desktop application GUI and other software is the presence of the user. Users are those that will ultimately use the product, but in current development-centric engineering settings, they are usually completely neglected. 'Focusing on users' means focusing on human details – cognitive factors such as perception, memory, learning, problem-solving and so on – rather than implementation factors such as system and business requirements, software architecture, hardware, and so on. User-centered design is a well-established set of practices that place users at the heart of GUI design and development. This is currently the only way known to obtain software that behaves as users expect, ideally becoming transparent to them – they don't realize they are using it – and not getting in the way of getting work done. Adopting, or even merely being aware of, the user-centered approach is critical, not only in the design phase, but throughout the whole development process.

A number of practices have been established for centering the design and overall iterative development on end users:

- Understanding users, their objectives, their current working practices, and the general context in which the software will be used, all of this before starting the design of the user interface.
- As part of this, an important role is played by two deeply intertwined central issues: users and their tasks. User analysis–providing groups of users with their goals – and task analysis – breaking down tasks in smaller subtasks – are two disciplines that aim at defining these issues in useful terms.
- Involving end users or user representatives in the design from the early phases. This practice is referred to as *participatory design*.
- A useful means for understanding users is to interview and observe them while at work in their normal work environment. Techniques such as *contextual enquiries*[1] and adopting an ethnographic[2] approach to user studies are widely used in this respect.
- Usability tests help to ground an application on user's needs after various iterations of design and development.

We will see user-centered techniques applied throughout this book, but apart from these techniques, it is essential to always bear in mind that being aware of the end user – playing the role of the *advocate of the user* – is essential in producing a professional user interface, especially on fast-paced projects in which it is hard to fully apply these techniques when other, more urgent deadlines are pressing.

1.3 A functional decomposition for user interfaces

Graphical user interface applications are a vast class of software systems with recurring properties. In a GUI there is always a portion of the screen that is designed for interacting with users, there are various forms of reactions to user interactions, perhaps through some form of an internal representation of the business domain at hand, and so on. Decomposing these functionalities into a set of

1. During a contextual enquiry, several potential users of an application or a process that we want to capture in software are observed in their day-to-day work. The interviewers focus on a specific objective and adopt a partner-like approach with users, rather than being judgemental or inquisitive.
2. Ethnography is a method of studying and learning about groups of people. Typically, it involves the study of a small group of subjects in their own environment in order to develop a deep understanding of them.

layers is useful as a key to aid discussion of the various aspects of GUI development, as a reference for discussions, and as a conceptual tool to tame the complexity of GUI design and development. This is illustrated in Figure 1.1.

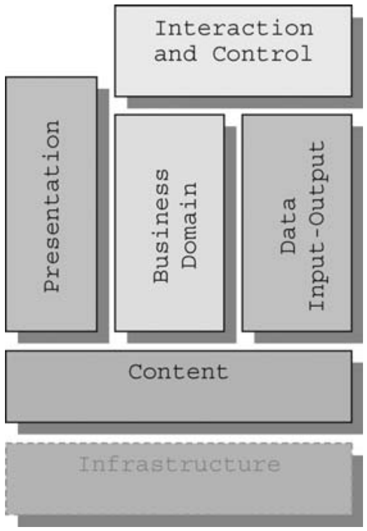


Figure 1.1 An abstract model for user interfaces

The layers in our reference model are:

- *Business Domain*. A representation of the domain of interest, separated from GUI and other non-business details.
- *Content*. The 'structure' of the GUI: widgets, windows, and navigation flow among different windows, screens, and so on.
- *Data IO*. The interface with the rest of the world other than the end user. Data formats, communication protocols and the like are represented by this layer.
- *Infrastructure*. Low-level support, runtime environment, utilities, and so on. The graphical toolkit of choice, libraries, frameworks and hardware support belong in this layer.
- *Interaction and Control*. Low-level events and control logic are gathered in this layer. Note that this layer can be thought of as the 'glue' that holds the rest of the GUI implementation together.
- *Presentation*. This layer represents graphical details that are dependent on the given presentation technology, such as pixels, colors, and fonts. This layer can be thought of as (theoretically) orthogonal to the other layers.

An example of the application of this functional model to a simple form-based GUI is shown in Figure 1.2.

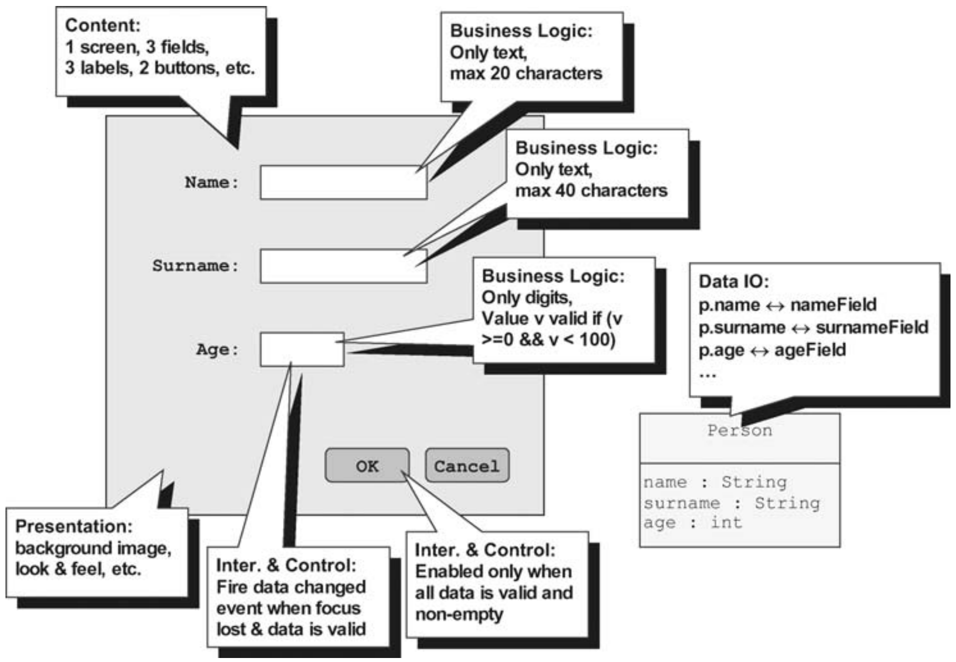


Figure 1.2 Applying the abstract model to a simple form GUI

The figure shows a very simple form-based GUI that has been decomposed using our model. Graphical aspects, no matter how implemented, belong to the presentation layer. Widgets and their layout are part of the content layer. Widget's behavior in reaction to user input – for example, the 'Age' field accepts only digits – is enforced at a low level by the interaction and control layer, but the ultimate logic lies in the business domain layer, where it is defined that an age is a numeric, integer entity with values between 0 and 100. It is the responsibility of the interaction and control layer to understand when the user has completed data input and how to handle invalid values.

This model is general – for example, it could be applied to Web GUIs, wireless device applications, touch-screen kiosks and so on – and somewhat arbitrary. It is just one of the various possible decompositions of GUI functionalities: it focuses on simplicity and practicality, and is independent of the particular implementation technology. You can use this model to represent any existing desktop application GUI, or to organize the development of new ones.

We are going to apply it to Java GUIs, which are high-level component-based user interfaces based on a strongly-typed object-oriented language and on an operating system-independent execution platform. This simple model will form the basis of our discussion about user interfaces in Java. We will analyze complex GUI implementations using this model, discussing disparate GUI technologies, and we will use it as a software architecture for one of the many examples proposed in the book.

1.4 Tool selection: the Java singularity

In this book we will discuss GUIs of any size and complexity, all built with Java technology. One of the most striking aspects of Java is its openness and the wide range of companies, tools, and technologies that flourish under its umbrella. This applies over a wide array of hardware, ranging from wireless devices and card readers to powerful back-end enterprise servers.

Of course, similar to Pliny's paintings, and the line of Alan Kay's 'inverse vandals' in the introduction, there are wild differences in quality. Along with open source tools with a rock-solid reputation, such as Hybernate and some good pieces of server-side facility, and in many other application domains, we have hundreds of ill-documented, partially working, hard-to-use libraries and would-be tools. Nevertheless, the whole Open Source Software (OSS) movement is a remarkable feature of Java technology, typical of the cooperative spirit that dominates this community.

A characteristic that assumes special significance for Java projects is the tool selection phase. With the sheer abundance of tools and technologies – *thousands* of OSS tools for Java development – the selection of the 'best' set of tools for a Java project can prove hard. Some choices can be changed later with acceptable cost, such as for example switching to another issue tracking system, as long as the old one provides some data export facility, but others cannot be changed without throwing away most of the work done. Choosing the right presentation technology – specifically, choosing between Swing or SWT – is a strategic choice that cannot be reversed easily. Peer opinions, Web forums and the like often provide biased opinions, or might not take your particular context and needs in account.

Commercial tools are usually better than OSS ones, but the same care in selection and evaluation should be applied. My personal experience has guided me to start with an OSS tool, and then, only if really necessary, move to a commercial one: buying a tool of which you have no previous experience can prove a costly mistake. It is wiser to start with an Open Source alternative and use it as long as possible, and this will also help to clarify your real requirements.

One thing that always strikes me when I get involved in a new project is the carelessness shown in choosing portions of the base tool set. Sentences like 'Oh, well we started off with 'X' and 'Y' together with 'Z' because they were available on the market and…' – and then usually there is a pause. Sometimes in large teams developers don't even know who started using a particular XML library or GUI testing tool, 'We just tried it, and it worked.' Then, after months of quiet work, they find themselves dealing with, perhaps, unmanageably huge XML files for acceptance tests[3].

Of running little green men and wrong choices

A little green man dashing to a door is the universal icon for emergency exits in public buildings. Similarly, alternatives and emergency plans should always be considered when choosing a project's ingredient technologies. When preparing a list of technologies for a project, let's not forget to make an alternative list with the emergency sign icon on it, because some of choices we must make are irreversible without losing much of the work already completed.

This process is like going to the forest to pick mushrooms. They are free, and they could be so tasty – but they could be poisonous. As every developer knows, tools usually work well at first, on simple tests and in recurring situations. But after some month of use, or, even worse, after more than few months, you may find that a tool can no longer support what we want to do, and you are then faced with the need to switch to another solution. Sometimes it is impossible to step back, so that there are no emergency exits for the situation. Rarely is technology the real problem, although it is always a great excuse: most of the time it is having access to the right people that will make the difference.

I have made several mistakes in tool selection. Some were inevitable at the time, others were just my fault. The most classic mistake I made was to see problems like nails to be driven in with my favorite hammers[4], the tools I was most familiar and confident with. At other times, more often than the wrong technology, the problem was a wrong adoption of a given technology.

1.5 Organizational aspects

Developing a non-trivial professional GUI is perhaps the most interdisciplinary kind of task to be found in software development. Many different roles need to interact closely: the alchemy of such interaction is so delicate that the resulting

3. Chapter 11 is entirely devoted to the practice of tool selection.
4. To paraphrase the American psychologist Abraham Maslow, '*If you only have a hammer, you tend to see every problem as a nail.*'

final outcome may be disappointingly poor, despite the dedicated people involved and the substantial resources employed.

The user interface of an application is the most visible part of a software product and the part where people external to the development team clash most. In some projects non-developers in the organization feel entitled to advise on the user interface, especially higher management. Repercussions on the UI may be not explicit or even foreseen – for example, decisions taken in database design can affect the UI, or the absence of a capable graphical artist can influence usability, thus the overall performance of users, and so also of the system, right down to the back-end servers.

People and GUIs

In this section we explore some of the issues related to the development of user interfaces and the involvement of the people who build them.

Dermaphobic and graphic hedonists

GUIs are software artifacts with a strong human component. This is apparent for end users, but it is true also for developers. In particular, there are usually two main types of developers, of which the first is the more common: the GUI-phobic developer.

There is a widespread tradition of distaste for GUI-related development. GUI toolkits are perceived as cumbersome, complex and ultimately useless – 'It's just cosmetics,' 'I've more to do than struggle with pixels.' There are a large number of implicit assumptions behind this attitude. All the 'real action' goes on the server side, and putting GUI technical skills in a resume is seen as something to avoid, like a sound engineer who has worked on lipstick design.

This implicit phobia for the 'skin' of client applications, a sort of software dermaphobia, surfaces in many ways and in various aspects of development. Sentences like 'We need to make our application totally decoupled from the GUI layer,' 'Completely hide the presentation technology,' 'I don't have the vocation for GUI stuff,' or 'Let's hand this to GUI specialists' are indicators of such ancestral fears. Of course, hiding and decoupling are good qualities for any software, and GUI toolkits tend to be complex and frustrating to master at first, but this phobic attitude can only harm a project. The GUI-phobic developer often puts together something that works, maybe by cutting and pasting some tutorial code found on the Web, and then rushes back to a nobler task.

Sometimes GUI specialists are just developers who find it harder than others to say 'No,' or that just don't want to be on the front line. You can hear them rationalize their phobias: 'We decoupled things so that we are GUI toolkit-independent' is the official line, but a closer inspection of the code shows that this decoupling doesn't work in practice and that is not even required at all.

The other dangerous class of developers, although much rarer, is the GUI enthusiast, such as those who can happily spend an entire working day finding the perfect gradient texture for the company's new look and feel.

Both these kinds of people tend to perceive GUI development as a developer-centric activity: the end user experience is just a by-product of a simple-to-build and possibly fun implementation.

Developers' attitude towards GUI development, and the resulting architectural choices, shape the way the final product will look. GUI-phobic developers tend to build bulky, low-bandwidth GUIs with fewer interactions, while GUI enthusiasts tend to present useless fanciness to the user while overlooking more substantial features. In both these extremes, the overall development cost is higher and the quality of the final product is compromised.

Who owns the GUI?

Apart from simple cases in which only a small number of people are involved in building simple applications, the implementation structure has social ramifications. Abstract, formal decisions about an architecture or, worse, no substantial decisions at all, affect the real nature of the implementation structure only shallowly, as mentioned in countless books[5].

If no strong force is at work in a project, developers will 'own' the development process. This can be fine for server-side applications, but needs special care for client-side applications. The most common owners of a project are its customers or other such stakeholders – although they rarely correspond to the application's final users. When a project is owned by its customers, a number of issues may arise that are specifically confined to GUI design – we detail these in Chapter 2 and 3. In these cases, though, the implementation can be designed to reflect this climate by providing effective mechanisms to absorb change at the GUI design and interaction level.

Particular care is needed in those cases in which the project will build a product or a service in a market where existing alternatives are available. Shareware software, or competing Web sites, are examples of software that is ultimately chosen by the end user, in contrast to, say, an intranet corporate portal or an ERP application, whose end users have no power of choice. In cases in which users have a low barrier to switching to a competing product or service, extra care is needed to safely ground ownership with the end users, or there is a risk of producing software that nobody

5. We would just mention here two classics: (Brooks 1995) and (De Marco and Lister 1999).

will buy. This is achieved by adopting a fully-fledged user-centered design approach with extensive usability testing and feedback from users, and in which developers and the other stakeholders are constantly focused on the end user ownership.

'Ownership' dictates the overall attitude toward the implementation. If you ask a developer what a good architecture should be, they can hold forth for hours, mentioning powerful virtues like usefulness, robustness, maintainability, scalability, agility, responsiveness, extensibility, fitness to purpose, and so on. Customers, in contrast, are often dangerously vague. For a user, anything is fine 'as long as it helps with the business.'

Cost can be a major factor as well. Projects driven by cost tend to have their implementation and architecture deeply shaped by their financial climate.

An often overlooked aspect of any technical decision (languages, architectures) is the emotional connotation people attach to it. One developer may not like Swing (or SWT), while another might find it a wonderfully comfortable choice. Architectures, tools and approaches have their own advantages and drawbacks, but they are merely instruments to aid in to solving the problem at hand. It is dangerous to let our feelings drive critical choices biased by personal feelings, as choosing the wrong tool can prove disastrous in the long term.

Team composition

GUI development is a multidisciplinary activity that involves a number of diverse skills. Here are some of the roles involved in a GUI development project:

- *UI designer*. This role is responsible for driving the UI design and ensuring a UI's usability, enhanced after usability testing.
- *Analyst*. Part of the analysis phase is often performed by means of discussions about user interface prototypes.
- *Developers*. Programmers are the main resource in building a professional desktop application GUI. The wide range of scenarios and requirements make the use of GUI application frameworks and rich client platforms impractical in some situations. Developers and labor-intensive development is the only practical way to achieve professional GUI applications.
- *Application architect*. This role is perhaps the most important of all. A GUI architect must be knowledgeable about GUI design, GUI implementation technologies, programming, business and application domains, and server-side issues, as well as being capable of dealing effectively with customers and other stakeholders. Architectural decisions impact directly on the GUI. For example, the decision to adopt a Web service architecture for client–server

communication dictates the kind of interaction available on the client GUI. The application architect is needed effectively to bridge the gap between customers and end users' unclear needs and the detailed information required to translate such needs into working code.

- *Usability expert*. This role oversees usability issues throughout the whole application lifecycle.
- *Graphic artist*. An artist design icons, colors and other graphics for the application. Rich client applications have a wide range of graphical possibilities, much wider than Web applications. This power can be misused, producing confusing and unusable GUIs, if not properly mastered.
- *Business domain expert*. People expert in the client's business domain should work closely with GUI developers to ensure that the GUI reflects the actual business domain terminology, skill, procedures, and so on. If a domain-driven approach has been adopted for developing a rich domain model, effort should be expended to verify with expert users that such a model doesn't remain buried behind the scenes, away from the user interface and the end users, wasting the effort required for its creation.
- *Client management*. The management of the client organization can play an important role in the development of the GUI.
- *Stakeholder*. This generic term includes any person or organization that may be affected by the success or failure of the software project. End users, developers, and managers are examples of stakeholders.
- *UI tester*. Personnel skilled in GUI testing and GUI testing tools.

Quality assurance[6] *experts*. The feedback from the QA team involves the user interface.

Of course, depending on the project, many of these roles might be performed by the same person or team.

The composition of the team that will design and build the GUI is also important. A multidisciplinary development team is essential to achieve a high-quality design. The contribution of people with different backgrounds and points of view is extremely important in building a professional GUI.

For example, a graphic artist is indispensable, even if only working part time as a consultant. You can see the difference a good artist can make by looking at the (very unprofessional) icons used in this book – excluding the standard ones from the graphics repositories from Sun and Eclipse.

6. We use the more general term 'quality assurance' without distinguishing it here from 'quality control,' although they are in fact distinct disciplines.

In very small development teams a common problem is the 'usability death spiral': if they don't try it out with external people, either other colleagues or end-users, developers get accustomed to their own design. The longer a developer – either a designer or a programmer – deals with building a GUI, the more reasonable and usable it appears to be to them!

1.6 Early design

Requirements are the backbone of any analysis. Requirements should be:

- *Clear* and *unambiguous*, and usually expressed in natural language.
- *Complete* – that is, covering the whole system.
- *Consistent* – they should form a consistent set of constraints for the system.
- *Testable* – for requirements than cannot be made testable, one cannot prove their fulfilment.
- *Traceable* – it's usually a good idea to establish a hierarchy among requirements, so that is possible to trace lower-level or newer system requirements to older or more general ones.

Traceability can be also done graphically. We could trace requirements or their equivalent counterpart, such as acceptance tests in XP practice, directly to screen areas in our GUI. Chapter 2 introduces a general technique, A3GUI, that can be used to tag screen areas with requirements or other useful information.

Use case diagrams and GUIs

In this book we will use UML notation extensively. This section introduces UML use cases and class diagrams, a popular analysis and documentation device. Use case diagrams are especially useful for defining functional requirements in the early stages of GUI design[7].

There are many books on UML: in particular we will refer to (Fowler 2003). Although not strictly related to user interface design, use cases are commonly used in real-world development for describing the requirements for a given application. UML use case diagrams are used as the preliminary stage to elicit the expected features of a software artifact.

7. We assume that the reader is already familiar with UML notation for use cases.

Use case diagrams describe a system in terms of the functionalities provided to its users. They consist of actors and use cases. Actors are entities external to the system that interact with the use cases, such as human users, other systems, and so on. These in turn are generic functions the system provides to the rest of world.

A single function can be thought as a flow of actions. The example in Figure 1.3 shows a simple use case diagram that describes an arcade video game. We have modeled the system with one external actor only, Player, and three main use cases: join the game, play the game, and insert a high score. Possible actions could be: push the 'start' button, insert coins, push the 'fire' button, and so on.

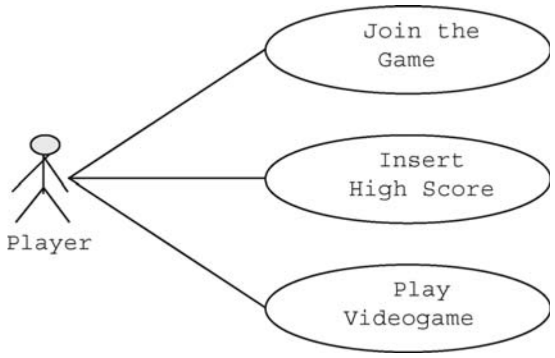


Figure 1.3 An example of use case diagram for an arcade video game

Use case diagrams are often used as inputs to the GUI design process, because they identify actors and functionalities within the system. *Scenarios* are used to represent a set of paths of possible events through single use cases. Scenarios are often described by means of natural language.

A possible scenario for the application in Figure 1.3 could the following:

- Player inserts two coins into the game console
- Player pushes the 'one player' button
- Player plays the game
- The game is over
- Player breaks the record and inserts their name into the high score list

Scenarios are also used, often in a more complex way, as a technique for identifying the typical interaction paths of a user interface. The next step could be to refine the previous scenario, including a first description of the system behavior. This is shown in Table 1.1.

Table 1.1 An example of a scenario from an arcade video game system

Player	System
The player inserts a coin into the console.	The system shows the message: 'Insert another coin and try this game.'
The player inserts one more coin into the console.	The system displays its availability to join the game by pressing the 1P button.
The player pushes the 1P button.	The player joins the video game, starting a new game.
The player plays the game.	The system engages the player in the video game.
The game is over.	The system shows the message: 'Game Over,' signaling the possibility of joining the game again by inserting more coins.
The player breaks the record and inserts their name into the high score list.	The system displays the high score list and lets the user insert their name.

Use cases can be refined into more general and more detailed ones. Use case diagrams say nothing about the implementation of the system. Use cases are not the functional modules of the system: rather, they are functionalities offered to external actors. UML does not prescribe how use cases should be represented – they can be described in any way, although usually as a list of numbered items.

Apart from narrating the user's experience of the system, use case diagrams can be helpful in understanding how use cases relate to each other, such as frequent functionalities instead of critical ones, or the possible event sequence's interactions, or as a way to expose the system analysis to customers.

For connections between use case diagrams and user interface prototyping, see for example (Elkotoubi, Khriss and Keller 1999), (Shirogane and Fukazawa 2002). Later in this chapter we will see an example of an extension to use case models to account for GUI design and usability.

1.7 Lifecycle models, processes and approaches

This section sets user interface development in the wider perspective of the whole software lifecycle. If we are to have usable software, it is essential to focus the whole design and development process around usability and GUI design issues.

We will introduce some different approaches to modeling the software lifecycle that take GUI design into particular consideration.

Rational Unified Process

The Rational Unified Process, or RUP, is a software engineering process made up of a number of best practices, workflows and various products (here called *artifacts*).

The key aspect of RUP lies in its iterative model for software development. RUP organizes projects in terms of disciplines and phases, each consisting of one or more iterations. There are four different phases: *inception, elaboration, construction,* and *transition*. The importance of each workflow depends on the given iteration. Using an iterative approach makes the development process more robust, with demonstrable progress and frequent executable releases.

Don't confuse RUP with UML. UML is a modeling language for software systems, while RUP is a software engineering process that provides a controlled approach to assigning and managing tasks and responsibilities within a development organization. RUP uses UML notation extensively in its guidelines and best practices, however.

RUP supports the following best practices:

- Develop iteratively – that is, adopt an iterative lifecycle model
- Manage requirements explicitly
- Use component architectures – a wise use of OOP plays an important role
- Model visually – that is, adopt UML
- Manage change in the form of a number of best practices
- Continuously verify quality, an essential aspect for minimizing risk

RUP defines a set of roles for modeling people involved in activities. One actual person can have the responsibility for many roles. For example, a 'stakeholder' role can represent customers, end users, buyers, and so on, or anyone who represents them in the developer's organization.

A *discipline* in RUP terminology is a group of homogeneous activities that shows all the different procedures needed to produce a particular set of artifacts.

RUP considers the following disciplines:

- *Business Modeling* is a discipline that aims at comprehending the structure and the dynamics of the target organization – that is, where the system will be deployed – to understand problems and identify possible solutions within such an organization.

Business modeling aims to ensure that customers, end users, and developers have a common understanding of the target organization, produce a vision of the new target organization, and based on that vision, define the processes, roles, and responsibilities of the organization in a business use-case model and a business object model.

- *Requirements*. This discipline aims to establish and maintain an agreement with customers and other stakeholders about what the system should do, the definition of the system's boundaries, an estimate of the technical contents of iterations, and the cost and time to develop the system. Part of this discipline is to define the user interface, focusing on the needs and goals of the users. As a part of this activity stakeholders are identified, together with their requirements.
- *Analysis and Design*. The objective of this discipline is to transform the requirements into a design for the future system.
- *Implementation*. The purpose of implementation is to define the organization of the code, in terms of implementation modules, to implement classes and objects in terms of components (source files, binaries, executables, and others), to test the developed components as units, and to integrate the results produced by implementers or development teams into an executable system. Unit testing is included in implementation, while system test and integration test are part of the Test discipline.
- *Test*. This discipline oversees the proper integration of all software components. It verifies that all requirements have been correctly implemented, and tries to isolate all defects prior to software deployment.
- *Deployment*. Prior to deployment the software is tested at the development site, followed by beta-testing before it is released.
- *Environment*. This discipline focuses on the activities necessary to configure the process for a project. It describes the activities required to develop guidelines to support a project. The purpose of the environment activity is to provide the software development organization with the software development environment – both processes and tools – that will support the development team.
- *Project management*. The objective of this discipline is to provide a framework for managing software-intensive projects, providing practical guidelines for planning, staffing, executing, and monitoring projects, as well as to provide a framework for managing risk. This discipline focuses mainly on the important aspects of an iterative development process: risk management, planning an iterative project, both through the lifecycle and for a particular iteration, monitoring progress of an iterative project, metrics.

- *Configuration and change management* (CM and CRM). These disciplines involve identifying configuration items, auditing changes, restricting access to those items, and defining and managing configurations of those items. A CM system is an essential and integral part of the overall development processes. It is useful for managing multiple variants of evolving software systems, tracking which versions are used in given software builds, performing builds of individual programs or entire releases according to user-defined version specifications, and enforcing site-specific development policies.

To describe what the system will do, RUP requires that a number of documents be written: a vision document, a use-case model, a number of use cases, and eventually a supplementary specification document.

- The vision document provides a complete vision for the software system under development, and supports the contract between the customer's organization and the developer's organization. It is written from the customers' perspective, focusing on the essential features of the system and acceptable levels of quality. The vision should include a description of the features that will be included, as well as those considered but not included.
- Use cases focus on describe functional requirements. A use case describes a significant amount of functionality using narrative text. The use-case model serves as a contract between the customer, the users, and the system developers for the functionality of the system, which allows customers and users to validate that the system will become what they expected, and system developers to build what is expected.

  The use-case model consists of use cases and actors. Each use case in the model is described in detail, showing step-by-step how the system interacts with the actors, and what the system does in the use case. Use cases function as a unifying thread throughout the software lifecycle: the same use-case model is used in system analysis, design, implementation, and testing. A use case should always describe the intended functionality – what a system should do – and not *how* it will be done.
- The supplementary specifications are an important complement to the use-case model, because together they capture all software requirements, both functional and nonfunctional, that need to be described, to serve as a complete software requirements specification.

Complementing these documents, the following are also developed:

- A requirements management plan. This specifies the information and control mechanisms that will be collected and used for measuring, reporting, and controlling changes to the product requirements.

- A glossary, defining a common terminology that is used consistently across the project or organization. Note that the glossary can overlap with the Ubiquitous Language[8] document if Domain-Driven Design has been used in the project.
- Use-case storyboard and user-interface prototype, both results of user-interface modeling and prototyping, which are done in parallel with other requirements activities. These documents provide important feedback mechanisms in later iterations for discovering unknown or unclear requirements.

The RUP project structure is usually represented in two dimensions:

- The horizontal axis represents time and shows the lifecycle aspects of the process.
- The vertical axis represents the disciplines (Business Management, Requirements, Analysis & Design, Implementation, Deployment, Configuration and Change Management, Project Management, and Environment).

  This is illustrated diagrammatically in Figure 1.4. The first dimension represents the dynamic aspect of the process as it is performed. This is expressed in terms of phases, iterations (initial, elaboration, construction and transition), and milestones. The second dimension represents the static aspect of the process: how it is described in terms of process components, disciplines, activities, workflows, roles, and artifacts. The graph shows how the emphasis varies over time.

The key difference between small and larger projects is the level of formality used when producing the different artifacts: project plan, requirements, classes, and so on. Furthermore, only a limited number of artifacts can be produced by small projects.

Use cases alone do not specify user interface details. Perhaps the most common objection against RUP from GUI designers is the strong bias for requirements over design aspects.

This has been addressed in a number of ways, providing custom approaches and various extensions to the standard process. As an example of these customizations, there is an optional extension to RUP called the User Experience Model, or *UX*, for handling GUI design issues (Kruchten and Ahlqvist 2001), (Conallen 2002). Building a UX model is a non-trivial task, needed only when the GUI design needs a special focus within the whole project.

8. Rather than a methodology, Domain-Driven design is an approach and a set of techniques aimed at dealing with the construction of software for complicated business domains: see (Evans 2004). 'Ubiquitous Language' is one such technique, focusing on building a language that defines the domain model and is used by all team members to connect their activities, including the construction of the software.

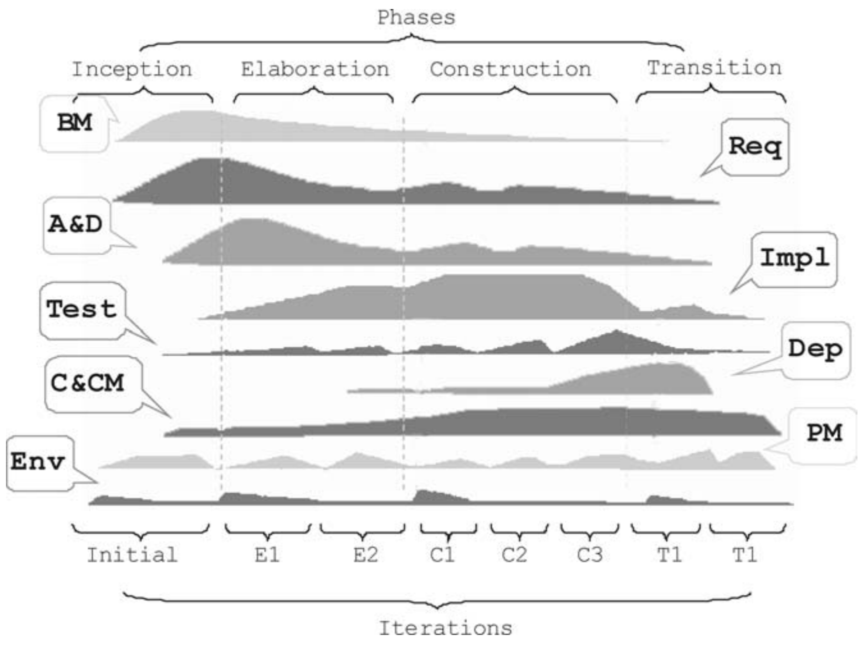


Figure 1.4 RUP Overview

User experience storyboards

The User Experience Model bridges the gap between analysis and GUI design, enriching the use case model with GUI design information. The UX model is a conceptual model that specifies visual elements (the content layer) in an abstract representation. It helps architects and GUI designers determine what will go into the UI before committing to technology details such as widget toolkits and GUI technology.

A UX model and its storyboards describe actors (user characteristics) and screens, as well as input forms, screen flows, navigation between screens, and usability requirements. The actor characteristics and usability requirements are added to the use-case descriptions. The other elements are described in UML and remain part of the UX model.

The two most important RUP disciplines relative to UX storyboards are Requirements and Analysis & Design:

- Use cases are developed in Requirements, while in Analysis & Design they are used to design the system, including the UI. RUP uses models to represent the various parts of a software system. The use-case model is the most important one to build in Requirements.

- The design models are developed in Analysis & Design. For systems with a significant amount of user interaction, the development team should also create a UX model and storyboards within that discipline. This integrates usability issues into the RUP development approach.

There are five steps to creating UX storyboards:

1. Add actor characteristics to the use case. Being non-functional, this information should be added in the special requirements section. This may include the users' average level of domain knowledge, general level of computer experience, working physical environment, frequency of use of the system, and the approximate number of users represented by an actor.
2. Add usability guidance and usability requirements to the use case. Usability guidance provides hints on how users should use the system, including average attribute values and volumes of objects, and average action use. Usability requirements might specify how fast a user must be able to do something, maximum error rates, maximum number of mouse clicks, learning times, and so on.
3. Identify UX elements. UX models use the same appearance as UML but with a different meaning: screens are rendered with UML classes, using the special stereotype «`screen`», navigation maps are expressed with class diagrams, screen instances with objects, and screen flow diagrams with UML sequence diagram.
4. Model the use-case flows with the UX elements.
5. Model screen navigation for the use case using UML navigation diagrams. These are essentially class diagrams with oriented links for navigation.

UX is just one possible approach to capturing GUI design and usability within the RUP.

Extreme Programming and other Agile approaches

Agile software methodologies are a radical departure from the traditional, document-heavy (usually) waterfall processes still in widespread use. These methodologies share a set of common values. They all try to find a useful compromise between informal development processes and formalized, traditional ones (Larman 2003).

Extreme Programming (Beck and Andres 2004) is perhaps the most 'extreme' of these Agile methodologies. XP is composed of the following practices:

- *Customer as a team member and on-site customer.* Development teams have one person (or a group of people) that represents the interests of the client, referred to as 'Customer.' Customer decides which features to add to the system.

- *The planning game*. Customer and developers cooperate to determine the scope of the next release. Customer defines a list of desired features for the system. Each feature is written out as a user story (see below). Developers estimate how much effort each story will take, and how much effort the team can produce in a given iteration, typically of two weeks. Customer decides which stories to implement and in what order, as well as when and how often to make available production releases of the system.
- *User stories*. These represent small features of the system that can be completed by a single developer in one iteration. Customer gives the user story a name, and broadly describes what is needed. User stories are typically written on paper cards.
- *Small releases*. Development starts with the smallest set of features that are useful. Releases are kept small by releasing early and often.
- *Simple and incremental design*. The simplest possible design that works is favored. Providing more design than is needed can be a waste of time, given that requirements can change, and is a needless cost for the project.
- *System metaphor*. Each project may have an organizing metaphor, which provides an easy to remember and guiding naming convention. This practice can be slightly confusing when adopted for GUI design: other design approaches, such as domain-driven design, suggest a focus on the core domain model to shape naming conventions and development abstractions.
- *Test-driven development (TDD)*. Before writing any code, developers devise a test that defines the expected behavior of the new code, and write the test first. These are typically unit tests. Each unit test usually tests only a single class or a few classes.
- *Acceptance tests*. These are specified by Customer to test that the overall system is functioning as specified. Acceptance tests typically test the entire system, ideally automatically. When all the acceptance tests pass for a given user story, that story is considered complete.
- *Refactoring*. This is the practice of making small changes to a portion of code to improve its internal structure without changing its external behavior. This is a practice born in Smalltalk development and popularized by Martin Fowler (Fowler 1999). Refactoring fits nicely with continuous testing, because after every change, tests are run to ensure code integrity.
- *Pair programming*. All production code is written by a pair of programmers working at the same machine.
- *Collective code ownership*. No single person 'owns' a package or any portion of code. Any developer is expected to be able to work on any part of the code base at any time.

- *Continuous integration and ten-minute builds*. All changes are integrated into the code base at least daily. A build should not last more than ten minutes. A build encompasses building the whole system and running all the tests, which should be able to be run both before and after integration, and deploying the system.
- *A sustainable pace of work ('energized work')*. Some XP practices advocate a forty-hour working week, to avoid the prolonged strain of work overload, usually a warning signal for a project.
- *Coding standards*. Homogeneous coding standards are applied by every member of the development team.

Official XP doctrine doesn't go into the details of user interface design, which are left to designers. A first version of the GUI design can be built up front (that is, the customer, together with developers) then used to feed the project's user stories. Alternatively, GUI design can be focused on iterations built on top of a reference framework consisting of GUI design guidelines and other constraints. Other approaches are also possible. No matter what GUI design details are chosen within the XP approach, a stable and continuous feedback loop from story creation through usability and user acceptance testing, and involving end users, is always instrumental to effective GUI design and development.

Early critics of the effectiveness of GUI designs performed with XP noted that user interface designers and usability engineers don't have a defined role within XP, and that the whole approach risks being developer-centric. However, a closer look at XP shows a number of strong points in this approach that favor sound GUI design practices. By building on the XP practices of communication, simplicity and continuous testing, usability can be achieved, not only in terms of end user acceptance and satisfaction, but also for other tenets of XP, such as implementation efficiency, developers comfort, and shared responsibility for the final product.

LUCID methodology

Classic LUCID methodology, as described for example in (Shneiderman 1998) and more recently updated, is an example of a user interface–driven approach to the whole software lifecycle, in contrast to the iterative approaches discussed previously. It is essentially a variant of the classic waterfall process, focused on usability and GUI design[9]. This is illustrated in Figure 1.5.

9. See http://www.cognetics.com/lucid/

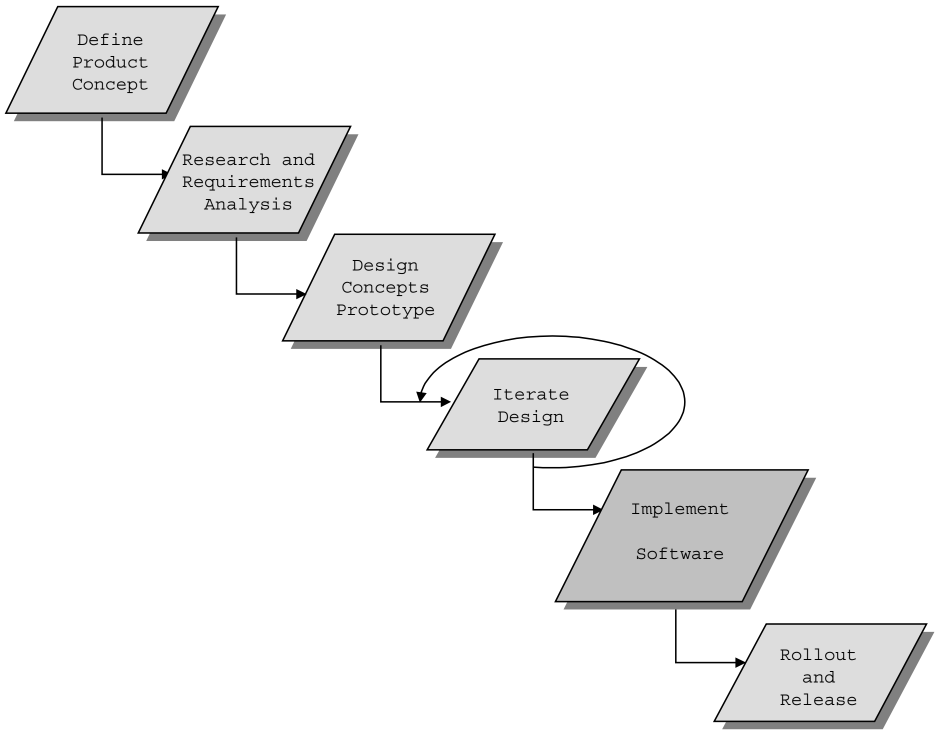


Figure 1.5 The LUCID lifecycle model

This lifecycle model can be broken down into the following elementary activities:

1. Develop the product concept:
   – Define the product concept. Begin writing down early use case diagrams.
   – Establish business objectives.
   – Set up the usability design team.
   – Identify the user population.
   – Identify technical and environmental issues.
   – Produce a staffing plan, schedule and budget.
2. Perform research and requirements analysis:
   – Partition the user population into homogeneous groups.
   – Break job activities into task units.
   – Conduct requirements analysis through construction of scenarios and participatory design.
   – Sketch the process flow for sequence of tasks.

   - Identify major objects and structures that will be used in the software interface.
   - Research and resolve technical issues and other constraints.
3. Design concepts and an initial prototype:
   - Create specific usability objectives based on user needs.
   - Initiate the guidelines and style guide.
   - Select a navigational model and one or more design metaphors.
   - Identify the set of key screens: log-in, major processes, and so on.
   - Develop a prototype of the key screens using a rapid prototyping tool, paper mock-ups, or other prototyping techniques.
   - Conduct initial reviews and usability tests.
4. Perform iterative design and refinement:
   - Expand key-screen prototype into a full system.
   - Conduct heuristic and expert reviews.
   - Conduct usability tests.
   - Deliver the prototype and the application specifications.
5. Implement the software:
   - Develop standard practices.
   - Manage late state change.
   - Develop on-line help, documentation and tutorials.
6. Provide rollout support:
   - Provide training and assistance.
   - Perform deployment, logging, evaluation and maintenance.

Modern software projects tend to require more flexible and rich models than this: we introduced it essentially for didactical reasons, because all main activities related to GUI development are listed in a sequentially ordered, simple arrangement.

Evolutionary Prototyping process

Many user interface design approaches use intermediate prototypes in order to produce the final GUI design more easily, reducing the risks (and costs) of the design phase[10].

The natural evolution of the prototype idea is to base the whole development around prototypes of increasing functionality. With this approach the prototype is never abandoned, but is constantly refined and expanded until it is good enough

10. We discuss prototyping in Chapter 3.

to be the final product. The discussion of this lifecycle approach is inspired from (McConnell 1996). The methodology is represented graphically in Figure 1.6.

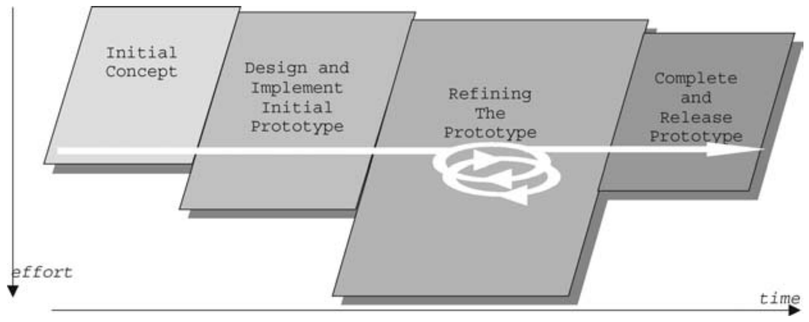

Initial Concept
Design and Implement Initial Prototype
Refining The Prototype
Complete and Release Prototype
effort
time

Figure 1.6 The Evolutionary Prototyping lifecycle model

This approach can be useful when requirements are changing – for example, when the customer is reluctant to commit to a defined set of requirements. It may prove useful in situations in which nobody fully understands the application domain at first, for example in advanced research projects. This model tends to produce visible progresses thanks to the steady prototype evolution.

There are however several drawbacks and potential risks when adopting this approach. First, as the application concept evolves as you develop the prototype, there are no predefined time and qualitative deadlines for ending refinement iterations. The risk is that as an important deadline approaches, the current prototype stage is declared 'good enough' to be released. Customer judgment may also not be a reliable criterion for concluding refinement iterations.

Another common risk is production of a poor-quality implementation in which code maintainability is low – if not addressed properly, continuous changes may produce code full of patches. *Feature creep* is another potential risk. When no clear and definitive requirements are set at the beginning of the development process, there is the concrete risk of adding too many new features to the prototype during its refinement.

Some guidelines help to tackle the commonest risks with this approach:

- It is essential to focus on a limited set of important aspects of the product before starting the development. These aspects will be the focus of the prototyping activity. An obvious choice is the GUI. Beginning the prototyping process with the GUI is a good way to give usability and GUI design top priority.

- The code used in evolving the prototype should be of the best possible quality, continuously refactored (Fowler et al. 2000) and enhanced, because frequent changes risk deteriorating it.
- For this reason, it is essential to avoid employing entry-level programmers when adopting this development model.
- Be sure of getting high-quality feedback from customers and end users, otherwise the prototype will prove poor and ineffective no matter what effort has been spent in refining it[11].
- Avoid evolving a throw-it-away prototype with this model. It should be clear from the initial inception of the concept whether to create a throw-it-away prototype, or to keep working on the prototype until it is refined into a final product. All members of the development team should be committed to this choice.

Evolutionary prototyping shares many characteristics with other iterative processes, such as RUP and the family of Agile models.

1.8 UML notation

This section introduces some UML notation that will be used in the rest of the book. Readers that are knowledgeable about UML's state, interaction and class diagrams may choose to omit this section.

Class diagrams

We introduce UML class diagram notation without discussing it thoroughly: if you are not familiar with UML class diagrams, many books are available on the topic.

In this book we will use simplified class diagrams. We won't use visibility indicators ('+, #, -' symbols for showing public, protected and private fields) nor other details such as initial values.

Figure 1.7 shows a sample class diagram that illustrates the level of detail of the class diagrams used in the book.

11. See the discussion on prototyping in Chapter 5.

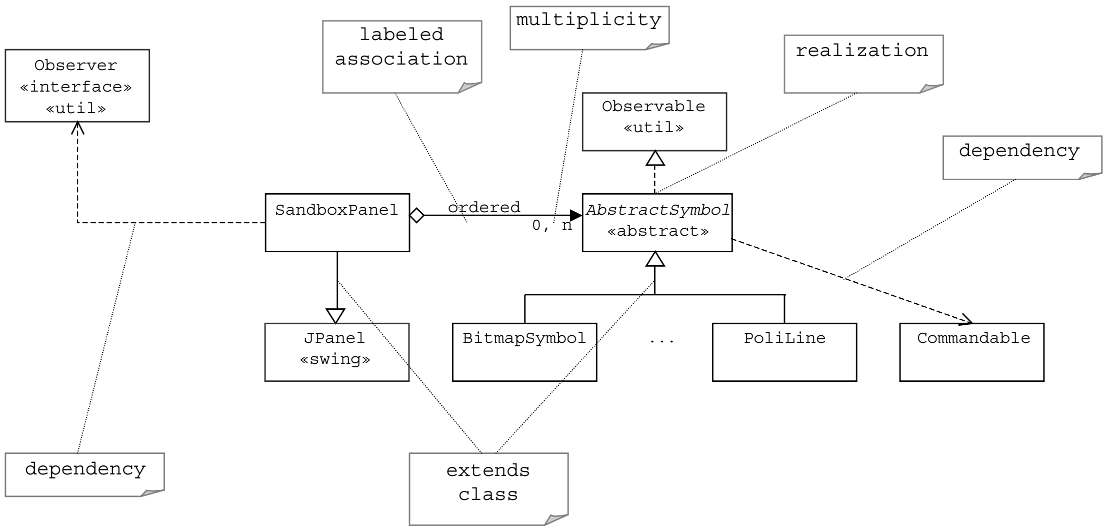


Figure 1.7 A sample class diagram

We use stereotypes – for example «`swing`» in the `JPanel` class – to represent the Java package to which the class belongs, or whether the Java type is an interface, an abstract class, or (when absent) a normal class. Figure 1.8 represents the class details we will use for documenting code.

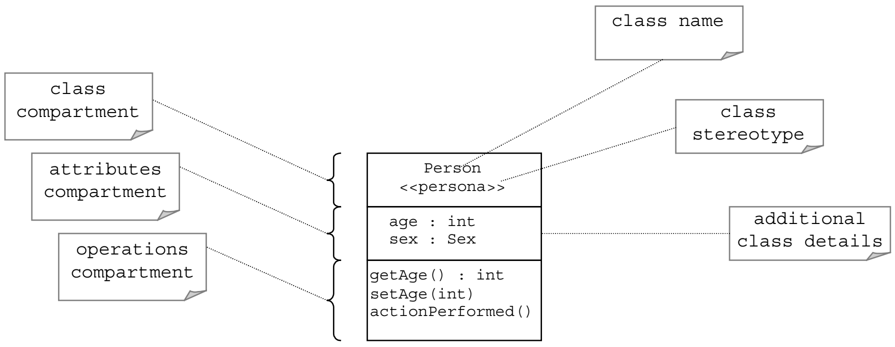


Figure 1.8 Class details

For brevity we will avoid the stereotype «`abstract`» for abstract classes, using only the italicized name, as in the `AbstractSymbol` abstract class in Figure 1.7.

We will also highlight the slight difference between realization and dependency relations:

- Realization means that a class implements behavior specified by another class. This is the common case when a class implements an interface or an abstract class.
- Dependency indicates that the implementing class *depends* on the other.

Whenever the interface `Observer` in Figure 1.7 changes, the `SandboxPanel` class may also have to change. The dependency relation is also used to express dependencies among different class packages.

Sequence diagrams

Throughout the book we will use both UML sequence and collaboration diagrams. Such diagrams, which are interchangeable, describe a behavior by means of a number of objects and the messages they exchange in a given temporal sequence.

Figure 1.9 shows an example of a sequence diagram that describes the typical behavior of a `CustomListener` instance that is registered for a `JButton`'s `ActionEvents`.

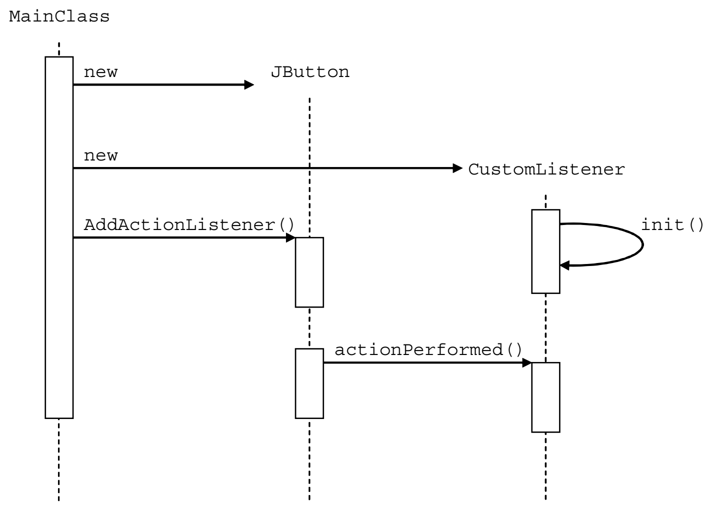


Figure 1.9 A sample sequence diagram

In the figure, an unspecified instance of the class `MainClass` creates a new instance of the class `JButton` and a new instance of the class `CustomListener`.

This in turn invokes the method `init()` on itself asynchronously, then invokes the method `addActionListener()` to the unspecified instance of `JButton`. After some time – not related to the previous sequence of method calls – the unspecified `JButton` instance method `init()` is invoked onto the unspecified instance of `CustomListener`.

For brevity we usually avoid indicating instance names: Figure 1.9 only specifies class names.

This section does not detail all the UML conventions used in the book for reasons of space – we have not mentioned collaboration diagrams, even though we will use them. Given the ubiquity of UML, we leave the interested reader to conssult one of the many sources available in literature or on the Web.

State diagrams

UML state diagrams are useful for describing the internal state transitions within a GUI.

Figure 1.10 shows an example of an UML state transition diagram.

As shown in the legends in the diagram, the initial state is indicated with a jagged arrow, while state transitions are indicated by arrows tagged with the event that triggers the transition from one state to the other. States are drawn as circles. Hence the state of the class described by the diagram in Figure 1.10 gets to State B after Event x happens when the class is in its initial state. Other events might change the internal state of the class, either restoring the initial state again or bringing it to a final state.

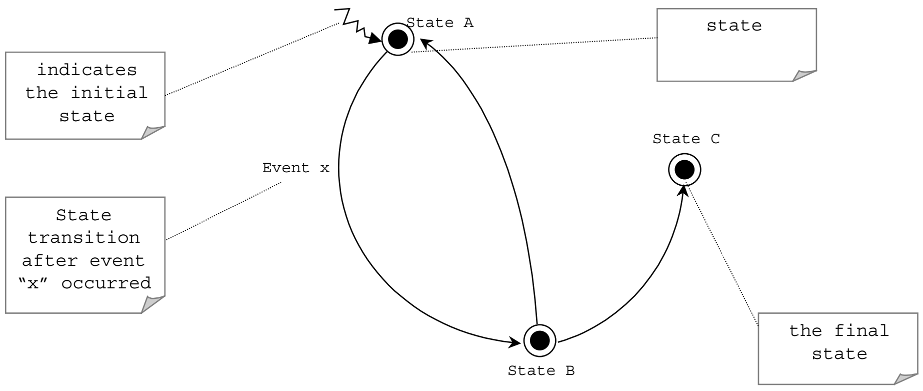

State A
state
indicates the initial state
State C
Event x
State transition after event "x" occurred
the final state
State B

Figure 1.10   State changes within a class

1.9 Summary

This chapter has presented some introductory discussions about effective software and GUI design. In particular:

- We discussessd the important concept of focusing on end users throughout the whole development process.
- We illustrated a general decomposition of GUI implementations based on functional criteria.
- Tool selection (the topic of Chapter 11) was presented briefly.
- We briefly introduced organizational aspects related to UI development.
- We discussed lifecycle issues, showing some design methodologies focused on usability and GUI-centered development.
- We introduced UML use case diagrams as a means of documenting systems from an end user's perspective.
- We also introduced sequence and state UML diagrams.

2 Introduction to User Interface Design

The field of user interface design and human-computer interaction is complex and vast. It has many different contributors and perspectives, and still lacks a uniform descriptive language. It is fragmented into different approaches and practices, a fact that stems directly from the very nature of human-computer interaction (HCI): the presence of the human component makes it impossible to develop an exact foundational theory. HCI, its derived design guidelines and criteria are thus based mainly on empirical evidence and practical principles.

The term 'HCI' was adopted in the mid-1980s. HCI is an interdisciplinary practice that aims at improving the utility, usability, effectiveness and efficiency of interactive computer systems. SIGCHI, the special interest group in HCI, defined it as 'a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them[1].'

In this chapter we introduce some HCI concepts that are fundamental to the professional design of user interfaces. The chapter is structured as follows:

2.1, The human factor discusses the role of people in the design process.

2.2, Display organization introduces the esthetics of GUI design, and discusses ways in which an application can interact with its users.

2.3, Interaction styles goes into more details about human-computer interaction, and presents the five major categories of human-computer interaction.

2.4, Conceptual frameworks for UI design describes a set of coherent concepts that structure the different phases of development of UIs, and which provide a reliable and proven mindset for organizing the design, thus reducing risks and improving quality.

2.5, Assessing the quality of a GUI describes ways in which testing of user interfaces can be conducted and its results collected.

1. Many disciplines contribute to HCI: computer science, cognitive psychology, ergonomics, social and organizational psychology, design, engineering, anthropology, sociology, philosophy and linguistics.

For simplicity we use the term 'GUI' as a general term to refer to any graphical user interface. In this chapter we distinguish between different classes of graphical user interfaces that are gathered under the common name of GUIs when applicable.

2.1 The human factor

We begin our quick tour of HCI with end users and how they perform actions.

A model of interactive systems – seven stages and two gulfs

One of the simplest approaches to modeling interactive systems is to describe the various actions users go through when faced with the task of using an interactive system (Norman 1998). Users:

1. Form a goal
2. Form an intention
3. Specify an action
4. Execute the action
5. Perceive the system state
6. Interpret the system state
7. Evaluate the outcome

The user first forms a conceptual intention from their goal – for example, deleting an item in the application shown in Figure 2.1 – then tries to adapt this intention to the functionality provided by the user interface. From these commands, as perceived by the user, they execute the action – for example by dragging an element in the tree to delete it, as Figure 2.1 shows. The user then tries to understand the outcomes of the action. This is particularly important in computer systems, where the inner workings are hidden and users have to guess the internal state from few artificial hints (Norman 1998). The last three stages listed above help the user to evolve their idea of the system. The entire interaction process is performed in such cycles of action and evaluation: by interpreting the outcome of their actions, the user refines their mental model of the system.

Action and evaluation are often illustrated by means of the *gulf* metaphor, after (Hutchins et al. 1986):

- *The gulf of execution*. This phrase describes the mismatch between a user's intentions and the allowable actions: for example, a user might be used to removing items by dragging them into a wastebasket, but in some applications items may be not draggable. *Gulf of execution* describes the practical difficulty of performing tasks with a GUI.

- *The gulf of evaluation*. This phrase refers to the difference between a user's expectations and the system's representation. Referring to Figure 2.1, the user observes that even if an item can be dragged onto the wastebasket icon, the intended delete operation did not work, because the application displayed the message shown in Figure 2.2. *Gulf of evaluation* describes the difficulty users experience in evaluating the outcome of an action they perform with a GUI.

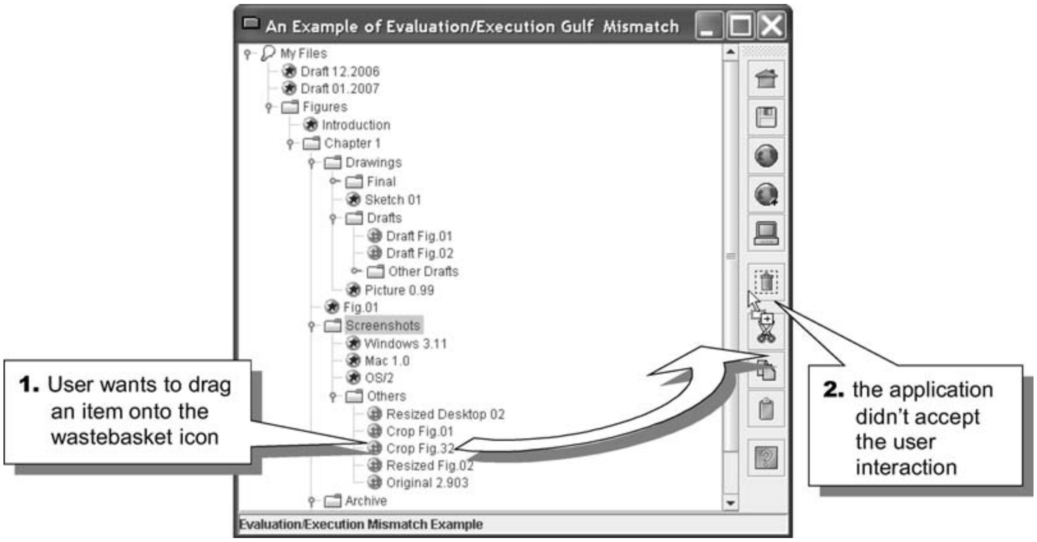
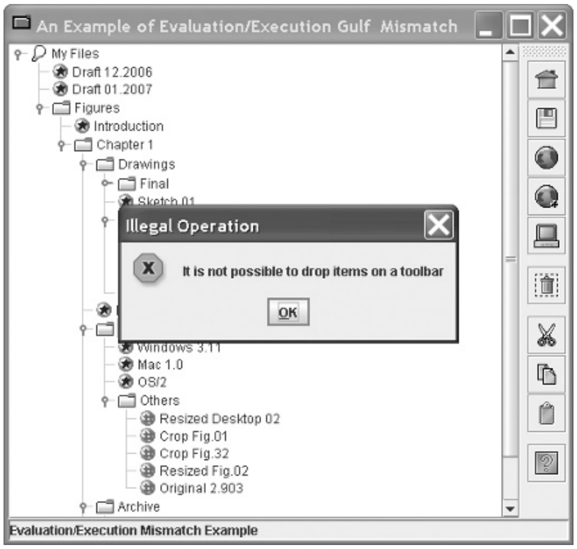


Figure 2.1 The gulf of execution: execution mismatch

The cognitive distance between two such worlds – the user's and the software's – corresponds to the potential mismatch between the way a person thinks about a task and the way it is represented in the GUI (Preece 1994). This mismatch, and the distance between the two gulfs, can be reduced by designing the user interface in a way that reduces the differences between the users' goals and the GUI's state and form.

As an aside, messages such as that shown in Figure 2.2 are rather intrusive. A better choice would be to signal the error with a less intrusive form of feedback, such as changing the mouse pointer shape or producing a beep.


An Example of Evaluation/Execution Gulf Mismatch

Illegal Operation

It is not possible to drop items on a toolbar

Evaluation/Execution Mismatch Example

Figure 2.2 The evaluation gulf: evaluation mismatch

Developers are part of the design process

So far we have only described the user's perspective of the GUI. We have not yet talked about the designers and the programmers who design the application, including the user interface. Just like end users, designers and programmers also have their own vision of an application. What for the end user might be an incomprehensible command description, such as *clear action stack*, may be an 'obvious' choice for the programmer that implemented it. A graphical design inspired by some new award-winning product might represent an accolade for the designer, but a nightmare for the developer to implement and an awkward thing for an end user to work with.

Many of the problems involved in creating effective user interfaces stem from the differences between the designer's and the end user's viewpoints. Designers can sometimes become so absorbed in their work that they lose focus and overlook the importance of the user's needs.

Ideally, the designer is the mediator between users and developers. Unfortunately professional GUI designers are thin on the ground and often expensive to hire. Hence developers often fill the role of designers, especially in small and medium sized organizations. This creates a potential problem: such developers-turned-designers often adopt their habitual programmer's mental model unconsciously, producing less usable GUIs as a result. On the other hand, fortunately, the effect of good design is contagious. Design guidelines, which are often promoted by organizations that can afford a team of full-time professional designers, are slowly making their way in everyday software, not just that produced by large corporations.

The refinement of a user's model of an application is often distorted by accidental interaction, bad design or software bugs. Even developers themselves, as users of other software, sometimes struggle to understand the internal model of a buggy application. Suppose that a developer uses an application that shuts down unpredictably, corrupts data, or causes other serious trouble. They will of course try to bypass the internal states that cause such harmful behavior. To do that in the absence of implementation documentation, they must develop a mental model of the application's inner workings. Computer programs, contrary to other types of technology, are both complex and inherently abstract. A mechanic can guess from the weird noise a car makes that it probably has a problem with its suspension, but even a seasoned developer cannot determine the actual implementation behind a GUI merely by using it.

End users only have the direct experience of the GUI with which they are interacting, coupled with their previous knowledge, to work out what is going on inside an application. Humans need semantic models to enable them interact with the world sanely, and always build such models, even unconsciously. Users act like the early philosophers, trying to make sense of an incomprehensible world using only their current and past experience – it is common to hear them explaining how an application works in their own terms. As personal computers have been around for decades, many people are accustomed to concepts such as files, databases and mouse gestures[2].

In some ways this is a problem – ideally we should be able to use a complex device such as a car or a software application without having to be aware of its inner working, although a minimum coupling with the underlying technology is unavoidable. A professional GUI design should therefore start with an abstract model in the designer's head. In addition, what might seem a natural choice for the designer can later reveal awkward details that are difficult to understand and to employ by users. It is important therefore for designers to adhere to a conceptual model that is as close as possible to the prior knowledge of the intended end user population.

2. The term *gesture* in HCI denotes a single basic interaction performed by the user. Usually it refers to mouse-based systems in which sequences of gestures will make the software perform certain operations. Sequences of gestures can be organized in a specific syntax, such as 'press right button-drag mouse-release right button.'

To recap, we have highlighted some important issues:

- Software is abstract. Good user interfaces are those that communicate their internal state to users effectively, encouraging the seven-stage cognitive sequence described on page 32. In computer applications, the inner workings are hidden, and human beings have to figure out the internal state from few artificial hints. Such hints should be coherent, otherwise the GUI won't be successful: it will be difficult to use, producing convoluted mental models that are hard to remember, inducing a negative response from users. Hence it is important to develop a sound conceptual model to stand behind the GUI. The basic concepts, visible items, their interaction, names and everything else should be carefully thought through at the design stage.
- People use conceptual representations of reality based on their current and past experience. Consequently, different mental models of the same application exist in the minds of its designers, its developers and its end users. It is important for designers to be aware of the different mental models involved in the development and subsequent use of a user interface as a *social artifact* – something that will be used by more than one person.

We mentioned that cognitive psychology was a contributing discipline to HCI. The next section discusses some simple cognitive models that underlie well-designed user interfaces.

Short term memory and cognitive modeling

We will now discuss some basic principles of cognitive modeling, and include some practical advice on their application to HCI design. In particular, we discuss briefly a useful – although rather crude – model of human memory, and some of its implications for interface design.

In human beings, *short-term memory* (STM) is a limited form of memory that acts as a 'buffer' for new information, used to process perceptual input. Empirical studies have shown that humans usually have an STM capacity of between five and nine items. Such items can be single objects or coherent chunks of information. The size of non-atomic pieces of information that can be stored in STM depends on the individual's familiarity with the subject, but usually the information survives no longer than 15–30 seconds.

You can try this for yourself: it is relatively easy to remember seven random colors, but it is not easy to remember seven Spanish words unless you speak Spanish – not to mention seven Urdu words. STM is very volatile. Distractions, external 'information noise' or other interrupting tasks quickly disrupt its contents.

The other type of memory is *long-term memory* (LTM), more stable and with far greater capacity, but with slower access than STM. A major problem with LTM is the difficulty of the retrieval phase. Many of us use mnemonic aids to access LTM, such as mental associations for remembering a personal code or password.

STM influences the efficiency of an HCI interaction. Interactions that can be processed using only STM are easier and faster to accomplish than those that require LTM or some external cognitive help. Complex interactions are made more difficult by the need to maintain a data context throughout the whole process, using working memory and STM.

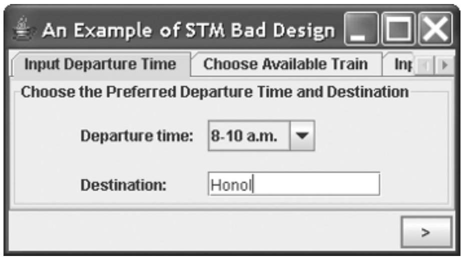
GUIs should be designed as much as possible to let users work with STM, but this kind of memory has its own limits as well. To illustrate these ideas, here is an example of the pitfalls of placing excessive trust in STM.

An example of STM misuse

Our example GUI here is designed to allow users to reserve a train seat. It is organized as a sequence of dialog in which only partial information is shown at any one time. Such an interaction style is often referred to as a *wizard*, a term popularized by Microsoft's extensive use of it.

Our GUI has been designed only for this example, and is not intended to be an example of good user interface design – see for example our weird use of tabbed panes!

In the first dialog we are asked for the basic details for our trip, as shown in Figure 2.3.



Choose the Preferred Departure Time and Destination

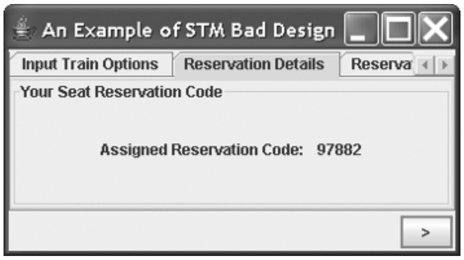
Departure time: 8-10 a.m.

Destination: Honol

Figure 2.3 An example of excessive STM burden: entering some data

After some input, such as reserving a window seat, food options and so on, we are presented a reservation code, as shown in Figure 2.4. This is meant to help the user to choose between different reservations. Users can remember this sort of code for varying times.

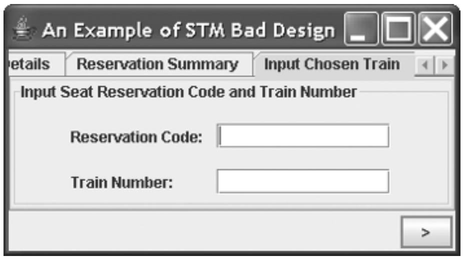
A recap screen is then presented, and the user is asked to choose one reservation, prompting for the data of choice (as shown in Figure 2.5).



Your Seat Reservation Code

Assigned Reservation Code: 97882

Figure 2.4 *An example of excessive STM burden: memorizing the reservation code*



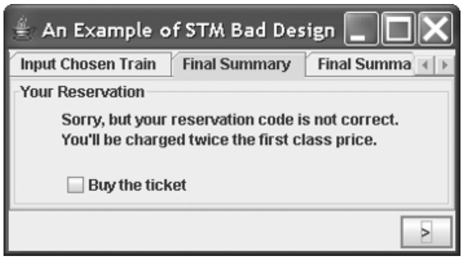
Input Seat Reservation Code and Train Number

Reservation Code:

Train Number:

Figure 2.5 *An example of excessive STM burden: retrieving data from STM*

Naturally, few users can remember this information, and the wizard will probably produce the message in Figure 2.6. At this point the user's STM memory has been overloaded. Clearly, more cognitive aids, such as displaying a list of reservations made, or providing some way to point to them, would make the GUI much more robust and usable.



Your Reservation

Sorry, but your reservation code is not correct.
You'll be charged twice the first class price.

☐ Buy the ticket

Figure 2.6 *An example of excessive STM burden: negative feedback*

Nobody design GUIs like this one any longer, but such design inconveniences were frequent before the widespread use of mature user interface solutions and the advent of direct manipulation techniques. In this situation the GUI shouldn't rely exclusively on the user's short term memory. But consider the case in which the client device cannot support a rich user interface, such as a mobile phone. Even in such a case, an alert message, such as 'Please print confirmation for your records…' is needed.

These problems have lead to the widely-used practice of exploiting the context for selecting and manipulating information. For example, users are now familiar with contextual menus, usually activated by right-clicking on an object, or some other platform-dependent gesture, that make all the possible commands for the selected item in the given context available. There is then no need for extra memory load on the part of the user. A designer should always try to design the user interface to make users work as much as possible with STM, as this lightens their memory load and makes the interaction speedier and less error-prone.

In contrast, something like a Unix command-line interface needs continuous access to LTM or some external cognitive aid. It is not uncommon for Unix novices users to use post-it or paper notes to remember commands and their syntax, or the sequences of commands required to carry out a certain task. With the advent of GUIs, this situation has changed. Now designers have a powerful set of tools for designing expressive, easier-to-use interfaces.

Another means of avoiding placing an excessive memory burden on users is to adopt a standard and consistent design. In this way, users can reuse the knowledge acquired from use of other parts of the GUI, or of other GUIs the adopt the same standards. Later we will show how expensive providing an arbitrary menu organization, with an incoherent command organization can be in user memory.

In conclusion, STM is a valuable aid to well-designed interfaces. STM requires concentration, so in general users should be in a proper environment for maximizing their performance. Users should feel at ease with the application, and have a predictable idea of how it works, without the fear of making catastrophic errors or of excessive time pressures. We cannot, of course, control the environment in which the application will be used, but we can consider it in our design.

UI design can be organized around basic criteria that are derived from cognitive modeling-based human psychology considerations: try to eliminate distractions, minimize user anxiety, provide feedback about task progress, and either avoid errors or handle them gracefully.

Interacting with human beings

In this section we discuss some practical issues related to interacting with users.

Response time

Even a brief introduction to cognitive modeling would be incomplete without mention of an important dimension of a user's experience when interacting with a computer system. *Response time* is a significant factor, in that slow response time is a cause of errors and user frustration. This is particularly true for Java-based applications, where performance can be a serious bottleneck[3].

Response time affects users in different ways. Their expectations and past experience play an important role in their reaction. If a user is accustomed to a task completing in a given amount of time, both excessive or too rapid a completion can confuse them. Short response times also aid more easy exploration of the GUI where such behavior is encouraged, for example by undo-able actions and low error costs.

Balancing human control and automation

It is often useful to provide automation of some features in an application, but this takes away some control from users. People become frustrated and nervous if they feel they don't have full control over the work they are doing. It is therefore important to provide the *sense* of control to end users.

In contrast, by definition a GUI should provide a high-level, easy-to-use view of an application's services and data, hiding irrelevant details from the user. A critical factor in a successful GUI design is determining the balance between automation and user control, between showing meaningful details and hiding the rest, and in doing so adaptively depending on the particular user. For example, a user may want to skip some automatic feature by taking full control of it as they become confident with the application.

It is useful to assess the levels of control that can be exerted in a GUI. This helps to make explicit in the design the layers of automation that can be provided, such as defining macros, providing wizards for most common operations, and so on. Nevertheless, a computer program is an inherently limited artifact, in that it cannot take into account all possible situations, only a restricted set of combinations thought out in advance.

Consequently, balancing human control over automation is a typical trade-off of GUI design. Providing fully-automated GUIs could be too risky, especially when the task is a critical one, such as managing a chemical plant, and there are many

3. In general, interpreted bytecode and the overhead of a Java virtual machine only impacts the overall performances of Java applications to a minor degree, thanks to sophisticated technologies like 'just-in-time' compilers, garbage collectors and various other optimizations. However, GUI technologies like Swing implement low-level graphical details and infrastructure interactions as fine-grained Java objects, requiring a high number of runtime objects just to implement simple user interfaces.

independent variables that may cause unforeseen behavior. On the other hand, allowing users to have a too much control could create GUIs that are too difficult or even dangerous to use. When exposing too much detail that can be manipulated in a GUI, the risk arises that users could modify some valid data or use the interface in unanticipated ways.

To recap, in designing a correct level of automation, much depends on the user population and the nature of the application domain. For a business-oriented application, some simple rules can be applied:

- For the same application, always provide two routes through the UI: one for experienced users, with more control and less automation, and a simplified (that is, more automated) set of functions for inexperienced users.
- Provide warning messages when critical data is being manipulated directly, even by experienced users.
- Whenever possible avoid automated features and the pro-active behavior exemplified by some 'agent-like' applications ("Hi, I'm Tom, I'll check the mail for you"). This latter kind of approach hasn't proved successful and also can quite expensive to implement. Proactivity is still an untamed beast[4].

Showing the application's internal state

Users build their own system model unconsciously while interacting with the GUI. It is essential to provide the right hints and to correctly signal the system state to the user. This can be achieved using various techniques. Those most commonly used are modifying the mouse pointer or using some form of animation:

- *Changing the pointer shape*. Changes of pointer shape are widely employed for signaling an application's internal state, for example a 'busy' pointer, and currently-available operations, for example resizing a window by dragging its corner. The Java Look and Feel guidelines, which we introduce in the next chapter, prescribe the use of the 'busy' pointer for any operation that takes more than two seconds.
- *Animation*. can be used to show both the progress of an operation, often by means of a progress bar component, or generic activity, using for example an ad-hoc animation. The Java Look and Feel guidelines suggest that a progress indicator should be updated at least every 4 seconds. The example GUI in Figure 2.7 below shows a progress bar that can be set to update at various

4. At a recent international conference on autonomous agents the speaker (an authority in the field, praising the many benefits of proactivity) was interrupted abruptly by his laptop crashing. It took many embarrassing minutes to recover his presentation. It turned out to be an unforeseen interaction between the screen saver (the proactive, yet unintelligent agent) and an operating system patch he had loaded the night before the presentation.

intervals in the range of 1 to 4 seconds. The task is completed in one minute. Readers are encouraged to download and try this sample GUI to directly familiarize themselves with update times.

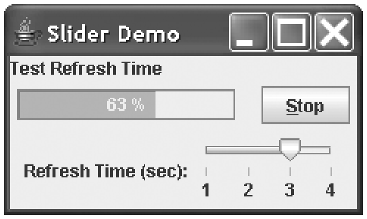


Figure 2.7 *Testing the update time for a progress indicator*

One controversial notion that should be considered is the use of *modes* in an application GUI. Modes are specific states or an application that affect some of the user interface behavior. A designer or developer can think of them as contexts in which some previous user interaction changed the meaning of current actions. For example, on a cellphone the 'hang up' red button has different functionalities, depending on whether a call is in progress or not. An application can behave in a completely different way in different modes in response to the same user input. Design guidelines normally discourage the use of modal interfaces, or even ban them altogether. Modes are difficult to manage by users and easily confuse them.

It is vital to show explicitly the current mode within the GUI. This is usually done by modifying the pointer shape, for example in drawing software when a specific graphic tool is selected from the palette, or by means of toggled buttons, status bar icons or messages.

Techniques for getting the user's attention

Techniques for getting the user's attention are employed widely in user interfaces. These techniques are derived from empirical studies and can be summarized as follows:

- *Animation*. Animation is often used to express the internal state of the GUI, showing work in progress or generically signaling activity. Often this latter use is the only one suggested by official design guidelines. However, flashing items on the screen easily capture a user's attention – often too easily: this technique can be disturbing and invasive.
- *Color*. Like animation, this technique should be used carefully. As with animation, the Java Look and Feel uses few system colors. Too many colors tend to produce confusing GUIs.

- *Audio cues*. This technique, used carefully, can be very effective[5].
- *Bold fonts and other graphic adornments*. When used carefully and coherently, such graphic conventions can be effective without being disrupting. As we will see later, the Java Look and Feel design guidelines adopt some simple graphic conventions to signal importance.

Relying on professional design guidelines avoids many obvious errors. This is especially true for attention-catching devices such as flashy labels, colors and the like. All major UI design guidelines provide reliable but noninvasive mechanisms for catching user attention.

Some general principles for user interface design

Before going further it is useful to recap what we have said so far. Here we distil the previous discussions into a few high–level general principles that should always be kept in mind when designing a user interface:

- *Minimize the load on users*. Reduce the memory and cognitive load on users by providing informative feedback, memory aids and other cognitive support. Ensure that a work session can be interrupted for a few minutes without losing the work in progress, as users are only able to focus attention for a limited time.
- *Ensure overall flexibility and error recovery.* Flexibility is essential when dealing with users. Human beings make errors: providing mechanisms for reversing actions allow users to explore the GUI free from the anxiety of being trapped in an unrecoverable mistake.
- *Provide user customization*. The interface should be customizable by the user. Flexibility also includes the provision of different use mechanisms for different classes of users: novices can use wizards or other simplified means for easy interaction, while expert users can take advantage of keyboard accelerators and other shortcuts, all within the same GUI. For specific users, such as those with disabilities, such flexibility could provide the only way in which they could use the application.
- *Follow standards to preserve consistency.* Many standards and guidelines exist for interactions, abbreviations, terminology and so on, such as the Java Look and Feel Design Guidelines (Java L&F Design Guidelines 2001), (Advanced Java L&F Design Guidelines 2001). Such user interface design standards are essential for the support of consistency between applications. They ensure professional quality while reducing the design effort.

  Consistency within a single GUI is even more important than correctly adopting a set of GUI design guidelines: for example, in labeling, terminology,

5. Audio clues are supported in J2SE's Swing from Version 1.4.

graphical conventions, component layout and so on. In this book we will discuss many such guidelines and principles, as well as systematic approaches to software design that are oriented towards consistency.

User-centered design

Perhaps the single most cited rule in user interface guidelines is '*know thy user.*' Without a reliable model of the end-user population, a design may be too general, relying only on the designer's, possibly restricted, cognitive model of an application's use[6].

A good designer not only knows the target users, but *thinks* like them. Learning to think as a user is essential for building a high-quality GUI.

Taking the user into account in the design process leads to an approach known as *user-centered design*, in which users are central to both the early design process and to later testing and evaluation. A user-centered technique known as *participatory design* stresses the active involvement of users throughout the design process, especially in the evaluation phase.

User-centered design focuses on three concepts:

- *Users*. Usually the following general user categories apply:
  - Novice users
  - Intermediate users
  - Expert users
- Users' *tasks*. The most common categories are:
  - Frequent tasks – designers should optimize these tasks
  - Infrequent tasks – such tasks can be assigned a lower priority than frequent ones as regards design/development resources and time
  - Critical tasks – those that should be engineered most carefully in the GUI
- *Context*. In what context will users be performing their tasks?

The involvement of end users in GUI design should be carefully managed. Users are not GUI designers, and their interaction should be managed so that their role in the design process is mostly reactive, providing feedback over proposed designs rather than producing new designs from scratch. The use of a prototypes is central to early design iterations (design-evaluate-refine): such prototypes can that function both as the current GUI representation as well as its requirements documentation. Prototyping is an important activity in the design of high-quality user interfaces, and we discuss it in Chapter 5.

6. The concept of 'users' should also include those whose activity is affected by the software, such as system administrators, support staff, customers, etc. We refer to these as *stakeholders*.

In the following sections we discuss the two major issues in user-centered design: users and tasks.

User analysis

User analysis is a vital part of the design process. The output of user analysis is a model of the end-user population. Such a model is usually a decomposition of the intended user population into homogeneous classes identified by some characteristic such as domain knowledge, skill level, role, system knowledge and so on. Such a model, and its underlying knowledge of the user population, is often documented by means of *user profiles*. This kind of information is always needed in any GUI design process, even at an informal level.

An example of a user profile – for a cellphone Java music player – could be the following:

- Buys a new wireless game or ring-tone at least every month.
- Buys at least one CD every two months.
- Is proficient with high-functionality cellphones.
- Is aged 16-30.
- Uses an MP3 player, possibly combined with a cellphone.
- Listens to/watches the following radio station/music shows: …
  (etc.)

As far as possible designers must study representative users directly, possibly in their workplace, to take into account their typical working environment.

Task design and analysis

The concept of *tasks* is an important one in GUI design. Tasks implicitly define users, not just the details of the actions needed to accomplish a certain result within the GUI. For example, the task of creating a sophisticated glossary in a word processor automatically underlies an expert user, while checking e-mail is a task that can be performed by any kind of user and should be thought of as characteristic of novice ones.

Tasks are used also for usability testing. To test specific parts of the GUI, designers create particular tasks the users have to accomplish in the testing environment. Task analysis studies the way in which users accomplish such tasks while using the system. This analysis produces a list of the tasks users want to achieve using the GUI, together with the information needed and the intermediate steps needed for completing them. Task analysis is be performed by interviewing users and by observing the way they complete preset tasks.

For example, suppose we are designing a music manager applet for J2ME-enabled devices, such as that shown in Figure 2.8.

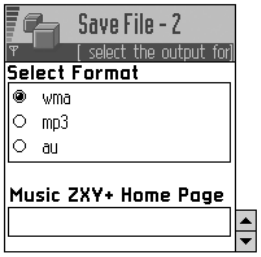


Figure 2.8 A music manager applet for J2ME wireless phones

Examples of the tasks and subsequent task realizations for this kind of GUI may be:

- Convert an MP3 into a ring-tone:
  - from the main menu, select **Format Conversions**
  - in the 'Format Conversions' screen, select the input MP3 file
  - in the 'Format Conversions' screen, select the **Export** option
  - in the 'Export' screen, select the ring-tone format and rename the resulting file
- Send a piece of music to another cell phone:
  - from the main menu, select **Clipboard**
  - from the 'Clipboard' screen, select the desired file
  - from the 'Clipboard' screen, select the **Send** option
  - in the 'Send' dialog, select the **Another cell phone** option
  - in the 'Send to another cell phone' screen, select the recipient's number or type it using the numeric pad
- Configure the application preferences:
  - from the main menu, select **Preferences**

Tasks depends on the GUI – the same task performed on two different GUIs may result in completely different task realizations.

Simplified thinking aloud

This technique prescribes testing the GUI with users who are asked to express their thoughts verbally while interacting with the system. An observer can use such additional insight into user's interaction process to identify unforeseen misunderstandings in the interface design. Users are usually videotaped while interacting with a GUI, as this allows better analysis.

A simplified, more practical version of this technique involves observers who take notes while the user is interacting with the GUI. Precision and exhaustiveness are traded for economic feasibility and practicality. Even in this simplified version, this type of test can reveal extremely useful information. Testers need to be aware of the added strain on users that this type of testing usually entails, and manage it accordingly, for example by limiting the duration of individual tests to a few minutes.

Graphical user interfaces are all about the visual arrangement of information. The next section moves the focus of our discussion from the human user to the computer, discussing the visual organization of graphical user interfaces.

2.2 Display organization

The organization of the display is clearly one of the most important aspects of the design of a graphical user interface.

It can help a developer to think of display organization as a language made up of the following basic constructs, which can be combined together to produce very complex display organizations:

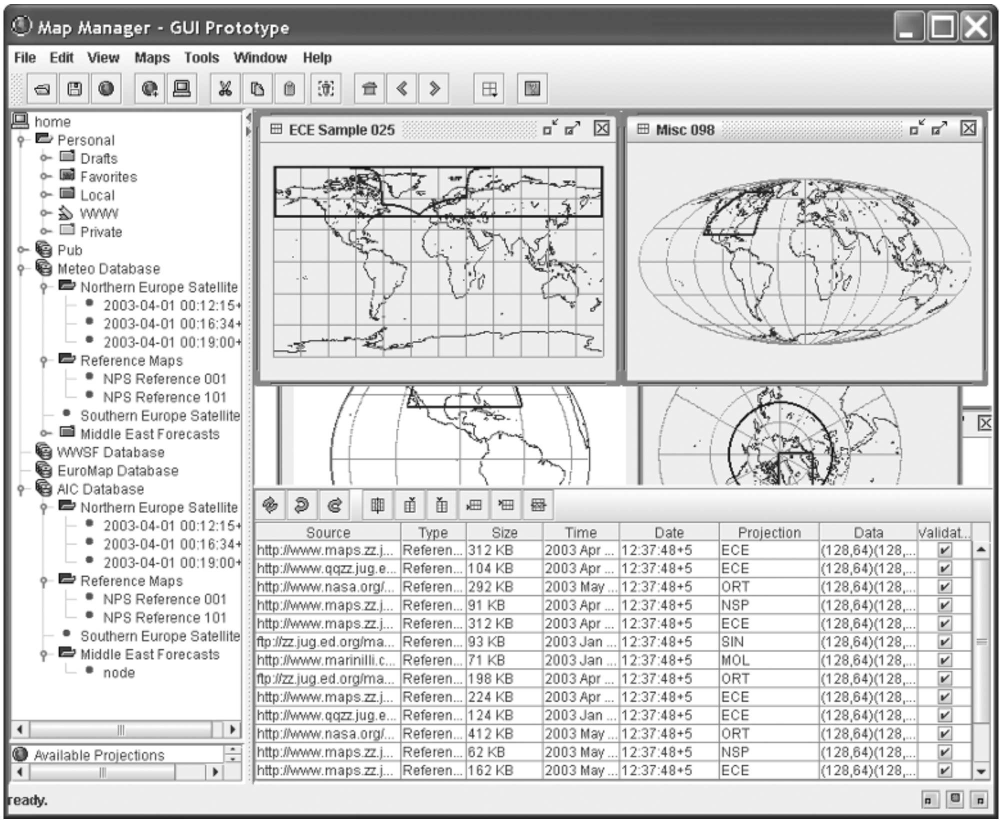
- *Composition*. Display organizations can be nested into others, recursively. Readers familiar with software design patterns know this mechanism as the Composite Pattern (Gamma et al. 1994).
- *Separation*. Specific portions of the display can be separated from others for semantic reasons. Static or dynamic separators, using rules or other graphic cues such as different windows, resizable areas in the same window and so on can be used. For example, a set of check boxes can be grouped and separated from other items with additional space.
- *Layout strategy*. This is the final element of our hierarchy of visually nested area organizations. We will discuss strategies for laying out the items in our GUI below.
- *Temporal sequence*. The display content depends upon external input such as user interaction or task completion. Such temporal 'screenplay' should be carefully thought through by the designer.

Layout strategies for a display area can be of two types:

- *High-density*, for conveying an high volume of information.
- Its opposite, which we call the *limited information* strategy, in which the aim is to reduce the amount of displayed data.

Such strategies are complementary and should always be used together in the design of every window or portion of it. Depending on the individual case, one or other will be dominant, but it is essential to take care with the balance of both. Interfaces that are either too cluttered, or too uncommunicative, are both hard to use. Figure 2.9 shows an example of a design in which the high density strategy is predominant. This interface has been designed mainly for expert users.

The limited information strategy is also used in the GUI in Figure 2.9 – see the use of drop-down menus, configurable areas, collapsing items and so on.



Source	Type	Size	Time	Date	Projection	Data	Validat...
http://www.maps.zz.j...	Referen...	312 KB	2003 Apr ...	12:37:48+5	ECE	(128,64)(128,...	✓
http://www.qqzz.jug.e...	Referen...	104 KB	2003 Apr ...	12:37:48+5	ECE	(128,64)(128,...	✓
http://www.nasa.org/...	Referen...	292 KB	2003 May ...	12:37:48+5	ORT	(128,64)(128,...	✓
http://www.maps.zz.j...	Referen...	91 KB	2003 Apr ...	12:37:48+5	NSP	(128,64)(128,...	✓
http://www.maps.zz.j...	Referen...	312 KB	2003 Apr ...	12:37:48+5	ECE	(128,64)(128,...	✓
ftp://zz.jug.ed.org/ma...	Referen...	93 KB	2003 Jan ...	12:37:48+5	SIN	(128,64)(128,...	✓
http://www.marinilli.c...	Referen...	71 KB	2003 Jan ...	12:37:48+5	MOL	(128,64)(128,...	✓
ftp://zz.jug.ed.org/ma...	Referen...	198 KB	2003 Apr ...	12:37:48+5	ORT	(128,64)(128,...	✓
http://www.maps.zz.j...	Referen...	224 KB	2003 Apr ...	12:37:48+5	ECE	(128,64)(128,...	✓
http://www.qqzz.jug.e...	Referen...	124 KB	2003 Jan ...	12:37:48+5	ECE	(128,64)(128,...	✓
http://www.nasa.org/...	Referen...	412 KB	2003 May ...	12:37:48+5	ORT	(128,64)(128,...	✓
http://www.maps.zz.j...	Referen...	62 KB	2003 May ...	12:37:48+5	NSP	(128,64)(128,...	✓
http://www.maps.zz.j...	Referen...	162 KB	2003 May ...	12:37:48+5	ECE	(128,64)(128,...	✓

Figure 2.9 A predominantly high-density display organization

A high-density layout strategy can be usually achieved using three general mechanisms:

- *Tabular organization*. Data is organized in a list of (possibly) structured values. Typical examples are spreadsheets and database grids.
- *Hierarchical organization*. Information is structured into a tree-like hierarchy, such as in the file system's graphical representation shown on the left of Figure 2.9.

- *Graphical organization*. Data is represented graphically in the form of a chart or diagram.

A limited information layout strategy instead aims at minimizing the displayed data. There are several approaches to controlling the volume of displayed data:

- *Step-by-step interaction*. Data is serialized and shown in stages separated in time. A classic example of this approach is the wizard.
- *Details on demand*. Optional data is only shown on user request. A common example of this strategy is dialogs that have a **More details** button that enlarge the dialog to provide further information. This type of mechanism should be used with care, however, because users prefer predictable windows and may feel uncomfortable with a GUI that changes its appearance too much.
- *Minimize irrelevant information*. There are many ways to minimize data; for example by shading it. Figure 2.10 shows a contextual menu in which some commands are grayed out to signal that they are currently unavailable. We often take such features of user interfaces for granted, but think how frustrating it is for the user to select a command only to be slapped in the face with an error message because the command is currently unavailable.

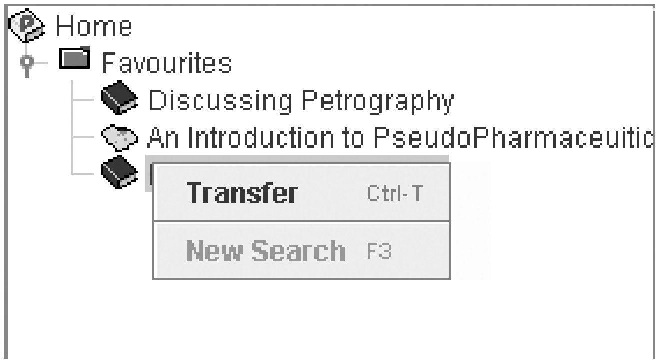


Figure 2.10 Disabling unavailable information at its simplest

These general techniques can be applied to many parts of the GUI – not just to menus, but also dialogs, radio buttons and so on.

Stated in this way, however, such principles are of limited help. We will see more concrete examples of the use of these ideas in the following chapters.

Esthetic considerations

Undoubtedly, professional-designed GUIs are pleasant to look at, but wrong assumptions are often made about the meaning of the term 'pleasant.' One of the common pitfalls in GUI design is to get stuck into creating an excessively elaborate visual experience on the (wrong) assumption that more is better. GUIs should be as least astonishing as possible. A successful GUI is one that is barely noticed, that works smoothly, swiftly and predictably.

I like to call such counterproductive and 'fancy' GUIs 'Louis XIV-style user interfaces' – this is often the case with novice or amateur designers who indulge in too much baroque design. This is also a common error even for seasoned designers. In fact, given the current pace of software releases, the most obvious and visible place to add new features, and so justify the new release, is always the user interface. Hence, *feature creep* is often concentrated in the GUI[7].

Feature creep is a well-known phenomenon, referred to as *featuritis* in the classic *The Mythical Man-Month* (Brooks 1995): 'The besetting temptation of the architect… is to overload the product with features of marginal utility, at the expense of performance and even of ease of use.'

On the other hand, esthetics *are* important. Too often developers take little interest in the visual appearance of their user interfaces, producing unusable designs as a result. Some find details of appearance such as buttons size, overall visual balance and the rest boring. Such developers are mostly implementation-driven, tending to automate the user interface as much as possible, implicitly seeing it as a dull, unnecessary activity. Unfortunately there is no substitute for human design: devices such as dialogs that automate the layout of the data they contain without semantic input can seem attractive, but produce poor user interfaces.

On the other hand, such appearance details can be hidden using wise use of object-oriented software architecture[8], in which you get all the benefits of a professional visual appearance with only a little extra work. We will show many techniques for promoting such advantages, from general approaches to practical, reusable classes.

This book promotes an *industrial* approach to user interface design, especially as regards visual appearance. Our idea of a good-looking user interface is one that adheres to official guidelines, is sober and usable as much as is economically feasible and provides extras only in a limited, 'withdrawn' fashion. This is the reason why, for example, Java libraries for advanced graphics handling, such as the 2D package, are covered only marginally. Some examples of visual details in a professional design are shown in Figure 2.11.

7. You can find many examples of wrongly-designed GUIs – not only limited to the purely visual aspects – by searching the Web for 'User Interface Hall of Shame.'
8. Alternatively, GUI layout may be done in a semi-automatic way, in which the semantic data that describes the layout of the visual components is stored externally, for example in property files or some special field derived from the class documentation or metadata.

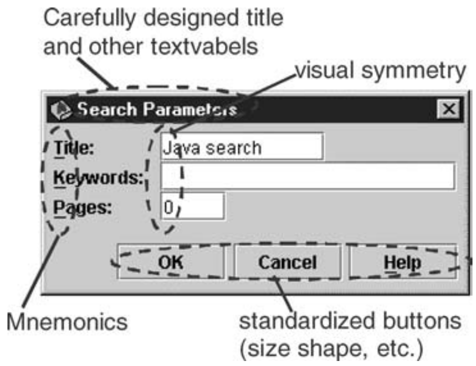


Figure 2.11 A simple dialog design

The interested reader can see (Mullet and Sano 1995) for an introduction to the art of visual design of software user interfaces, or the classic trilogy from Edward Tufte (Tufte 1990), (Tufte 1997), (Tufte 2001).

Abstract-Augmented Area for GUIs

Abstract-Augmented Area for GUIs (A3GUI or A3GUI) is a term that describes a simple approach to the definition and general management of graphical user interfaces. The key idea is to organize a GUI and all its underling dynamics conceptually – user interactions, intended behavior, design requisites, constraints, implementation and so on – by *areas*, that is, the 'real estate' of our GUIs.

A3GUI represents a GUI as a set of *augmented areas*. These areas are abstractions over the real GUI that help the design, implementation, testing or any other aspect of the GUI in which we are interested. A3GUI can be thought of as describing a general mindset that is independent of the chosen UI design approach.

By the term *real GUI*, we mean one concrete execution of our application at a given time. This will in turn depend on the surrounding context that may affect our GUI, such as the current user, the OS on which the application is running, and so on. All such context data can change the GUI's behavior and appearance for a given execution in a given situation. For example, a GUI may change depending on the given locale, or show only specific features to specific users, depending on their roles. We all deal with GUIs by managing abstractions of real executions.

Augmented areas, which for brevity we refer to merely as *areas*, are pictorial representations of specific facets of the GUI (whole windows, panels or single widgets) augmented with other information. Areas can be represented as paper sketches, in electronic form as drawings, diagrams, bundles of files, UML 2.0 diagrams and so on, or in any other convenient way. Areas can be visually nested inside other areas, can be related to other areas, and documentation attached to them, such as requirements, documents, the implementation's Java classes, and so on. The need is to provide a pictorial representation, a unique ID and an explicit or implicit set of abstractions we are representing throughout the area in the real GUI. The A3GUI concept also happens to dovetail nicely with modern OO GUI toolkits such as those used to build Java GUIs.

The type and level of abstractions really depend on our purpose. For example, Figure 2.12 shows a number of possible abstractions for a real GUI in a specific execution context.

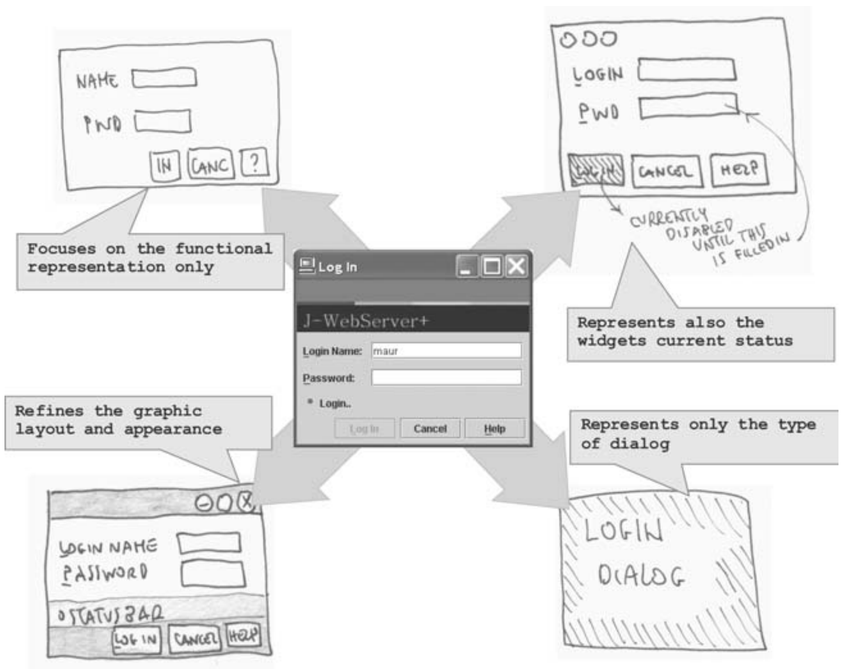

NAME
PWD
IN CANC ?
Focuses on the functional representation only
LOGIN
PWD
LOGIN CANCEL HELP
CURRENTLY DISABLED UNTIL THIS IS FILLED IN
Represents also the widgets current status
Log In
J-WebServer+
Login Name: maur
Password:
Login..
Log In Cancel Help
Refines the graphic layout and appearance
LOGIN NAME
PASSWORD
STATUS BAR
LOG IN CANCEL HELP
Represents only the type of dialog
LOGIN DIALOG

Figure 2.12 Some possible abstractions over a given GUI execution

A3GUI provides a formalized yet flexible framework for designing and managing GUIs that helps to solve the conceptual twists that we commonly face when defining GUIs at various levels of detail.

An example

Suppose we want to design a very simple GUI to display the bank accounts of specific customers. We want to provide a list of all transactions recorded for a given customer (which for simplicity are chosen outside our GUI). Only certain users may have access to transaction details: for example, if a clerk is inspecting a customer's account, we don't want to allow access to customer-sensitive data.

For simplicity we can think our GUI as having only two requirements:

- (R1) Our GUI has to show a list of all available transaction for a given customer.
- (R2) Depending on the role of the current user, only a small subset of a transaction's details can be seen.

We start by devising the following areas[9] – see Figure 2.13.

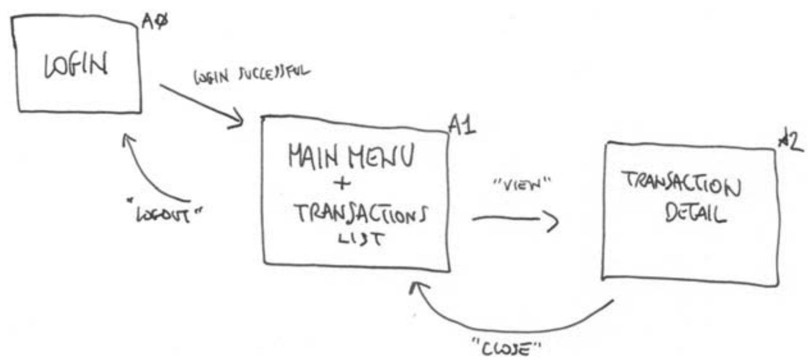


Figure 2.13 A possible GUI representation in three areas (adopting functional abstractions)

The following abstractions relate to each pictorial representation shown in Figure 2.13:

- A0 represents the login functionality each user has to accomplish to access the rest of the application.
- A1 represents the access to all the functionalities provided by the main menu, and also provides the list of available transactions to the user (fulfilling R2).
- A2 represents access to the transaction details, available only for the given role.

The diagram also shows some relationships between the various areas. These relationships express informally the intended navigation between the various areas. We will use the same notation to express navigational relationships between areas in the following figures also.

9. In this example we use 'A' as a prefix for areas ids to specify that these areas came from analysis, and are meant to capture requisites and decompose functional behavior only.

The areas shown in Figure 2.13 were the result of a functional refinement activity – we can think these areas as roughly equivalent to the use cases for the GUI.

A further step is to refine the previous areas for the UI design, deciding whether areas will become fully-fledged windows, or parts of other areas. A possible UI design refinement step is shown in the following figure[10].

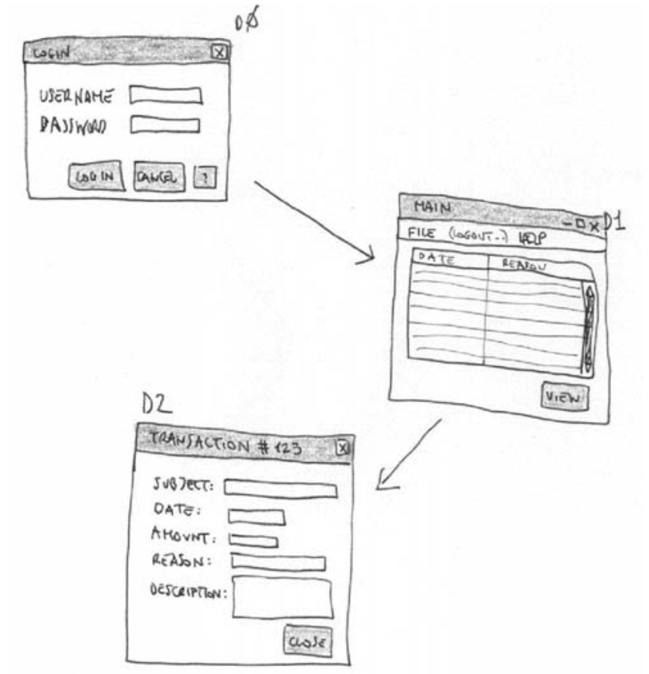


Figure 2.14 A UI design refinement step

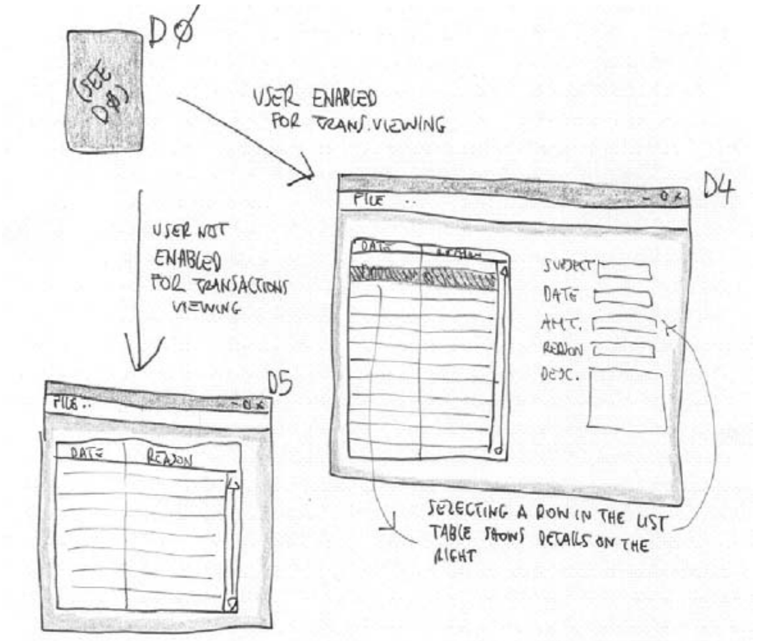
The following abstractions related to the pictorial representation depicted in Figure 2.14:

- D0 represents the log-in dialog.
- D1 represents the main windows with the available commands and the list of all the transactions for the chosen customer. Whenever the current user (already logged in from D0) is not entitled to see transaction details, the **View** button is disabled.
- D2 represents the pop-up modal dialog that appears whenever the user has selected one transaction and presses the **View** button.

10. In keeping with the previous step of this example, we use 'D' to prefix areas ids, to specify that these areas arise from UI design refinements.

We have omitted some implicit relationships between the areas in Figure 2.14 and those in Figure 2.13 for simplicity. For example, D0 is the GUI design refinement of A0. These implicit relationships are useful in a number ways, as we will see in the many examples provided later in the book.

We could have provided an alternative UI design for the same area as Figure 2.13. Imagine that we provide two different windows, depending on the user's role. This design is shown in Figure 2.15, in which D0 is the same area as that shown with the same id in Figure 2.14, while the other two areas are new.



DØ
(SEE DØ)
USER ENABLED FOR TRANS.VIEWING
USER NOT ENABLED FOR TRANSACTIONS VIEWING
D4
FILE ..
DATE
REASON
SUBJECT
DATE
AMT.
REASON
DESC.
D5
DATE
REASON
SELECTING A ROW IN THE LIST TABLE SHOWS DETAILS ON THE RIGHT

Figure 2.15 An alternative UI design

However, we decide to use the design in Figure 2.14, because it avoids having different navigation paths based on user role.

We then focus on refining area D1 in Figure 2.14 further. Now that the requirements are clearly addressed and the overall UI design is almost complete, we want to further refine the GUI for implementation reasons. This is usually performed by an experienced developer when setting out the implementation architecture for the GUI. We focus on refining area D1 into two reusable areas for technical reasons[11]. The resulting areas are shown in Figure 2.16.

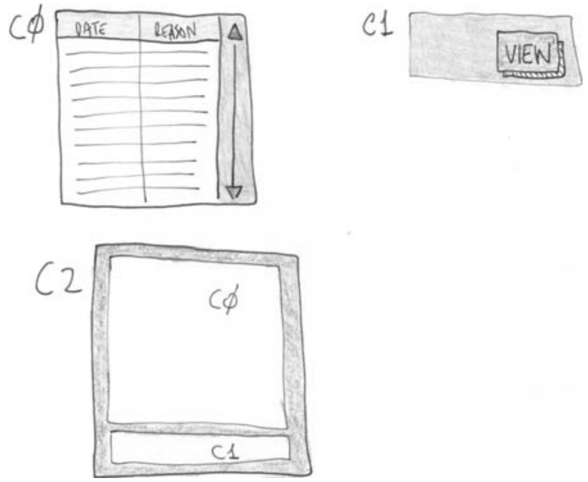


Figure 2.16 Representing an area as composed of two other areas (adopting the visual containment abstraction)

Note how we iteratively refined our GUI as a set of augmented areas. The areas have a number of semantic relationships between them. Figure 2.17 shows the iterative refinements we made to get to the final UI.

We have obtained the following benefits by using this conceptual approach:

- Centralizing several notions such as functional decomposition, requisites, technical aspects and so on in one representation, organized by GUI areas at various levels of abstraction.
- Iteratively refining our application at several levels of granularity and at several stages of the development lifecycle.
- Clearly assigning responsibilities – for example, requirement R1 is now handled by area C1.

The A3GUI approach will be used as a common expression language throughout the book for the discussion of the various aspects of GUIs.

We go into more detail of available user interface interaction styles in the next section.

11. These areas are mapped directly to one or more Java classes to define the implementation criteria of our GUI.

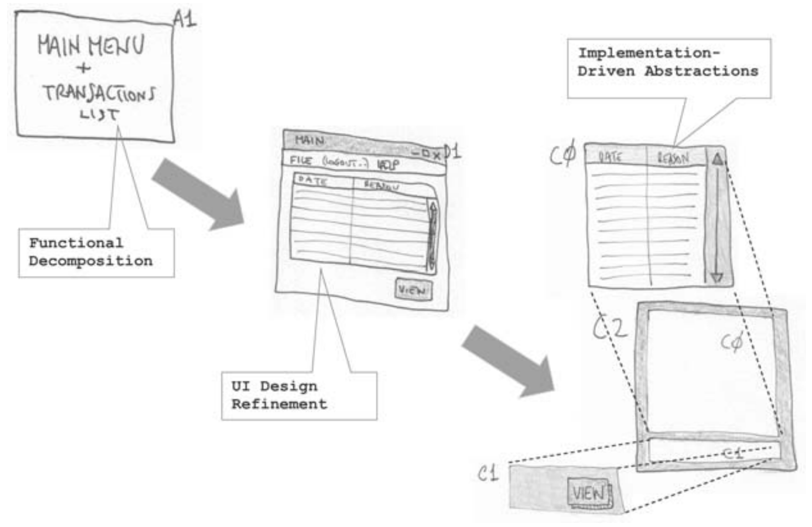


Figure 2.17 Refinement relationships among areas

2.3 Interaction styles

It is possible to identify several basic interaction styles for user interfaces (Shneiderman 1998), (Tidwell 1999).

- *Menu selection*. Here the user interacts with the system by selecting items in the UI from menu.
- *Form filling*, used for simple data input such as when inserting data in a Web form.
- *Direct manipulation*, used for example when performing operations by dragging or dropping items in a working area.
- *Language–based* interaction styles, such as interpreting users' natural language directly, or interacting by means of simple form of artificial language such as command line or scripting languages.

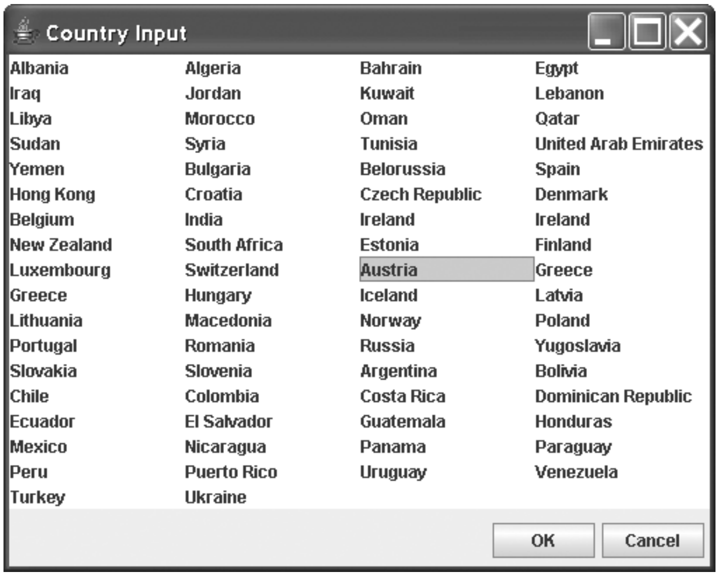
Such interaction styles are often combined together. Below we summarize design aspects that derive from the general principles outlined above for each style. Such styles can be thought of as abstract recurring patterns for GUIs.

Menu selection

The *menu selection* interaction style is an academic term in which a user selects items from a list of available choices. With this term we mean here a generic, abstract situation that doesn't only refer to GUIs – for example, graphical menus are only a particular case of such an interaction style – but the any type of user interface.

Menus are used for selecting items such as commands in a systematic way. The generic term 'menu' covers any selectable item, such as links in a hypertext page, commands in a drop-down menu, buttons, voice commands in voice interfaces and so on.

Figure 2.18 shows an example of a two-dimensional graphical menu implemented as a list. In the following we focus on graphical menus only, because they are the most common form of menu selection in Java user interfaces. The selection process is quick and users have a clear view of all possible choices. However, as the number of items grows, or if items lack a clear indexing organization such as a geographical or alphabetical ordering in this example, this approach reaches its limits.



Albania	Algeria	Bahrain	Egypt
Iraq	Jordan	Kuwait	Lebanon
Libya	Morocco	Oman	Qatar
Sudan	Syria	Tunisia	United Arab Emirates
Yemen	Bulgaria	Belorussia	Spain
Hong Kong	Croatia	Czech Republic	Denmark
Belgium	India	Ireland	Ireland
New Zealand	South Africa	Estonia	Finland
Luxembourg	Switzerland	Austria	Greece
Greece	Hungary	Iceland	Latvia
Lithuania	Macedonia	Norway	Poland
Portugal	Romania	Russia	Yugoslavia
Slovakia	Slovenia	Argentina	Bolivia
Chile	Colombia	Costa Rica	Dominican Republic
Ecuador	El Salvador	Guatemala	Honduras
Mexico	Nicaragua	Panama	Paraguay
Peru	Puerto Rico	Uruguay	Venezuela
Turkey	Ukraine		

Figure 2.18 An example of a (rather unusable) two-dimensional graphical menu

Organizing menus is an important issue, especially when many items are available for selection.

The criteria mostly used are:

- *Task-related organization*. This is the single most successful strategy for organizing menu items. If items are organized at design time following some relevant semantic criteria, it greatly helps users in accessing them at runtime – as long as the semantic criteria used is clear to the end users.

- *Hierarchical grouping* in tree structures. Hierarchical menus are characterized by the number of levels (depth) and the number of items per level (breadth). Empirical studies have shown the advantage of breadth over depth in menu hierarchies. As a rule of thumb, menu hierarchies shouldn't be deeper than *three* levels. There are some practical rules for choosing the right hierarchical structure. A greater depth at the root is recommended, taking care to make sure that items are distinct and not overlapping. A broader range can be adopted on the leaves – the lowest-level items in a menu hierarchy.
- *Standardized organizations.* Adopting a standard menu organization helps users to acclimatize to new applications quickly, minimizing their required memory load when working. Later will see the Java Look and Feel guidelines prescriptions for organizing menus.

These strategies, used in combination, relieve users from the time-consuming task of finding an item in a potentially large menu space.

Figure 2.19 shows a simple **File** menu that follows the Java Look and Feel guidelines. Even in such a simple case, adhering to well-established conventions is essential. Providing non-standard, arbitrary menu structures confuses users and can wreck an application's productivity potential.

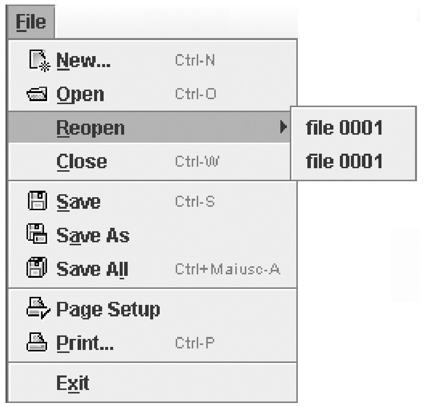


Figure 2.19 An example of commands menu using the Java Look and Feel

Form filling

Form filling is also used widely in user interface design. The general principles of a form-filling interaction style are those for general data entry (Shneiderman 1998) or (Nielsen 1993), among others. These are:

- Ensure that data-entry transactions are consistent
- Focus on minimizing end-user input actions

- Keep memory load on users as low as possible
- Ensure compatibility of data entry with data display
- Allow user flexibility and control of data entry

These criteria closely resemble the general ones discussed previously (see *Some general principles for user interface design* on page 43). There are a number of general guidelines for designing data entry forms:

- Group and sequence fields logically, for example by grouping together related items using the separation display strategy introduced on page 47.
- Supply clear instructions. Specifically, provide meaningful labels and titles for forms, and add explanatory messages for fields. This can be achieved both via contextual help and through *tooltips* – pop-up labels that appear if the mouse pointer is held over a particular item.
- Adopt consistent terminology and abbreviations. Avoid cumbersome terminology and provide names as close as possible to the business domain to which users are accustomed.
- Design the form's appearance using a visually-appealing layout – we detail one in Chapter 3 for J2SE – and by means of signaling visually which fields are mandatory and which are optional.
- Provide effective navigation focus[12]. Apart from the mouse, other navigation options such as the keyboard should be considered.
- Handle errors using two strategies: prevention, whenever possible, or displaying meaningful error messages if error states cannot be avoided.
- Provide an effective completion signal, making it clear how to complete the data entry task associated with the form.

The representation of text, wherever it appears in the GUI (buttons, menu items and so on), is another important aspect in GUI design that is not limited to form design. Some general rules apply:

- Keep labeling text brief, and preferably locate it beside the related component, as shown in Figure 2.19 on page 59.
- Use ellipses in menu items and buttons to show that another dialog will appear to accomplish the command. For commands that show a dialog as their entire result, the ellipses are redundant and should not be used. For example, an **Open** command doesn't need ellipses, because its sole purpose is to show a File Chooser dialog, while a **Print...** command does, because it

12. The term *focus* refers to the active area in a window or a panel where the user's next keystroke will be received. Focus represents which GUI area or widget is going to receive keystrokes.

will prompt the user with a supplementary print properties dialog, instead of directly starting printing.

To avoid visually overloading them, do not use ellipses in toolbar buttons.

- Adopt a coherent rule for titles in windows. An example of such a rule could be *object name - application name*, while titles in secondary windows could follow the format *descriptive name - application name*. This rule is adopted in several standards, including the Java Look and Feel guidelines. If you prefer to create your own rules, be sure to apply them consistently.
- All messages in English, such as command names, labels windows title, tab names and so on, should follow the *headline capitalization* rule:
  - Every word is capitalized except articles (‘*the*,’ ‘*an*,’ and ‘*a*’), coordinating conjunctions (‘*but*,’ ‘*or*,’ and ‘*and*’) and short prepositions (such as ‘*to*,’ ‘*in*’).
  - The first and last words in a sentence are always capitalized, no matter to what category they belong.

  These rules are not used for full sentences, where normal sentence capitalization is used instead – only the first word is capitalized. Examples of such text include alert message boxes, error or help messages, status bar messages, or general labels that indicate a status change, for example ‘Download is 30% complete,’ in contrast to a text label, which would follow the headline capitalization rule: ‘Download Progress.’

  These rules only apply to English text. For other languages, you should refer to language-specific official guidelines, where these are available.

Creating effective forms

Creating an effective form requires some care. Figure 2.20 shows an example of a simple yet well-designed form dialog. Navigation has been enhanced and the whole interaction process smoothed with few simple details:

- Every field in the form is easily reachable using a related mnemonic – for example typing **alt+x** moves the focus to the *combo* box that indicates the gender of the person to be input. (A combo box is a GUI widget that users click to show an associated list of possible values. Some combo box options can be selected directly using the keyboard, while others allow new values to be input only through the drop-down list, thereby constraining input to the available values).
- Tab traversal – using the **tab** key to move the focus from field to field – is logical. For example, tab traversal skips the disabled **File** name field, so that if tab is pressed when the focus is on the combo box, it transfers to the **Browse…** button.
- When the dialog first appears the focus is automatically set on the **Name** field to facilitate input.

- Standard globally-applicable buttons are added at the bottom of the dialog. In this way the user knows from past experience how to dismiss the dialog.
- Effort has been spent on the visual appearance of the dialog to avoid extraneous graphics and provide a pleasant overall effect.
- Information about accessibility – providing ease of access for users with disabilities – has been added to the GUI, even if this is only partly visible. In fact, some accessibility information is stored in the GUI to allow it to be accessed by special tools such as text magnifiers or interaction facilitators. Such invisible features could be very important in some situations and should always be used.
- The field's alignment and layout of widgets provides a pleasant overall visual appearance.

Such little details greatly enhance a dialog's usability. Try it for yourself by downloading and running the relevant code.

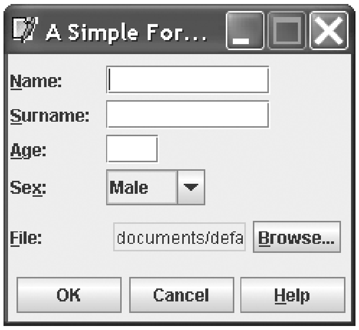


Figure 2.20 An example of form dialog

Well-designed forms should always provide a clear completion signal. As shown in Figure 2.20, the prescribed mechanism in the Java design guidelines is to provide buttons – usually at the bottom or right-hand side of the window – with explicatory text such as **OK** and the associated behavior of closing the dialog and accepting its contents. Figure 4.15 on page 138 shows a prototype GUI that does not respect this rule, and in which the completion buttons are included within the information area, potentially confusing the user.

We will return to form design in Chapter 4.

Language-based styles

There are two other interaction styles that we will mention here for completeness, although we won't cover them extensively in this book.

- *Command language*. Using a language that must be input by users via a command line is sometimes the only solution in certain situations. For example, using a command-line user interface sometimes is the only solution in certain domains, such as a rich programming environment in which rules represented as scripts needs to be manipulated and executed. Providing a graphical UI for representing such scripts can be expensive and might ultimately result in lower usability.
- *Natural language*. Though quite complex to implement, natural language (both via text or speech recognition) can be useful in some cases.

In the next section we discuss an important interaction style for building high-quality user interfaces.

Direct manipulation

Figure 2.21 shows an example of a direct-manipulation interaction mechanism for dealing with visual objects. The items in the user interface can be dragged, edited or deleted by performing operations on them in a consistent way.

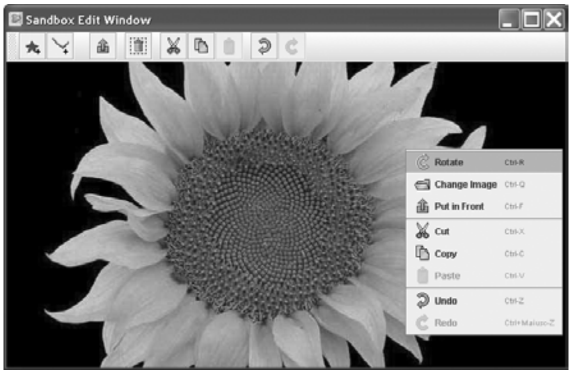


Figure 2.21 A direct manipulation interaction example

We will come across many examples of direct manipulation GUIs throughout the book. Direct manipulation is attractive from an implementation viewpoint, because it can be implemented easily using Java 2 Standard Edition (J2SE) technology, rather than another client technology such as Web pages. Direct manipulation is more difficult to implement on Java 2 Micro Edition (J2ME) devices, due to its limited pointing facilities and graphics display[13].

13. J2SE is the Java environment designed for desktop computers and laptops (those that provide a mouse pointer, a large graphic display, keyboard and so on), while J2ME is the Java environment for handheld and portable devices (ranging from palm top devices to wireless phones).

The appearance of the mouse cursor is often used to give some hints about the *affordances*[14] of the given items. For example, when a dragged object can be dropped onto another, the cursor appearance changes accordingly. We will see some examples of this feedback technique in the book.

2.4 Conceptual frameworks for UI design

In designing complex artifacts, the gap between the intended results and the chosen technology can be so wide that some conceptual structure is needed. By *conceptual framework* we mean a set of coherent concepts that structure the different phases of development – design, implementation, and so on – of UIs. Developers and designers therefore follow an abstract, principled approach to organizing the UI[15].

Conceptual frameworks provide a reliable and proven mindset for organizing the design, reducing risks and improving quality. Furthermore, by leveraging reuse, designs can be standardized and various economies derived.

Figure 2.22 shows some of the major conceptual frameworks used in today's UI design.

Various approaches to UI design are shown in the figure:

- *Entity-based*. This is a family of conceptual frameworks that structure UI development around the concept of abstract *entities*, their properties and interactions. The members of this family of conceptual frameworks vary only in the way in which the abstract concept of entity is defined. Object orientation applied to UI design is a member of this family of conceptual approaches.
- *Metaphor-based*. This approach focuses the whole design around metaphors. By leveraging those metaphors, users can use the software without having to learn the underlying system model.

14. The term *affordance* was introduced by Gibson and subsequently used by Donald Norman (Norman 1990) to describe the possible functions of an object. In Norman's words "a chair affords support, a pencil affords lifting, grasping, turning, poking, supporting, tapping and, of course, writing".
15. We focus this discussion on conceptual framework for UI design, and not its implementation, or other development phases. This discussion is completely separated from the actual implementation of the UI itself. Some of the approaches described here can in fact be applied to the entire software development lifecycle, not just UIs.

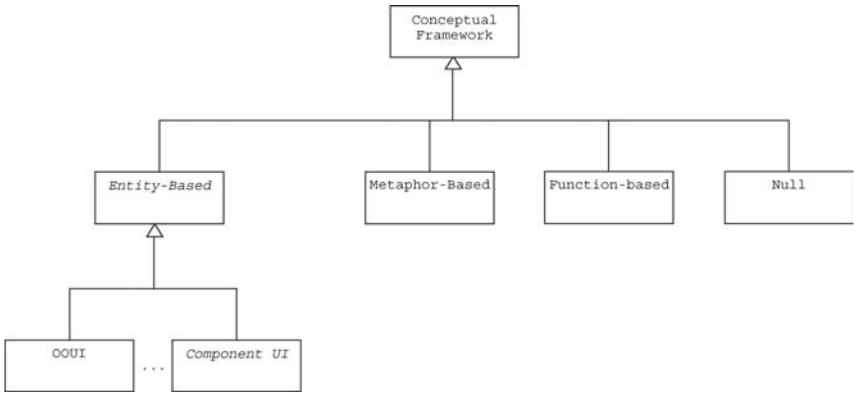


Figure 2.22 Major conceptual framework for GUI design

- *Function-based*. UIs developed using the function-based approach can be thought of as set of functions derived directly from the analysis of use cases and requirements. This UI design approach is also known as *application-oriented*. For example, we can think of a word processor as of a set of functions like 'save current document,' 'reformat selected text' and so on. The resulting UI is simply the most usable way to provide these functions given the chosen implementation technology.
- '*Null' conceptual framework*. This represents the default conceptual approach of novice designers that have never came across a sound UI design introduction.

Entity-based approaches to UI design

Entity-based approaches to UI design are characterized by the notion of the abstract concept of an entity and its relationships.

Such entities provide different views of their internal state to users, and interact with other entities within the UI environment. When designers adopt such an approach, any item in the UI is thought of as being part of an abstract entity, and users directly manipulate these entities to perform their tasks. This is usually accomplished through contextual interactions – that is, users select an entity and perform some operation on it, such as invoking a pop-up contextual menu.

A particular class of entity-based conceptual frameworks leverage object-orientation theory for defining the abstract model. We introduce this in *Object-oriented user interfaces* on page 69. Other members of this family of approaches are component-based UI design, in which the term *component* refers to a higher level of granularity than 'classic' OO objects, and various others such as the Naked Object approach to UI design – see www.nakedobjects.org.

Metaphor-based approaches to UI design

UI designs can also be shaped around the concept of metaphors. Designing a UI that resembles some real-world situation – a *metaphor* – helps users better understand the system, leveraging their knowledge of the metaphor instead of the system's actual implementation. Thus for example dragging icons could be equivalent to moving items in the physical world, and dropping an icon onto the wastebasket in the GUI is equivalent to invoking the 'delete' operation on that item. Such idioms are coherent with a desktop metaphor and do not require the user to learn implementation-dependant commands.

Historically the metaphor approach to UI design represented the first major improvement over the functional approach. Its most famous example is the desktop metaphor for operating systems originally adopted in the Xerox Star software around 1982. Metaphors can also be used at various levels in UI design even without using a fully-fledged metaphor-based conceptual framework. Examples of limited metaphors could be the wastebasket for deleting items, or using the metaphor of a restaurant menu to display the available features in a product.

Metaphor-based approaches are different than entity-based ones in that the latter devise a generic abstract world that is applicable and repeatable in different domains. For example, one can model both a bank account and a file system directory with the same abstract concept of an 'entity.' Metaphors, in contrast, are domain-dependent, and are defined in an ad-hoc way. For example, we can model bank accounts in our UI as following a 'personal logbook' metaphor, which would not be useful for representing the file system UI.

This approach suffers from two problems:

- Good metaphors are hard to find.
- Even when a good metaphor is found, it may turn out to constrict our design.

As a trivial example of this, consider the way in which file systems are rendered in current operating system GUIs. At first it may seems that a file system is adequately modeled by means of the folder and file metaphor. However, if we had to follow the real-world metaphor of files and folders, as we know them in the physical world, we would end up of a deficient UI. First of all, real world folders cannot be nested indefinitely – this is only a mathematical abstraction that software provides us, constrained by memory resources. Second, the folder and file metaphor works fine for certain interactions such as renaming and moving, but seems a bit odd for other, such as cutting and pasting folders. It also has no parallel at all in the real-world metaphor for some operations, such as compressing the content of a folder.

Many software development approaches advocate metaphors in software design and implementation. Software documentation also benefits from the use of clear, higher-level, lifecycle-wide metaphors. Chapter 11 shows a practical example of employing an articulate metaphor in a GUI design.

Examples of metaphors used as tools can be found in different fields of computer science, such as software development (Beck and Andres 2004), with a hands-on perspective (McConnell 1993) or in the analysis process (Fowler 1997).

There are many resources available on the Web for the use of metaphors in UI design: for a critical view of such a UI design approach, see for example (Cooper 1995).

Function-based approaches to UI design

Function-based UI designs are built around the functions the system is required to perform and the interactions between them. The academic meaning of the term 'GUI' historically refers to the first generation of UIs using rich graphical technology that leveraged the underling procedural implementation approach. Menu bars, toolbars and menu items are extensively used in this approach because they help to bundle together a set of disparate *functions* the UI performs on behalf of the user. This UI design approach, as well as the 'null' approach, are the most common in current software.

'Null' approach to UI design

The so-called 'Null' approach is the approach implicitly used by developers when they do not appear using any explicit approach at all. As we said on page 35, human beings always interact with the world through semantic models, even when they are unaware of them. The Null approach, given a UI software technology, consists of putting all user requirements on the screen in one way or another, often trying to mimic other existing UIs. This is not really a bad approach in itself, but clearly lacking the backing of a sound theory, this approach has a tendency to produce confusing UI designs that do not scale well.

To better grasp the differences in the approaches described above, let's take a common example: the GUI of an OS. In order to make our point we will simplify our discussion and overlook details:

- The GUI of Windows 3.1 (Figure 2.23) can be seen an example of a functional design approach mixed with a limited use of metaphors (mainly for files and folders): the GUI was designed around a set of operations to be performed on the file system.

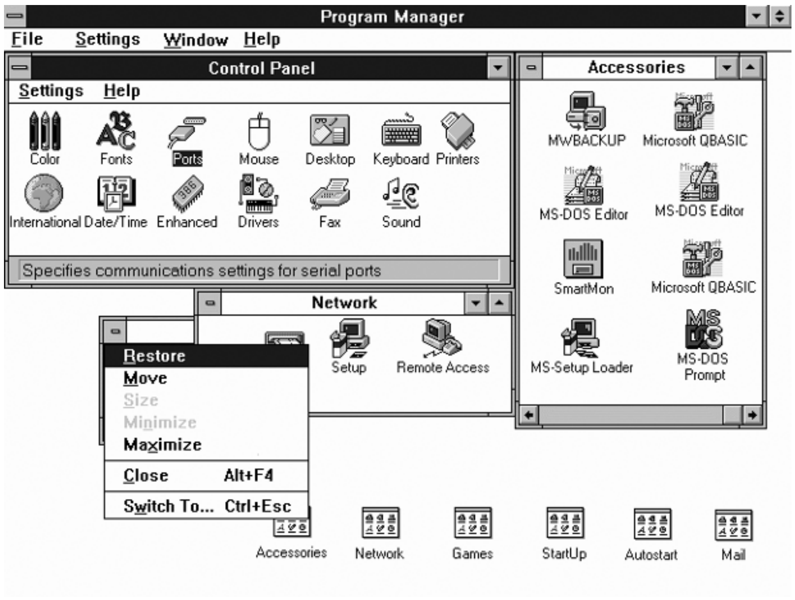


Figure 2.23 Windows 3.1

- In contrast, the original Macintosh UI (Figure 2.24) was designed following the desktop metaphor approach, with very few exceptions, thus providing a homogeneous and reliable concept model for end users.

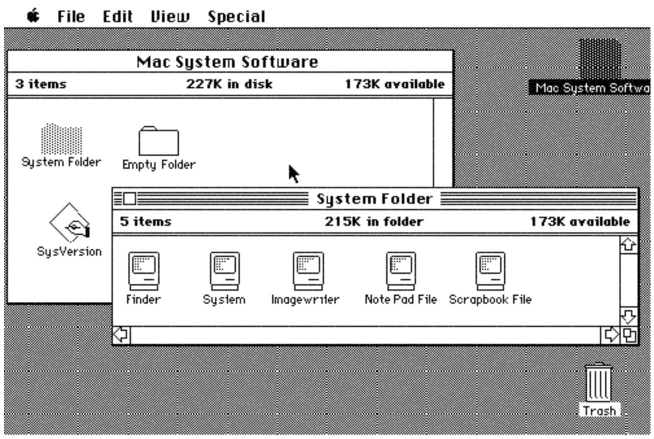
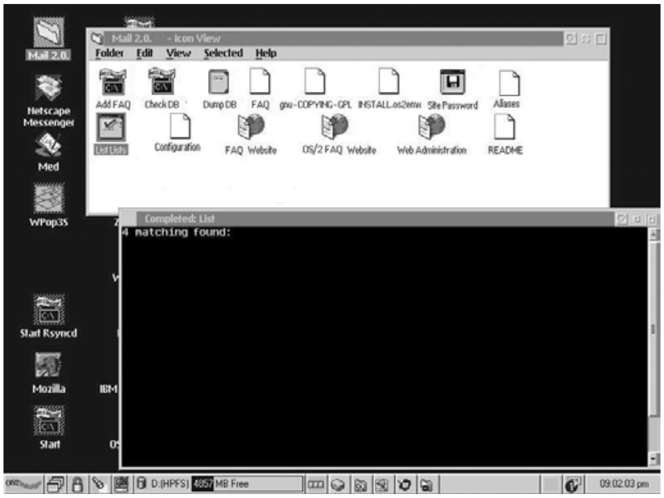


Figure 2.24 The original Apple Macintosh UI

- OS/2 (Figure 2.25) was designed completely as an entity-based GUI – every item accessible through the GUI was a conceptually well-defined entity, with its own set of available operations, properties and configuration attributes.

Despite these GUIs basically representing the same domain – the file system and basic OS functionalities – the UI design approach behind them was very different, shaping the UI in its various detail aspects.

The next section describes a particular case of entity-based approaches to UI design in detail, the Object-Oriented User Interface approach.



```
4 matching found:
```

Figure 2.25 The OS/2 UI

Object-oriented user interfaces

The idea behind the Object-Oriented UI (OOUI) design approach is simple: apply OO abstract principles to UI design. An OOUI consists of a set of *abstract objects* designed following OO principles such as abstraction, implementation hiding, and so on.

Unfortunately the term OOUI, despite widely accepted in the literature (see for example (Mandel 1997)) is rather confusing when applied in OO programming contexts such as the Java language. In fact, despite being two approaches founded on the same conceptual footing (object orientation) they are separate in practice. OOUI relates to UI design, while OOP focuses on software programming. Avoid confusing the two approaches, by ignoring for now the underlying technology on which the UI will be implemented.

OOUI focuses on defining abstract objects with which the user will interact via the user interface. Unlike the metaphor-based design approach, such 'objects' are not required to follow any metaphor from the physical world. OOUIs are a coherent collection of such objects – usually referred to as an *ecosystem* – that are available for user interaction.

The direct manipulation interaction style couples naturally with the object-oriented paradigm. Think for example of the windowing metaphor used in the Apple Macintosh, IBM OS/2, Microsoft Windows, and others, on which you can manipulate objects such as files and directories directly. A file object behaves consistently throughout many different applications, providing the same set of functionalities – move, copy, rename, and so on – like an abstract object.

Once you have designed your application GUI as a coherent object ecosystem, it is natural to interact with it by means of direct manipulation, because the UI appears as a virtual world made up of objects that can be operated on by the user. We will explore such a GUI design approach extensively, because it happens to dovetail nicely with the object-oriented nature of Java.

Java developers should be careful over some subtleties. Object-oriented programming and object-oriented user interfaces are different. One could implement an object-oriented user interface using non-OOP languages and platforms, while an OOP language like Java can be used to build any kind of user interface, command-line ones included. Furthermore, OOUI is limited to the software as it appears to the user – that is, the concepts, tasks and overall semantics exposed by the application to its users – while OOP is used to implement all of the application. A specific OOUI object, as perceived by the end-user, can be implemented with many Java classes, and, conversely (although more rarely) one Java class can implement several different OOUI objects.

Despite these differences, with thoughtful software design it is possible to bridge the two worlds systematically, providing a natural mapping between the OOUI GUI design and its underlying Java implementation.

OOUI objects are used to represent the internal state of the application and to enable user interaction. Accordingly, there is little point in providing OOUI *classes*. These are an OOP mechanism for conveniently creating objects. While interacting with an OOUI, users create new objects by manipulating existing ones.

To better grasp the OOUI concept, consider the main differences between traditional graphical user interfaces (function-based or application-oriented) and OOUIs[16]:

16. These differences hold also between generic entity-based UI approaches and function-based ones.

- In an OOUI users interact with objects, while in application-oriented interfaces the interaction is organized by function. In functional-based GUIs the software is rigidly organized by function. In OOUIs, in contrast, the user interacts with objects in a less structured, freer environment.
- In OOUI there are few, common objects. Combining and manipulating them produces many different results. The aspect of a coherent metaphor for object interaction is key. In traditional GUIs there are many applications, one per task, while in OOUIs the environment is common and functionalities lie within objects and their possible interactions.
- Each approach fosters different cognitive theories: traditional GUIs enforce the traditional cognitive model (a set of predefined operations that need to be learned by end users as conceived at design time by the developers), while OOUIs allow for a learning style closer to the *constructivist* cognitive approach in which users are free to interact with the system at their own pace, constructing their user experience without strongly predetermined constraints.
- Functional-based GUIs are composed of global menus. Groups of items are represented with lists. In OOUIs the objects themselves essentially convey all possible interactions.

Function-based user interfaces can be best suited for stand-alone programs, in which the user wants to accomplish one or more well-defined, circumscribed tasks. GUI designed following the OOUI approach can be useful for large applications such as operating systems, in which many functions are available and a large number of possible combinations are legal.

In this book we will combine these design approaches, with the ultimate aim of providing the most usable user interface depending on the current situation.

Object views and commands

In OOUIs, each object can be manipulated in several ways. Following Donald Norman's terminology (Norman 1990), each object has its own *affordances*. For example, some objects can be dragged, dropped onto other objects, or can provide a list of their available commands via contextual menus. Other objects cannot be dragged at all. Generally, every object provides a set of commands with which it can be manipulated. Contextual menus are the proper place to provide object command access. By convention, clicking an object on the screen with the right mouse button (or in other ways, depending on the given platform) triggers the contextual menu that contains all the valid commands for the object.

Objects can be viewed in different ways. Suppose you have a file directory. You can see it as a 2-dimensional container of icons, or as a tree in which each node can be a file or a folder. Thus the same items are viewed in different ways. You can also open a file to see its contents, providing yet another view of your file object. (Mandel 1997) mentions four basic types of object views: composed, contents, properties, and help:

- *Composed views* are views of an object obtained by combining other objects.
- *Contents views* show the contents of an object, used especially for containers objects.
- *Properties views* are used to show specific details of an object, and can also allow editing by inspecting a value and modifying it as required if this is meaningful within the application. Properties views for discrete data usually use the form-filling interaction style.
- Help views shows help data.

We don't adopt a fully-fledged OOUI approach in this book: all the OOUI examples we provide use a simplified version of the OOUI approach. For a 'full' OOUI-driven design methodology, see for example IBM's OVID (Objects, Views, and Interaction Design).

We will see the OOUI approach implemented in Java in Chapters 14 and 15.

2.5 Assessing the quality of a GUI

The quality of a user interface is dependent on its *usability*. Software usability is the characteristic of a given application of being easy to use within a set of constraints such as the target user population, development budget, and so on.

Ease of use can be measured by the number of mistakes made in the use of the application by a sample user group, how quickly they can perform given tasks, users satisfaction, and how quickly the system is learned by novice users.

We won't discuss robustness and other implementation-related parameters here. An example of testing for robustness is systematically trying all combinations of buttons and other controls to see whether the GUI responds coherently, or produces unforeseen behavior.

Assessing the quality of a user interface is not a trivial task. There are many aspects to consider, and much depends upon the particular situation – the design approach followed, the end user population and other constraints. Over time several approaches have consolidated, although the fact that there are so many different criteria for GUI quality assessment underlines the complexity of such an activity.

Some of the main approaches are:

- *Expert review and survey*. Usability experts review the GUI and produce a document in which GUI weak points are identified and suggestions proposed. The review may involve a formal inspection in which the user interface is discussed with designers.

- *Usability testing*. This term encompasses all types of trial that test the GUI for usability. These involve considerations such as choosing the usability parameters to measure, the way in which such parameters will be evaluated and so on. In general usability testing is a complex discipline that need specialized personnel.
- *Acceptance testing*. Here the developer's quality assurance department define objectively measurable tests for the final GUI. A key point is the establishment of precise acceptance criteria. Acceptance tests usually cover:
  - Novice user's performance, in which the first part of the learning curve for users new to the application is measured.
  - Regular user's performance, the most commonly used acceptance tests.
  - Testing for retention, in which user expertise with the system is measured after a period of non-use of the application under test, usually of some 2–3 weeks.
- *Robustness* and other software-related tests. Usability depends on the reliability of the implementation. Buggy GUIs, no matter how well-designed, result in a poor-quality end user experience.

Other approaches to GUI assessment exist, for example Cognitive Walkthrough. The interested reader can find more details in (Nielsen 1993) or (Preece 1994).

Cognitive Walkthrough is an approach for evaluating user interfaces. A group of evaluators first determine the major tasks the system must perform. They then analyze each task, decomposing it in a sequence of steps. For each step they adopt a cognitive approach – they evaluate how difficult is for the user to identify and operate the interface element most relevant to their current subgoal, and how clearly the system provides feedback to that action. This approach is especially useful for assessing the usability of a system for users in exploratory learning mode – that is, first-time or infrequent users. Cognitive walkthrough can be performed on early prototypes as well as the final GUI.

The next section discusses a common approach to evaluate a GUI by adopting a set of rules (heuristics) that have been devised for assessing its overall quality.

Usability heuristics

When evaluating a GUI, whether in review or in usability testing, experts use this simple set of criteria in order to assess its effectiveness. The 'classic' set of such heuristics is:

- *Visibility of application status*. This involves checking whether the GUI expresses its current internal state by appropriate feedback. This is usually done by means of a status bar, mouse pointer shape, progress dialogs and so on. This criterion checks whether these means are properly used in the GUI, and whether they effective, or merely disturbing?

- *Match between application and the real world*. Terminology and the overall GUI should be as 'current' as possible. This criterion checks whether the GUI uses weird metaphors or other unnatural kinds of interaction.
- *Consistency and standards*. When checking for this evaluators should ask themselves whether the given GUI is compliant with required design guidelines, and whether any specific part of the GUI is coherent with the remainder. Evaluators look for consistency by asking themselves questions like 'Do the completion buttons always appear at the same place in a dialog?'
- *User control and freedom*. This criterion checks whether the GUI encourages exploration and error recovery. Typical hints are the effective support for undo/redo functionalities.
- *Error prevention*. This criterion checks whether the GUI is designed in such a way as to minimize user errors. A common means to achieve this is an apt use of constraints and metaphors. Another common expedient for avoiding user errors is to disable commands when they are not meaningful.
- *Helping users recognize, diagnose and recover from errors*. Not all possible errors may be prevented by clever design. This criterion checks whether the application provides helpful messages and constructive communication in the case of errors, as well as assessing the quality of error messages.
- *Recognition rather than recall*. Users need to remember specific commands or a particular interaction, and the GUI need to offer a clear visual route through all the available options. This criterion checks how effectively the users STM is exploited.
- *Flexibility and efficiency of use*. This criterion checks the extent to which it is possible to customize the GUI, and whether the GUI is suitable for expert users. It checks for the availability of accelerator keystrokes and other shortcuts that can make the GUI suitable for expert users as well as for novices.
- *Aesthetic and minimalist design*. This criterion focus on the rational and functional graphic appearance of the GUI. It checks whether the GUI is appealing visually, without being distracting or annoying.
- *Help and documentation*. This criterion checks the quality of the help system. It verifies that the supplied documentation is practical and concise, easy to search and effective in solving user needs.

Appendix A shows a simple questionnaire for evaluating Java user interfaces. This is am empirical adaptation of general questionnaires – see for example (Shneiderman 1998).

2.6 Summary

In this chapter we presented some introductory discussions about effective GUI design. In particular:

- We introduced some basic principles for human-computer interaction, showing how a basic understanding of human cognition can help in the design of high-quality user interfaces.
- We presented five main interaction styles. We will deal with three of these in the remainder of the book: *menu selection*, *form filling* and *direct manipulation*.
- We introduced object-oriented user interfaces (OO UIs) as a special case of direct manipulation. This approach will be adopted in some of the examples provided in the book.

In the next chapter we introduce practical GUI design for Java platforms, and introduce the Java Look and Feel guidelines.

3 Java GUI Design

In this chapter we introduce user interface design for the Java platform, focusing our attention on J2SE GUIs. The chapter is structured as follows:

3.1, *Java technology for GUIs* introduces the components that Java provides for building user interfaces.

3.2, *Cost-driven design* describes how cost constraints can be taken into account in user interface development.

3.3, *Exploring the design space for a point chooser* gives some examples of practical GUI design, using as an example the design of a component for selecting points on the earth surface.

3.4, *Design guidelines for the Java platform* introduces the idea of user interface design guidelines, specifically those for Java.

3.5, *The Java look and feel design guidelines* describes the Java look and feel guidelines in detail.

3.1 Java technology for GUIs

This book deals principally with graphical user interfaces composed of visual components. This kind of interface is made up of widgets and windows, following the well-established syntax of point-and-click GUIs.

Assembling the components

This section discusses the basic organization of a Java-based GUI. Java GUIs are organized in reusable units that are directly mapped onto groups of Java classes. For example, in the Swing library a visual tree component (also called an *expandable list*) is implemented as a set of more than a dozen standard classes and interfaces that can be configured or specialized as necessary. Such classes include specialized event listeners, cell renderer, and data models – see for example (Geary 1999).

In contrast, the analogous component in the SWT library is implemented using only three Java classes.

Focusing on the Swing library, even the simple dialog in Figure 3.1 below is implemented using instances of several different Java classes. Figure 3.2 shows the conceptual layering of the main user interface components that implement the dialog in Figure 3.1.

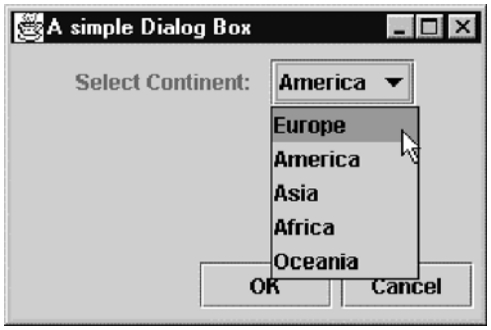


Figure 3.1 A simple dialog

The number of Java classes involved in the previous example is in fact much larger – Figure 3.2 shows only some of them. For example, the main container `JDialog` and the bottom panel use a layout manager instance that supervises to the contained widgets layout. Note that we have employed two separate panels

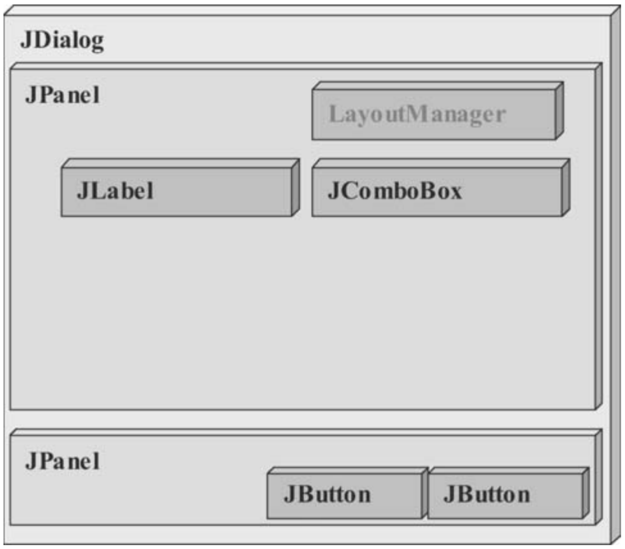


Figure 3.2 The conceptual layering behind a simple dialog using the Swing toolkit

in our design, one for the buttons at the bottom and the other one for the content area at the center. This is essentially for engineering reasons – it allows us to reuse the standard buttons panel. We discuss component reuse in *Leveraging object-oriented programming* on page 166 in Chapter 4. For a complete list of widgets available in the Swing, SWT, and AWT toolkits, refer to Chapter 11.

Three levels of component cost

In my experience I have found it helpful to distinguish between three kinds of visual component, depending on their relationship to the existing base libraries (such as Swing, AWT, or third-party ones like SWT). Categorizing GUI components in this way is useful for driving top-down development, from GUI design to software development, testing and so on:

- *Standard* components. These are standard GUI objects that are typically used with only shallow customization. These are therefore the cheapest components to use.
- *Custom* components. These are non-trivial subclasses of the standard library objects. They are also relatively inexpensive to develop, but limit designers in the degree to which they can customize existing components.
- *Ad-hoc* components. These visual components are developed to solve some special problem that cannot be solved by extending an existing component. These are of course expensive to build, as they often require additional GUI design effort, but can provide the highest quality resolution of requirements.

Figure 3.3 shows some examples of these components.

In Figure 3.3, from the left-hand side we have an example of the `JTree` standard component for the Swing library. Developers only need to change a few properties from the default values, and populate it with the required data. The center of the figure contains a custom `JTree` component, in which the same widget has been deeply customized and some of the standard classes have been extended to provide custom behavior. Finally, the right-hand side shows an example of an ad-hoc component that needed to be built from scratch because the standard library does not provide it.

This categorization is based on standard libraries such as Swing and SWT, is cost-driven and somehow arbitrary. Depending on the target Java environment, designers can rely on various GUI libraries (AWT for basic Java 1.x applets, Swing for Java 2 GUIs, and some specific toolkits for J2ME profiles).

This classification approach can be used by designers based not on standard libraries, but rather on third-party ones such as specialized set of components, or proprietary, in-house developed GUI toolkits, for example).

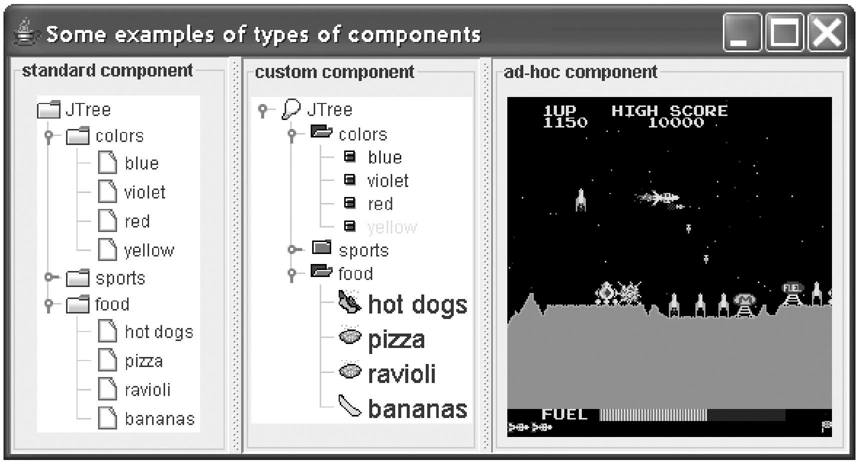


Figure 3.3 *Example of the three kinds of visual component*

The components in Figure 3.3 are organized by increasing development cost, from the cheapest on the left to the most expensive on the right.

This classification is however blurred, because it depends on many factors, not least the experience of the developers involved. For example, a design team that has a background in video game development may find easier to use a particular ad-hoc component than to employ a custom tree component from a complex GUI toolkit, potentially reducing development costs.

At this point some readers may wonder why, when we are discussing GUI design, we are making such an implementation-driven distinction among different visual components? The reason is because, in my opinion and experience, quality-driven industrial design cannot be separated from its implementation.

We have used three words in the latter sentence that need explanation:

- *Quality-driven* design. Quality should always drive a GUI design. Quality here means usability. For example, when it would enhance usability, a good GUI design should employ direct manipulation instead of other cheaper but more convoluted metaphors.
- *Industrial* design. As long as you are not developing your GUI for fun or for some artistic purpose, you should remain grounded in basic principles like the development cost and usefulness of the final product. No-one works in an environment of limitless resources. In real-world projects, both time and human resources are often limited, and your design cannot ignore this. The search for perfection is limited by practical industrial constraints.

- *Implementation*. Designers should always consider the final implementation, especially when developing for Java. The richness and sophistication of the platform shouldn't waylay professional designers into over-using objects and classes, degrading performance.

Generally speaking, in practice a final design is determined by the trade-offs between quality and practical constraints – this is one of the main assumptions of this book. Distinguishing visual components by their cost impacts directly on the overall GUI design, and can make the difference between an inexpensive or a costly GUI. The cost is comprised of design (plus usability testing) and the required software development, debugging and testing. A standard component is almost ready-to use: designers don't have to design its GUI from scratch or test it for usability, having only to adapt it to the current situation.

When the customization cost passes a specific threshold – for example, non-trivial subclasses need to be written – we call them *custom* components.

3.2 Cost-driven design

The term cost-driven design describes an approach to GUI design that explicitly takes into account development costs.

Ad-hoc versus custom – the difference between 'run' and 'ride'

What is the actual difference between an ad-hoc component and a customized one? When should one employ one instead of the other?

A number of parameters influence this design choice:

- *The application domain*. Sometimes the application domain dictates the kind of components used in a GUI. We will see some examples in the following sections.
- *Required GUI quality.* The quality of a GUI is a further input parameter to the design.
- *Types of users*. Depending on the user population – for example, novice users accustomed to drag-and-drop GUIs – one type of components could be preferred.
- *Practical constraints*. These include time-to-market, development costs, context-dependent constraints and so on.

In this section we discuss the difference between ad-hoc and customized components by means of practical examples.

Figure 3.4 shows the prototype GUI of a hypothetical control panel for the underground railway network in Rome, Italy.

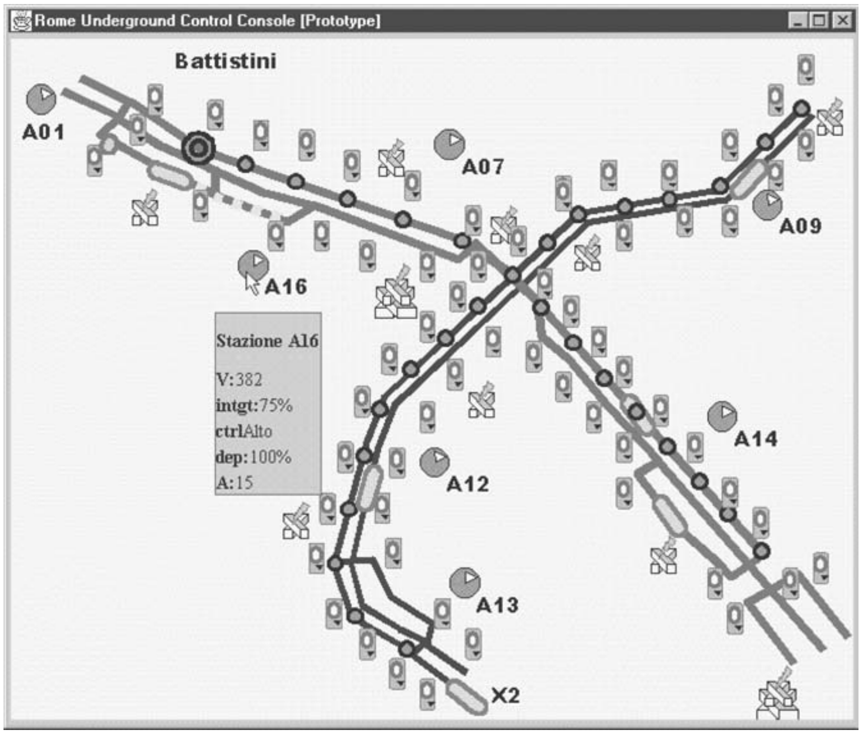
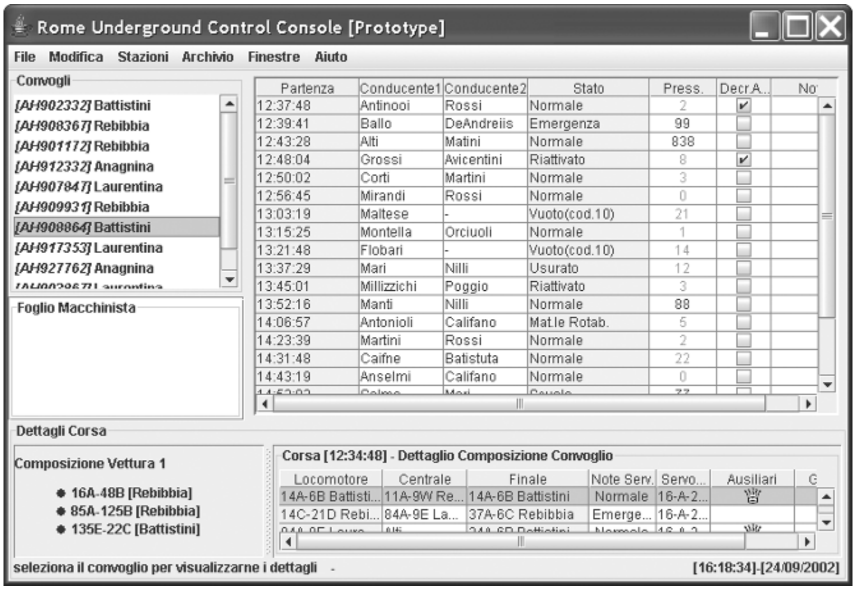


Figure 3.4 Ad-hoc prototype of the Rome underground system control console

Such a design is quite intuitive, pleasant to interact with, as it is essentially based on the direct manipulation interaction style, and relatively simple to use. Accidentally, because of its audience – it is intended for railway technicians – it is full of acronyms and technical jargon. It looks like a simple 2D-video game. Trains move on the tracks, data is queried both via tool tips for a brief summary or by double-clicking on the particular item, when a dialog pops up with the details.

This design has one major shortcoming. Such an attractive GUI is quite hard to develop. Because of the domain – complex but well-formalized – and the specialized technical audience – some of the complexity can be transferred out of the GUI to the users themselves. This can be done by a greater dependence on manuals, help support and internal training. Sadly, however, this is also often the case with badly designed GUIs.

We can imagine that such an approach might produce a much cheaper prototype like the one shown in Figure 3.5.


Rome Underground Control Console [Prototype]
File Modifica Stazioni Archivio Finestre Aiuto
Convogli
Foglio Macchinista
Dettagli Corsa

[AH902332] Battistini
[AH908367] Rebibbia
[AH901172] Rebibbia
[AH912332] Anagnina
[AH907847] Laurentina
[AH909931] Rebibbia
[AH908864] Battistini
[AH917353] Laurentina
[AH927762] Anagnina

Partenza	Conducente1	Conducente2	Stato	Press.	Decr.A..	No
12:37:48	Antinooi	Rossi	Normale	2	✓	
12:39:41	Ballo	DeAndreiis	Emergenza	99	☐	
12:43:28	Alti	Matini	Normale	838	☐	
12:48:04	Grossi	Avicentini	Riattivato	8	✓	
12:50:02	Corti	Martini	Normale	3	☐	
12:56:45	Mirandi	Rossi	Normale	0	☐	
13:03:19	Maltese	-	Vuoto(cod.10)	21	☐	
13:15:25	Montella	Orciuoli	Normale	1	☐	
13:21:48	Flobari	-	Vuoto(cod.10)	14	☐	
13:37:29	Mari	Nilli	Usurato	12	☐	
13:45:01	Millizzichi	Poggio	Riattivato	3	☐	
13:52:16	Manti	Nilli	Normale	88	☐	
14:06:57	Antonioli	Califano	Mat.le Rotab.	5	☐	
14:23:39	Martini	Rossi	Normale	2	☐	
14:31:48	Caifne	Batistuta	Normale	22	☐	
14:43:19	Anselmi	Califano	Normale	0	☐	

Composizione Vettura 1

- 16A-48B [Rebibbia]
- 85A-125B [Rebibbia]
- 135E-22C [Battistini]

Corsa [12:34:48] - Dettaglio Composizione Convoglio

Locomotore	Centrale	Finale	Note Serv.	Servo...	Ausiliari	G
14A-6B Battisti...	11A-9W Re...	14A-6B Battistini	Normale	16-A-2...		
14C-21D Rebi...	84A-9E La...	37A-6C Rebibbia	Emerge...	16-A-2...		

seleziona il convoglio per visualizzarne i dettagli - [16:18:34]-[24/09/2002]

Figure 3.5 *Prototype of the Rome underground system control console using specialized components*

This second design is at first much less appealing. An additional memory burden is placed on the user, more training is needed – technicians understand technical acronyms, but would need to be specifically trained in how to use table views such as those in Figure 3.5 – and novice users can initially be lost and therefore unproductive. Even if the Rome underground system is relatively simple (consider Figure 3.4), a design such as that in Figure 3.5 gives the impression of a more complex application than does its ad-hoc counterpart: its design favors a high density layout approach over the 'lean' strategy preferred in the design shown in Figure 3.4.

When ad-hoc is the only way to go

It is not always possible to resort to a GUI composed of customized components, no matter how cleverly they are used. Consider the next two examples, illustrated in Figure 3.6 and Figure 3.7.

The first prototype is an editor for UML class diagrams. Users drop symbols on the diagram and manipulate them as needed. We will see the actual code for something similar to this in Chapter 9, but here we are interested in discussing the design issues involved in making the choice between ad-hoc versus specialized component development options.

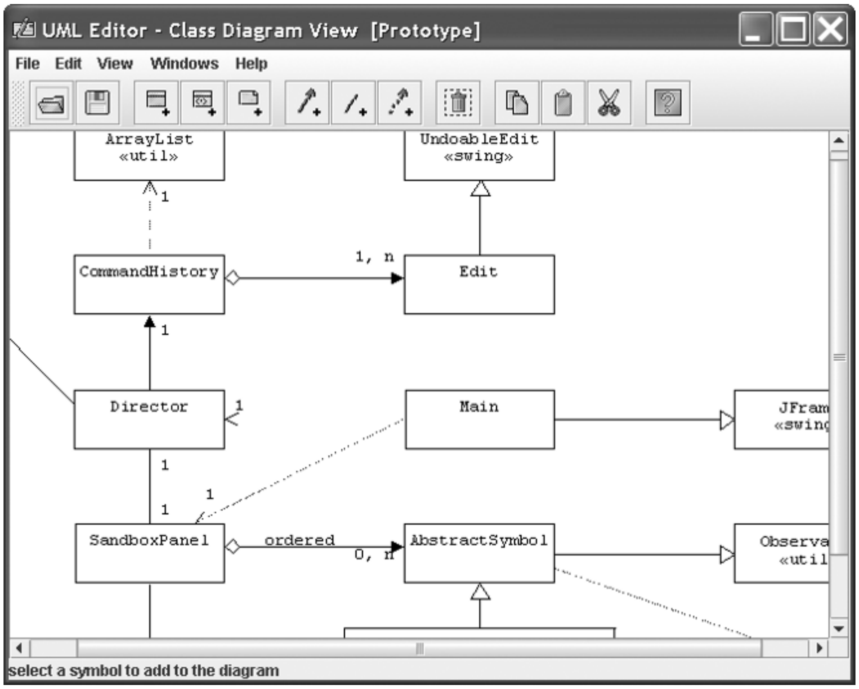


Figure 3.6 *A UML class diagram editor*

In this case we could imagine ways to present the same information that are simpler to develop, for example using a tree that gathers the relations of each class. Though a much poorer interaction method to the direct manipulation proposed in Figure 3.6, it is a viable and relatively usable interaction mechanism. The only problem is that one of the benefits this editor is supposed to provide lies in *the visual representation* itself. This GUI oversees the manipulation of UML class diagrams, which are themselves pictorial representations. In this case, the ad-hoc design approach, and its associated cost, is justified because it is part of the very purpose of the application. This is the case in many application domains, such as video games.

Consider another example. Even if not strictly related to graphical issues, there are domains that are intrinsically hard to manipulate via discrete widgets such as those provided by general-purpose GUI libraries. Such libraries were developed to serve well-formalized discrete domains like business management, data base manipulation and so on. The scientific domain, for example, is one such 'difficult' domain. In Figure 3.7 we show a fictitious viewer for physical data related to Oceanography. The data, rendered with an equidistant, cylindrical equatorial projection, is not a mere image, but something that can be manipulated, queried and processed (although in this prototype is a mere bitmap). There is a database behind such GUIs, but the best way to structure the interface is often radically different from those such as the underground railway network shown in Figure 3.5.

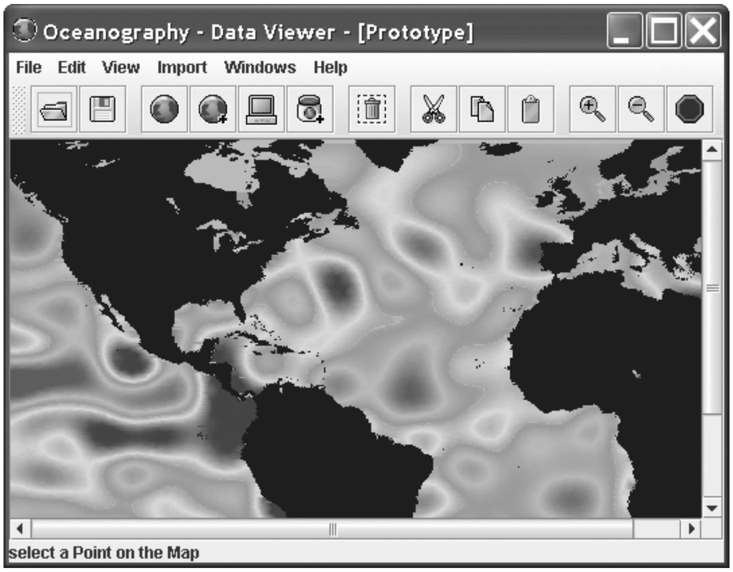

Oceanography - Data Viewer - [Prototype]
File
Edit
View
Import
Windows
Help
select a Point on the Map

Figure 3.7 An example of a scientific data viewer and editor

Here it would be unthinkable to present the user with a set of grid views extracted from our database. It would be literally like an ocean of numbers, impossible to read, not to mention manipulate properly.

In these cases an ad-hoc development route is unavoidable. The only alternative would be to reduce the quality of the GUI by lowering interaction. In the application in Figure 3.7, for example, instead of providing complex commands to manipulate the scientific data, one can imagine an almost batch-like interaction style in which the user is prompted with a form that defines all the details of the required data manipulation, together with a **Submit** button that dismiss the dialog, launches the command and then displays an image that is the user cannot manipulate. While this interaction style might be acceptable in some cases, it can be intolerable in other contexts.

Cost-driven design is a form of system-centered design, and as such appears to conflict with user-centered design, as described in Chapter 2. This conflict is only apparent, though. Whenever in doubt, user requirements (for usability) should prevail over system-centric considerations. If development costs are not taken into consideration, even a good GUI design can be implemented poorly, producing applications that are de-facto less usable.

3.3 Exploring the design space for a point chooser

In this section we examine some diverse examples of practical GUI design. We introduce practical design using the case of a simple visual component that illustrates the many possibilities the designer would consider as they relate to the chosen technology (which in the example is J2SE Swing).

Let's suppose we have to design the user interface of a component for selecting points on the earth surface, which itself forms part of a wider GUI. We will refer to this as a *Geopoint chooser*. We won't go into the details of the GUI design process here, but we will explore the design space in order to discuss few of the many design choices available, even for such a limited problem.

Of course we know that there is no absolute 'good' design. GUI design depends on many factors that include tasks[1], users, and cost. We deliberately do not commit to a fixed scenario, so that we have the freedom to discuss some more of the practical subtleties GUI designers often face in their work.

The functional requirements for our example GUI component are really simple. The related use case diagram is shown in Figure 3.8.

Use case diagrams can also be employed for describing the details of specific parts of GUIs, such as this example. See for example the 'Complete Selection' use case above, which can be further detailed using 'Commit' or' Cancel' selection use cases independently of the given component used to implement a GUI component.

Having seen the functional requirements for our component, it's time to focus on the GUI design itself. For simplicity we focus on the selection use case only. There are established guidelines for implementing a selection completion use case, for

1. By *task* we mean one of the tasks performed by the user, coinciding with the same term used during task analysis (see Chapter 2, page 45)

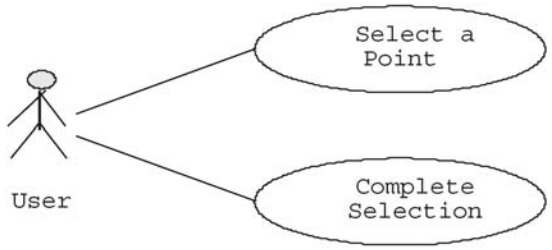


Figure 3.8 The use case diagram for a geographic point chooser component

example by placing **OK** and **Cancel** buttons at the bottom of a model dialog, as we will see later in this

chapter. Here we will focus on the design of the selection area only, the use of which is illustrated in the simple paper sketch[2] in Figure 3.9.

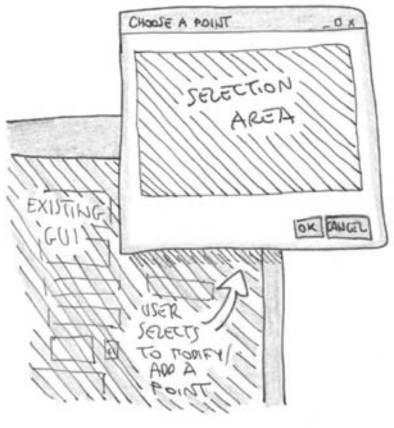


Figure 3.9 The intended use of the Geopoint chooser

In the following subsections we show a number of possible designs for these requirements, and introduce them in relation to their underlying implementation. Our purpose is to illustrate a number of design details that appear only in practice.

We begin with designs that are implemented by means of standard components.

2. Chapter 5 discusses paper prototypes in more detail.

Standard designs

The first and most obvious idea is to rely on existing cultural conventions, such as latitude and longitude, for selecting a point on the Earth's surface. We could adopt the form-filling interaction style, as shown in Figure 3.10. Fortunately measurements expressed in latitude and longitude are widely accepted around the world, so we don't have to worry about localization subtleties.

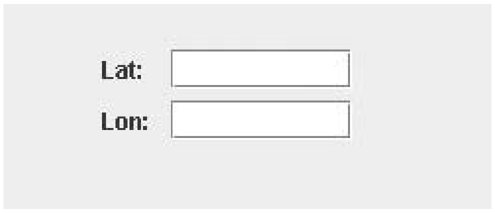


Figure 3.10 Using a form-filling interaction style

We can refine our simple design by leveraging the usual form-filling techniques, for example by providing a *history* facility – a drop-down window showing past input values – or by separating latitude and longitude values into degrees and minutes, as in Figure 3.11.

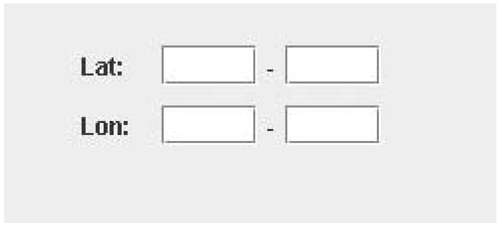


Figure 3.11 Using a form-filling interaction style – refined

Depending on whether latitude or longitude can be accessed separately in the remainder of the GUI, it could be useful to address this concept explicitly, for example by providing an icon for value. This makes sense only if such icons are used elsewhere in the GUI and with the same meaning, otherwise it is just useless visual 'noise.' Apart from additional visual clue, which might be difficult to enforce, for example in third-party components to whose GUI design we do not have access, another useful enhancement might be to use spinners for selecting input values, as shown in Figure 3.12.

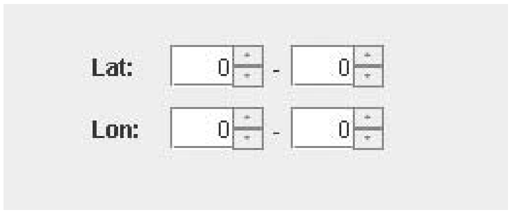


Figure 3.12 Using a form filling interaction style – even more refined

Of course we could choose to employ this sort of standard component because of the nature of our application domain, for example the required precision for latitude and longitude.

Without a proper task analysis, a design remains incomplete, because there are so many possible twists that an effective general design is often not viable.

Adopting the form-filling interaction style in our design is very cheap, as it requires only standard GUI components. It can work well if (in this case) our GUI tasks use latitude and longitude and our end-users are accustomed to such measurements. If, alternatively, we were designing a component for choosing a time zone in a non-technical GUI, or for setting locale data such as choosing a home country, clearly this design wouldn't work well.

We can use these designs to help make a point about a common situation that arises when using sophisticated GUI toolkits like Swing. Consider the following situation, which gives rise to an unexpected additional implementation cost, even for the cheap design in Figure 3.10–3.12. Suppose we are developing a GUI for a client that employs our Geopoint chooser as shown in Figure 3.10 and, somewhere else in the GUI, a date input field. Such a date input field is implemented using Swing widgets that provide extra behavior and formatting for dates. One of the features provided automatically by Swing is the ability to use arrow keys to increase days, months, and years directly in the date field (mainly through the `JFormattedTextField` class). This feature is appreciated by users, so they find it odd that the same handy mechanism is not available in our Geopoint chooser.

Thus we have an unforeseen problem of consistency with the rest of the GUI, because of the automatic facilities provided by the standard library we are using. If we decide to fill this gap, our design shifts from a standard to a custom one, as we have now to implement the behavior in our Geopoint's **Lat/Lon** text field that is available in date fields. This underlines an important point: in practice, effective GUI design is always an iterative process, no matter how simple the design may appear at first.

A geopolitical design

Even in its simplest form, the previous design is not suited to some tasks. The design shown in Figure 3.13 illustrates a different approach that uses the menu selection interaction style – which is by no means limited to commands menus – for selecting an area. This can be thought of as a point with a degree of tolerance. The advantage here is that such an area is identified by geopolitical coordinates, such as continent name, region and so forth. Depending on the user population or the nature of the task, this could be the most usable solution.

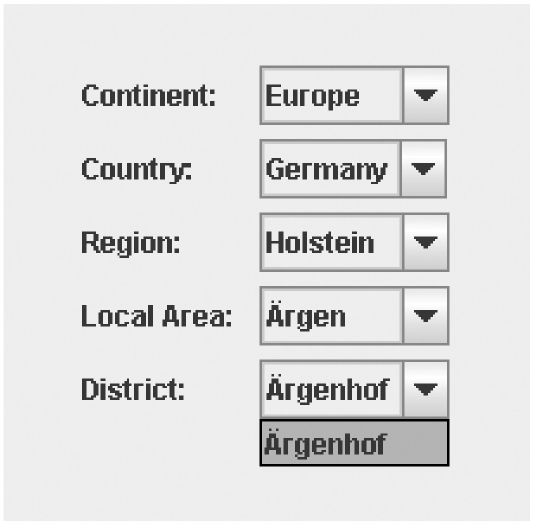


Figure 3.13 A geopolitical chooser

The design in Figure 3.13 highlights an interesting point about the low cost of building a GUI using standard components. Such cost savings relate only to the GUI's appearance, not to the remainder of the implementation. Even if we use standard components, the data needed to make this design work (countries, regions, counties, etc.) could be expensive to gather, offsetting or cancelling the cost savings.

Nevertheless, this solution is very robust: as long as the combo boxes automatically populate themselves with valid data, users cannot choose an impossible value. This is often referred to as the *power of constraints*. Well-designed GUIs should be like that – by careful design of their interaction rules, they should reduce to a minimum the possible sources of errors at the outset.

A cryptic design

An important aspect that we have not yet had the chance to discuss in detail is the importance of operational feedback. To illustrate this, we consider an absurd design choice: form filling-based selection without operational feedback.

We are usually unaware of its importance, but while we are typing into a field on a form, we actually watch what we are doing. This is a basic form of operational feedback, like seeing the mouse pointer move while we move the mouse to select a point in a GUI that employs direct manipulation. To illustrate the importance of such feedback, try out the interface in Figure 3.14, in which password fields are used for latitude and longitude input!

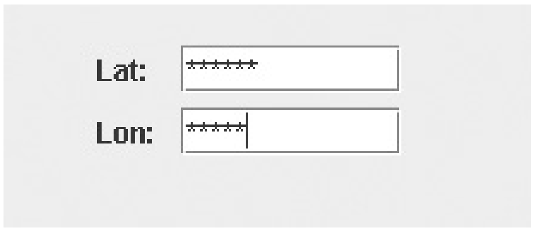


Figure 3.14 Using a cryptic form-filling interaction style

Ad-hoc designs

The simplest way to indicate a point on a map is by pointing at it with the mouse. Such as design is shown in Figure 3.15. From a technical viewpoint, this design choice needs the use of ad-hoc components, and is therefore usually more expensive to develop than those that use toolbox components.

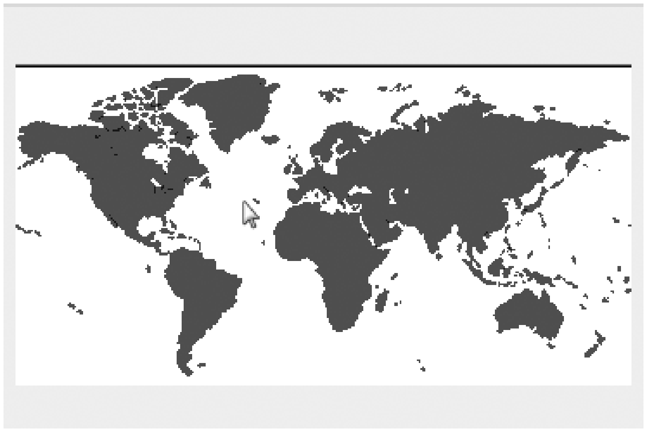


Figure 3.15 A direct manipulation Geopoint chooser

As natural and user-friendly as it may seems, there are cases in which this type of design is not the best. In highly repetitive scenarios, for example, in which users need to input many points routinely, extra consideration should be given to the use of the keyboard as the main input device to speed up selection. The design in Figure 3.15 does not provide such a facility, and this can be a serious shortcoming in such cases.

Let us now consider the details of the design of a direct manipulation Geopoint chooser.

The direct manipulation design in detail

Given the nature of the application, we assume that our users are non-occasional and skilled experts in the domain of interest.

Configuration settings such as the precision of mouse hovering and other preferences are kept separate in another dialog. We won't discuss such configuration issues here.

The direct manipulation interaction employed in the design in Figure 3.15 seems a perfect choice. However, it may not be obvious to a novice user – there is nothing into the GUI that suggests the point-and-click behavior.

Instead of adding a label or a tooltip to signal the intended interaction, and so risking annoying regular users, we chose to change the cursor shape and add a label that indicates the geographical point indicates by the mouse focus, shown on the left bottom of the map, together with a label that shown any point already selected, as seen in Figure 3.16. These are all discreet hints that 'invite' the user to click on the map to see what happens. There is no need to overload the design with explicit signals – in this way both first-time and regular users are well served.

When the user selects a point, it is signaled by an 'X' on the map and represented numerically at the bottom-right of the map. On the left- hand side of the screen the point corresponding to the current mouse position is shown.

We place the selected and the current point in the status bar, following the Java look and feel design guidelines (introduced later in this chapter), although further usability testing should be done to check that our choice is not confusing to users.

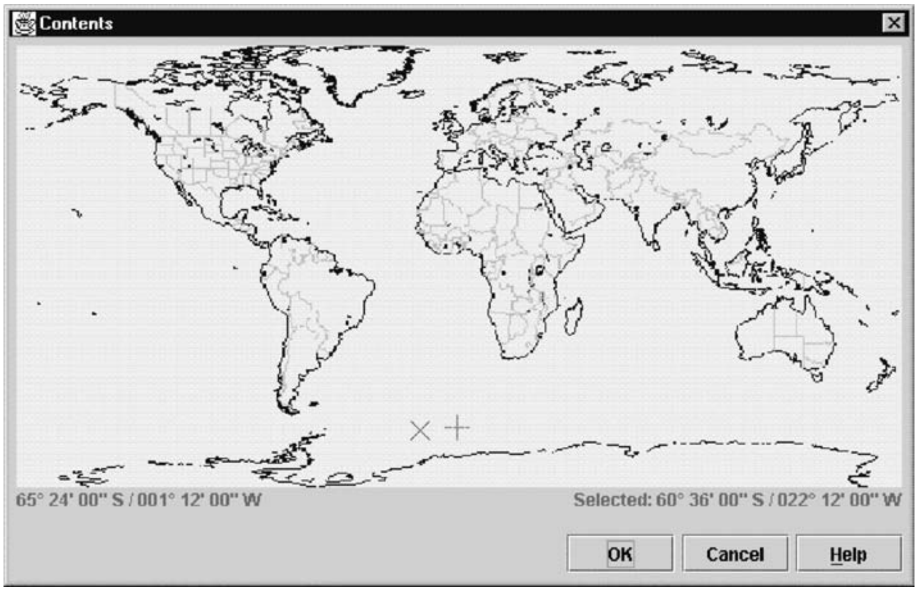

Contents
65° 24' 00" S / 001° 12' 00" W
Selected: 60° 36' 00" S / 022° 12' 00" W
OK
Cancel
Help

Figure 3.16 An implementation of a Geopoint chooser visual component

When the user has chosen a point on the map, they can dismiss the chooser, so committing the operation by clicking the **OK** button, or just cancel the selection operation by using the **Cancel** button.

Allowing editing of the currently selected point helps fine-tune inputs. Figure 3.17 shows a typical interaction with an enhanced version of the chooser.

This version allows users to edit latitude and longitude values directly, and so refine a chosen point more easily, to any level of precision.

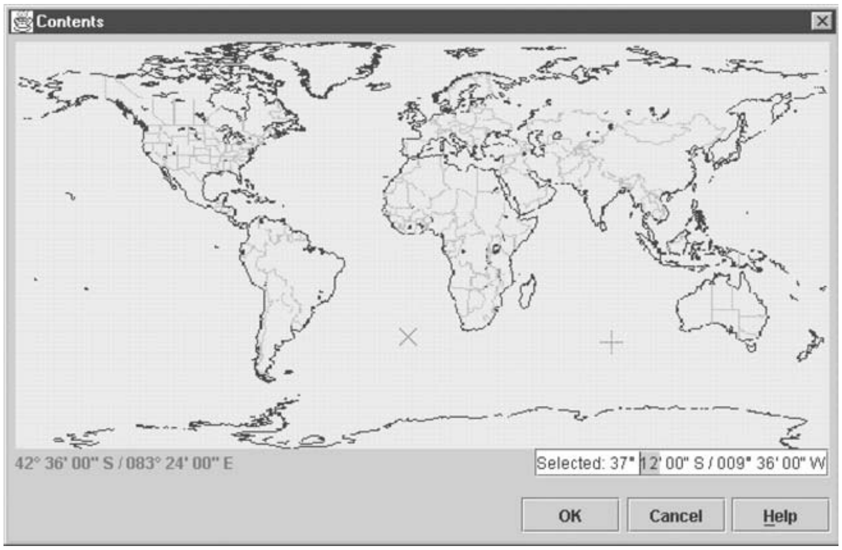

Contents
42° 36' 00" S / 083° 24' 00" E
Selected: 37° 12' 00" S / 009° 36' 00" W
OK
Cancel
Help

Figure 3.17 An enhanced version of the Geopoint chooser

The design in Figure 3.17 illustrates a subtlety regarding the commit behavior of the editable field for the selected point. The designers have to decide how the editing in the field is going to be committed and so change the location of the 'X' mark on the screen. One possibility is to accept the editing as soon as the user types valid numbers into the field (this is called *immediate* mode). This has the unpleasant side-effect of making the 'X' mark scatter all over the map while users are typing the digits of a coordinate. The other option is to commit the value after a special 'completion event' is performed by the user, such as pressing **Enter** or pressing a button (this is called *deferred commit* mode). A third possibility is to delay the commit for a specific time after the last user keystroke (perhaps a few seconds), giving users the time to fully input the value. This option (*delayed immediate* mode) and deferred mode are important when the commit cost could be high, for example to send data to a remote server.

Quality assurance testers love to fiddle with such subtleties. What if the chosen commit mode is delayed immediate – say after 2 seconds after last user keystroke in the field – and as soon as the user types a digit, they quickly close the dialog by pressing the **OK** button? The new value does not have time to be committed, and they developers can find themselves dealing with a new and unpleasant bug.

A further evolution of this design could involve spinners – using the `JSpinner` Swing component – instead of free text. Figure 3.18 shows such a solution.

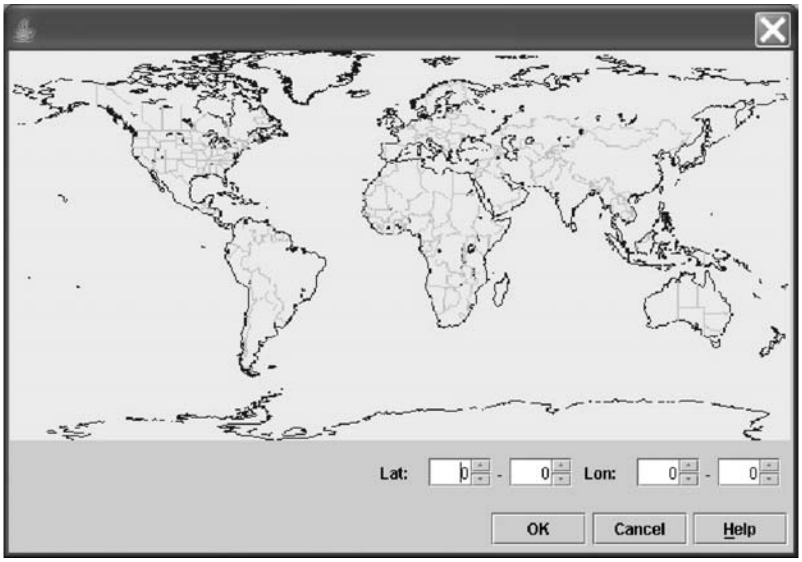


Figure 3.18 A Geopoint chooser that employs spinners

The design shown in Figure 3.18, although visually loading the chooser window a little, allows for a finer user data input. The use of spinners is also self-explanatory – users understand their actual purpose easily, so that they can use this additional control whenever a fine, but constrained, input is needed.

Figure 3.19 shows another version of our design, in which users can specify the current geographical projection adopted at the top right. Whenever the projection is changed, the underlying map and the selected point change accordingly.

Changing the map projection is an example of a configuration item that can become a part of the operational GUI, depending on the situation. If this feature is used by unskilled users, it might be distracting or even confusing. This is a common dilemma, where the user population is not easily predictable at design time[3].

3. As one can imagine, designing 'catch-all' visual components isn't an easy job.

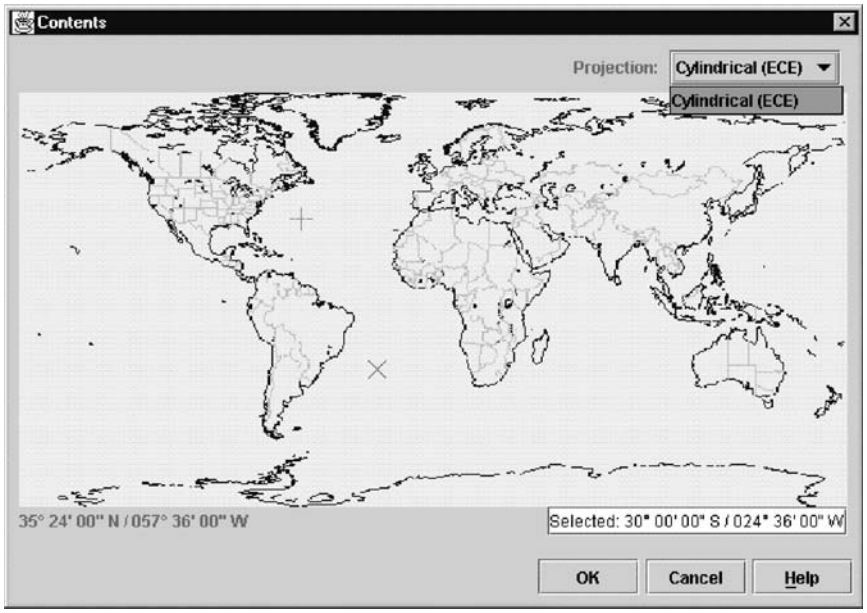

Contents
Projection: Cylindrical (ECE)
Cylindrical (ECE)
35° 24' 00" N / 057° 36' 00" W
Selected: 30° 00' 00" S / 024° 36' 00" W
OK
Cancel
Help

Figure 3.19 Interacting with the chooser

Finally, the class diagram related to the version shown in Figure 3.16 is shown in Figure 3.20. The implementation code for this Geopoint chooser is available on the book's Web site – see the `GeoPoint` and related classes.

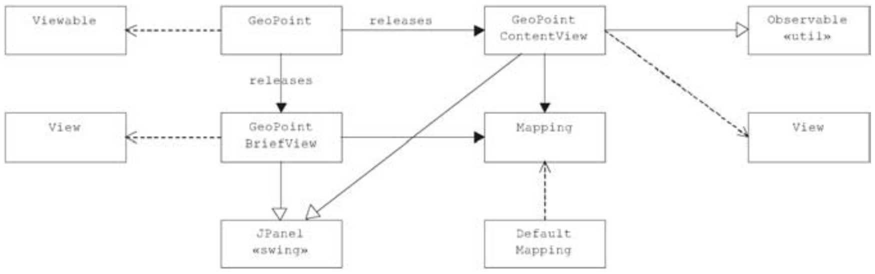


Figure 3.20 The Geopoint chooser class organization

Next we look at alternative designs that employ combinations of design approaches.

Mixed designs

As seen in the previous section, it is possible to combine direct manipulation and the use of standard components in a GUI design. These are the most expensive designs, due to the cost of building the different representations, plus the extra cost of establishing the coordination between the two. The use of such an approach should be thought through carefully, because it can actually produce more cluttered – and so less usable – designs. This is a classic phenomenon known as *feature creep*: designers feel somehow more reassured by adding extra functionalities to the GUI in a vague attempt to make it more usable.

An obvious solution for increasing the ease of use of our Geopoint chooser design is to employ two different representations of the same data simultaneously. Choosing the two representations carefully can lead to larger usable selection areas, for example one quicker to use, but less precise, together with a slower but more accurate one.

A set of different designs are possible. For example we could employ sliders for selecting the point indirectly on the map, as shown in the design in Figure 3.21.

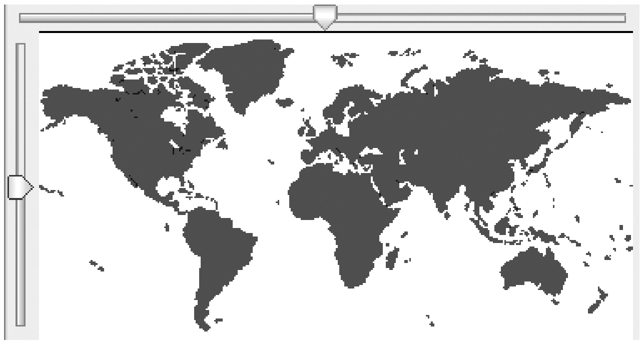


Figure 3.21 Indirect manipulation

This solution has a flaw. Depending on the projection used for the map, the sliders could indicate meaningless measurements (the geographic projection used in Figure 3.21 is only a mock-up).

One possible solution is to decouple the sliders from the visual representation of the map, as shown in Figure 3.22. This new solution has the advantage of combining the two required parameters (which may not necessarily be latitude and longitude) with the powerful visual feedback given by the chosen point indication on the map. More importantly, it does not depend on a specific map projection.

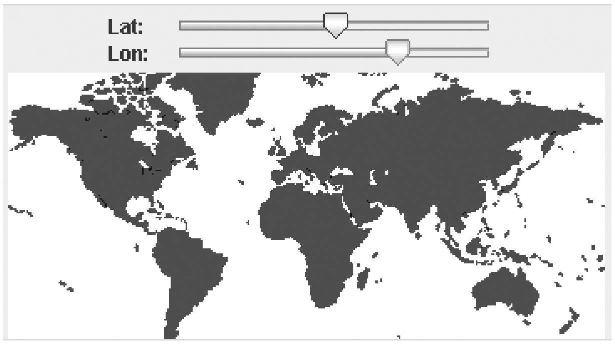


Figure 3.22 Another attempt

Like the design illustrated in Figure 3.21, this design imposes a degree of coordination between the two representation of the same data: the two sliders above being the indirect representation, and the map in the center being the direct representation of a point on the Earth's surface. When the user changes one of the sliders, the point in the map changes accordingly. This is an example of the concept of different *views* – that is, different representations – of the same data. We will see in the second part of the book how Java GUIs, by leveraging OO design pattern technology, can implement such constraints in complex applications.

Combining two designs in one

In some case is not possible to accommodate both expert and novice users with the same design without hampering one or both of the groups. In these cases one solution is to provide two slightly different versions of the same UI in combination, providing the simpler path for novice users and a more elaborate but powerful one for expert users.

Returning to our Geopoint chooser, suppose expert users want to define the information about a point on the earth surface in a more articulate way. To avoiding cluttering the UI for novice users, who are happy with point-and-click interaction, we can devise a design that conceals more complex data input in a separate area. The design in Figure 3.23 shows this solution.

We can draw a number of lessons from the design in Figure 3.23. When providing such a two-way UI differentiated by user skill, it is always a good idea to favor novices over experts, for example by starting up the GUI with the default view for novice users, or by providing simpler interactions for them. This is not always

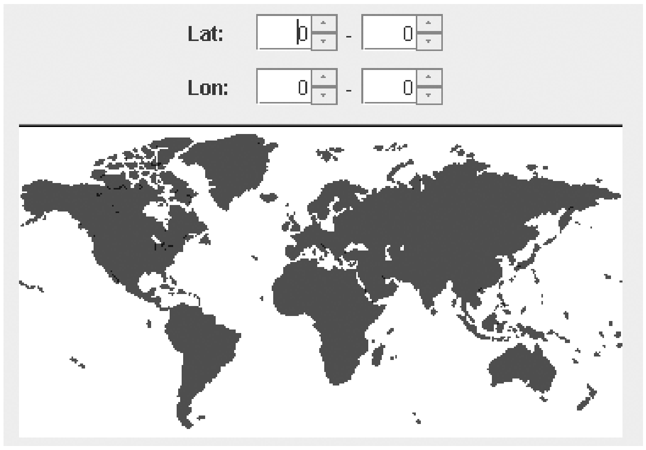


Figure 3.23 A two-way UI differentiated by user skill

possible, though. Sometime the GUI needs to be engineered for expert users over novices, for example to optimize user's interaction speed.

From a visual viewpoint, the 'expert' form-based view could be switched on and off in a number of ways, for example by means of a button, or by adding a tab pane with two tabs, one for the map and the other for the numerical representation. Two tabs would avoid confusing novice users, who can use the less precise, direct manipulation map and ignore the more elaborate form-based input area. But with two tabs, the UI loses the very useful operational feedback of seeing the point selected with spinners directly on the map.

Conclusions

Even in these simple examples we find many design choices that complicate our GUI design process. We can see how categorizing components based on their development cost can sometime be misleading, because it doesn't take account of non-GUI costs, such as data collection, such as that needed to make the design in Figure 3.13 work.

One aspect that recurs in each design we have examined is the phenomenon of *feature creep*. The more designers work on a design, the more they are tempted to add extra functionality, overloading the design beyond what is needed and potentially making it less usable.

In the next section we enter the world of user interface guidelines, introducing the official design guidelines for GUIs built with the J2SE Swing toolkit.

3.4 Design guidelines for the Java platform

Fortunately it is not necessary to start from general principles when designing a new GUI for a given computing platform[4]. The platform provides many conceptual and coding constraints that help us to build a professional GUI economically. However, many developers aren't aware of such guiding principles. This can be seen in many GUIs, in which the designer didn't understood the principles behind the visual components employed, or even misused them altogether.

Using a sophisticated and powerful GUI toolkit doesn't make one immune from gross UI design errors, as shown in Figure 3.24.

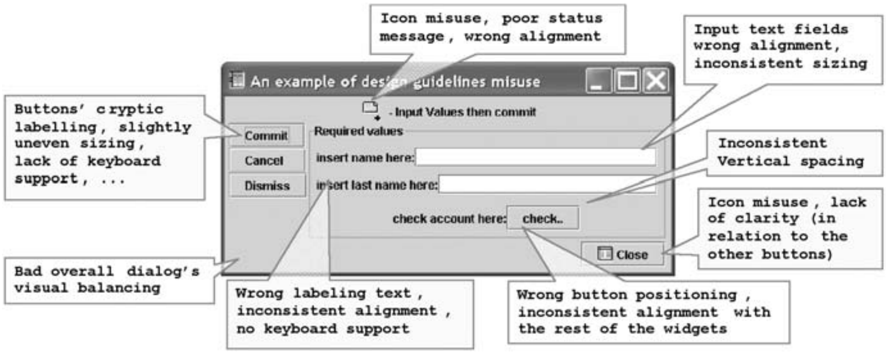

Icon misuse, poor status message, wrong alignment
Input text fields wrong alignment, inconsistent sizing
An example of design guidelines misuse
Buttons' cryptic labelling, slightly uneven sizing, lack of keyboard support, ...
- Input Values then commit
Commit
Cancel
Dismiss
Required values
insert name here:
insert last name here:
check account here:
check..
Close
Inconsistent Vertical spacing
Icon misuse, lack of clarity (in relation to the other buttons)
Bad overall dialog's visual balancing
Wrong labeling text, inconsistent alignment, no keyboard support
Wrong button positioning, inconsistent alignment with the rest of the widgets

Figure 3.24 A badly-design form

What is missing from the figure is a coherent, systematic organization of the layout and intended user interaction. Such an organization is required to ensure UI consistency – users expect dialogs, panels and other GUI parts to have the same mechanisms and conventions, possibly sharing those of similar products – and ensuring the required levels of usability.

Introduction to the guidelines

Professional UI designs are the result of many contributions, ranging from the UI toolkit in use to the general UI design guidelines available for that platform, and

4. Java is not only a mere development environment in the traditional sense, in that a Java runtime is also deployed with the execution code, thus providing a sort of 'Java platform' in which a minimum set of services (constantly growing with each release) are available for all Java applications. At the same time, the Java platform is not always totally independent of the underling native OS.

also comprising the general international standards and guidelines for usability, design best practices and so on. In Figure 3.25 shows some of the contributions to the final design of a simple J2ME MIDP form for a handheld device.

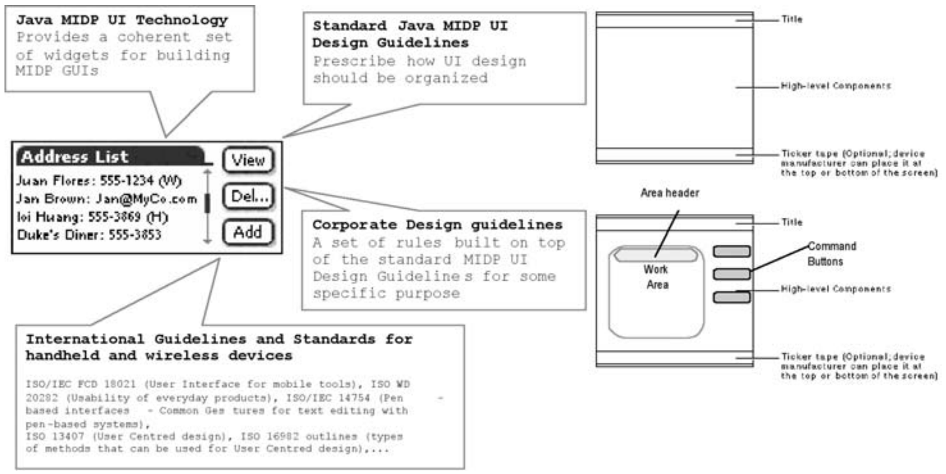

Java MIDP UI Technology
Provides a coherent set of widgets for building MIDP GUIs
Standard Java MIDP UI Design Guidelines
Prescribe how UI design should be organized
Address List
Juan Flores: 555-1234 (W)
Jan Brown: Jan@MyCo.com
Ioi Huang: 555-3869 (H)
Duke's Diner: 555-3853
View
Del...
Add
Corporate Design guidelines
A set of rules built on top of the standard MIDP UI Design Guideline s for some specific purpose
International Guidelines and Standards for handheld and wireless devices
ISO/IEC FCD 18021 (User Interface for mobile tools), ISO WD 20282 (Usability of everyday products), ISO/IEC 14754 (Pen - based interfaces - Common Ges tures for text editing with pen-based systems),
ISO 13407 (User Centred design), ISO 16982 outlines (types of methods that can be used for User Centred design),....
Title
High-level Components
Ticker tape (Optional; device manufacturer can place it at the top or bottom of the screen)
Area header
Title
Command Buttons
Work Area
High-level Components
Ticker tape (Optional; device manufacturer can place it at the top or bottom of the screen)

Figure 3.25 Every good design is the final result of many guidelines

Note that in general UI design guidelines are built on top of other more general ones, to provide a complex and coherent set of UI design directions – that is, guidelines that don't contradict other more general guidelines. This can be seen in Figure 3.25 above, in which the corporate UI design guidelines restrict the standard general design guidelines for MIDP GUIs. The presentation technology, including widget toolkits, is also built following standard guidelines.

Guidelines provided by the platform vendor are not exhaustive, and organizations can expand them to meet their needs, to add extra features, or to provide a 'branded' look and feel. One could provide further design guidance for a family of applications that in turn specializes corporate design guidelines. Figure 3.26 shows the general layering of user interface design constraints for any graphical interactive platform.

The layering metaphor in Figure 3.26 is used to convey the idea of a set of *harmonized* guidelines, which, when put together, form a coherent language for building graphical user interfaces.

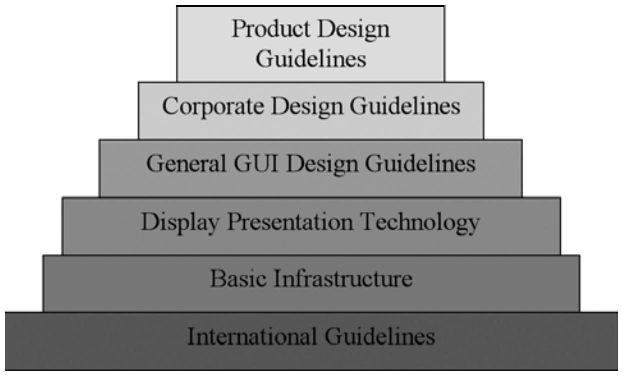

Product Design Guidelines
Corporate Design Guidelines
General GUI Design Guidelines
Display Presentation Technology
Basic Infrastructure
International Guidelines

Figure 3.26 Stacking up design guidelines in general

Starting from the bottom layer:

- Basic concepts, pointing devices and the remaining items that make up the 'plumbing' of modern GUIs are based on broader and more general guidelines such as:
  - ISO 9241 (ergonomic requirements for office work with visual display terminals)
  - ISO 20282 (usability of everyday products)
  - IEC TR 61997 (guidelines for the user interfaces in multimedia equipment for general purpose use)
  - ISO/IEC 10741-1 (dialog interaction – cursor control for text editing)
  - ISO/IEC 11581 (icon symbols and functions)

  while, for J2ME other standards apply:
  - ISO/IEC 14754 (pen-based interfaces – common gestures for text editing with pen-based systems)
  - ISO/IEC 18021 (information technology – user interface for mobile tools)[5]
- Above this is the basic infrastructure for interactive GUI features provided by the platform. Such an infrastructure in modern multipurpose software environments is usually organized around the concept of component-based GUIs. These are graphical items (also called *components* or *widgets*) that can be assembled to create a large number of different GUIs. Some specialized

5. There are many hardware standards too: for computer displays, keyboards, etc. For a comprehensive list of the various usability and HCI standards, see: http://www.hostserver150.com/usabilit/tools/r_international.htm.

platforms (or those with limited hardware, such as hand-held devices) may use other approaches to model the basic infrastructure of their user interface.

- The display presentation technology is built using a conceptual UI architecture and a set of basic guidelines and standards. This software allows developers to build UIs by means of specialized APIs.
- At a higher abstraction level, presentation technology alone is not enough to guarantee effective and usable UIs. A set of UI design guidelines and best practices needs to be taken into account during the UI design process. Such a set of guidelines is strictly dependent on the underling presentation technology: for example, a set of voice interfaces design guidelines is meaningless for graphical-only presentation technologies. An example of a UI design guideline for a GUI could be 'command buttons should all be the same size.' These guidelines are usually provided by the same companies that develop the related display presentation technology, or by independent standard bodies.
- Corporate design guidelines are built on top of the standard UI design guidelines by private organizations to provide a higher level of consistency for the software developed in or for the organization, and to enable other benefits such as support for a proprietary toolkit, product documentation purposes, quality assurance, UI cost estimation, and so on.
- Above corporate UI design guidelines could be further specification for single products, perhaps for providing special UI features, branding, better user targeting, and so on. Imagine for example the GUI of a software music player, as opposed to the GUI of an e-mail client built by the same company. This and the corporate level of guidelines are usually owned by organizations and not available for public use.

J2SE user interface design guidelines

The same layering of design guidelines shown in Figure 3.39 also exists for Java 2 standard edition (J2SE) too. Figure 3.27 shows how the final design of a simple J2SE Swing GUI is influenced by the different UI design guidelines layers introduced in the previous section.

The various design guidelines are compounded, enforced by the GUI technology, here Swing, to create the final result.

This layering is illustrated in Figure 3.28. The pyramid of constraints and guidelines for the design of GUIs stands on top of the same international standards mentioned above – the hierarchies in Figure 3.26 and Figure 3.28 share the same lowest level. The architects of Java adopted a common approach based on components for modeling GUIs, indicated by the *Basic Infrastructure* layer in Figure 3.26.

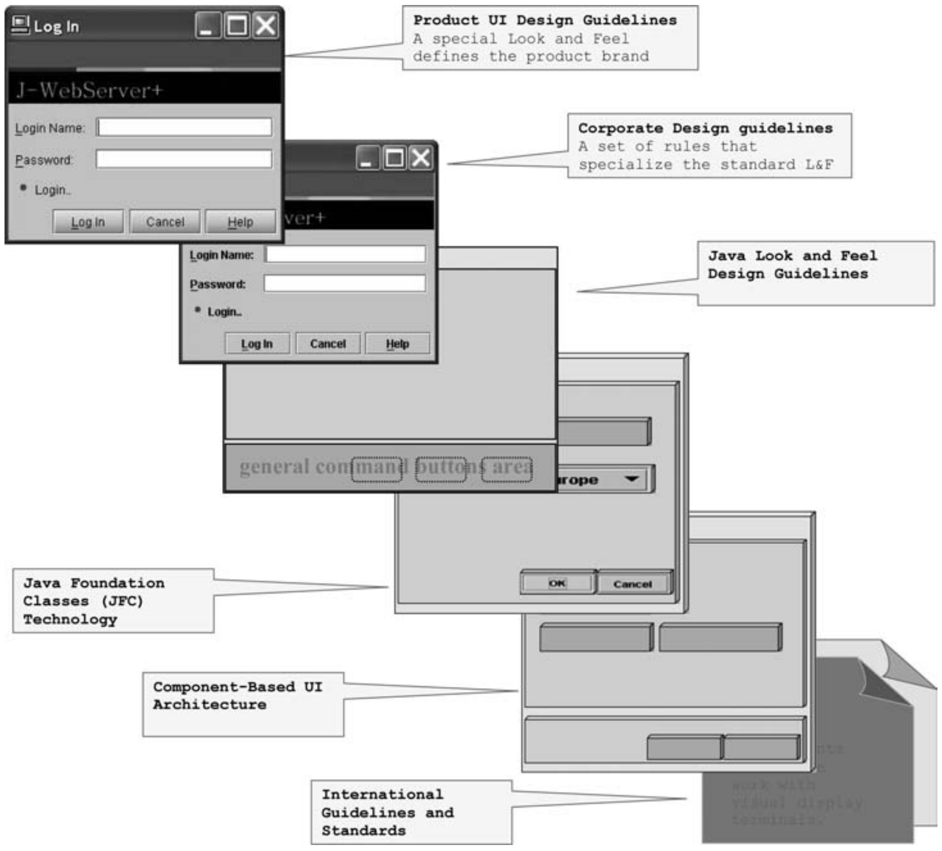

Log In
J-WebServer+
Login Name:
Password:
Login..
Log In
Cancel
Help
Product UI Design Guidelines
A special Look and Feel defines the product brand
Corporate Design guidelines
A set of rules that specialize the standard L&F
Java Look and Feel Design Guidelines
general command buttons area
OK
Cancel
Java Foundation Classes (JFC) Technology
Component-Based UI Architecture
International Guidelines and Standards

Figure 3.27 The final design of a J2SE GUI

The idea that a J2SE GUI is inherently composed of elementary, reusable components impacts both the design and implementation of GUIs. Such components can be visual objects, such as a combo box, more abstract ones, such as a layout manager, or non-GUI objects, such as the data model behind a list[6]. Building on top of this conceptual model, we are offered a number of visual components that can be combined to build GUIs.

Sun provides two, in part overlapping, toolkits created around this component approach: Swing and AWT, plus a number of auxiliary libraries such as Java2D

6. Values displayed in widgets are usually stored in runtime memory structures known as *data models*. When users modify the value in the widget through the UI the changes are transferred to the related data model.

and JavaHelp. These are the more popular GUI toolkits for J2SE, but there are others. At a higher level of abstraction, Sun also supplies a set of design criteria and guidelines for harmoniously composing the building blocks provided in these libraries. Finally, developers are free to add their own design constraints and guidelines by building on top of other guidelines. Figure 3.28 shows the layering of user interface design constraints for J2SE platform.

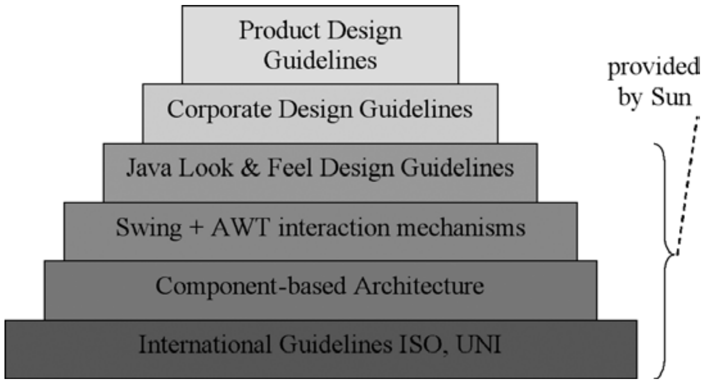
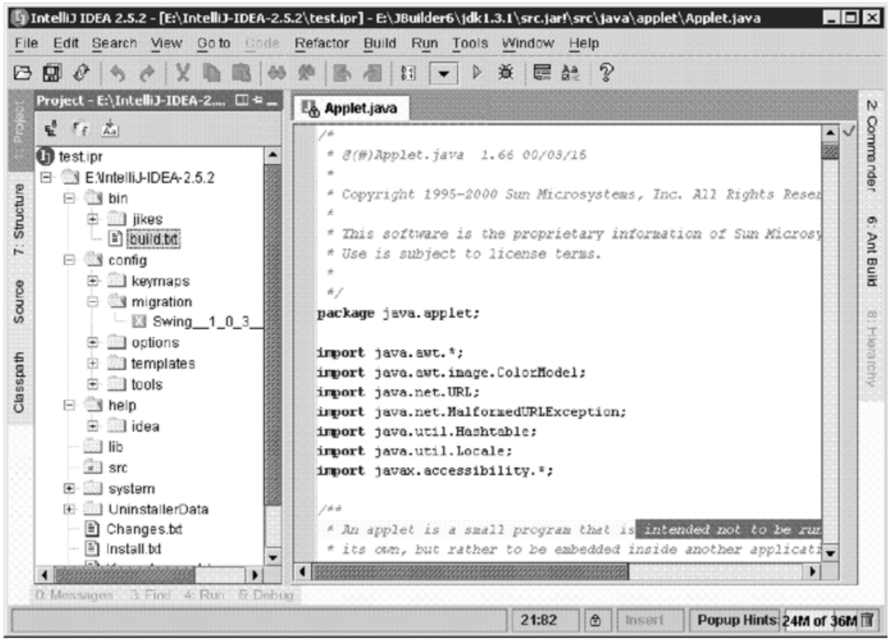

Product Design Guidelines
Corporate Design Guidelines
Java Look & Feel Design Guidelines
Swing + AWT interaction mechanisms
Component-based Architecture
International Guidelines ISO, UNI
provided by Sun

Figure 3.28 Stacking up design guidelines for J2SE

Any GUI toolkit include abstractions and mechanisms related to the use of the widgets it offers. Such interaction mechanisms may be closely linked to higher-level design guidelines. This is the case with the Java look and feel design guidelines and the underlying Swing library – in fact, the Java look and feel design guidelines have been designed specifically for the Swing toolkit. For example, the Java look and feel provides detailed guidelines for changing the visual appearance of the whole GUI at runtime, and such a feature is technically available only for Swing-based GUIs.

By taking advantage of corporate design guidelines, it is possible to create new GUI styles that highlight the product's identity, or that are specialized for some particular case. Figure 3.29 shows an example of such a custom style, built on top of the Java look and feel, used for the JetBrains IDEA[7] integrated development environment.

7. IntelliJ IDEA is a trademark of IntelliJ Corp.



```
/*
 * @(#)Applet.java  1.66 00/03/15
 *
 * Copyright 1995-2000 Sun Microsystems, Inc. All Rights Reser
 *
 * This software is the proprietary information of Sun Microsy
 * Use is subject to license terms.
 *
 */
package java.applet;

import java.awt.*;
import java.awt.image.ColorModel;
import java.net.URL;
import java.net.MalformedURLException;
import java.util.Hashtable;
import java.util.Locale;
import javax.accessibility.*;

/**
 * An applet is a small program that is intended not to be run
 * its own, but rather to be embedded inside another applicati
```

Figure 3.29 The IntelliJ IDEA GUI

As the figure shows, the designers had to solve various GUI-related problems, and resorted to adopting a specialized version of the Java look and feel style. Many of the conventions used in the standard Java look and feel were maintained, but new visual components were provided.

It is important to point out that Swing, although the most popular, is not the only toolkit available to GUI designers using Java. Developers can create their own toolkits that build on top of standard libraries, or even substitute them altogether, as IBM did for Eclipse[8]. On platforms such as Eclipse, its SWT library still offers a component-based approach to GUI building, but also provides an alternative set of widgets to developers. The design guidelines also differ from those proposed by Sun. The Eclipse design constraints are shown in Figure 3.30.

SWT design guidelines are different than Swing guidelines, as can be seen from the example GUI developed for Eclipse shown in Figure 3.31. Notice, for example, the status/message bar at the top of the dialog just below **Java Settings**. The SWT library is described in Chapter 11.

8. See Chapter 11.

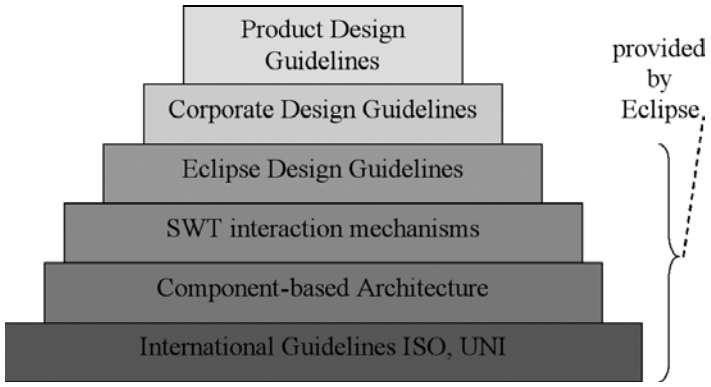
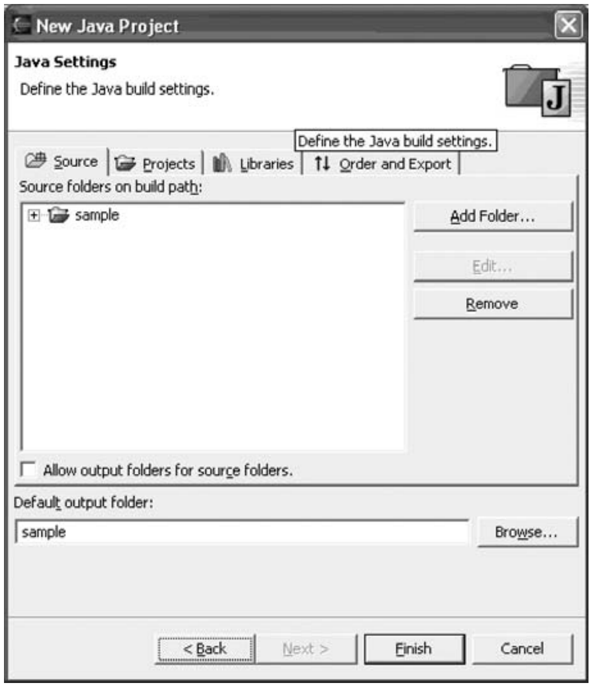

Product Design Guidelines
Corporate Design Guidelines
Eclipse Design Guidelines
SWT interaction mechanisms
Component-based Architecture
International Guidelines ISO, UNI
provided by Eclipse

Figure 3.30 Stacking up design guidelines for the Eclipse platform


New Java Project

Java Settings

Define the Java build settings.

Source folders on build path:

sample

Allow output folders for source folders.

Default output folder:

sample

Figure 3.31 An Eclipse standard GUI

The standard Java Look And Feel design guidelines provided by Sun is not the only such set of guidelines available. The layering shown in Figure 3.26 on page 102 can be highly customized, and each guideline layer can be replaced with others. This is a powerful feature in the hands of seasoned designers, as it is expensive and time-consuming to create an original yet professional set of design guidelines. An easier and safer way is to build on top of existing guidelines. Fortunately, the Java look and feel provided by Sun is an effective set of design guidelines that fits J2SE's technical constraints and allows easily for some customization.

In contrast to the look and feel of single components, the style (the systematic layout of widgets in windows and the set of interaction patterns recurring in the GUI) cannot be strictly enforced by a class framework no matter how clever it is devised, and it should be put into practice explicitly by designers and developers in their applications.

3.5 The Java look and feel design guidelines

Adhering to a particular set of design guidelines is key to the creation a professional GUI on any platform, and on Java in particular. But Java software can be run on many platforms. This raises the issue of which design guideline to adopt. While the visual appearance of the GUI can be changed easily – as long as the Swing library is used – the underlying window layouts, interaction mechanisms and other important aspects of the GUI cannot. It would be quite expensive to provide a single GUI that can look and behave like a Windows application on Windows and like an Aqua application running on an Apple Macintosh. And even this wouldn't really solve the problem, because Java applications are different than native ones, no matter how cleverly you code them.

To address this problem, Sun proposed a standard set of design guidelines specific to the J2SE platform. If your application is compliant with these guidelines, it will look and behave (almost) the same on all the platforms Java on which can run. Even if you are not planning to exploit the multi-platform capabilities of Java, you will be able to create professional-looking GUIs with little effort by adopting the Java look and feel design guidelines.

Our aim here is to provide a general introduction to the Java look and feel design guidelines, and for J2SE in particular, rather than provide a thorough exposition of topic such as how to space items in a window, how to handle raster graphics on different platforms, and so on. Readers interested in the detail can refer the official guides provided by Sun[9], *Java L&F Design Guidelines 2001, Advanced Java L&F Design Guidelines 2001*.

Some definitions

First, there is a small terminological twist related to two different meanings of term 'look and feel.' In Java code, 'look and feel' refers strictly to the visual appearance of GUI components, and is also known as 'Metal' in the code. In a design context, however, the same term may indicate both the visual appearance *and* a set of abstract behaviors that identify the design's style at large . We therefore use the term 'look and feel design guidelines' to describe collectively the set of abstract behaviors and design guidelines *plus* the resulting visual appearance of the GUI components.

A set of look and feel design guidelines is therefore more than a mere collection of appearances for visual components. It implies also a set of behaviors and conventions that are used throughout the applications. To take an analogy, you might build a house from bricks. and wood, but look and feel design guidelines would define the architectural style and how your constructional materials should be used to produce an effective and comfortable design. A look and feel implementation is a set of coherent components that comply with these guidelines.

The designers of the Java look and feel tried to cope with the diverse habits or users by creating a rather 'neutral' set of design guidelines that could be employed to create GUIs that could be used easily by Mac, Linux or Windows users. The Java look and feel was designed therefore as far as possible to be cross-platform. To have an idea of what such a design guideline is all about, we will examine some of its details in the following sections. As long as you employ standard or custom components in your GUI, you are not required to master all the details of the Java look and feel visual appearance, because Swing's designers have already worked them out for you. You need to be aware only of some general style guidelines – we will discuss these later in this chapter, and in the many examples in the rest of the book.

The Java 'look'

This is the most visible part of any GUI, the part that creates a user's first impressions. Three visual elements characterize the 'classic' Java look and feel:

- The *flush 3D style*. This describes the way in which component surfaces appear, making use of beveled edges. From a graphical viewpoint, component surfaces with beveled edges appear to be at the same level as the surrounding screen area.

9. Available on the Web at http://www.java.sun.com/products/jlf.

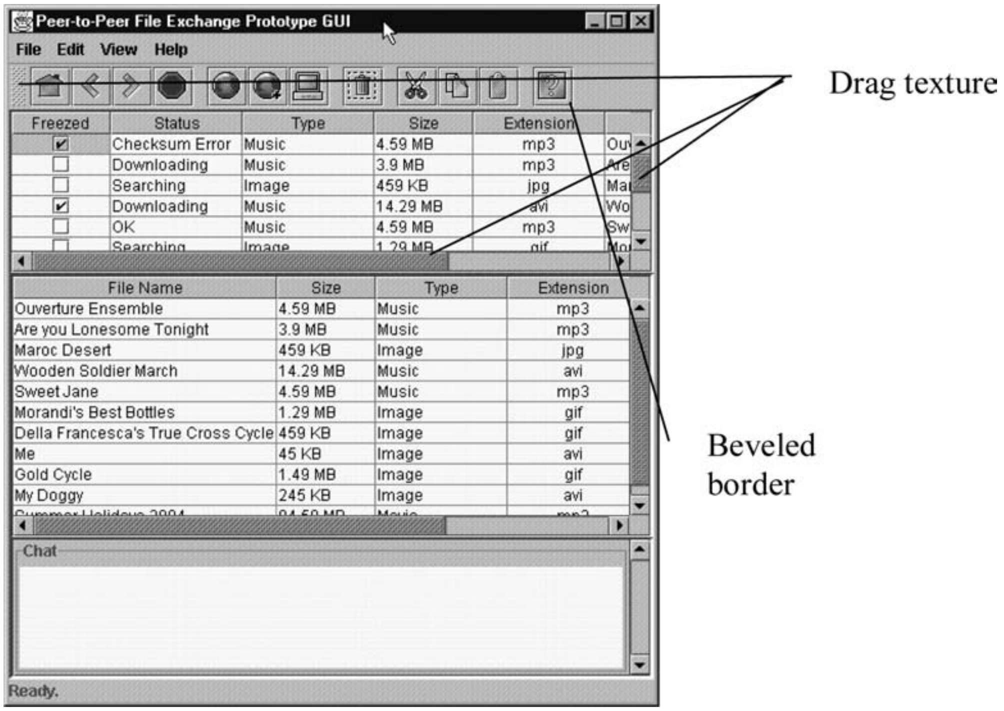
- The *drag texture*. A particular graphic pattern indicates items that users can drag with the mouse.
- The *color model*. A simple set of theme colors ensures a consistent look across different platforms. The Java look and feel uses eight system colors – three primary and three secondary colors, plus two general colors for the display of text and highlights.

Figure 3.32 shows an example of an application that uses the Java look and feel, highlighting the three basic elements of the Java look and feel. To grasp the difference, Figure 3.33 shows the same application, but using the Windows look and feel.

The visual appearance of widgets is a shallow part of a GUI. Another important part of a user's experience is the way in which the GUI reacts to user manipulation – the 'feel.'

The Java 'feel'

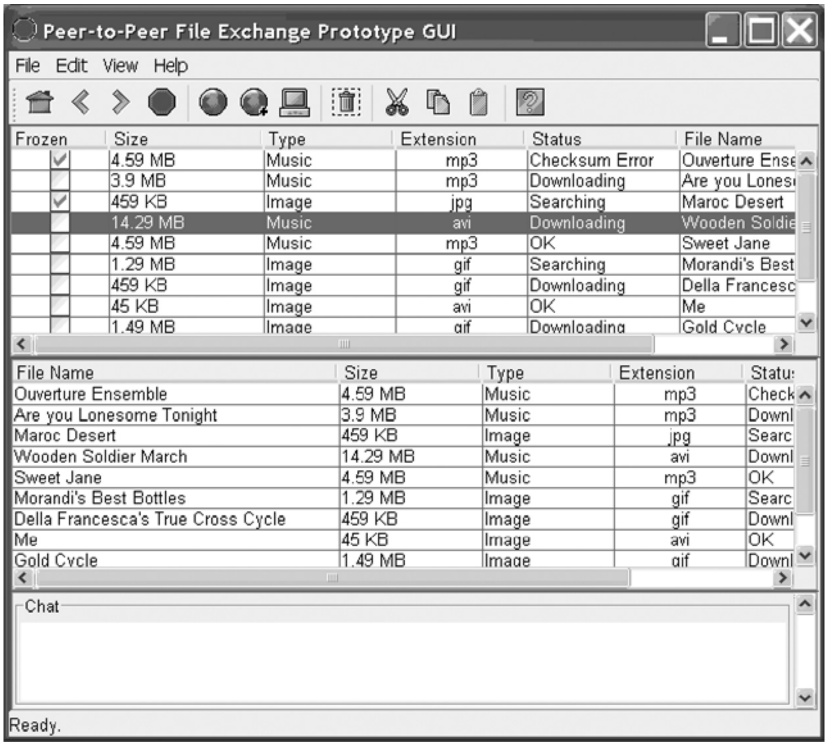
A set of look and feel design guidelines doesn't only define the visual appearance of an application's components. An important part of the design guidelines defines the way they respond to user interaction.



Freezed	Status	Type	Size	Extension	
✓	Checksum Error	Music	4.59 MB	mp3	Ou
☐	Downloading	Music	3.9 MB	mp3	Are
☐	Searching	Image	459 KB	jpg	Ma
✓	Downloading	Music	14.29 MB	avi	Wo
☐	OK	Music	4.59 MB	mp3	Sw

File Name	Size	Type	Extension
Ouverture Ensemble	4.59 MB	Music	mp3
Are you Lonesome Tonight	3.9 MB	Music	mp3
Maroc Desert	459 KB	Image	jpg
Wooden Soldier March	14.29 MB	Music	avi
Sweet Jane	4.59 MB	Music	mp3
Morandi's Best Bottles	1.29 MB	Image	gif
Della Francesca's True Cross Cycle	459 KB	Image	gif
Me	45 KB	Image	avi
Gold Cycle	1.49 MB	Image	gif
My Doggy	245 KB	Image	avi

Figure 3.32 An application using the Java look and feel



Peer-to-Peer File Exchange Prototype GUI

File Edit View Help

Frozen	Size	Type	Extension	Status	File Name
✓	4.59 MB	Music	mp3	Checksum Error	Ouverture Ense
☐	3.9 MB	Music	mp3	Downloading	Are you Lones
✓	459 KB	Image	jpg	Searching	Maroc Desert
☐	14.29 MB	Music	avi	Downloading	Wooden Soldie
☐	4.59 MB	Music	mp3	OK	Sweet Jane
☐	1.29 MB	Image	gif	Searching	Morandi's Best
☐	459 KB	Image	gif	Downloading	Della Francesc
☐	45 KB	Image	avi	OK	Me
☐	1.49 MB	Image	gif	Downloading	Gold Cycle

File Name	Size	Type	Extension	Statu:
Ouverture Ensemble	4.59 MB	Music	mp3	Check
Are you Lonesome Tonight	3.9 MB	Music	mp3	Downl
Maroc Desert	459 KB	Image	jpg	Searc
Wooden Soldier March	14.29 MB	Music	avi	Downl
Sweet Jane	4.59 MB	Music	mp3	OK
Morandi's Best Bottles	1.29 MB	Image	gif	Searc
Della Francesca's True Cross Cycle	459 KB	Image	gif	Downl
Me	45 KB	Image	avi	OK
Gold Cycle	1.49 MB	Image	gif	Downl

Chat

Ready.

Figure 3.33 A Java Application using the Windows look and feel

Any visual component has its own interaction rules. These rules describe how components react to user manipulation. As an example, here are the rules for user selection for some of the Java widgets.

For text – multiple line or single line text-based components, such as text fields or text areas:

- A single click deselects any existing selection and sets the insertion point.
- A double click on a word deselects any existing selection and selects the word.
- A triple click on a text line deselects any existing selection and selects the whole line.
- A shift-click extends a selection by the same unit as the previous selection (single character, word, line, etc.).

- Mouse dragging deselects any existing selection and selects the currently selected range.
- Direct manipulation for cutting or copying a text selection is not provided.

While for lists and tables:

- A single click on an item deselects any existing selection and selects the object.
- A shift-click on an item extends the selection from the last selected item to the new one.
- A control-click on an item toggles its selection without affecting the previous selection.

Even from simple rules such as those above, it is clear that if you allow users to change the look and feel at runtime, you should also change the underlying behavior, terminology and standard layouts (what we called the *style*) to match the chosen look and feel. This is much trickier than simply changing the widgets' visual appearance. For this reason, if you plan to deploy your application on different target platforms, the wisest choice is to adopt the standard Java look and feel guidelines. Although technically possible, the official design guidelines strongly discourages the provision of features that allow end users to switch to a different look and feel at runtime.

Some terminology

We introduce here some terminology related to user GUI interaction that we will use throughout the book to describe the support of keyboards and other input devices.

- *Mnemonics* are look and feel and locale-dependent combinations of a letter and a modifier key such as Alt. Mnemonics are used for menu item selection and for setting the focus. They are shown by an underline under the given character. For example, the Windows or the Java look and feels allow menu items to be selected by combining the underlined letter with the Alt modifier.

  Figure 3.34 and Figure 3.37 on page 115 show examples of use of mnemonics for focus control in dialogs. Note that mnemonics are used also for command buttons.
- *Accelerators* or *keyboard shortcuts* are key combinations completely defined by the designer. For example Ctrl-x is the keyboard shortcut for activating the 'cut' command on the selected items. Figure 2.21 on page 63 shows a pop-up menu in which every command is provided with accelerators. The Java look

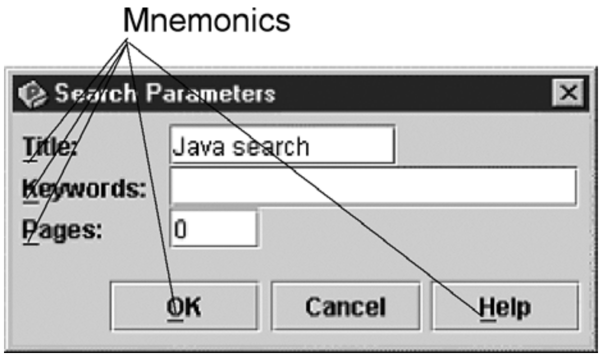


Figure 3.34 An example of mnemonics

and feel requires that accelerators are indicated to the right of the command label for menu items, and in the tooltip as well where relevant.

- *Focus navigation*. Using the keyboard, it is possible to switch the focus from one component to another. This provides a quick way to manipulate the GUI, and is very convenient for data entry forms.

The scope of accelerators and mnemonics is limited to the current window. When deciding which characters to use, some guidelines apply:

- Use standard accelerators whenever possible. The official guidelines[10] provide a list of the most common ones.
- If this is impossible, use the first letter, as long as it doesn't conflict with other mnemonics. In the example in Figure 3.37, 'L' is used for 'Log in.'
- If the first letter of the label is not available, resort to the next suitable consonant. For example, if 'l' is reserved, 'n' could be used. If this also fails, choose a suitable vowel. Locales with non-Latin alphabets should use the English mnemonic. (For languages other than English, internationalization guidelines are provided.)
- Finally, do not provide mnemonics or accelerators for potentially dangerous commands such as 'delete,' 'cancel,' or for the default button in a dialog, as this can be triggered merely with the Return key.

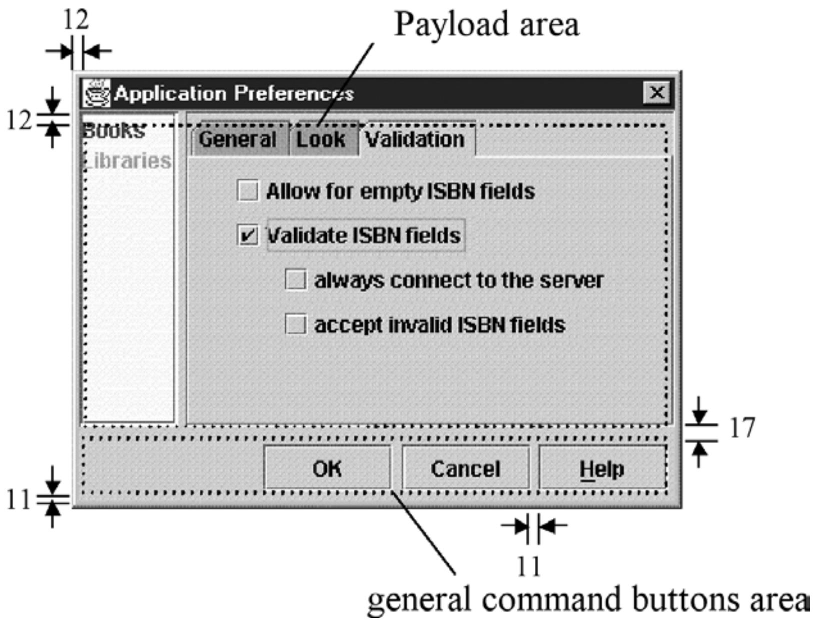
10. See http://www.sun.com

An example – applying the guidelines for designing dialogs

Dialogs provide a useful means delivering an application's functionalities in logical 'chunks,' which can enhance user's understanding. Dialogs also provide an indication of task completion, providing feedback to users.

We introduce some general guidelines for the design of dialogs here: in Chapter 5 we will illustrate these with coded examples.

Figure 3.35 shows a dialog designed following the Java look and feel guidelines. Some of the minor details, such as standard dimensions in pixels are also shown.



Allow for empty ISBN fields: [ ]
Validate ISBN fields: [✓]
always connect to the server: [ ]
accept invalid ISBN fields: [ ]

Figure 3.35 A Java look and feel guideline–compliant dialog

It is important to emphasize the prescribed structure – the *style* – for dialogs. Figure 3.36 shows the two standard arrangements for area organization within a Java look and feel compliant-dialog. Note that the second arrangement, using vertically placed buttons, is less common in practice.

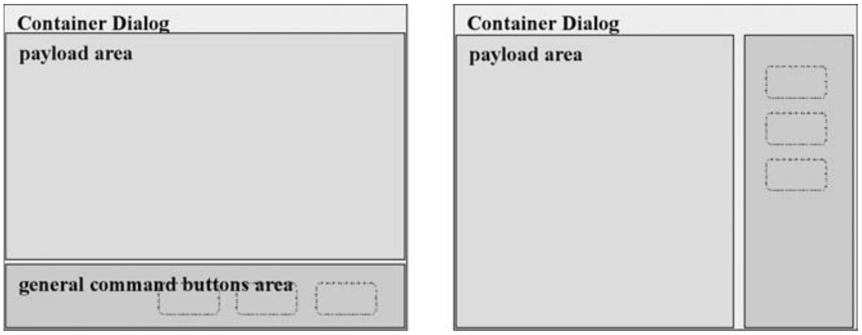


Figure 3.36 General structure for Java look and feel dialogs

Note in Figure 3.36 that buttons and other component can also be employed in the *payload* area. The general command buttons refer to the dialog as a whole, while Content-specific components will be organized within the payload area.

This simple structure guarantees a systematic and predictable layout for dialogs. Users easily discover how to dismiss a dialog or to perform the intended operation, which is always associated with the left-most button. This is illustrated in Figure 3.37, which shows an example of a log-in dialog: instead of the **OK** label, the dialog's acceptance button has a more expressive label, **Log In**.

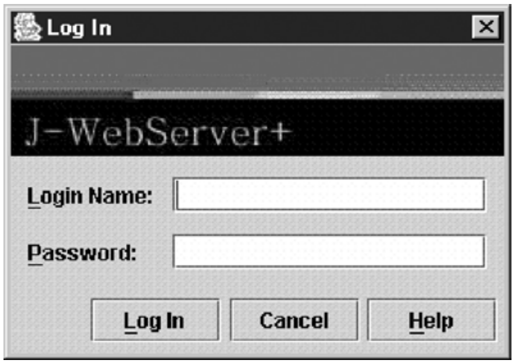


Figure 3.37 An example of standard Java look and feel login dialog

We will see the graphic details of such a scheme when we discuss some real cases in Chapter 5.

Regrettably, not all the dialogs provided by the standard Java libraries are compliant with this simple organization. This is partly because the arrangements described in Figure 3.36 can sometimes result in inefficient use of space. Figure 3.38, for example, shows an example of a well-known JFC standard dialog that doesn't follow the suggested area organization. In the 1.4 release of J2SE, Sun's designers amended the design to that shown in Figure 3.39.

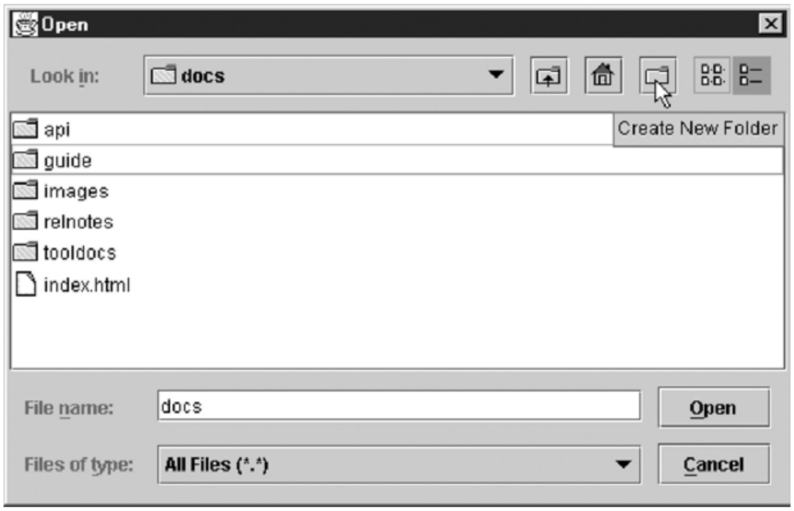
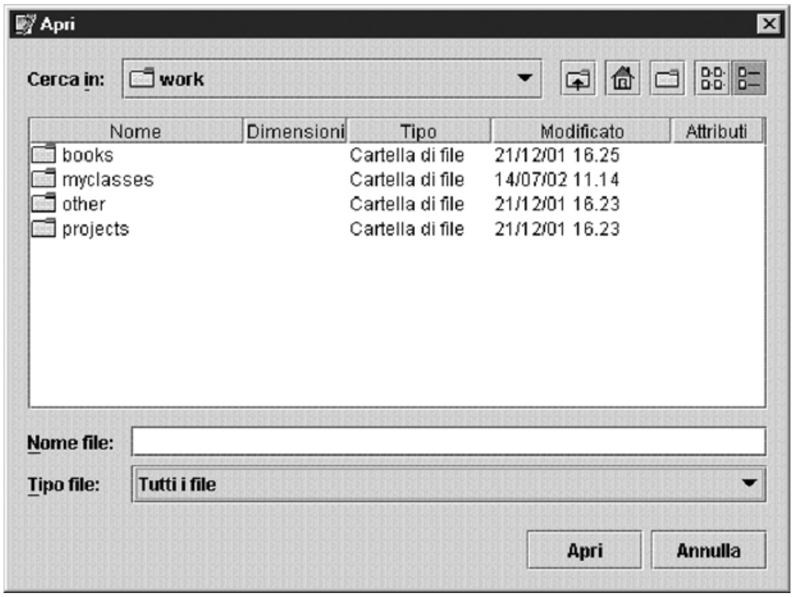


Figure 3.38 The not-so-standard file chooser of J2SE 1.3

The design in Figure 3.39 illustrates internationalization support for standard components – it shows an open file dialog for the Italian locale.

We will use a dialog classification scheme extensively in rest of this book. We group dialogs by the way they allow interaction:

- *Modal dialogs*. Users are forced to interact with the currently-open dialog. If the user wants to interact with the remainder of the application, they must first dismiss a modal dialog. Typical general commands for this kind of windows are '**OK** and **Cancel**, or some other context-dependent command such as **Log In**.
- *Modeless dialogs*. Modeless dialogs don't prevent users from interacting with other windows in the same application. Such dialogs can be used for toolbox



Cerca in: work

Nome	Dimensioni	Tipo	Modificato	Attributi
books		Cartella di file	21/12/01 16.25	
myclasses		Cartella di file	14/07/02 11.14	
other		Cartella di file	21/12/01 16.23	
projects		Cartella di file	21/12/01 16.23	

Tipo file: Tutti i file

Figure 3.39 The file chooser dialog of J2SE 1.4

or other auxiliary windows that assist users with details of the operations being performed on the main window.

3.6 Summary

In this chapter we have discussed the general principles of user interface design, mentioning common aspects of user interface design, and have provided a brief introduction to the Java look and feel guidelines. We also discussed the GUI design space for a simple chooser.

Here are some of the ideas we discussed in this chapter.

- We saw that Java user interfaces for J2SE are organized into components that can be assembled to create complex user interfaces.
- We distinguished three types of visual components based on their construction complexity:
  - *Standard components* are obtained from standard library components with few adaptation to their code.
  - *Custom components* are major customizations of standard components, involving the creation of new, non-trivial specialized classes.

  - *Ad-hoc components* are components created from scratch for solving specific problems that aren't addressed by existing libraries, either those provided by Sun, or by other third-party component vendors.
- We suggested that user interface design guidelines can be visualized as a hierarchy for building a coherent framework for professional GUI design.
- We briefly introduced some of the aspects of the Java look and feel, that we will assume as the reference look and feel throughout this book.

In the next chapter we discuss some frequent GUI designs for Java GUIs.

4 Recurring User Interface Designs

This chapter illustrates some common GUI designs. We will present them in a practical way, sometimes sacrificing exactness and completeness for practical utility and intuitiveness. The idea is to make you aware of some common issues, together with their possible solutions, that have been developed and refined over recent years. Unfortunately, user interface design is a human-dependent task, and it doesn't make sense to constraint it in precise, formal rules.

Following the multidisciplinary approach of this book, we will see both GUI design and development issues together, often switching between the designer's and the implementer's viewpoints. We discuss both Sun's Java Look & Feel design guidelines, as available for Swing applications, and IBM-backed Eclipse GUI design guidelines, as available for SWT applications, although focusing more on the former: we discuss SWT extensively in Chapter 13.

This chapter is organized as follows:

4.1, GUI area organization discusses the issues related to the GUI design of screen areas in the main GUI window.

4.2, Choosers deals with a GUI design strategy that focuses on allowing users to select items and objects.

4.3, Memory components discusses the use of widgets that remember previous user choices and input, to enhance GUI usability.

4.4, Lazy initialization discusses the important approach of instantiating objects only when needed from a GUI design viewpoint.

4.5, Preference dialogs illustrates some typical GUI designs for application preferences and configuration information.

4.6, Waiting strategies introduces the most common choices for interacting with users during long-running tasks.

4.7, Flexible layout discusses the use of dynamic layout managers.

4.8, Common dialogs introduces some standard dialogs – About, Log in, and first-time dialogs, splash windows, providing reusable code.

4.9, Command components illustrates the GUI design issues related to providing toolbars, menus and buttons in GUIs.

4.10, Accessibility discusses how to provide accessibility support in Java GUIs.

4.11, Navigation and keyboard support introduces keyboard input and tab navigation to allow a GUI to be used from the keyboard.

4.12, Internationalization discusses the problems and solutions involved in internationalizing and localizing Java GUIs.

4.13, Help support describes the adoption of a help system in applications.

4.15, Leveraging object-oriented programming discusses how to employ OOP to build better Java GUIs more effectively.

4.14, Icons and images illustrates some GUI design issues related to icons and images with Java GUIs.

4.1 GUI area organization

User interfaces often need to show different information at the same time. In this case it is essential to determine a suitable organization for the screen area. Over the years several arrangements have been established for this purpose. A common layout for 'average' applications implements an area devoted to the work itself, such as the text editor pane in a RAD (rapid application development) environment, a command area, usually at the top of the frame, containing the menu bar and some toolbars, and a selection or exploration area on the left.

When there is more data to show, additional areas can be combined with these basic ones. We will see some examples of area organizations in the sections that follow.

Terminology

For container visual components, we will use the following terms in this chapter, which are taken from Swing terminology:

- A *window* is a visual container used for organizing the information that users see in an application. We will use this term to indicate both *dialogs* and *frames* (generic screens) or to indicate 'plain' windows – that is, those without the top header – used for example in splash screens.
- A *frame* is a window in which the user's main interaction takes place.
- A *dialog* is a secondary window that is dependent on a frame or on another dialog, and is used to support the main interaction that takes place in frame(s).

- Finally, a *panel* is a generic visual container that represents an area assembled with visual components. Panels can be composed within other panels or within any window.

Main frames

Essentially, except for the command area – the upper area, which gathers the menu bar and the toolbars – and a status bar on the bottom, the rest of the window is left to the designer's creativity.

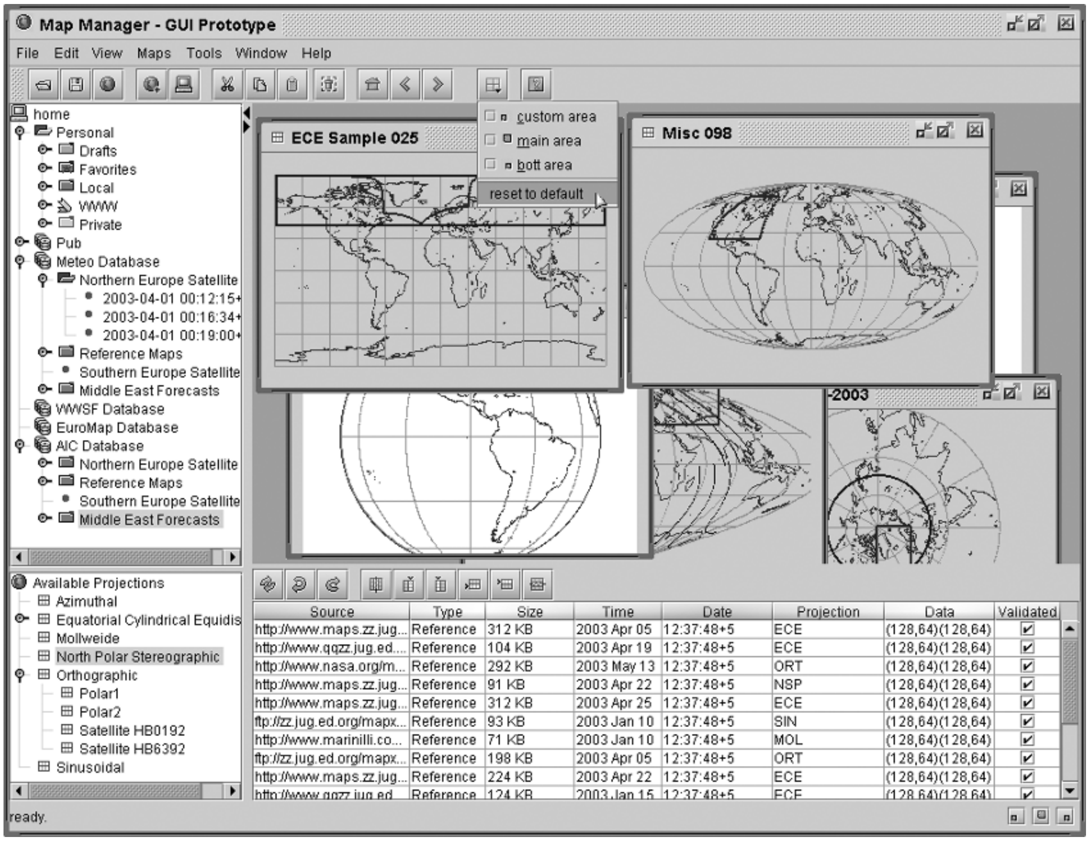
A top-down design approach begins with the identification of the following standard areas in a GUI, or those of them that are required:

- *Selection area*, situated on the left of the main area. This usually contains a tree view or other selection components.
- *Work area*. the main area of the dialog, and where the user's attention is focused most of the time.
- *Secondary area*, which can be devoted to the details of the current operation, or to messages, or to some notification message not captured by the work area.
- Other areas. Depending on the GUI's complexity, additional display areas can be needed.

This type of organization has some common properties. Apart from the main area or application-specific areas, the other areas should be made visible and customizable as required by the user, and a means provided to make such settings persistent. This may be done with toggle buttons in the toolbar for the most commonly-used areas, while others may be located in a related drop-down menu. Areas other than the selection and work areas should be designed as simply as possible, in order not to distract the user's attention. For complex interactions that are not supported by the work area, a modal dialog is often the best choice.

Figure 4.1 shows a sample area organization for an application's main frame. This (fictitious) application manages a set of geographic databases containing images of the earth using different projections.

As we know from previous chapters, many design choices ultimately depend on the end user population. Their working habits, the tasks they regularly perform, and other variables all contribute to the final design. In the application above, for example, the need for comparison of different images prompts the use of a multiple document interface (MDI) display organization, implemented by using the internal frames in the main area (top right). The selection sub-area in the bottom left of Figure 4.1 has been added to accommodate the frequent task of selecting the various available projections for a given image.

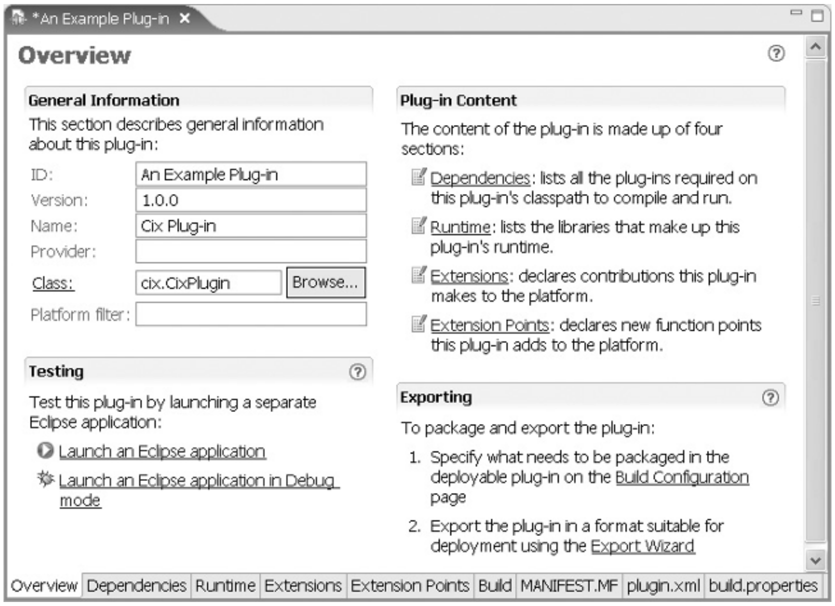


Source	Type	Size	Time	Date	Projection	Data	Validated
http://www.maps.zz.jug...	Reference	312 KB	2003 Apr 05	12:37:48+5	ECE	(128,64)(128,64)	✓
http://www.qqzz.jug.ed....	Reference	104 KB	2003 Apr 19	12:37:48+5	ECE	(128,64)(128,64)	✓
http://www.nasa.org/m...	Reference	292 KB	2003 May 13	12:37:48+5	ORT	(128,64)(128,64)	✓
http://www.maps.zz.jug...	Reference	91 KB	2003 Apr 22	12:37:48+5	NSP	(128,64)(128,64)	✓
http://www.maps.zz.jug...	Reference	312 KB	2003 Apr 25	12:37:48+5	ECE	(128,64)(128,64)	✓
ftp://zz.jug.ed.org/mapx...	Reference	93 KB	2003 Jan 10	12:37:48+5	SIN	(128,64)(128,64)	✓
http://www.marinilli.co...	Reference	71 KB	2003 Jan 10	12:37:48+5	MOL	(128,64)(128,64)	✓
ftp://zz.jug.ed.org/mapx...	Reference	198 KB	2003 Apr 05	12:37:48+5	ORT	(128,64)(128,64)	✓
http://www.maps.zz.jug...	Reference	224 KB	2003 Apr 22	12:37:48+5	ECE	(128,64)(128,64)	✓
http://www.qqzz.jug.ed	Reference	124 KB	2003 Jan 15	12:37:48+5	ECE	(128,64)(128,64)	✓

Figure 4.1 A typical main frame area organization (Compiere)

GUIs built using Eclipse can take advantage of the ‘flat look’ GUI library, an example of which is shown in Figure 4.2. Essentially, it exploits HTML-like widgets to save space in high-density form-based panels. It is available only within specialized panel subclasses, and it cannot be used in toolbars and other general-purpose containers.

The Eclipse ‘flat look’ is a valuable tool in the GUI designer’s toolbox for SWT applications. As of Eclipse 3.1, though, it is still needlessly hard to use for



*An Example Plug-in

Overview

General Information

This section describes general information about this plug-in:

ID:	An Example Plug-in
Version:	1.0.0
Name:	Cix Plug-in
Provider:	
Class:	cix.CixPlugin
Platform filter:	

Testing

Test this plug-in by launching a separate Eclipse application:

- Launch an Eclipse application
- Launch an Eclipse application in Debug mode

Plug-in Content

The content of the plug-in is made up of four sections:

- Dependencies: lists all the plug-ins required on this plug-in's classpath to compile and run.
- Runtime: lists the libraries that make up this plug-in's runtime.
- Extensions: declares contributions this plug-in makes to the platform.
- Extension Points: declares new function points this plug-in adds to the platform.

Exporting

To package and export the plug-in:

1. Specify what needs to be packaged in the deployable plug-in on the Build Configuration page
2. Export the plug-in in a format suitable for deployment using the Export Wizard

Figure 4.2 Eclipse's flat look to the rescue of crammed forms

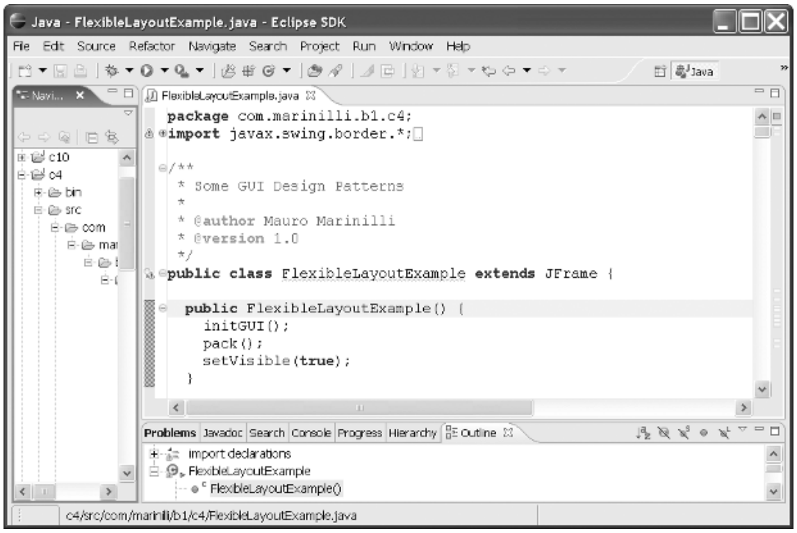
developers. Ironically, an 'old-fashioned' desktop application GUI should take advantage of a newer and possibly more limiting technology: advanced Web forms.

Multiple document interfaces

Multiple document interfaces (MDIs) are GUIs in which several different fully-fledged internal windows are responsive to user interaction at the same time, like the application shown in Figure 4.1. MDIs can be implemented in different ways, for example as a collection of frames or non-modal dialogs in Swing.

The Swing library contains a special set of components, called *internal windows*, for providing a way to manage multiple windows that are confined inside a main window. From a usability perspective, internal frames and MDIs in general are difficult to manage for average users, and their use is usually not needed for most applications. Eclipse itself is an example of a complex GUI in which designers succeeded in minimizing the use of MDI, as shown in Figure 4.3.

Another common approach to the efficient exploitation of precious GUI real estate is to serialize it – that is, to split a long task into a sequence of simpler steps, each rendered with the same subset of the screen area. This approach has different names, but is most commonly known as a *wizard*.


Java - FlexibleLayoutExample.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Navi...
c10
c4
bin
src
com
mar
FlexibleLayoutExample.java
Problems Javadoc Search Console Progress Hierarchy Outline
import declarations
FlexibleLayoutExample
FlexibleLayoutExample()
c4/src/com/marinilli/b1/c4/FlexibleLayoutExample.java

```
package com.marinilli.b1.c4;
import javax.swing.border.*;

/**
 * Some GUI Design Patterns
 *
 * @author Mauro Marinilli
 * @version 1.0
 */
public class FlexibleLayoutExample extends JFrame {

  public FlexibleLayoutExample() {
    initGUI();
    pack();
    setVisible(true);
  }
```

Figure 4.3 Eclipse's GUI design avoids the use of multiple document interfaces

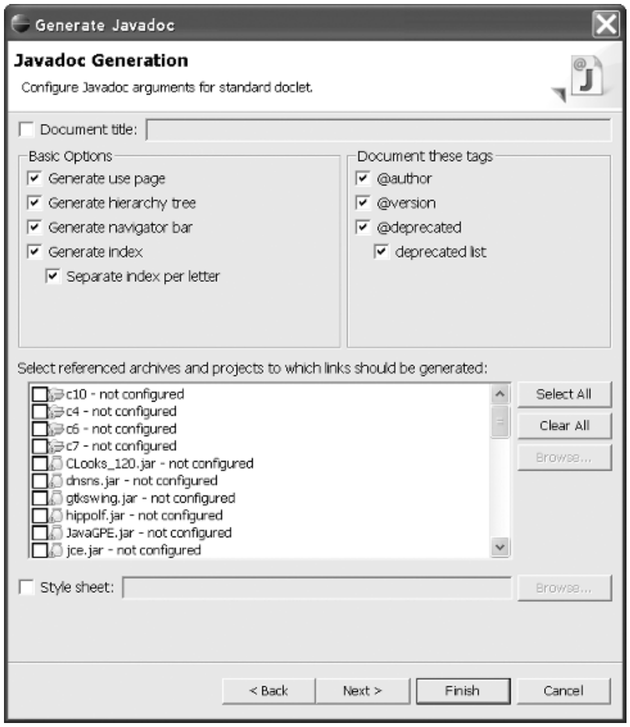
Wizards

Wizards are a well-known and widespread way to organize GUI functionalities in order to support inexperienced users. A wizard guides the user through the features provided by the application in a simplified way, proposing only a few choices and limited information at a time. By narrowing the available choices, novice users are better guided through an interaction with the application. Historically, wizards made their debut in the mass market with Windows 95.

The (Advanced Java L&F Design Guidelines 2001) provides some useful advice on designing wizards for the Java Look and Feel, and the Eclipse guidelines also provide such advice. An example of a wizard designed to Eclipse design guidelines is shown in Figure 4.4.

The buttons at the bottom of the wizard are used for navigating through the various panels, and are disabled according to the semantic state of the current pane.

In an attempt to lower the cognitive burden needed to understand GUI interactions, all interactions should be kept localized and their context narrowed as much as possible. If user data prompts a GUI notification, this should be kept linked with the triggering cause as far as possible, so that the user will interpret it more easily. In the case of a wizard, an awkward situation arises when a data item inserted in a previous panel affects the behavior of another panel. In this case the user might not be able to notice the connection, and so fail to understand the behavior of the application.



Generate Javadoc

Javadoc Generation

Configure Javadoc arguments for standard doclet.

☐ Document title:

Basic Options
- ✓ Generate use page
- ✓ Generate hierarchy tree
- ✓ Generate navigator bar
- ✓ Generate index
  - ✓ Separate index per letter

Document these tags
- ✓ @author
- ✓ @version
- ✓ @deprecated
  - ✓ deprecated list

Select referenced archives and projects to which links should be generated:
- ☐ c10 - not configured
- ☐ c4 - not configured
- ☐ c6 - not configured
- ☐ c7 - not configured
- ☐ CLooks_120.jar - not configured
- ☐ dnsns.jar - not configured
- ☐ gtkswing.jar - not configured
- ☐ hippolf.jar - not configured
- ☐ JavaGPE.jar - not configured
- ☐ jce.jar - not configured

☐ Style sheet:

Figure 4.4 An Eclipse wizard to Eclipse design guidelines

Designing a wizard

We conclude this section by discussing some basic, general advice for designing wizard GUIs. Fortunately there is much literature and infrastructure support for building wizards in the form of high-level reusable API and classes.

Designing a usable wizard is usually not a complex task if a few simple rules are followed. Conversely, given their simplicity and the relatively cheap development cost of using a general framework to implement them, the opposite problem often the case – a proliferation of wizards in application that do not really need them. As a rule of thumb, wizards should provide an alternative interaction mechanism to an existing feature, or be employed for non-repetitive tasks for occasional users only.

Well-designed wizards clearly declare their boundaries, so that users know where the wizard begins, where it finishes, and the sequence of operations within it. The inputs and the final result should be clearly outlined, so that the user can be sure of what they are doing.

Usability studies have demonstrated some interesting aspects of this kind of user interface device. Users tend to employ a wizard only to complete a given task, and are relatively uninterested in learning new concepts in the meantime.

Wizards can be particularly useful for the following activities:

- Dividing a complex input process into many sequential steps. 'Serializing' a complex input form, such as for the creation of complex data structures, is a common case of this.
- Performing tasks that are inherently composed of a well-defined sequence of steps. By focusing on each step, correct task completion is much more likely to be achieved.
- Whenever users lack domain knowledge in some field and need to be guided through operations. Consider the integration of new hardware into an operating system, or the completion of some task involving decisions and knowledge in other fields.

Wizards for occasional, hard-to-undo operations should also provide a final recap screen where all the important data is summarized before the user leaves the wizard and completes the task.

The design ideas for organizing the application display area that we have shown in this section are rather general and can be applied to many different situations. Next we discuss another common pattern in practical GUI design – choosers.

4.2 Choosers

Principled, systematic area organization is essential in secondary windows, dialogs and choosers, as well as in any other part of the GUI. By *chooser* we mean a screen area specialized for a performing a selection task on a given item. Choosers deserve their own discussion, both because they happen to be a useful means of interaction in Java GUIs, and also because quite often their use has been misunderstood.

Figure 4.5 shows an example of a chooser with three distinct areas where the user can focus their attention:

- On the left-hand side the selection area contains a list that shows the items that can be picked.
- The right-hand side is the preview area, where the currently-selected item is shown.
- The bottom-most part of the dialog is occupied by the standard command buttons for deferred mode interaction and preview option.

Note that in choosers the main area coincides with the selection zone, because of the purpose of these components. To make the design coherent with the standard layout, the selection area is still organized on the left-hand side.

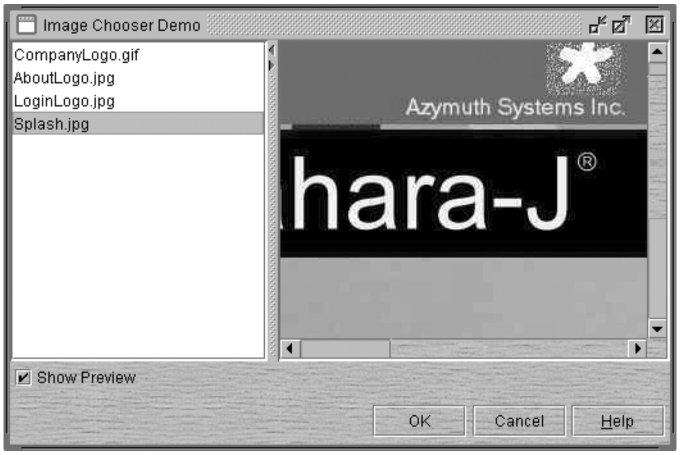


Figure 4.5 Area organization in an image chooser (Wood)

Often users need to specify one or more items while using the GUI. When this type of choice is occasional and involves a dedicated interaction because of its complexity, a chooser should be designed to accommodate it.

Choosers are often designed as pop-up dialogs that contain the information needed to specify the given item. Once the item has been chosen, the dialog is dismissed and the new value is used in the application.

Choosers are often activated by means of a button, usually a **More…** button. Such a button indicates the availability of further related data that can be showed by clicking the button. Such a behavior is signaled by the ellipsis (**…**) in the button's caption, together with a brief description of the planned action. For example, when selecting a file from a file chooser, instead of directly entering the file's path, a common choice for the related 'more' button label is **Browse…**

Chooser activation mechanisms

There are two main ways to show a chooser in a new window: as a pop-up window, or as a fully-fledged dialog (often a modal one).

A useful convention is to use a downward arrow to signal a pop-up chooser for buttons only, referred to as *drop-down buttons*[1]. In all other cases, a **More…** button that triggers the related chooser dialog is the most common solution.

1. See icon images in *Graphic conventions* on page 156.

A chooser can be also contained in a lightweight pop-up such as the one used by Combo boxes, which in this respect can be seen as choosers for one text value among a list of available alternatives.

Chooser interaction styles

As we know from Chapter 3, we can have two main kinds of interaction modalities for dialogs: *immediate* and *deferred* mode.

The following two examples illustrate these two modalities and their implication for choosers: some of this discussion can be generalized for any other dialog type.

Figure 4.6 shows the standard file chooser provided by the JFC[2] in a fictitious text editor application. The file is first selected, and only when the choice is committed by hitting the **OK** button is the dialog dismissed and the new value transmitted to the underlying application. This is the deferred mode interaction style.

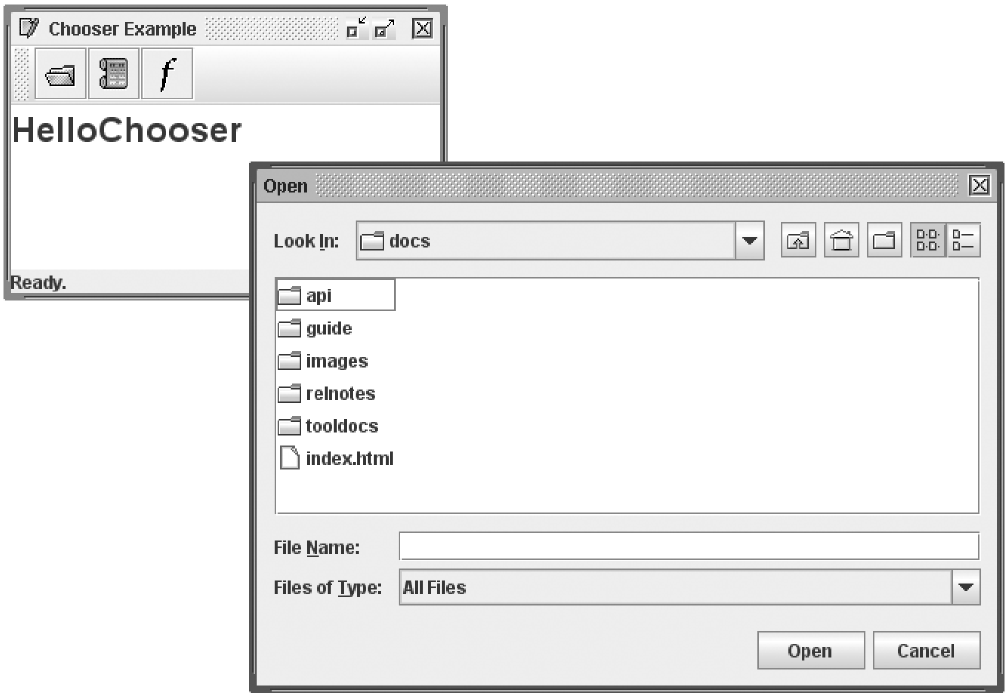


Figure 4.6 A deferred mode chooser (Smooth Metal)

2. Java Foundation Classes (JFC) are the foundational libraries needed for building GUIs with Java for JSE. They comprise the AWT and Swing libraries.

Figure 4.7 shows activation of the 'choose color' button in the toolbar, causing the color chooser to appear. The difference from the previous example is two-fold: the changes made in the chooser dialog are instantly transmitted to the application, and consequently the dialog is modeless.

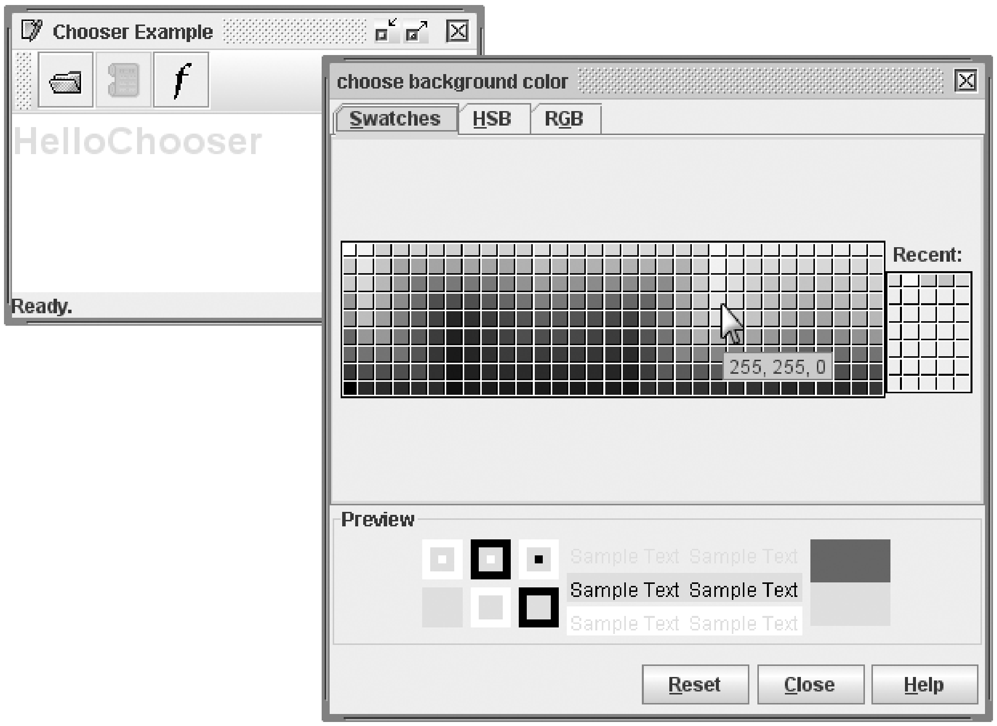


Figure 4.7 An immediate mode chooser (Smooth Metal)

There is another possible kind of interaction, which can be thought of as a combination of deferred mode – an explicit user commit action is required – and immediate mode: the window is not dismissed after user commit. In our fictitious text editor, we click the font button in the toolbar, and we are prompted with a font chooser dialog, as in Figure 4.8. Note the standard Java Look and Feel-compliant button organization.

In this example, using an immediate interaction style for the font selection could be confusing for the user and resource-consuming for the application. On the other hand, a deferred mode interaction style like the one in the file chooser is not optimal, because users prefer to see changes take place in a more interactive way. By using a multiple-use, deferred mode interaction style, we allow users to interact

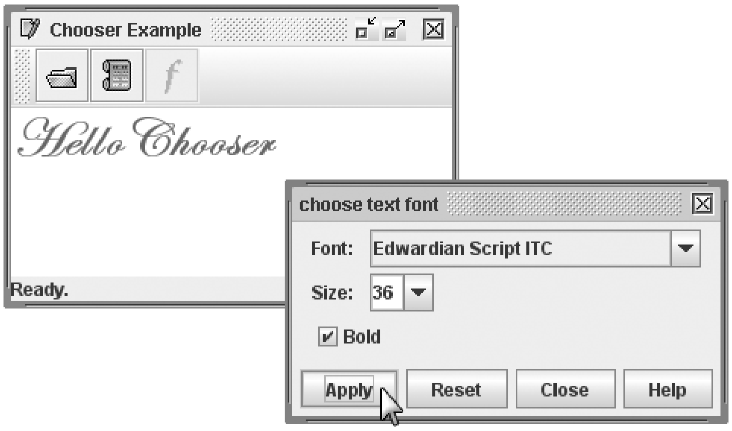


Figure 4.8 A multiple-use, deferred mode chooser (Smooth Metal)

more closely throughout the choice process at a level that is intermediate between the immediate and deferred interaction style (Figure 4.6 and Figure 4.7).

The presence of two modeless dialogs can bring some unforeseen combinations. Figure 4.9 shows our application when both modeless choosers are activated. Note that we allowed only one chooser for each kind, by disabling the corresponding command whenever the related chooser was up, although such a combination might be required in some applications.

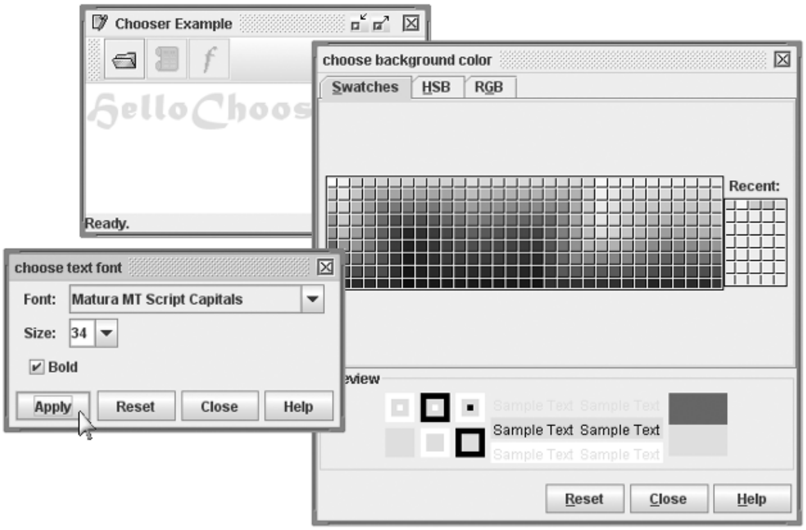


Figure 4.9 Possible combinations of modeless choosers (Smooth Metal)

Designing the interaction mode for choosers depends on several factors. The most important one is the kind of users that will be using them. Deferred mode should generally be preferred over immediate mode when the user population is made up of novices and inexperienced users. Because it provides an easier way of undoing a choice, it implies a cleaner interaction – the chooser dialog is modal, so it has to be dismissed to return to the application, hence the number of floating windows is kept under control – and keeps users more focused on the main task. Furthermore, deferred mode dialogs are more widespread in common, commercial GUIs, so users are more familiar with them. Essentially we trade usability for interaction power.

Immediate mode dialogs are used in cases in which a higher degree of interactivity is preferred and more freedom is left to the user. This happens when immediate feedback on a choice is important. Such a higher level of interactivity helps to enhance the choice process, because it allows the differences between the manipulated items to be seen immediately.

Item selection is distinct from item creation, but this is an artificial separation in practical GUIs. We discussed item selection first for clarity, but a chooser is often meant to allow users to create new items as well as selecting them from a list. In real-world choosers these two features are often blended.

Broadening the choice

From a practical viewpoint, the chooser approach is very close to the task of creating a new item. Real-world choosers frequently offer a way to create a new item as well as choosing from a list of available ones. The file chooser shown in Figure 4.10 below allows users to create a new file or folder as required.

It is always a good idea to provide an explicit way to create new items. In the case of file choosers, for example, new file creation is often obtained by entering a file name that doesn't yet exist. Unfortunately, there often is no visual hint that this is the way to do it, and users can be puzzled by this 'hidden' interaction mechanism, which often looks more like an implementation trick rather than an explicit design choice.

Other than creating new items and choosing existing ones, choosers shouldn't be overloaded with other functionalities. One of the benefits of this design solution is that it keeps the user's perception of the GUI highly structured, so that users feel comfortable and secure when interacting with it. Users *know* that they are dealing with such-and-such kind of item, and can be confused if the same chooser offers other functionalities. Real-world choosers are often polluted with management functionalities. Such features are less frequently accessed than choice-related ones, and are usually required only by experienced users.

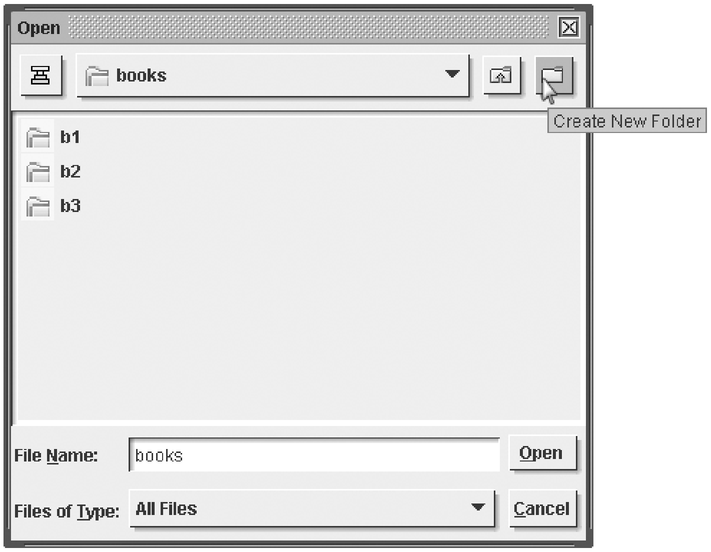


Figure 4.10 Creating new items through a chooser (Oyoaha)

More sophisticated dialogs, like management dialogs, can be built with the same design principles as choosers, but kept separate from them. In a word processor, for example, one could choose the style to be applied to a portion of text using a chooser, but the management dialog – where styles can be browsed in detail, edited, saved, imported and exported from the outside, and so on – should be kept separated from the simple chooser. A 'more' button, perhaps with a caption like **Styles...**, can be made available in the chooser if the design allows users to access the management dialog directly from the chooser.

The chooser approach can be used for selecting more than one item, as in Figure 4.11, in which users can select a list of items.

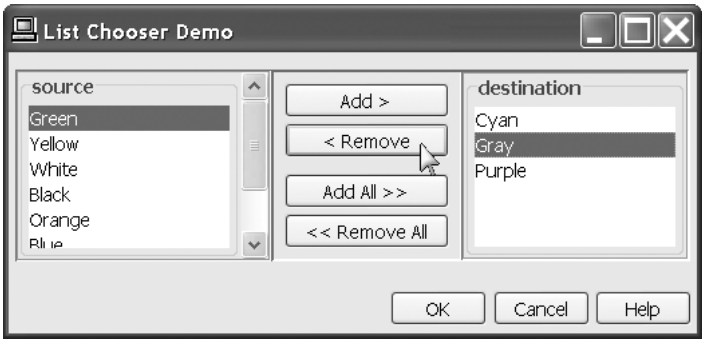


Figure 4.11 A list chooser (JGoodies Windows)

Conclusions

Choosers save GUI real estate in forms and selection screens, relegating selection to a specialized window. Choosers should be used only when the selection task is an infrequent one – in other cases, the use of a fully-fledged selection area or other more direct, faster selection mechanisms are better solutions.

Using choosers systematically in a GUI brings some benefits:

- The task of selecting an item is limited and circumscribed at a precise point, both in terms of interaction time and within the GUI.
- Completion signals indicating the end of the selection task to the user are provided automatically.
- Users understand the chooser concept, and by leveraging this repetitive interaction schema GUIs tend to be more predictable, enhancing their quality.
- Choosers are useful both for the GUI designer and for code developers.

Item creation and item selection are two conceptually separate tasks, although providing them in the same chooser is often good practice. When users are allowed to create a new item, there is usually a short-cut to such a feature within the chooser.

4.3 Memory components

Memory components are GUI components that are capable of maintaining a persistent state. This is an implementation-oriented distinction. As an example, a text field that keeps the history of previously-input strings in a drop-down can be implemented with a memory Combo box.

Memory components are usual Java visual components that can have one or more of their properties made persistent from session to session. Implementing them can be done through a specialized service provided in Service layer[3]. Designers usually need to specify only the persistent properties – in its JavaBeans meaning – of the given widgets.

There are many applications of memory components in GUI design. We will mention just a few here to better illustrate the concept.

3. See Chapter 7.

Input history

Keeping track of previous user inputs is a way to record the current user's context, thereby enhancing the overall usability of the GUI. Figure 4.12 shows a simple prototype example of a text field that registers the data inserted into it persistently throughout a session, or even across sessions[4]. Searching capabilities further enhance its usability.

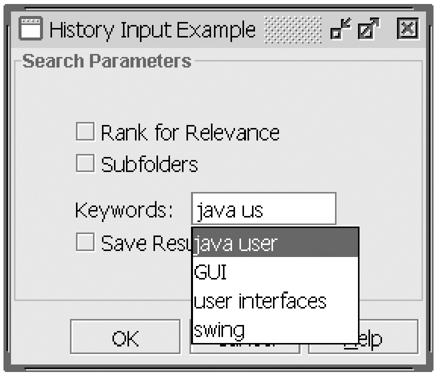


Figure 4.12 A memory combo box (JGoodies Plastic)

The field in Figure 4.12 is an example of support for user input provided automatically by the GUI. We will see in Chapter 13 that this and other stricter forms of control over user input can also be implemented for Web interfaces.

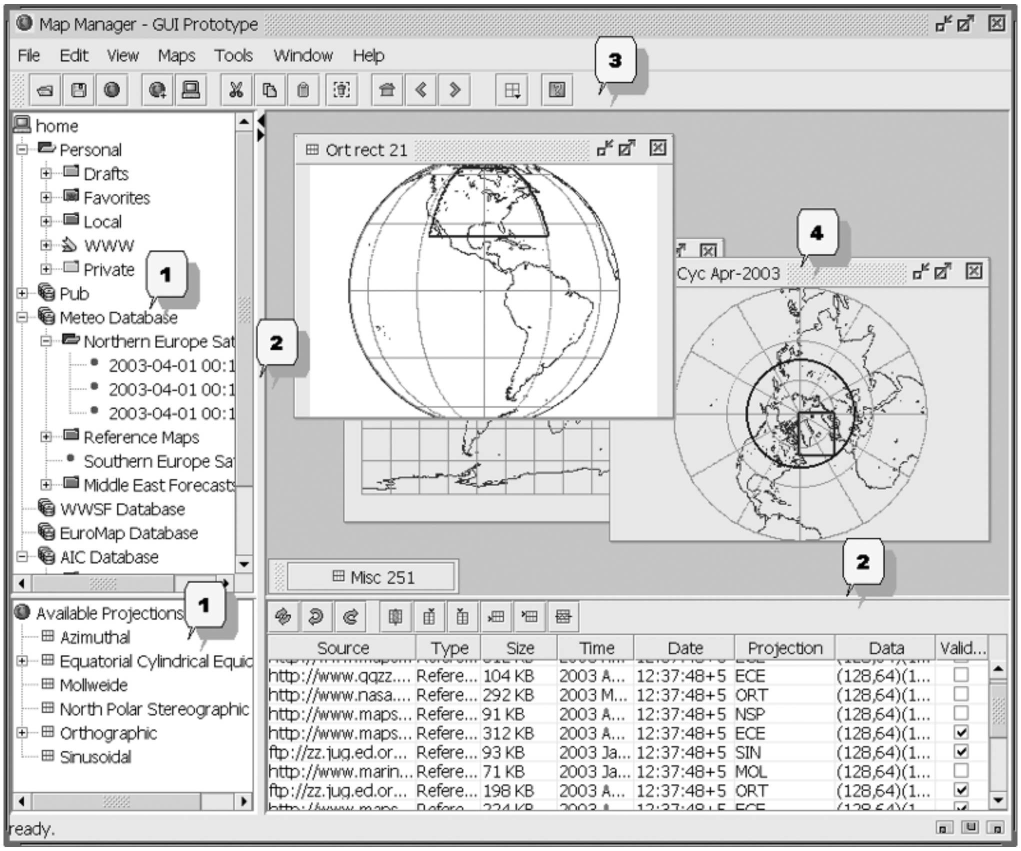
Saving user preferences

Memory components allow you to make GUI preferences persistent from session to session.

Figure 4.13 shows an example application that saves some GUI-related data persistently. The figure shows some example information that is retained for the user from session to session. Referring to the indicated areas on the figure, these are:

1. Tree structure and expanded path
2. Area separators
3. Toolbar customizations
4. Internal windows, their dimension and positions

4. This latter feature was implemented in the pioneering character-based GUI of Borland's TurboPascal.



Source	Type	Size	Time	Date	Projection	Data	Valid...
http://www.qqzz....	Refere...	104 KB	2003 A...	12:37:48+5	ECE	(128,64)(1...	☐
http://www.nasa....	Refere...	292 KB	2003 M...	12:37:48+5	ORT	(128,64)(1...	☐
http://www.maps...	Refere...	91 KB	2003 A...	12:37:48+5	NSP	(128,64)(1...	☐
http://www.maps...	Refere...	312 KB	2003 A...	12:37:48+5	ECE	(128,64)(1...	✓
ftp://zz.jug.ed.or...	Refere...	93 KB	2003 Ja...	12:37:48+5	SIN	(128,64)(1...	✓
http://www.marin...	Refere...	71 KB	2003 Ja...	12:37:48+5	MOL	(128,64)(1...	☐
ftp://zz.jug.ed.or...	Refere...	198 KB	2003 A...	12:37:48+5	ORT	(128,64)(1...	✓

Figure 4.13 A customized application (JGoodies Plastic)

Memory components are quite important and useful in professional GUIs. They allow user customization in a way that is natural from an end user perspective, and inexpensive for developers. When the display organization becomes complex it is important to provide personalization features to allow users to customize them.

4.4 Lazy initialization

The start-up time, especially for complex GUIs, is an important aspect of GUI responsiveness and overall user experience. This is even truer for Java applications. The latest JRE technology considerably enhances start-up time, but a professional GUI can't blindly rely on the invisible hand of the Java Runtime

Environment. There are cases in which start-up time is critical to a system's usability – think for example of a never-ending applet download and initialization, or a rich client application that takes ages to fill the screen with server-sourced data. In stand-alone applications, too, snappy start-up is a feature that end users will undoubtedly appreciate.

Optimizing start-up times should be a general habit rather than a circumscribed procedure to be applied only in specific cases. It originates from implementation considerations, but involves GUI design as well. GUI designers should be aware of such considerations when designing the first window the application shows to the user.

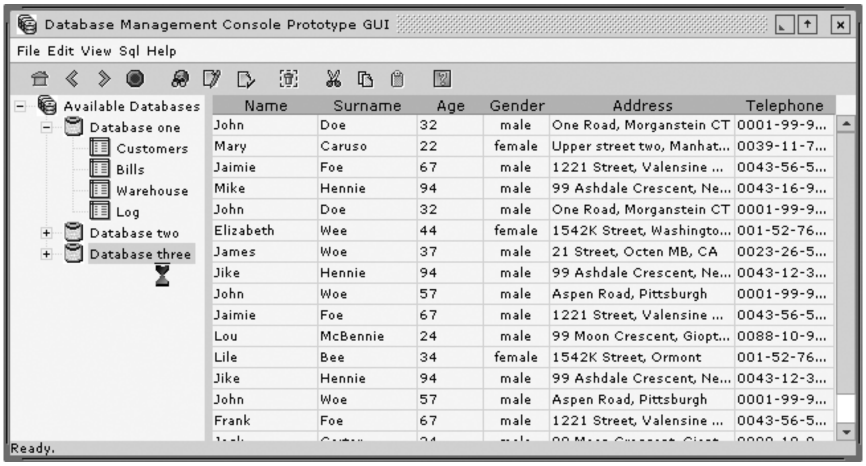
A note for Swing programmers. The `UIDefaults` class implements a common access point to all UI-related default values needed by Swing components. Some of these default values are rarely accessed, for example internal frame borders, so that employing a lazy instantiation mechanism make sense. Swing designers used an interface, `UIDefaults.LazyValue`, that is implemented by those classes that represent lazy values. Such an interface is composed of a single method, `createValue`, that returns an `Object` instance. The `get` method in `UIDefaults` first checks whether the type is an instance of `UIDefaults.LazyValue`. In this case the `createValue` method is invoked and the value is then returned.

Concretely, pieces of the GUI could be left hidden, and only when needed will they be instantiated on the fly[5]. Suppose we have a database management utility in which some databases are hosted on remote servers. To speed up GUI start-up, we could avoid the expensive (in time) remote connection, the application only connecting when prompted by the user.

Such an arrangement is implemented in the mock-up shown in Figure 4.14.

This prototype simulates an expensive connection time with a delay in expanding the third database node in the tree: you can try it yourself by running the prototype. To implement this mock-up we used some of the utility classes discussed in Chapter 5. What is interesting here is the addition of the connection delay simulation in the mock-up to make the prototype more realistic.

5. Lazy instantiation (or lazy initialization) is a strategy focused on deferring the allocation of costly resources that are not always needed until they become necessary. In this way the cost of those resources can be saved in cases where they are not required, both in terms of runtime and memory allocation.



Name	Surname	Age	Gender	Address	Telephone
John	Doe	32	male	One Road, Morganstein CT	0001-99-9...
Mary	Caruso	22	female	Upper street two, Manhat...	0039-11-7...
Jaimie	Foe	67	male	1221 Street, Valensine ...	0043-56-5...
Mike	Hennie	94	male	99 Ashdale Crescent, Ne...	0043-16-9...
John	Doe	32	male	One Road, Morganstein CT	0001-99-9...
Elizabeth	Wee	44	female	1542K Street, Washingto...	001-52-76...
James	Woe	37	male	21 Street, Octen MB, CA	0023-26-5...
Jike	Hennie	94	male	99 Ashdale Crescent, Ne...	0043-12-3...
John	Woe	57	male	Aspen Road, Pittsburgh	0001-99-9...
Jaimie	Foe	67	male	1221 Street, Valensine ...	0043-56-5...
Lou	McBennie	24	male	99 Moon Crescent, Giopt...	0088-10-9...
Lile	Bee	34	female	1542K Street, Ormont	001-52-76...
Jike	Hennie	94	male	99 Ashdale Crescent, Ne...	0043-12-3...
John	Woe	57	male	Aspen Road, Pittsburgh	0001-99-9...
Frank	Foe	67	male	1221 Street, Valensine ...	0043-56-5...

Figure 4.14 *A snappy startup GUI (Hippo)*

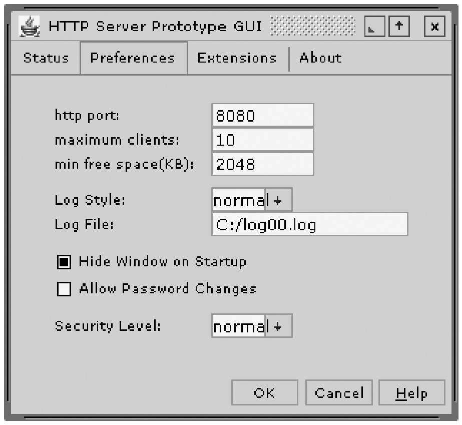
This GUI design does not need to sacrifice performance in its implementation. For example, we could keep a lightweight cache of the nodes the user expanded the last time they used the GUI. The net effect would be quick application start-up, with a delay being perceived by the end user only on node expansion. There are many possible enhancements such as this, for example keeping only few expensive nodes in memory at time, and re-adding them to the tree as needed, and so on.

4.5 Preference dialogs

User preferences are a common feature of modern GUIs. A widely-accepted practice that makes sense in terms of usability is to gather all user configuration-related commands into one configuration dialog. At design time it is important to decide what configuration information each UI object has. Usually the preference dialog is activated via a menu item and a standard button on the toolbar – see for example the Library application in Chapter 15.

Even in simple GUIs there is often a need for a preference dialog, especially when supporting a coherent means of expanding the application's features for future releases.

It is customary to organize preferences in a deferred mode dialog. To understand why, consider the application shown in Figure 4.15. This shows a fictitious GUI for a simple HTTP server. Given the simplicity of the application, operative and configuration commands are arranged together.



http port: 8080
maximum clients: 10
min free space(KB): 2048
Log Style: normal
Log File: C:/log00.log
[✓] Hide Window on Startup
[ ] Allow Password Changes
Security Level: normal

Figure 4.15 *A confusing GUI design (Hippo)*

This confuses users at first, because they can't easily understand the impact of the given commands on the application, even if they are neatly separated in different tabs. This is another case of developers dictating the GUI design. The 'catch-all' use of the tabbed pane stems directly from the implementation. Preferences should be gathered in a specialized dialog and triggered by the related option, as prescribed for example in the Java Look and Feel design guidelines.

The design choice for preference dialogs shown here of course differs from that prescribed in the official Java Look and Feel design guidelines. We discuss some of the different design choices for preference dialogs in the next section. Other visual errors demonstrated by Figure 4.15 are incorrect alignment of the check boxes and the incoherent vertical spacing between widgets.

Preference dialogs styles

Preference dialogs are an area in which designers' creativity is plentifully applied. One common design, demonstrated for example by Netscape Navigator's preference dialog, is that of using a tree to organize the selection area, like the one shown in Figure 4.18 on page 140.

In simple or medium-complexity applications particularly, using a tree results in a less usable design. It excessively burdens the user's memory ('*Where was that option?*') and obliges users to expand the selection area to look for a specific property, when a simpler design would have been more effective. The Java Look and Feel design guidelines suggest a different design choice, one that simplifies the selection area as a non-hierarchical list. Figure 4.16 shows such a design for a fictitious Java Internet browser application.

Unfortunately, such a design doesn't scale well to complex GUIs, such as those with many options. In such cases – when the exploration area on the left doesn't

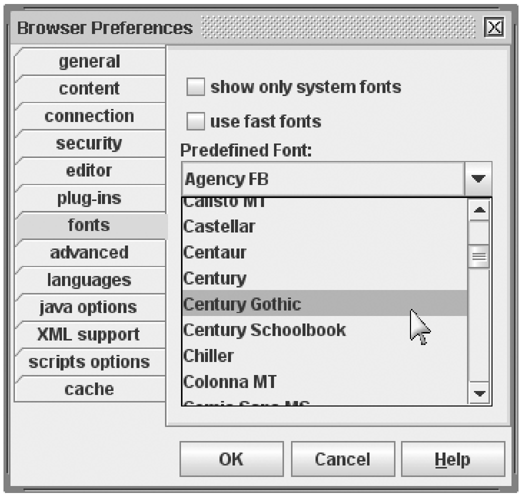


Figure 4.16 A preference dialog designed following the Java L&F guidelines (Smooth Metal)

result in sparse tree – the best solution is to organize the many options into a hierarchy. A possible solution, adopted in some of the Swing examples in this book, is shown in Figure 4.17.

The hierarchy is realized by means of a `JTabbedPane`, and the exploration area on the left-hand side, implemented with a list, points to the categories of options on the right-hand side. Icons can be used in the selection list to strengthen the mental association of label to options category, making options more recognizable for occasional users as well.

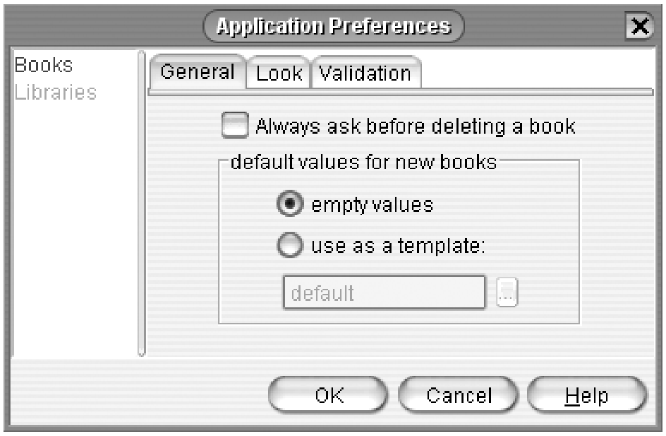
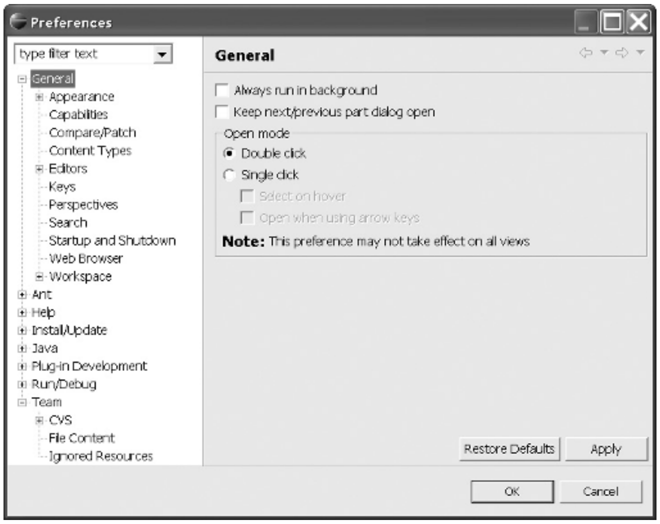


Figure 4.17 A preference dialog with a different design (Liquid)

This design isn't too dissimilar to that prescribed by the Java Look and Feel design guidelines and shown in Figure 4.16, but can accommodate more complex GUIs as well. Moreover, it forces designers to organize the options in a hierarchy at most only two levels deep.

When a GUI is complex, the previous design doesn't work well and we need to resort to a more powerful design, such as that shown in Figure 4.18, which is taken from the Eclipse 3.1 preference dialog.



General

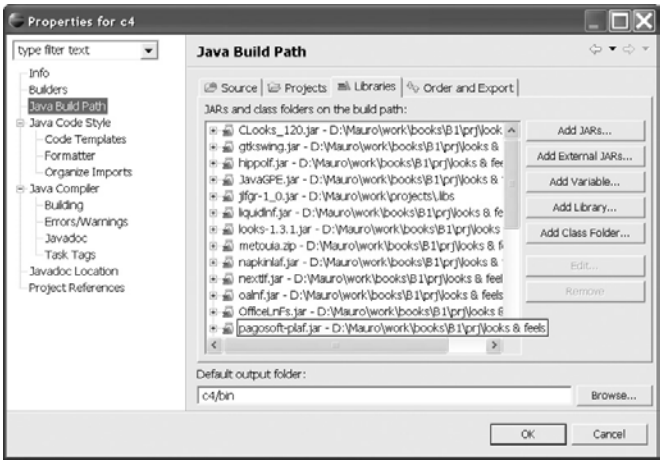
[ ] Always run in background
[ ] Keep next/previous part dialog open

Open mode
(●) Double click
( ) Single click
[ ] Select on hover
[ ] Open when using arrow keys
Note: This preference may not take effect on all views

Figure 4.18 A view of the Eclipse 3.1 preference dialog

The Eclipse 3.1 preference dialog employs a search facility – the Combo box at the top of the exploration area on the left-hand side of Figure 4.18 – that acts as a filter for showing only those pages with occurrences of keywords that match the filter text. This makes it possible for users to access the required preference pages by keyword, instead of walking the exploration tree looking for the right preference page.

For complex applications this preference dialog design can be used for functional purposes, for example to gather business domain configuration data. Figure 4.19 shows an example of this idea, again from the Eclipse GUI. See how the general Content structure is almost identical to that used for the dialog in Figure 4.18.



JARs and class folders on the build path:

```
CLooks_120.jar - D:\Mauro\work\books\B1\prj\look
gtkswing.jar - D:\Mauro\work\books\B1\prj\looks &
hippolf.jar - D:\Mauro\work\books\B1\prj\looks & fee
JavaGPE.jar - D:\Mauro\work\books\B1\prj\looks &
jlfgr-1_0.jar - D:\Mauro\work\projects\libs
liquidlnf.jar - D:\Mauro\work\books\B1\prj\looks & fe
looks-1.3.1.jar - D:\Mauro\work\books\B1\prj\looks
metouia.zip - D:\Mauro\work\books\B1\prj\looks & f
napkinlaf.jar - D:\Mauro\work\books\B1\prj\looks &
nextlf.jar - D:\Mauro\work\books\B1\prj\looks & feel
oalnf.jar - D:\Mauro\work\books\B1\prj\looks & feels
OfficeLnFs.jar - D:\Mauro\work\books\B1\prj\looks &
pagosoft-plaf.jar - D:\Mauro\work\books\B1\prj\looks & feels
```

Default output folder: c4/bin

Figure 4.19   The Eclipse 3.1 project properties dialog

Despite the fact that the GUI design device is almost identical, the two previous designs are different and should be kept distinct in the GUI to avoid confusion. A useful approach is to always stick to rigorous naming conventions: ‘properties’ are business-domain data, with one property dialog per business domain type, such as Person, Project, and Account properties, while ‘preferences’ are extra functional configuration data, with one preference dialog for the whole application. Designs should also be optimized for each user type: only repetitive users should need access to preferences, while properties dialogs should be made more easily accessible and usable, for example by providing contextual menu access.

4.6 Waiting strategies

Managing user interaction while tasks are being carried out by an application is a common issue in GUI design. Responsiveness, as we saw in the first part of this book, is an important feature in modern user interfaces. Just as with people, we have the feeling that a slowly-responding person is somehow unintelligent, and, false as it may be, we call them a ‘slow’ person. On the other hand, gadget-laden, baroque GUIs are no more usable than sober, plainer GUIs. The Java Look and Feel favors the latter approach both as a deliberate, wise choice, and as an undeniable practical necessity.

Java desktop GUIs – mainly J2SE, but also J2EE – may suffer from responsiveness problems. Indeed, competing platforms have listed this as a major drawback of Java GUIs. However, careful design and implementation can easily produce (relatively) snappy, responsive GUIs in Java. We will see some of the little details that can enhance the responsiveness of Java GUIs in this chapter. We will again consider both GUI design and low-level implementation details.

One of these techniques is quite effective when medium–long tasks must be accomplished, and turns out to be quite common and easy to implement in practice. A common problem is to inhibit user input during computation. A solution to this when using the Swing toolkit is to use the 'glass pane' component or similar methods to divert input events from the GUI, as it is temporarily unable to process them correctly. These can be neat technical tricks, but they often lack usability considerations and a sound cost–benefit balance.

A better solution would be to focus on communicating with the user, showing them the current application state. A modal progress dialog does the trick nicely: an example is shown in Figure 4.20.

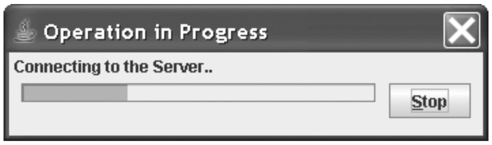


Figure 4.20 An example progress dialog (Ocean1.5)

This simple solution has a number of advantages:

i. It shows the user what is happening.
ii. It gives the user the option of canceling the process.
iii. As an aside, the modal progress window intercepts all the events directed to the underlying visual controls.

In practice there are many cases in which tight control over a task is not possible, for example a client–server connection to a Web service, where completion time is not known *a priori*. In this cases a simple solution is to provide an activity indicator only, using an 'indefinite progress bar,' as shown in Figure 4.21.

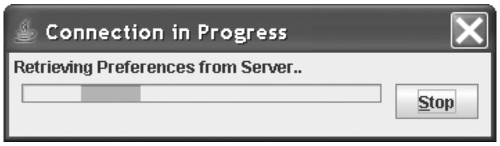


Figure 4.21 An example of a indefinite progress dialog (Ocean1.5)

Unfortunately, progress windows are not commonly seen in older Java GUIs, even in the simplistic arrangement proposed above, because of the cost of implementing them with low-level Swing components and the related threading infrastructure.

Things are simpler with SWT GUIs and the Eclipse RCP that provides a framework for supporting concurrent tasks. It is possible to choose between asynchronous (running in background) and synchronous (blocking user interaction until done) tasks, and this choice can be also offered to the end user, as shown in Figure 4.22, which is taken from Eclipse 3.1.

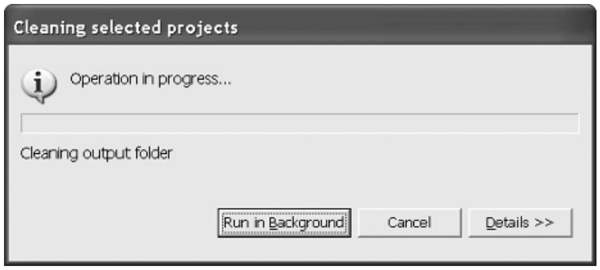


Figure 4.22 A progress indicator in Eclipse 3.1

Background execution basically uses the same implementation, but leaves users the ability to interact with the IDE while the task thread is running. When this is chosen, Eclipse 3.1 represent the task in the bottom right-hand side of the main frame in order to be less intrusive. Users can still interact with the task, stopping it, viewing details, and so on, by clicking on the button icon in the low-right corner, shown in Figure 4.23.

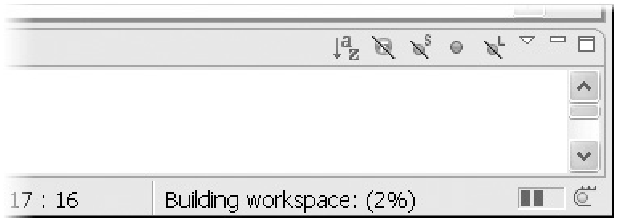


Figure 4.23 An Eclipse 3.1 progress indicator

The same indicator implemented with Flat Look is shown in Figure 4.24.

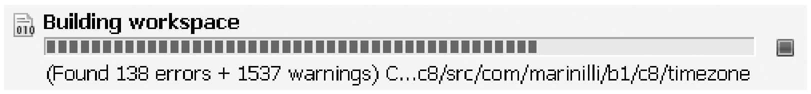

Building workspace
(Found 138 errors + 1537 warnings) C...c8/src/com/marinilli/b1/c8/timezone

Figure 4.24 An Eclipse 3.1 Flat Look progress indicator

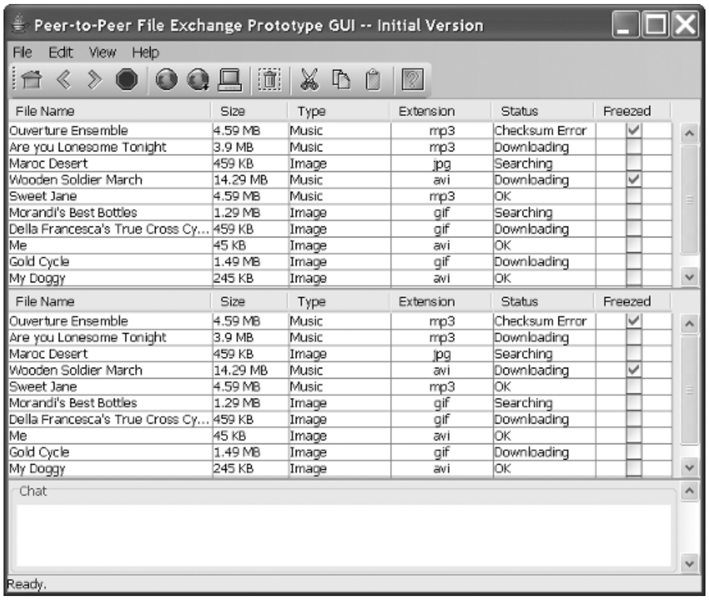
4.7 Flexible layout

Generally speaking, a well-designed window is first a usable one. Usability is frequently helped by the capability of resizing the window to enlarge it, or even to enlarge only a portion of it, according to the user's wishes.

Translating this into Java code means reconsidering our component layout philosophy. Usually designers tend to design 'static' windows, in which the widget visual organization is designed in a once-and-for-all fashion: such windows are easier and cheaper to design and build. Only for the main window or particularly critical windows is the layout is allowed to be variable, usually in the form of window resizing or using some `JSplitPane` here and there. This is easier than considering all possible user resizing needs or other related interactions, and also eases development.

Consider the fictitious GUI in Figure 4.25, which represents a hypothetical mock up for a peer-to-peer file exchange application. The first two areas list locally-available files and currently-exchanging ones. The bottom-most area represents a chat facility.

Consider the dynamic layout organization of the main frame. Allowing the window to be resizable is not a proper solution: the user might need to enlarging some of the internal lists, and this is not allowed by the design – its developers wrongly thought that a scroll pane would provide all the flexibility the user needed.

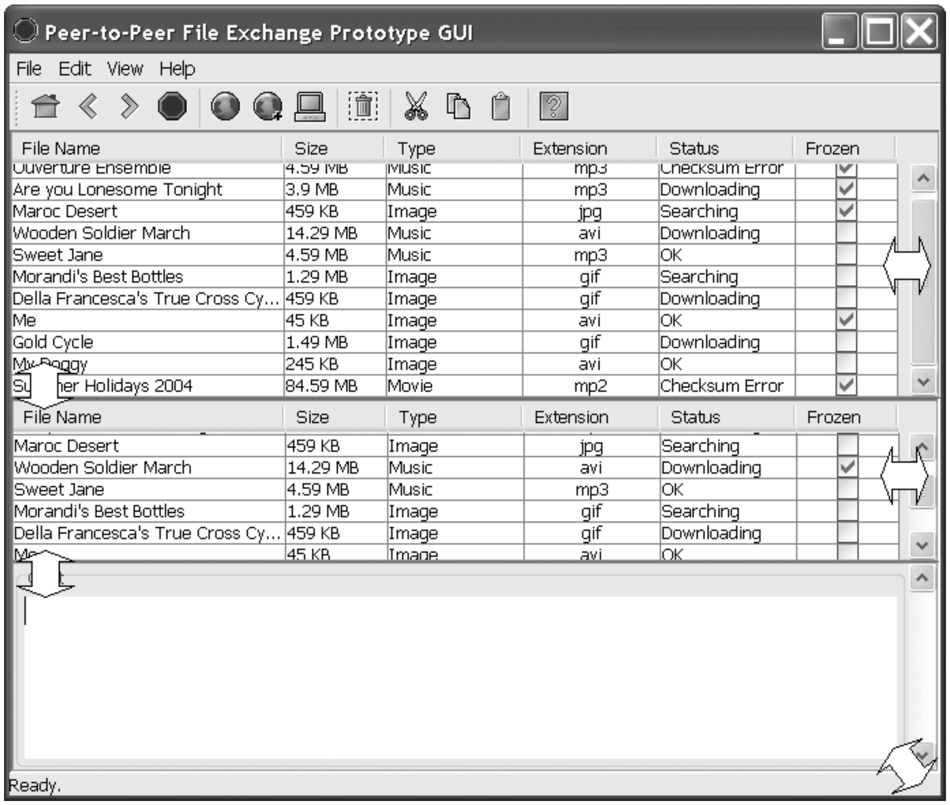


File Name	Size	Type	Extension	Status	Freezed
Ouverture Ensemble	4.59 MB	Music	mp3	Checksum Error	✓
Are you Lonesome Tonight	3.9 MB	Music	mp3	Downloading	
Maroc Desert	459 KB	Image	jpg	Searching	
Wooden Soldier March	14.29 MB	Music	avi	Downloading	✓
Sweet Jane	4.59 MB	Music	mp3	OK	
Morandi's Best Bottles	1.29 MB	Image	gif	Searching	
Della Francesca's True Cross Cy...	459 KB	Image	gif	Downloading	
Me	45 KB	Image	avi	OK	
Gold Cycle	1.49 MB	Image	gif	Downloading	
My Doggy	245 KB	Image	avi	OK	

File Name	Size	Type	Extension	Status	Freezed
Ouverture Ensemble	4.59 MB	Music	mp3	Checksum Error	✓
Are you Lonesome Tonight	3.9 MB	Music	mp3	Downloading	
Maroc Desert	459 KB	Image	jpg	Searching	
Wooden Soldier March	14.29 MB	Music	avi	Downloading	✓
Sweet Jane	4.59 MB	Music	mp3	OK	
Morandi's Best Bottles	1.29 MB	Image	gif	Searching	
Della Francesca's True Cross Cy...	459 KB	Image	gif	Downloading	
Me	45 KB	Image	avi	OK	
Gold Cycle	1.49 MB	Image	gif	Downloading	
My Doggy	245 KB	Image	avi	OK	

Figure 4.25 A not-so-flexible layout (Office2003)

A slightly more sophisticated design like the one in Figure 4.26 greatly enhances the usability of the application. Note that the two designs look pretty much the same from their (static) screenshots. It is in their dynamic behavior that the better quality of the second design becomes clear. Figure 4.26 shows this by using arrows.



File Name	Size	Type	Extension	Status	Frozen
Ouverture Ensemble	4.59 MB	Music	mp3	Checksum Error	✓
Are you Lonesome Tonight	3.9 MB	Music	mp3	Downloading	✓
Maroc Desert	459 KB	Image	jpg	Searching	✓
Wooden Soldier March	14.29 MB	Music	avi	Downloading	
Sweet Jane	4.59 MB	Music	mp3	OK	
Morandi's Best Bottles	1.29 MB	Image	gif	Searching	
Della Francesca's True Cross Cy...	459 KB	Image	gif	Downloading	
Me	45 KB	Image	avi	OK	✓
Gold Cycle	1.49 MB	Image	gif	Downloading	
My Doggy	245 KB	Image	avi	OK	
Summer Holidays 2004	84.59 MB	Movie	mp2	Checksum Error	✓

File Name	Size	Type	Extension	Status	Frozen
Maroc Desert	459 KB	Image	jpg	Searching	
Wooden Soldier March	14.29 MB	Music	avi	Downloading	✓
Sweet Jane	4.59 MB	Music	mp3	OK	
Morandi's Best Bottles	1.29 MB	Image	gif	Searching	
Della Francesca's True Cross Cy...	459 KB	Image	gif	Downloading	
Me	45 KB	Image	avi	OK	

Figure 4.26 A more flexible layout (Office2003)

Any component in the window can now be enlarged as required, greatly adding to the GUI's usability. The implementation of this enhancement came quite cheaply, as we used only two split panes to do the trick. The point here is in the *idea* of thinking of any of a GUI's window layouts as flexible ones.

Hence, thinking dynamically about the layout of windows is essential for quality design, and has a relatively low impact on their development. Considering the possible degrees of freedom of a GUI usually isn't a demanding operation.

Unfortunately, developers and designers tend to neglect this aspect, producing nice-looking but totally rigid windows that could be made much more usable with only a little additional effort. Systematically considering how to make your GUI flexible is essentially a change in design and implementation habits, but one that can greatly improve the quality of the resulting GUIs with only a little extra effort.

It is usually a good idea to make all dialogs – not to mention frames – resizable by the user. Unforeseen combinations of local and language locale settings, monitor resolution, and other factors can make your GUI unusable, even if it looks neatly designed in the development environment.

4.8 Common dialogs

GUI designers often tend to find themselves dealing with the same problems, such as showing information about their product, or notifying something basic to novice users. Over the years some design solutions have become consolidated in the industry. This section describes a few of the best-known, as implemented for the Java platform.

The 'About' dialog

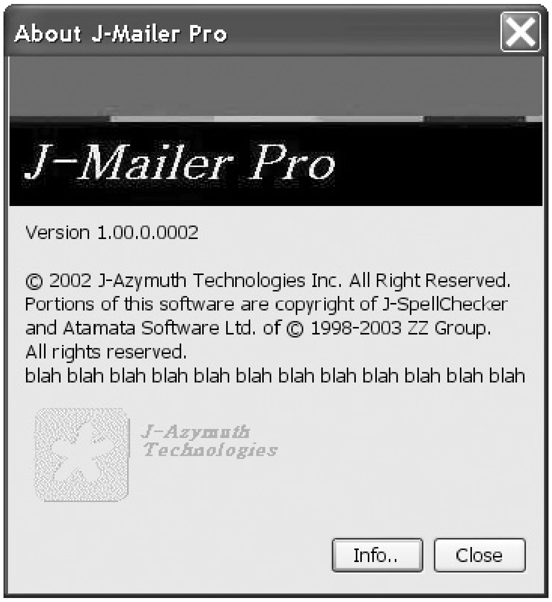
This is a very common feature of GUIs, where details of the application are shown. Such a facility isn't mere cosmetics, in that information such as the license data, the software version, or the list of the JAR files currently loaded, can be accessed by users. As prescribed by the Java Look and Feel design guidelines, this information is usually organized into two dialogs, one for the essential data, and a second with additional information.

Depending on the complexity of the application and – more importantly – the degree to which you want to make it visible to end-users, you may decide to show only a portion of such data in your 'About' dialog. Technologies like JNLP[6] solve many of the commonest debugging and deployment problems, so that showing too many details of your application in the 'About' dialog may not be really needed.

6. See the example application in Chapter 14

Main panel

The organization of the main panel is discussed in the official Java Look and Feel design guidelines. Figure 4.27 shows an example of an 'About' dialog.



Version 1.00.0.0002

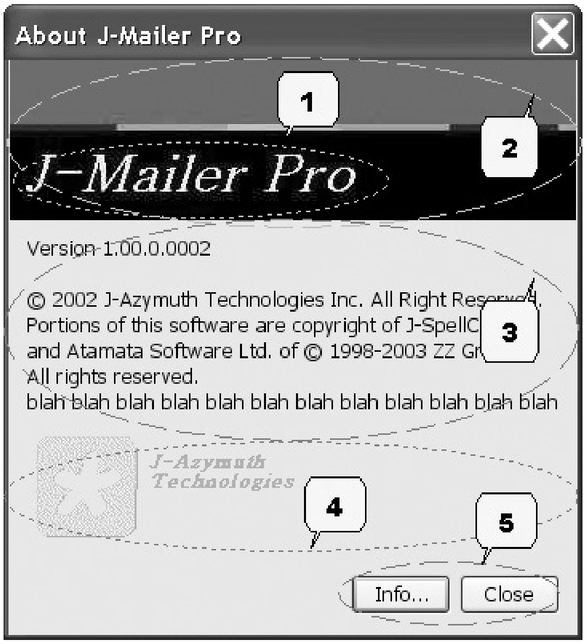
© 2002 J-Azymuth Technologies Inc. All Right Reserved.
Portions of this software are copyright of J-SpellChecker
and Atamata Software Ltd. of © 1998-2003 ZZ Group.
All rights reserved.
blah blah blah blah blah blah blah blah blah blah blah blah

Figure 4.27 An About dialog example (Hillenbrand Windows)

Exploiting Web visual conventions, some areas of the 'About' dialog, such as manufacturer's information, may be made clickable like a link on a HTML page. The 'About' dialog in the example application in Chapter 15 shows such a technical trick at work. Here however we prefer a plain implementation to introduce the issue.

The visual organization of the main information panel, as prescribed in the Java Look and Feel design guidelines, is shown in Figure 4.28. The highlighted areas are: (1) product name, (2) dialog banner, (3) text information, (4) company logo, and (5) interaction buttons.

The way to reach the additional information dialog is suggested in the form of a single **Info...** button in Figure 4.27 and Figure 4.28.



Version 1.00.0.0002

© 2002 J-Azymuth Technologies Inc. All Right Reserved.
Portions of this software are copyright of J-SpellC
and Atamata Software Ltd. of © 1998-2003 ZZ Gr
All rights reserved.
blah blah blah blah blah blah blah blah blah blah blah blah

Figure 4.28 JL&F About dialog main panel organization (Hillenbrand Windows)

Additional info panel

A few words about the additional information panel, displayed prompted as a separated modal dialog when the **Info...** button is clicked, are relevant.

You aren't obliged to provide an additional information dialog in your 'About' dialog, as long as the data you need to show can be neatly accommodated in the main panel. If required, information stored in the additional information dialog can include:

- JAR files listing, each with its dimension and exact version.
- Java system properties, such as the current JRE used, the heap size, the locale, and so on.
- The version of the application.
- Some of the more important application-dependent configuration data.
- Other configuration data, such as the version and type of some of the Java extensions currently used.
- External modules, required applications, third-party libraries and the like.

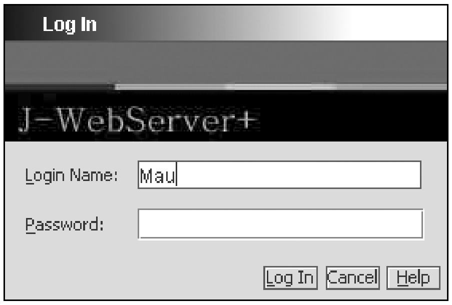
This data is usually organized in tables and labels ordered by means of a `JTabbedPane`.

A general-purpose implementation of an 'About' dialog component that complies with the Java Look and Feel design guidelines is provided with the code bundle for this chapter. Our class implementation offers many constructors: you can specify the parent frame, the additional information dialog, the main image, and the company logos. When the empty constructor is used, the dialog is instantiated with a set of default values documented in the source code for that class.

Log-in dialog

Some applications need to identify specific users before they are granted access to the full functionalities of the GUI. This is usually done by means of an authentication phase, in which the user is requested to insert a log-in name and a password. An example might be a thick client application that needs to access sensitive data on the server, or a personalized application that has been tailored to a particular user.

In such cases a log-in dialog is shown. The Java Look and Feel design guidelines prescribe principles for the design of such dialogs. Figure 4.29 shows an example of a standard Java log-in dialog.



Login Name: Mau
Password:

Figure 4.29 An example of a log-in dialog (Tonic)

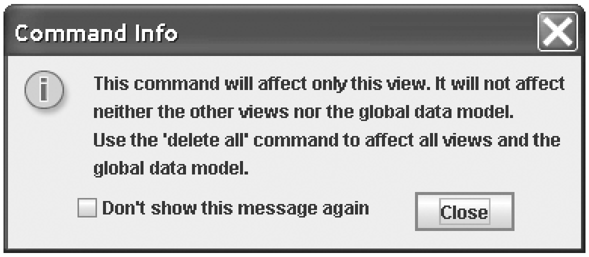
Simpler dialogs, such as those without a product header at the top of the dialog, can also be provided, but it is always a good idea to make the identity and the purpose of the authentication phase explicit to the user, providing a recognizable indication of your application.

First-time message dialogs

The first time an operation is performed, inexperienced users might need to be reassured about the GUI's internal state, to answer the mental question 'what is going to happen now?' This can be done neatly by using message dialogs that describe the operation that is about to be performed, or that has just been performed. Allowing these dialogs to be shown only when required avoids annoying the user in subsequent sessions and makes the application more usable.

Figure 4.30, Figure 4.31, and Figure 4.32 show three examples of this kind of explanation dialog:

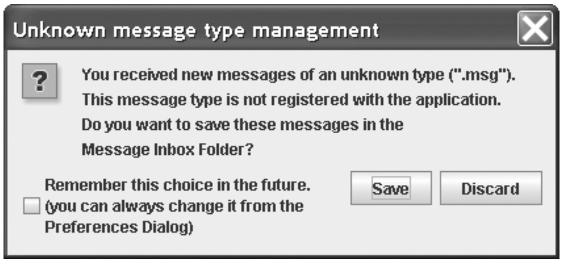
- The dialog in Figure 4.30 notifies the user of the consequences of an operation they have performed.



This command will affect only this view. It will not affect neither the other views nor the global data model.
Use the 'delete all' command to affect all views and the global data model.

Figure 4.30 An example of a first-time only explanation dialog (Ocean1.5)

- The dialog in Figure 4.31 allows the application to both acquire an answer from the user and to avoiding asking the question again.



You received new messages of an unknown type (".msg").
This message type is not registered with the application.
Do you want to save these messages in the Message Inbox Folder?

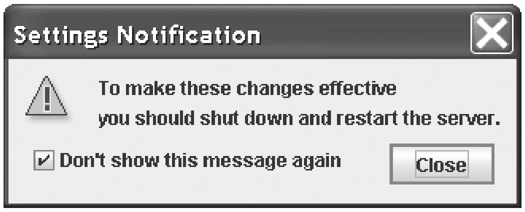
Remember this choice in the future. (you can always change it from the Preferences Dialog)

Figure 4.31 A first-time only explanation dialog (Ocean1.5)

In Figure 4.31 the dialog also explains where to find the option even when turned off. Sometimes insecure users avoid switching off a feature, fearing that they won't be able to restore it easily in the future. If wisely employed, such little details greatly increase the overall usability of a GUI.

- Finally, Figure 4.32 shows another example of a first-time dialog, in which such a facility is used to warn the user explicitly about the effect of the operation they have performed.

First-time dialogs can be implemented easily using simple memory components.


Settings Notification

To make these changes effective
you should shut down and restart the server.

☑ Don't show this message again

Figure 4.32 Another first-time only explanation dialog (Ocean1.5)

Splash window

Another commonly-used window in non-trivial GUIs is the screen that appears during application start-up. This window entertains the user during start-up and informs them of what is going on while waiting for the application. The Java Look and Feel design guidelines suggest a way to organize the visual appearance of a splash window.

A splash window is a good place to show an application's identity. A GUI compliant with the Java Look and Feel design guidelines doesn't loose its identity – rather, it becomes more usable and recognizable by users. The splash window is one of the correct places to put your application's 'personal' touch, so is important to not to waste such a chance.

An example is shown in Figure 4.33.

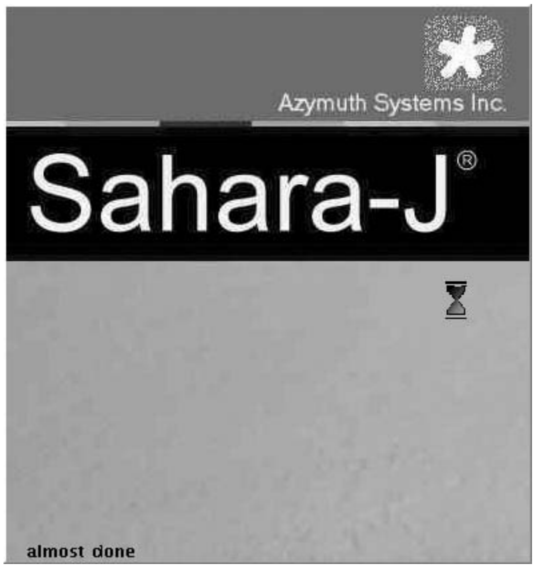


Figure 4.33 Splash window example

In this case you might find it interesting to look at the source code. This consists of a reusable yet simple class that provides all the functionality for a splash window. It provides a way to set up the static image shown in the window, the text in the message label at the bottom, and a mechanism to hide or show it as needed. The `SplashWindow` class is provided in the code bundle for this chapter.

4.9 Command components

This section discusses GUI components for managing user commands.

Menus, toolbar buttons and other means of asserting commands are an important part of a GUI. It is therefore no wonder that the Java Look and Feel design guidelines describe how such components should be organized in detail. We will give some examples here.

Figure 4.34 through Figure 4.38 show examples of a fictitious application that adopts the official guideline's suggestions for menu organization. As with all the other examples, you can run these on your own computer. Apart from menus, Figure 4.34 shows the use of command palette internal frames, which can be used in a multiple document interface (MDI) environment.

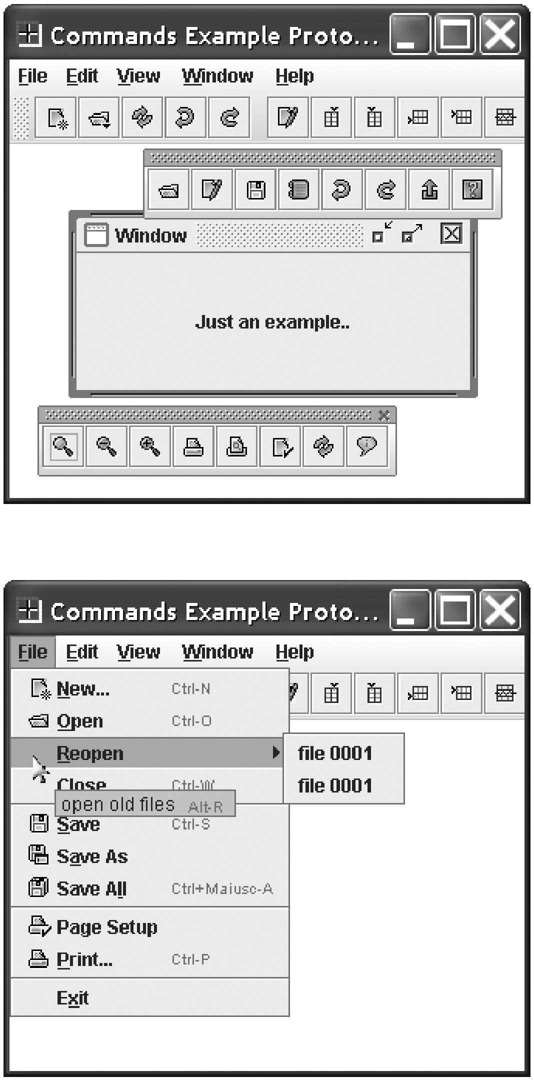


Figure 4.34 Examples of various command components (Ocean1.5)

Let's focus on menus first. The Java Look and Feel design guidelines prescribe a suggested structure for common menus, like **File**, **Edit**, and **Help**. Figure 4.35 shows the **File** menu. When the same commands are available through a toolbar, it is customary to associate a unique icon to the command to make it more recognizable by the user.

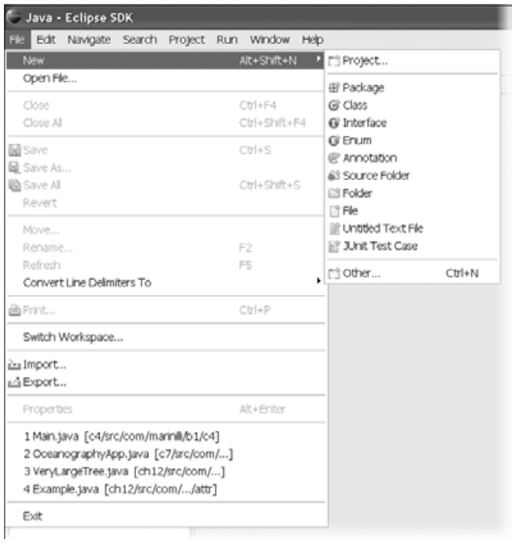


Figure 4.35 Eclipse menus

The standard **Edit** menu is illustrated in Figure 4.36 for both Swing and Eclipse.

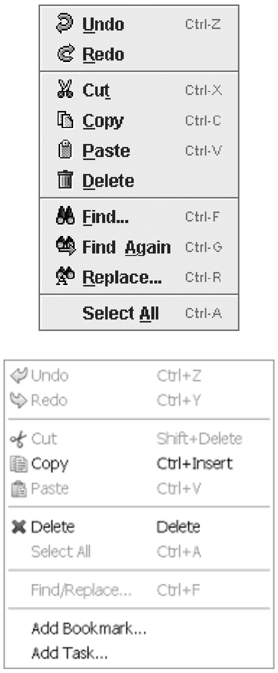


Figure 4.36 The suggested Edit menu organization for Swing (Metal) and Eclipse

Figure 4.37 illustrates an example of the **View** menu that employs radio button menu items for selecting the application's icon size.

When information is accessed only infrequently, as in the case of the icon size for the application shown in Figure 4.37, the information can be put in a global configuration (preferences) dialog instead of directly in a menu.

Another example of menu organization is shown in Figure 4.38, which is taken from a fictitious graphics application.

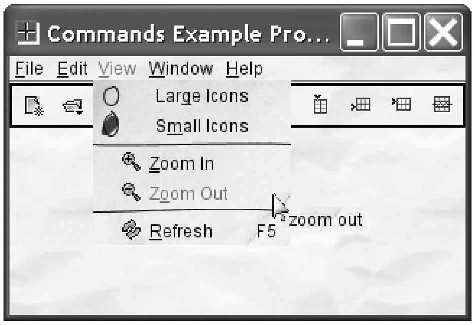


Figure 4.37 An example of view menu organization (Napkin)

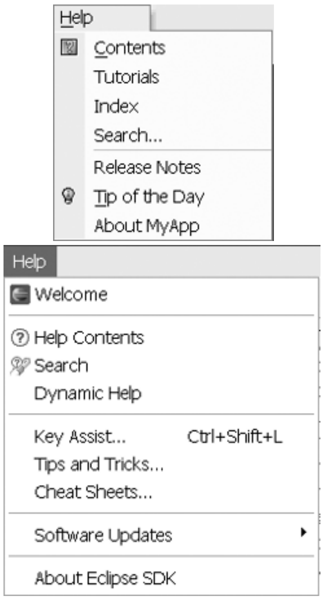
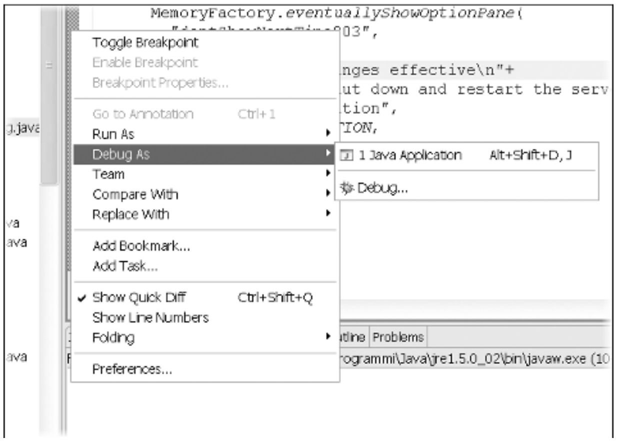


Figure 4.38 Help menu suggested organization for Swing (OfficeXP) and Eclipse

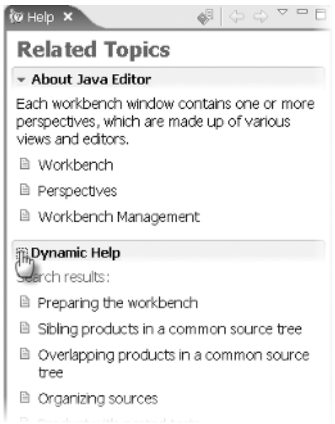
Contextual menus are another important category of menu, one that should be made available for medium–large, non-form based applications, like that shown in Figure 4.39.



```
MemoryFactory.eventuallyShowOptionPane(
nges effective\n"+
ut down and restart the serv
tion",
TION,
```

Figure 4.39 An Eclipse contextual menu

Expandable menus are a menu variant that is supported natively by SWT, as shown in Figure 4.40, which is also available for Swing through third-party libraries. Clicking on the title minimizes or expands the menu as required.



Each workbench window contains one or more perspectives, which are made up of various views and editors.

Figure 4.40 An expandable/collapsible menu

Graphic conventions

A number of conventions are adopted in SWT and the Java Look and Feel design guidelines. The latter provides four standard adornments for expressing common functionalities in button icons, as shown in Table 4.1.

Table 4.1 Graphic conventions for Java L&F button icons

Indicator name	Use	Example icon
Drop-down menu	A pop-up menu appears when clicking the button	🔈
New object	A new object of the given type is created following the current GUI metaphor	📄
Add object	An object of the given type is added following the current GUI metaphor	📄
Properties	Prompt a property/setting window for that object	📄

The Java Look and Feel design guidelines describe the graphics for any of the adornments in Table 4.1 in full detail. The guidelines also warn designers about mixing two or more indicators in the same icon. Take the case of a 'new item' button that brings up a menu with a gallery of items available for creation. We should use both 'new object' and the 'drop down menu' indicators. We can slightly modify the button interaction to use only one indicator by resorting to an object gallery dialog that will work as the pop-up menu, leaving only the 'new item' adornment to the toolbar button. When the user clicks the button, a dialog appears that allows them to choose the type of new objects they want to create. In this way the button can be left only with the 'new' indicator.

An interactive example of the use of the button graphical indicators is provided with the code for this book, and shown in Figure 4.41.

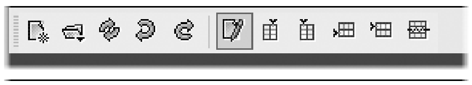


Figure 4.41 Examples of button indicators at work for the Swing L & F

Apart from the standard adornments, there are a number of other common graphical designs for buttons. For example, customizing the main window areas is a common task. You can use a toggle button for collapsing unneeded window areas. For more details, see (Java L&F Design Guidelines 2001).

Toolbar composition

Toolbar creation is a topic necessarily involves implementation considerations. From a software design viewpoint, the toolbar composition is mainly a creational problem. The issues with which developers are often concerned are how toolbars are assembled, and from where the commands are obtained.

We provide various implementation strategies for command management throughout this book that can be employed in a wide range of situations. Problems arise when the application needs to support dynamic toolbar composition. There are of course several different levels of features that can be supported. Loading new commands when new modules are plugged into the application can be achieved via JNLP technology, and does not require any special programming such as reading commands from properties files and the like. As the JNLP protocol becomes more popular and widespread, its more advanced features, such as the JARDiff format, which allows downloading of only the portions of the JAR files that change from version to version, can ease the development of this kind of feature, not only for mounting new modules, but also for applications updates. The more popular solution, though, is to employ a plug-in architecture.

Another typical toolbar feature is enabling users to customize application toolbars. We briefly touched on the issue of user customization in the section about memory components. Conceptually, menus act as a predefined and logically-organized repository of the available commands for an application, while toolbars are often left to the user to customize: they may contain a subset of all possible commands, or can be even switched off completely by the user.

Some software designers like to use more sophisticated mechanisms for toolbar creation, based on negotiation protocols between the GUI builder and the objects that are publishing their functions via the GUI. We will not discuss such architectures here, especially because they tend to needlessly complicate the class architecture and weigh it down at application start-up – which, as we have seen, is a critical issue for Java applications. Usually a thoughtful and neat class design can provide many such features without sacrificing runtime performance.

Command composition

Several implementation considerations affect GUI design.

Contextual menus are a useful way of organizing user interaction. The underlying implementation should be taken into account for cost-aware, professional GUIs. One common issue is the gathering of commands from different GUI items into one menu. The user is not aware of such composition, but this mechanism has several benefits:

- It organizes the menu commands in more rational groups or hierarchies.
- It allows for an elegant mapping into an OOUI and the OO implementation of the designed GUI. Each item is mapped into an OOUI object, then into

several Java objects. The complexity of the GUI is divided into smaller, coherent pieces, each one exposing some specialized commands.

- The creation mechanism of complex menus is made systematic and general in order to be extensively adopted in a wide range of GUIs.
- It establishes a standard, general logical division between the responsibilities of complex commands, possibly involving several objects. Many commands, belonging to different objects, can be composed together in a unique menu. They are kept separated in compartments by separation lines within a menu for a clarity.

Command composition is not merely the gathering of available commands from each of the relevant GUI objects. The most general scenario involves the negotiation of commands among the classes involved. In fact, some commands may be not applicable in the given context – for example, an administrative user often has more commands available than normal users – and they may not even appear in the menus, or some commands may depend on the interaction of several objects, and so on.

An example of this latter case might be a list of items: depending on the current selection, the contextual menu can show selection-dependent commands. In a file listing window, for example, when selecting all image files, a **Create animation** menu item could be included in the pop-up contextual menu.

Figure 4.42 shows another example of this technique, in which the Eclipse GUI requests all its loaded plug-ins to provide their available views.

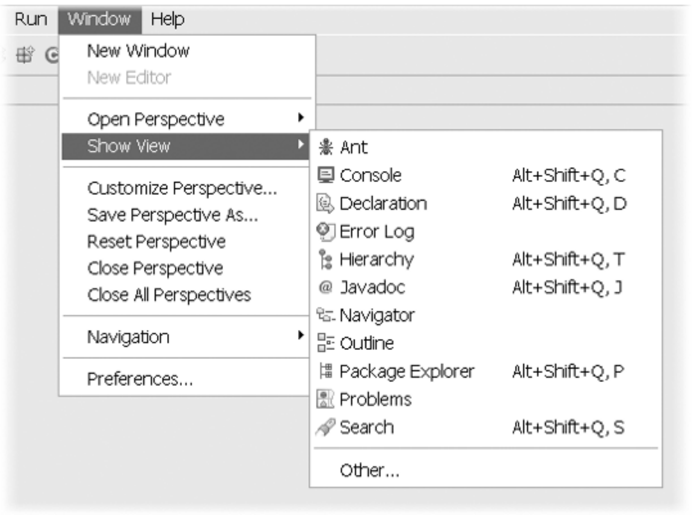


Figure 4.42 Eclipse 3.1 Example of command composition

From a programmer's viewpoint, in such complex cases it may be useful to employ a Java class devoted exclusively to negotiating the commands, populating the pop-up menu, and executing the more complex, cross-objects commands that are often needed.

4.10 Accessibility

Software accessibility is now legislated for in the USA and some other countries. It is an important commercial market, and supporting *assistive technologies* in Java is quite easy. When designing a GUI, the following four main disabilities should be taken in account as a minimum:

- Color blindness
- Partial or total deficit of vision
- Partial or complete lack of hearing
- Partial or total absence of mouse and keyboard use

Other more complex disabilities (including cognitive ones) exist, but we do not cover them. The interested reader can refer to some of the URLs provided at the end of this section.

Designers should prepare their application for interaction with external assistive technology tools, such as screen magnifiers. This is only one side of the coin, however. The GUI should be made highly customizable for fonts, their size, colors, and so on. Color-blind users can need color combinations that may seem strange to others, while users with impaired vision might require unusually large fonts, and so on.

As a default, JFC applets and applications – and with some limitations, AWT ones as well – use the settings from the underlying environment. Fonts, their sizes, system colors, and other settings are therefore inherited automatically, as long as the application does not explicitly set them.

When implementing ad-hoc components[7], Swing developers implement the `Accessible` interface, which provides the core of accessible data that is used by assistive technologies. Naturally, Eclipse support for accessibility is provided as well.

7. See Chapter 16.

Testing the final product for accessibility

No matter how diligently accessibility is designed into an application, the final test is its use by users with real disabilities. Although the next chapter covers GUI testing, there are some practical considerations worth mentioning here.

Firstly, we need to test the application for keyboard support without the mouse (you could even take it away). This allows an application to be tested entirely via keyboard: we need to verify that all the parts of the GUI remain accessible using only the keyboard. Usability should be verified as well – shortcuts, mnemonics, accelerators, and so on. Colors and fonts settings can be tested by choosing a large font size, say 24 points or more, and verifying what happens to each window in the application.

A special Look and Feel class is available from the Sun Web site for Swing that is designed for low-vision users. A GUI can also be tested with external assistive tools such as IBM's Self-Voicing Kit for Java.

Conclusions

This is only a brief discussion of accessibility in Java GUIs. There are many useful resources on the Web: IBM provides an excellent source of material on this issue, as does Sun's Web sites:

`http://www-3.ibm.com/able/guidelines/software/accesssoftware.html`

`http://www.sun.com/access/developers/developing-accessible-apps`

Many other resources on this important issue are available on the Internet.

4.11 Navigation and keyboard support

Navigation is the flow of control from one window to another. This section discusses navigation between elementary widgets. Navigation between screens has been touched on in various parts of this chapter, such as the discussion about wizard design, and will be discussed in depth in Chapter 9 in the section on Web user interfaces.

Keyboard support for command selection is essential in usable GUIs. Experienced users tend to use quicker ways of performing the same operation as they become knowledgeable with an application. The keyboard is a good way to shorten interaction times for expert users, as it doesn't make the application more complicated for novices.

Keyboard shortcuts

The JFC library provides a complete set of tools for handling keyboard input at various levels of abstraction. We won't get into programming details here, but it is important to consider these features when designing a GUI.

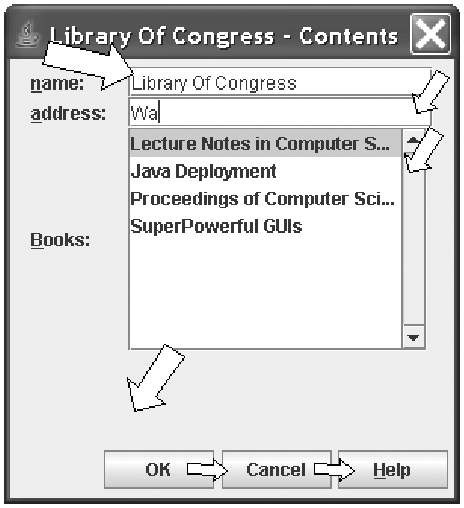
A useful feature for enhancing the navigability of your dialogs is to provide a default button that is activated when the user hits the **Return** key. The Java Look and Feel will signal this special button, as shown for example by the **Close** button in Figure 4.30. Support for the **Escape** key is also widely used in dialogs, whenever appropriate.

Keyboard support should be designed while bearing in mind that it will often be the main support for repetitive users. As such, it should be employed to cover all the application's functionalities, even though critical ones where data can be lost, such as **Delete**, or closing a dialog, shouldn't provide keyboard shortcuts.

Tab traversal

The tab key is used for moving the focus between components in a window. By repeatedly pressing the tab key, users can navigate through all a window's components. Designing the correct traversal sequence enhances the usability of the GUI for those users that take advantage of keyboard support.

This feature is especially important for windows that are used frequently, such as data input forms. The default sequence is dictated by the order of component's addition into the window, as in the code listing, and can be modified explicitly by the focus framework provided in J2SE 1.4 and subsequent versions. An example of tab traversal in a simple dialog is shown in Figure 4.43 – the arrows indicate the movement of the focus for repeated presses of the tab key.



Library Of Congress
Wa
Lecture Notes in Computer S...
Java Deployment
Proceedings of Computer Sci...
SuperPowerful GUIs

Figure 4.43 Tab traversal in a dialog

Tab traversal is a form of keyboard support, and as such it follows the general rules discussed here and in the Java Look and Feel design guidelines.

4.12 Internationalization

The design of applications suitable for a global marketplace, referred to as *internationalization*, and the related topic of customizing an existing application for a given locale, *localization*, are important issues in GUI design.

The cost of localizing an application can be roughly thought of as the sum of the development costs of the required infrastructure, plus the required messages translation. For this latter cost, (Maner 1997) indicates a sum of between $0.25 and $0.75 per word. For Java applications, however, such figures are usually an over-estimate – thanks to the Java internationalization architecture, the translation process can be accomplished cheaply, for example by sending the relevant text files to be translated by a suitable localization company.

The key point is the provision of technical support for internationalization. Even if it is not planned to distribute the application in different countries, it is a good idea to consider the internationalization issue from the start of the GUI design process. Unfortunately, for effective localization, it is not enough to provide different translation files and a sound software design that supports external resource bundles. Apart from the software architecture, the following factors for international GUI design should be considered as a minimum:

- Translating messages and any other textual data, such as mnemonics, accelerators and help data, by means of properties or other support files.
- Other Java-specific technical facilities, such as input frameworks, good-quality font sets, and so on.
- Flexible layout, which is essential to accommodate labels, buttons and other text-based widgets in different languages.
- A thoughtful design of the general interaction style, to be as culture-neutral as possible – a loquacious GUI that displays many information messages can be viewed as polite in some cultures and arrogant in others.
- Specific cultural issues, such as:
  - Images, colors, sounds and other graphics conventions: icons, images and other locale-sensitive data references can be put in resource bundles so that they can be easily localized.
  - Currency, units of measurement, and any other number formats.
  - Various conventions such as date formats, phone numbers, salutations, and so on.

- Cultural issue in general. This is a complex problem, and involves the help of specialists in the target culture. A large number of 'cultural' accidents can be found in commercial GUIs. Some are unimportant, such as a progress bar that starts from the left in a country in which text is written from right to left, but others are more serious. Even some apparently neutral associations like using a Red Cross logo, for example, can be found offensive in some non-Western cultures.

Using resources bundles for all the relevant resources (icons, text messages, and the rest) can also be useful even if an application is not planned for internationalization, as it allows all messages, icons, and other resources such as audio clips to be polished more easily, by non-programmers if necessary.

A problem arises on platforms with different locales. From J2SE 1.4 onwards, multilingual support covers standard JFC components such as the file chooser dialog. This engenders the risk of providing users with fragmented multilingual GUIs, for example with the main frame in English and other standard dialogs in the application's current language. As a work-around, the locale can be overwritten or labels can be set explicitly by developers, although this latter practice results in a hack rather than a disciplined design.

It is always good practice to consider internationalization issues in the first place when designing a GUI. This involves not only providing a flexible and dynamically-adjustable layout to handle text of unforeseen dimensions, or other technical tricks, but also to rethink icons, interactions, and even GUI concepts from a multiculturally-aware perspective. Daunting as it may seem, such a task is well repaid in the long run. The cost of localizing an already-developed application from scratch is always much greater than the effort of designing it and testing it for usability with internationalization in mind. Even if internationalization is not foreseen in the near future, a preemptive minimal internationalization-aware design, for example implementing global icons, flexible layouts, and text files for messages, is always a wise choice.

4.13 Help support

J2SE ships with a library for full client-side help support. The JavaHelp library is an example of this kind of support, which provides context-sensitive help of two types: user-initiated and system-initiated. User-initiated help can be activated in four different ways:

- By pressing the **F1** key it is possible to display the help data about the container that currently has the focus. This is called *window-level help*, as it is recommended for use only in windows, frames and dialogs.
- After clicking the contextual help button, usually in the toolbar, or choosing it from the **Help** menu, the mouse cursor changes to a special contextual help

cursor. This signals that the program is waiting for the selection of an item in the GUI, using the mouse or the keyboard, when the contextual help available for the selected object is displayed. This is referred to as *field-level help*.

- By using the standard **Help** menu in the menu bar. This can be used to provide help about specific tasks or objects. The **Help** menu contains a submenu of items that provide help about various tasks.
- In dialog boxes via a **Help** button. This provides help information about how to use the dialog. Clicking **Help** is usually equivalent to pressing the **F1** key while the dialog box has the focus.

System-initiated help is performed by the program itself reacting to some user action that is not explicitly related to help commands.

Help support can be useful both in prototype building and GUI extension. In a pre-release version for a selected user population, some of the functionalities to be added can be explained in the help system. By default, help information is displayed in the help viewer, but this can be customized as needed.

Other libraries also exist that provide help support, both for Swing and SWT applications, providing a different mix of runtime performances, simplicity, and range of available features.

4.14 Icons and images

A number of bitmap images are usually employed when creating a GUI with Java technology. Table 4.2 lists the most frequent ones. Designers should provide these images.

Table 4.2 Common images for swing applications

Description	Use	Size
Log-in app logo	Shown in log-in dialogs	~ 280 x 64
App icon	Shown in app frames and dialogs	small: 16 x 16 large: 24 x 24
‘About’ app logo	Used in the ‘About’ dialog	~ 280 x 64 or greater
Company logo	Appears in the ‘About’ dialog	
Splash window	Startup splash window	~ 392 x 412
Toolbar icons	Toolbar buttons	small: 16 x 16 large: 24 x 24
Other app-dependent graphics	Depends on the application	

The image sizes preceded by a '~' sign are merely illustrative.

We will provide a number of practical examples throughout the book. Chapter 14 discusses a complete application where all these images are instantiated for a real case.

4.15 Leveraging object-oriented programming

Reusability of software components tends to produce better quality GUIs, because behavior and appearance are replicated in a coherent way throughout the whole interface, and coding effort is saved. OOP reusability is a key point for high-quality inexpensive Java GUIs.

A common, concrete case is provided by the fact that some dialogs are served in two main modalities that depend on how the user's actions are recorded by the application: *deferred* or *immediate* mode interactions. Some GUI design guidelines prescribe dialog appearance. It is possible therefore to envisage a small component that implements the area where buttons are displayed. Such a widget is shown in Figure 4.44 and Figure 4.45 for a typical deferred interaction dialog in which changes are committed using the **OK** button, or dismissed by means of the **Cancel** button.

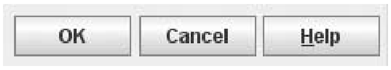


Figure 4.44 The OKCancelPane component for Java L&F

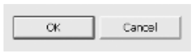


Figure 4.45 The OKCancelPane component for Eclipse

A **Help** button could optionally be provided as well – in Swing GUIs this is officially mentioned, but not in the Eclipse guidelines.

The practice of adopting customized, reusable components is very useful. The next logical step is to provide a deferred-mode dialog that can be used every time you need to perform such an interaction in a GUI. A simple component might contain an `OKCancelPane` such as the one shown in Figure 4.44, as well as some other standard behavior, such as being sensitive to the **Escape** key to dismiss the dialog, or automatically visualizing the help data when the **Help** button is clicked. This is provided out of the box by the Eclipse GUI libraries.

No matter which mechanism you use to assemble GUI Content[8], the idea is to engineer this activity in a coherent way, so that the final user experience will be uniform and predictable throughout the whole GUI. A small investment in development time in implementing such basic facility will be repaid many times during software development and in the final, systematic aspect of the GUI.

One flaw in this approach of employing only few, highly customized components lies in visual components provided by someone else. This shouldn't be a problem, because GUI design guidelines nicely dictate all GUI details. Unfortunately third-party vendors sometimes tend to ignore such prescriptions, especially in older products. Such incompatibilities are being resolved over time with the Swing library – at least as long as the latest versions are used. For third-party GUI libraries, be careful to check out their design guideline compliance before adopting them in your project. All your development effort can be wasted if you provide your customers with an inconsistent user experience, no matter how elegant the underlying software implementation.

The Swing implementation of the `OkCancelPanel` class is provided for readers that are interested. This provides global action buttons as prescribed by the Java Look and Feel design guidelines, and should be used extensively throughout the GUI, enforced by quality assurance if necessary. For usability reasons the appearance of the **OK** button may be changed in some cases. For example, in a **Print...** dialog it makes more sense to label the **OK** button with **Print** even if the underlying function remains the same. For the same reason the range of possible customization of this panel is limited. No icons should be used for the buttons, and the **Cancel** and **Help** buttons, although locale-dependent, cannot be arbitrarily labeled.

4.16 Summary

This chapter introduced some common design problems, together with their solutions for effective Java GUI design and subsequent development, and occasionally considered implementation issues. The approach was aimed at highlighting some often overlooked issues in GUI design, with particular relevance to the Java platform. Some of the issues were too broad to be addressed exhaustively in this chapter.

In particular, the chapter discussed:

- Window area organization, including some widely-accepted and used criteria for organizing the functional areas of a non-trivial GUI.

8. We discuss the main implementation alternatives available briefly in Chapter 6, in *Content assembly* on page 229.

- Choosers, including the preferred activation mechanism for choosers, and how to expand them to handle other features such as item creation. Choosers were also used for discussing the different types of dialog interaction: deferred, immediate, and mixed.
- Memory components, visual components that have a subset of their state made persistent.
- Lazy instantiation – complex Java applications can become excessively slow in some situations. Mixing design and implementation can substantially boost performance.
- The preference dialog, a common design: a centralized access point for configuration data is needed in all but the simplest applications.
- Command composition. Negotiating commands is a common practice in GUIs implemented with OOP, especially for OOUIs.
- Wizards. Although relatively easy to implement, wizards should be used only when needed, although they are a useful tool in a designer's toolbox.
- Waiting strategies, providing sound designs for situations in which the GUI is performing internal work and is currently unresponsive.
- Flexible layouts. It is not enough to provide scroll panes for the main components and a resizable window for the container of the dialogs or frames of your application.
- Common dialogs and windows – in current GUIs there are many de-facto standard windows and dialogs. We proposed only few of them with some examples, both to show their suggested design, and to provide a utility library that eases their development.
- Menu and toolbar organization, important and frequent design issues.
- Accessibility – it is always good practice to provide accessibility support in your GUI.
- Navigation and keyboard support – providing a planned keyboard support for any dialog or frame in your application is good design practice.
- Internationalization and localization, important aspects of modern GUIs that should be considered from the start of GUI design.
- Help support – integrating help support into an application using the JavaHelp library.
- Common icons and images.

We also discussed proposed design solutions, providing some practical examples that highlighted the main advantages such architectures provide.

In the second part of the book we will leave GUI design and move to the implementation aspects of professional Java GUIs.

5 Iterative GUI Development with Java

No design is ever perfected at the first attempt. Instead, a professional design in many engineering fields is the result of several refinement cycles. This is true for software engineering in general, and is even more true for GUI development, where the presence of end users makes the engineering task highly unpredictable and dependant on subjective criteria. In this chapter we will examine the major approaches and the available techniques for building professional Java GUIs through iterative cycles of refinement.

The iterative GUI development approach consists of frequent product releases that continuously and smoothly expand the application by means of small additive changes, implementation refinements (such as refactorings) and continuous, pervasive testing. Testing 'in the large' is essential for achieving an effective iterative development. We will discuss GUI testing, usability testing and memory profiling, an often overlooked aspect of GUI development.

Readers are not forced to adopt an iterative development approach if they don't want to. Despite being a powerful development approach – see the discussion in Chapter 1 – it is labor-intensive, involves mastering many techniques, and ultimately leads to good and cost-effective results only when developers genuinely embrace its philosophy. Nevertheless, the techniques discussed in this chapter can be applied to a wide range of software engineering approaches, ranging from XP (Extreme Programming) to traditional waterfall development.

Iterating a GUI design that has already been exposed to end users is a delicate art, requiring skill, as well as a different attitude to that required for software refactoring. As we saw in Chapter 2, to a user the GUI *is* the application. As the most externally visible part of a system, the user interface tends to evoke strong feelings. Once a GUI design has been agreed, the process of changing it is often complex and politically charged. Evolving a GUI design from one iteration to the next can put a strain on end users. Users learn the application through the GUI, and even minor refinements can be unpopular once familiarity is established.

One of the advantages of iterative development is the possibility of constantly evaluating and changing the application using end users. Without end users and domain experts working with developers on a GUI there is little possibility of progress – at most we are developing a nice, abstract application that probably doesn't solve actual users' needs, just the needs of our fictitious idea of end users.

This chapter begins by introducing the fundamental strategy behind effective iterative development, followed by an introduction to Java GUI prototyping. Various aids to prototyping are introduced as well (GUI builders and some examples of utility prototyping classes). After an initial and inexpensive prototype has been assessed with users, iterative development will take care of evolving the application to meet user's needs. Common GUI-specific refactorings are discussed together with testing and runtime memory profiling. This chapter covers all these heterogeneous aspects, to provide a unique reference for iterative GUI development, spanning diverse topics such as prototyping, refactoring, testing, and profiling.

This chapter is structured as follows:

5.1, Iterating wisely discusses the strategies behind iterative GUI development.

5.2, Introduction to prototyping deals with the basic concepts for the design of effective GUI prototypes.

5.3, Prototyping alternatives discusses the various approaches to prototyping available, such as paper prototyping, storyboarding, and so on.

5.4, GUI builders introduces this kind of tool, useful for prototyping as well as for building final GUIs.

5.5, Reusable prototyping widgets discusses some widgets specialized for prototyping purposes, along with their implementation.

5.6, GUI refactoring illustrates the practice of refactoring GUI code, going into the details of GUI-specific refactorings.

5.7, Introduction to user interface testing introduces the general topic of GUI testing, focusing on some of its most controversial aspects.

5.8, Software testing of Java GUIs illustrates the role of software tests in producing professional Java GUIs.

5.9, Usability testing of Java GUIs briefly touches the main points related to usability testing of Java GUIs.

5.10, JRE runtime management discusses profiling of Java desktop GUIs.

5.1 Iterating wisely

Before introducing the various techniques and approaches for effective iterative development, it is important to discuss the overall strategy behind the assignment of priorities to development activities. This focuses on the development activities that need to be carried out, as opposed to use cases or user stories. The latter will depend upon the given project and customers, but will be influenced by the development process chosen.

We will focus on questions such as how much interaction and control behavior should be provided from one iteration to the next, the correct amount of GUI design to implement in the first release, or whether an explicit domain model should be implemented now or moved to a future release. We will use another incarnation of the cost-driven principle introduced for GUI design in Chapter 3 as the subject of this discussion, but this time apply it to an iterative style of software design and implementation for desktop application GUIs.

At first glance iterative GUI development seems a perfect candidate for the well-known 80:20 rule, or *Pareto Principle*[1]. This states that for many phenomena 80% of the consequences stem from 20% of the causes. This principle has been empirically validated on many software projects, in various forms[2]. GUI development is a circumscribed and well-known application domain in which experience can be reused fairly well. If we suppose that this rule roughly applies to GUI development, wouldn't it make a big difference to the way we plan our development activities? Such an 80:20 rule may not, however, apply to the design and development of top-quality professional GUIs – those with a sophisticated, innovative GUI design and substantial resources for their development – which is a fine art, the result of many tiny details carefully crafted together. Nevertheless, even a rough match with this rule would give us a very useful planning principle.

Clearly we will never be able to demonstrate empirically that the 80:20 law, or something similar, applies to GUI design projects. The main problem lies in assessing objectively the overall 'quality' of a GUI. How can we tell that a design is 80% done while also accounting for subjective and ephemeral aspects such as its usability and its overall appeal to users? Any developer who has built a number of desktop application GUIs can observe that there are common patterns of development activity that constitute the bulk of the job, in terms of an 'effective GUI' (a subjective definition, of course). What is invariably needed is a mixture of

1. This principle can be seen as a special case of the Pareto Distribution, a power-law distribution found in various cases in nature, such as the frequency of words in long texts, the size of sand particles, the size of areas burnt in fires.
See http://en.wikipedia.org/wiki/Pareto_distribution
2. See for example: A. Ultsch, *Proof of Pareto's 80/20 Law and Precise Limits for ABC-Analysis.*

a 'minimum dose' of the various contributions: overall team attitude, testing, suitable software architecture, basic usability testing, and so on.

Apart from these abstract considerations, the *ranking* between development activities is important. Imagine having such a ranking documented neatly in the form of an ordered to-do list. Achieving cost-effective quality would then just be a matter of executing the items in the list using a 'greedy' style – starting from those activities that have the largest impact on the final result. Quality could be fine-tuned in this way depending on the budget, without risk of wasting precious resources in unproductive or counterproductive work.

Scheduling development activities following such an 'optimum' list minimizes risk, by ensuring that roughly 80% of the required result is achieved before focusing on inessential requirements. We can maintain the project in good shape from early releases: customers gain confidence that the project is progressing well, developers are gratified by their work, the project manager enters the room whistling merrily, and so on. (Guess how often this happens…)

Unfortunately, such a ranking is almost impossible to calculate, because it is the final result of many intertwined factors – project details, business domain factors, project timeline, the people involved – that vary widely from project to project. Some rules of thumb can be given, but ultimately it is the developer, the team leader, or the application architect, that has the last word and should actively focus on cost-effectiveness when ranking development activities. A prioritized list of development activities can be sketched by leveraging past experience and the contents of this book, but an exact assessment is largely unattainable – a situation that applies to non-GUI development projects as well.

Here is an example of a mythical list of activities ordered by cost-driven criteria. The example list refers to a simple form-based rich client project, with no need for localization and with many simplifying assumption (people have been assigned already, preliminary analysis has been performed, etc.).

Set up a basic production environment, choosing simple and reliable technologies such as GUI and unit testing tools, version control tools, clear and simple look and feel or presentation technology, GUI toolkits and application platforms, deployment technology, and so on.

1. Determine the basic contents for use cases X and Y from customers and implement the control layer completely and without dynamic layout managers, validating it with end user representatives.
2. Define the data handled by the use case and implement it, whether it is part of the business domain or data IO.

3. Identify and implement the minimum set of commands that realizes the use case, given the data and the content from the previous steps. Provide a minimal implementation of client-side data validation.
4. Verify the GUI by software testing of critical points and a brief usability-testing session.
5. Provide extensive software testing and basic profiling, checking memory leaks and thread deadlocks.
6. Add additional control logic to ease interaction in the form of further validation behavior.
7. Provide basic help support and keyboard navigation.
8. Supply further content details for dynamic layout support.
9. Add a branded Look and Feel/presentation style, evaluated with end users and available client runtime resources (such as memory, CPU power, hard disk space, screen size).
10. Provide customized content widgets for easier interaction.

This list implicitly assigns different weights to the quality of the final result, depending on the needs of the customers and the specifics of the project. It assumes that about the first five points in the list will deliver roughly 80% of the final result to users.

These assumptions are, of course, subjective and case-specific, yet intuitively appealing. For example, the choice to regard dynamic layout as optional, perhaps because localization is not needed, thereby ranking it ninth in the list, is debatable.

We are now ready to dip our toes into iterative GUI development with Java, starting with a well-known tactic: prototyping.

5.2 Introduction to prototyping

The development of a representation of a system for testing purposes is common practice in many engineering fields. It is an important method in GUI development as well. Design flaws or other incorrect assumptions can be individuated from the beginning, with resultant large savings in development costs. Prototypes can range from simple paper mock-ups to fully-functional products. Prototyping can be used not only for defining the GUI design, but also for eliciting requirements and as a mean of communication within the development team, with the customer, and with users. This chapter discusses the many different options available for prototyping Java GUIs.

Uses for prototyping

Prototyping is an essential aspect of any professional GUI development. During the analysis phase and later in the development lifecycle a prototype can be seen as another form of documentation. It can help the communication flow, both with the customer's organization and within the design and development team itself, and of course also with the final users of the product. Some of the most useful uses for prototypes are discussed below.

As a means of communication

Prototypes can convey a lot of information to people in a number of different roles within the development organization, as well as other stakeholders. A prototype can:

- Demonstrate to users and customers how the final GUI will look. This requires extra care, however, in order to avoid committing an early, sketchy design as the final one.
- Help to clarify the developer roles involved, especially on the client side – who is ultimately in charge of the GUI design, whether or not the representative users are the same as the end users, who has authority over the design of the GUI, and so on.
- Define detailed terminology, which can be used as the basis for building a *domain-driven ubiquitous language* for the project (see (Evans 2004)), as well as small details that would be tricky to guess from mere discussions.
- Document the GUI design: GUI prototypes are a powerful means of documenting a design, throughout the software lifecycle, especially for potentially risky aspect of the project.

Personally, and possibly unwisely, I love to amaze my clients. After a heavy analysis session in which they expect a recap document, I often release a functioning prototype instead, to much surprise. Pleasing clients early on in a project usually rebounds in the form of extra work and greater expectations, but I like to do so anyway. One of my favorite tricks is to add a general comment mechanism to the prototype application, so that end users can attach their own comments directly to specific areas of the prototype application. The comments they register in this way are precious, because they show how users think about the GUI in detail. They help to substantiate the A3GUI decomposition of screens, for analysis and design, and sometimes they even shape the final development.

Exploring the design space

Prototypes can also be used to explore the design space, especially for novel classes of systems for which no mature design has been established. Several parallel designs could be developed to try to generate as much diversity as possible, or just to focus on evaluating a few alternatives. A number of preliminary designs are created and the best ideas are used for the definitive design, as shown intuitively in Figure 5.1.

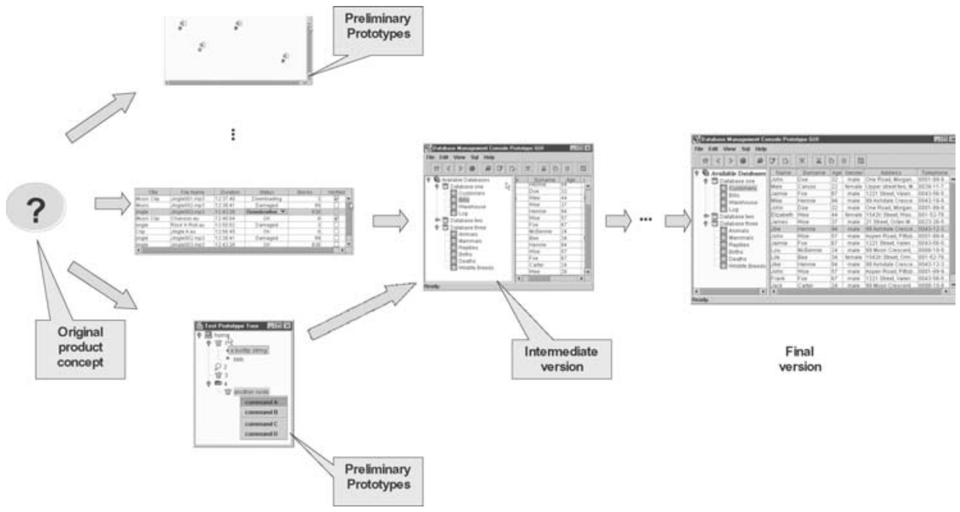


Figure 5.1 Exploring the design space with different prototypes

Developing parallel prototypes is clearly rather expensive because – to have the largest diversity possible – each prototype should ideally be developed by a separate team or individual, with little contact with other teams. However, in most actual cases, a single prototype is enough to produce a viable design.

As suggested in (Hunt and Thomas 2000), when nervous or insecure about the beginning of a new project, or just about the design of specific screens, it is wise to break the ice with a prototype instead of committing unwillingly to a tentative solution.

A common risk is to close the design space prematurely, choosing as final a design solution that has not been thoroughly tested and validated with users. On the other hand, keeping too many design choices open is needlessly expensive and could lead to incoherent, 'stratified' designs, to which different and unrelated approaches were added over time.

Some software development approaches like XP (see Chapter 1) push the prototyping approach outside the GUI domain, involving the whole software development at large (Beck and Andres 2004). XP projects can employ prototyping for exploring possible GUI solutions. These limited systems are called *spikes*. A spike solution is a very simple program built to explore potential solutions, addressing only the problem under examination and ignoring all other issues.

Capturing requirements

Prototypes are often used to elicit requirements for the system to be built. This is done both when building the prototype, and later when gathering feedback from the users on the prototype that has been built. Using prototypes in this way is a natural extension of the adoption of other functional requirement techniques such as use cases or XP's 'user stories.'

Using prototypes as a mere form of requirements-gathering can lead to rather unusable GUIs. Usability and GUI design are different from functional requirement gathering, and should be handled in a different way, by using approaches focused on GUI design, such as user-centric design techniques, rather than system-centric ones like system requirements.

The two dimensions of prototyping

A prototype is a reduced version of the final system. Such a reduction can be achieved either by implementing less functionality, or by reducing the level of functionality of each feature. The former approach is called *vertical prototyping* – demonstrating few features, fully implemented – while the latter is called *horizontal prototyping*, demonstrating many features but with each shallowly implemented. These two dimensions of prototyping are shown graphically in Figure 5.2.

Horizontal prototypes are easier to build, as shown later in this chapter, because they focus mostly on GUI aspects, and can help to test the whole prototype and the full picture it produces. By reducing the number of features and their implementation level, we can obtain cheap 'subsets of use' of the final application, called *scenarios*. A scenario describes a single interaction session limited to few functionalities.

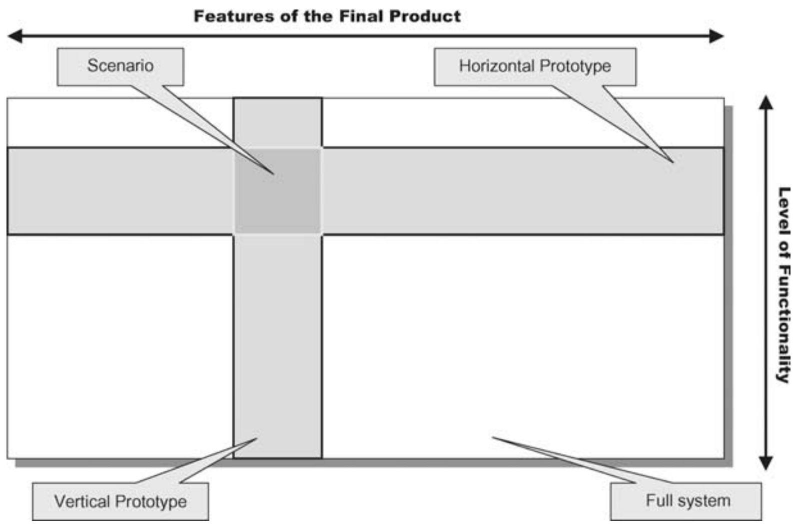


Figure 5.2 The two dimensions of prototyping (Nielsen 1993)

There are several different definitions of a scenario. We make the assumption that a scenario corresponds to the definition given in Section 1.4, when we introduced scenario use case diagrams.

Competitors' product as ready-made prototypes

A design approach known as *competitive analysis* considers similar products that are already available as a starting point for the design activity. A competing similar product is already fully implemented, and can be easily tested in detail. Even when we already have a prototype ready, we can compare concretely how well analogous tasks are implemented by the competing product and by our prototype application.

If several competing products are available, we can examine their differences and the way they approach the same abstract application using different GUI designs. This greatly helps the analysis and design phases – even if, as Nielsen points out, competitive analysis and design does not mean stealing other's hard-won designs, but rather taking them into consideration in your own design analysis, possibly to improve on them and overcome their weaknesses.

Prototyping as a philosophy for development

Evolutionary prototyping is a fully-fledged development philosophy in which the GUI development is just a part of the overall software lifecycle. Agile and other fully-iterative approaches are inspired by this type of highly iterative view of design and development. The prototype is constantly refined, expanded and validated with users until it becomes definitive, when the final product is released. This approach can be difficult to implement, due to the technical pitfalls involved in working with prototypes that constantly evolve. We mentioned such approaches and their lifecycle models in Chapter 1. For more details, see for example (McConnell 1996) or (Beck and Andres 2004).

In conclusion, prototyping deals with building a GUI incrementally and in a cost-effective way. The ability to state the quality of a given GUI is a key factor in driving the evolution of one or more prototypes into the final design correctly. In the next section we will see in detail the different kind of prototype presented above.

Prototypes and customers

Prototypes can have a significant impact on end users. Handling this aspect correctly is important in ensuring adoption of any prototype. Apart from the technicalities involved in creating prototypes, they also guide the perception of the product being developed by end users and customers. Customers often have a non-technical background, and a number of misunderstandings are possible:

- A bad GUI will impact negatively on the idea customers have of the product, irrespective of the fact that it is only a prototype. Customers often implicitly establish an emotional link with the software that will probably become part of their daily working life.
- Agreeing on a given prototype with customers is an important statement. From that moment on customers will be expecting that specific user interface, and anything different could be considered as a change in any agreement made with them.
- An overly-sophisticated prototype can convey to users the false idea that the product is almost complete. When presenting a prototype, it is essential to state the current state of development of the product, and not just focus on how the prototype is different from the final product. One can provide some graphical adornment such as watermarks to signal the fact that the prototype is just a prototype, no matter how good it might look. There is even a Swing Look and Feel that is expressly designed to provide this feeling of ‘sketchiness,’ as we will see later.

To recap, it’s important to remember when dealing with customers that prototyping often represents their perceived image of the product you are building: special care is needed to deal with such a delicate issue.

5.3 Prototyping alternatives

There are a number of possible approaches to prototyping, depending on which aspects designers want to focus on.

Different types of prototypes

This section introduces the main types of prototyping discussed in this book: the subsections that follow describe them in detail.

Storyboard prototyping

Storyboard prototyping is a technique for representing parts of an interface in a way similar to the 'storyboard' used to represent and evaluate the script of a film before committing to the expensive process of shooting the final motion picture.

Storyboarding is a simple, informal way of representing a scenario associated with a given task in the user interface. It is mainly useful for the initial phases of the design process, where accurate feedback from users is still not needed.

Figure 5.3 shows a simple storyboard for the task of selecting a color from a form on screen. Storyboards usually comprise more GUI screens than is shown in this example, as we will see later.

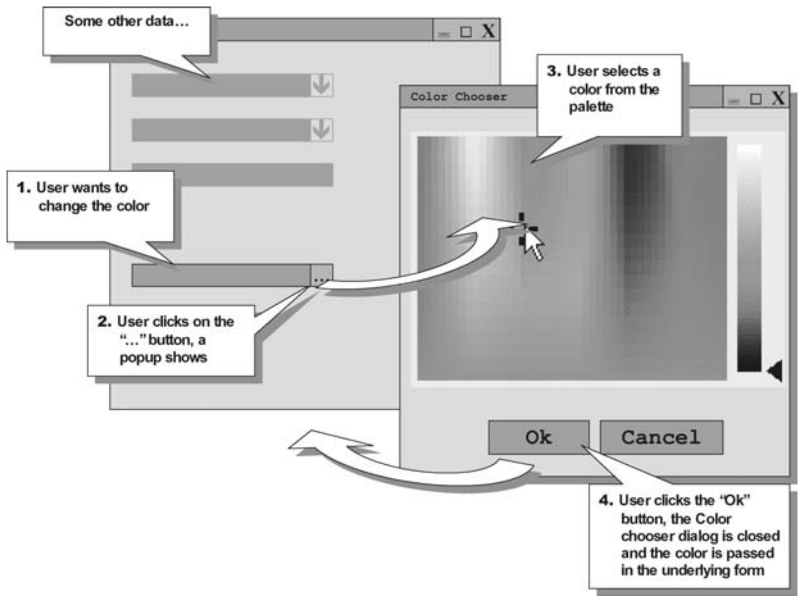

Some other data...
1. User wants to change the color
2. User clicks on the "..." button, a popup shows
Color Chooser
3. User selects a color from the palette
Ok
Cancel
4. User clicks the "Ok" button, the Color chooser dialog is closed and the color is passed in the underlying form

Figure 5.3 An example of a simple storyboard

The storyboard in Figure 5.3 has been designed using a computer graphics application. Storyboards are more often sketched informally, for example on paper, as can be seen in the examples in Figure 5.5 and Figure 5.6.

Paper-based prototyping

Paper-based prototyping needs as its technical support only a piece of paper and some pencils. Sketching out a GUI in this way usually produces rather coarse prototypes, but helps to make key ideas explicit quickly and cheaply. For a thorough discussion of this topic, see (Snyder 2003).

Several slightly different techniques are gathered under the term 'paper prototype.' In a later section of this chapter we will discuss in detail this family of techniques, maybe the most popular form of prototyping. Figure 5.4 shows an example of a paper prototype taken from the example in Chapter 14.

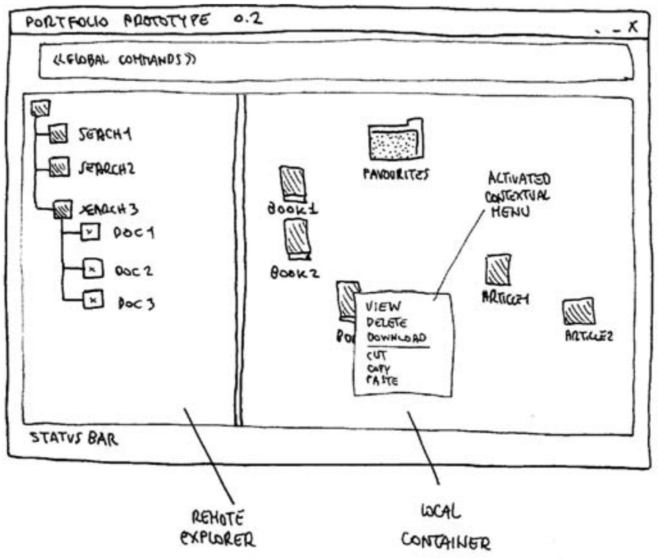


Figure 5.4 An example of a simple paper prototype

Paper prototypes can be used for usability testing with users (Snyder 2003). Following this approach, one or more paper prototypes are built to model the GUI and test it for usability. Testing for usability in this case means letting users try the prototype as if it was the real interface, and try to discover any difficulties and problems to which its design might give rise.

Rapid prototyping

Rapid prototyping (also known as *throw-it-away prototyping*) is the technique of building scaled-down applications, usually using the same technology as the final product. The prototype developed in this way is abandoned at some point in the development process, after it has accomplished its duty – for example in pinpointing defects in the design of the GUI with end users. The GUI prototype is cheap and serves as a first point for requirements gathering and defining the design space.

Rapid prototyping and GUI iterative development can complement each other. Iterative development focuses on building a working GUI starting from the most-needed and best-understood requirements, while rapid prototyping is usually employed to validate or elicit specific aspects, and focuses on those requirements that are poorly understood.

The different expressiveness of prototype techniques

The following table summarizes the different expressiveness properties of paper versus rapid prototyping.

Table 5.1 Expressiveness of prototyping techniques

Entity type	**Category of entities that can be represented using the given type of prototype**	**Prototyping method**	
		Paper	**Rapid**
Business	Main concepts	✓	✓
	Terminology	✓	✓
	Documentation, help	✓	✓
	Requirements, functionalities	✓	✓
	Data size, dimensions	-	✓
GUI	Navigation, work flow	✓	✓
	Appearance (Look and Feel)	-	✓
	Screen layout	✓	✓
	Response time	-	✓
	Keyboard, mouse, other input	-	✓

Clearly, paper prototyping has numerous disadvantages when compared with rapid prototyping. Nevertheless, given its cheapness and simplicity – even end users can come up with their own proposal – paper prototyping is widely used. Rapid prototyping can be used in cases in which specific development risks that need to be evaluated early in the development are not made explicit by a paper prototype. Consider for example an application that is required to be close to an existing application, with a high level of fidelity. Only a software prototype can fulfill this need.

Different types of prototypes can be used in combination to give the best of both approaches. Suppose we want to design the GUI for an application with a heavy data load – perhaps tens of thousands of items. This aspect is a potential risk that needs to be explicitly addressed as early as possible. The first informal prototypes are written on paper: when a suitable design emerges, it is rendered in a rapid prototype that simulates a large number of data items and their related latencies, so that the design can be validated and agreed with end users.

Prototyping technologies

Prototypes rely on specific technologies, whether the same technology as the final product (in our case Java) or another, for example using Web pages to sketch form-based screens. Comparing Java with other technologies:

- *Java technologies*. A number of visual tools that generate Java sources for GUI layouts and screens by direct manipulation are widely available. Open source software (OSS) tools such as NetBeans or Eclipse VE, as well as commercial products such as JBuilder and Idea, are commonly used in development. A number of stand-alone Java visual builders are available too – we discuss this in Chapter 11.
- *Non-Java technologies*. Prototyping technologies can be employed too: drawing or authoring tools such as Microsoft Powerpoint and Visio, CorelDraw, for sketching paper prototypes, or tools for building horizontal prototypes, such as Visual Basic or MacroMedia Flash. None of these tools effectively model the Java Look and Feel, however.

Storyboards

A storyboard documents how a part of a user interface is employed to accomplish a given task. A storyboard is a simplified representation of the GUI, usually drawn on paper, showing how a user interacts with the product to achieve a specific task. Storyboards usually represent the user interface at a higher level of abstraction than paper prototypes, allowing a wider perspective – storyboards are often drawn on large sheets and hung on the wall. They provide navigation, meaningful data, and all other details needed to represent the task performed in the GUI to a suitable level of detail. Figure 5.5 shows a storyboard for an example application.

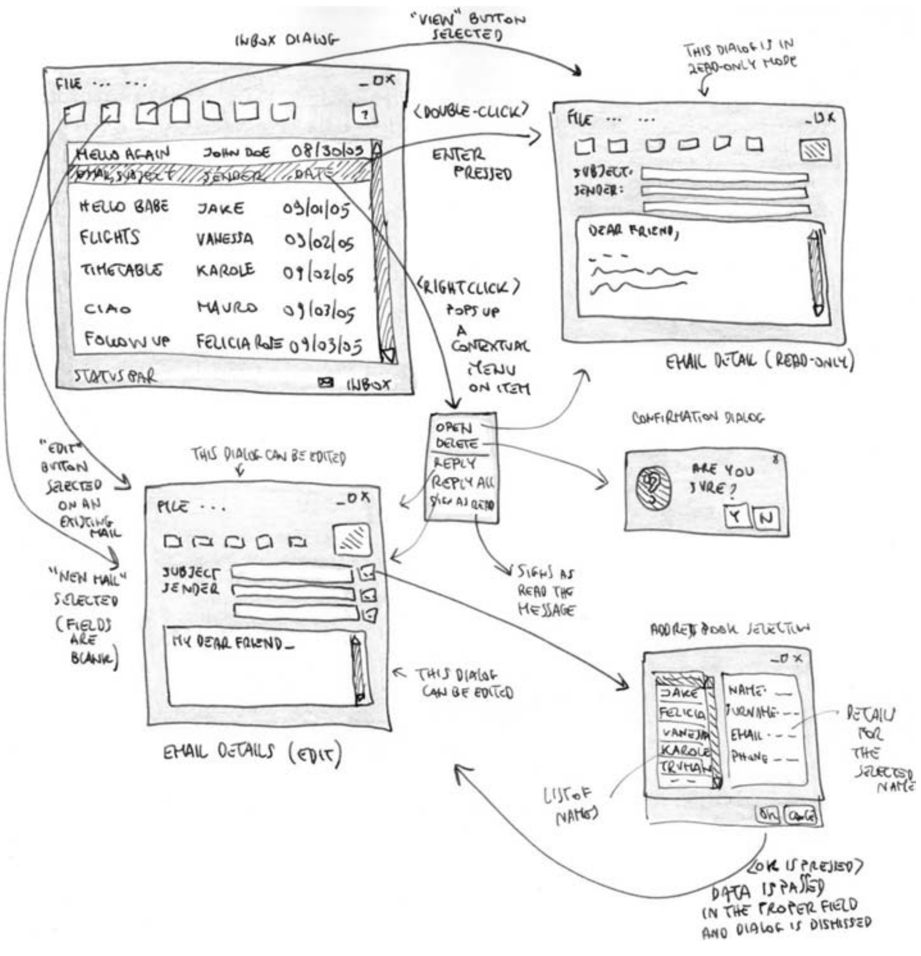

INBOX DIALOG
"VIEW" BUTTON SELECTED
THIS DIALOG IS IN READ-ONLY MODE
FILE ... ...
HELLO AGAIN JOHN DOE 08/30/05
EMAIL SUBJECT SENDER DATE
HELLO BABE JAKE 09/01/05
FLIGHTS VANESSA 09/02/05
TIMETABLE KAROLE 09/02/05
CIAO MAURO 09/03/05
FOLLOW UP FELICIA ROSE 09/03/05
STATUS BAR
INBOX
(DOUBLE-CLICK)
ENTER PRESSED
SUBJECT:
SENDER:
DEAR FRIEND,
EMAIL DETAIL (READ-ONLY)
(RIGHTCLICK) POPS UP A CONTEXTUAL MENU ON ITEM
OPEN
DELETE
REPLY
REPLY ALL
SIGN AS READ
CONFIRMATION DIALOG
ARE YOU SURE?
Y
N
"EDIT" BUTTON SELECTED ON AN EXISTING MAIL
THIS DIALOG CAN BE EDITED
"NEW MAIL" SELECTED (FIELDS ARE BLANK)
SUBJECT
SENDER
MY DEAR FRIEND_
THIS DIALOG CAN BE EDITED
EMAIL DETAILS (EDIT)
SIGNS AS READ THE MESSAGE
ADDRESS BOOK SELECTION
JAKE
FELICIA
VANESSA
KAROLE
TRUMAN
NAME
SURNAME
EMAIL
PHONE
OK
CANCEL
DETAILS FOR THE SELECTED NAME
LIST OF NAMES
(OK IS PRESSED) DATA IS PASSED IN THE PROPER FIELD AND DIALOG IS DISMISSED

Figure 5.5 An example storyboard

This storyboard describes navigation details as well as UI details. Storyboards usually focus on navigation and on providing a wider picture of the GUI. The storyboard in Figure 5.6 shows an example of this latter approach for an account management user interface.

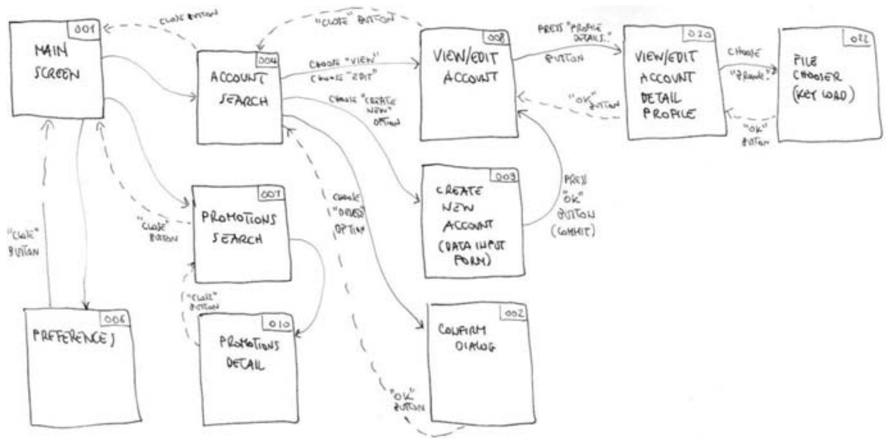


Figure 5.6 Another example storyboard

A number of details can be seen in Figure 5.6:

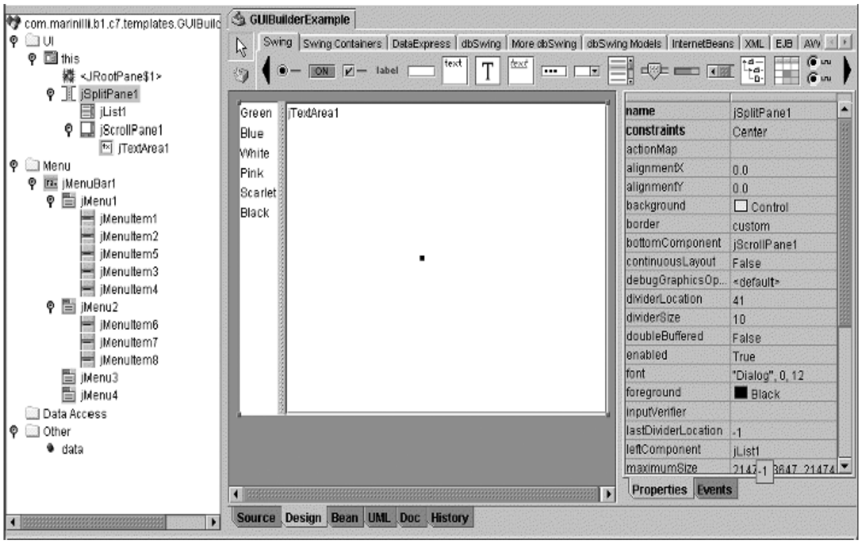
- Every screen is represented by a box showing a window title.
- Transitions from one window to another are shown by arrows labeled with the GUI action that triggers the transition.
- Dashed arrows represent the navigation when the current screen is dismissed.
- Windows are identified with a unique id number in their upper-right corners, for quick reference both during design and at runtime .

Storyboards are a valuable tool for describing GUI navigation and for sketching the GUI, especially at early stages of design.

5.4 GUI builders

GUI builders are another commonly-used aid for building prototypes, as well as entire simple GUIs. They consist of visual environments that ease the construction of GUIs by means of a user-friendly construction interface that creates the code behind the scenes. All major Java integrated development environments (IDEs) provide such a graphical UI editor. This section gives an example of the use of one such tool.

A screenshot of the JBuilder IDE visual designer is shown in Figure 5.7.



name	jSplitPane1
constraints	Center
actionMap	
alignmentX	0.0
alignmentY	0.0
background	☐ Control
border	custom
bottomComponent	jScrollPane1
continuousLayout	False
debugGraphicsOp...	<default>
dividerLocation	41
dividerSize	10
doubleBuffered	False
enabled	True
font	"Dialog", 0, 12
foreground	■ Black
inputVerifier	
lastDividerLocation	-1
leftComponent	jList1
maximumSize	2147-1 3647, 21474

Figure 5.7 The JBuilder IDE designer

The final result of using the builder took less than ten minutes to create, and is shown in Figure 5.8. No extra-GUI, functional code was provided, to keep the resulting code as short as possible.

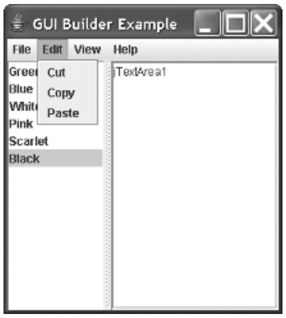


Figure 5.8 An example GUI

The JBuilder editor creates an auxiliary method (`jbInit`) that gathers all the GUI-related code. Code statements generated as commands are issued via the user interface shown in Figure 5.8. Widget visibility is provided for all components, which are created as instance variables of the visual container class. Instance variables are named automatically by the tool, but can be renamed manually.

Even in such a simple case we needed to modify/insert the generated code by hand. JBuilder is flexible enough to recognize lines of code added by hand. This is illustrated in the code example that is available on-line, in which the string array used for filling the list widget has been added in the source code. This is still successfully recognized by the tool, as can be seen in Figure 5.8.

Using a GUI builder tool

Whether code builders are used or not, achieving a professional GUI is always a matter of detail. You will therefore always need to dig into generated code to polish the details of even the simplest GUI. Visual editors do not simplify the overall coding effort if your GUI is complex enough to require massive extension or rewriting of the code automatically produced by the tool.

Visual editors can help quick development of content structure, such as widgets and their layout. They can be used as aids to create the structure of the required class quickly, which can be refined later manually. They can be useful for building rapid prototypes, or can be used by novice programmers for learning the basics of Java GUI libraries in a 'learning-by-doing' fashion.

Programmers tend to have their own opinions of these kinds of automatic tools. Some find them stimulating but limited, others are confident they can save a lot of time, while many simply hate them altogether. Clearly, the perception you have of a tool directly impacts your performance when using it: it's a matter of usability here as well, only here developers are the end users. The ultimate choice depends on your preferences.

Visual editors do have a number of practical shortcomings:

- It is cumbersome, if not impossible, to modify some parts of the generated code. For example, some editors allow only Java Beans – widgets with void constructors – to be created.
- Some editors, such as the one provided with Netbeans, rely on vendor-specific files as well as vendor-neutral comments in source code. This ultimately leads to a form of vendor lock-in. It also makes it harder to adapt the generated code to particular needs without breaking the compatibility between the tool and the edited GUI code.
- The general structure of the generated class is hard to tweak. Special methods can be hard or even impossible to circumvent. In some cases maintaining source code compatibility with the visual editor can become so complex that the simplest solution is to abandon the GUI editor tool altogether.
- Architectural issues are not supported by visual builders that tend to build weak and deeply-coupled code.

5.5 Reusable prototyping widgets

In this section we will examine practical applications of reusable classes to prototyping. In particular, we will look at two classes that are specialized for building rapid prototypes inexpensively.

A tree prototype utility class

The reusable class introduced here simplifies the creation of complex Swing tree components. Such trees are limited to use as rapid prototypes. Despite that, their use could be quite helpful for producing medium- and even high-fidelity prototypes. The class we describe here is implemented as a specialized `JTree` that reads its configuration from a properties file. In this way it is easy to populate a tree, change its appearance and provide contextual menus, tooltips and drag and drop (dummy) support. We will introduce and discuss a sample properties file first, then present the class code, and conclude with another example of its use.

In our implementation, all the appearance is delegated to the properties file, though for specific features it may be necessary to add an external listener class, or to subclass the prototype class itself. For example, in Chapter 4 we saw a tree prototype that simulated a delay by attaching a tree expansion listener to a `JProtoTree` instance. The properties file used in the constructor dictates the appearance and behavior of the prototype tree. Listing 5.1 below shows an example of such a properties file. The resulting output is shown in Figure 5.9.

Listing 5.1 The `Tree.properties` file

```
root=home,a tooltip string,root.gif,command A%%command B%%--%%command C%%command D,a1,a2,a3,a4
a1=1,tt1,bit1.gif,command E%%command F%%command G,a11,a12
a2=2,tt2,light.gif,-
a3=3,tt3,bit1.gif,-
a4=4,tt4,-,-,a41
a41=another node, and its tooltip,bit1.gif,-
a11=aaa,-,-,-
a12=bbb,-,-,-

# special properties
setShowsRootHandles=true
setScrollsOnExpand=true
setRootVisible=true
setDragEnabled=true
setClosedIcon=closed.gif
setOpenIcon=open.gif
setLeafIcon=dot.gif


# tree properties
JTree.lineStyle=Angled
```

The properties can be specified in any order. In Listing 7.1 above there are three main groups of properties:

- The first group dictates the structure of the tree. Line 0 says that the root node has a label '`home`,' a tooltip '`a tooltip string`' and should be rendered using the image '`root.gif`.'
- The contextual menu is composed of four commands: string commands are separated by '`%%`' and the '`--`' represents a menu separator.
- Finally, the root node has two children, identified by the labels `a1`, `a2`, `a3` and `a4` that in turn are defined in lines 01–04.

The final result. the `JProtoTree` instantiated with the properties file of Listing 7.1, is shown in Figure 5.9.

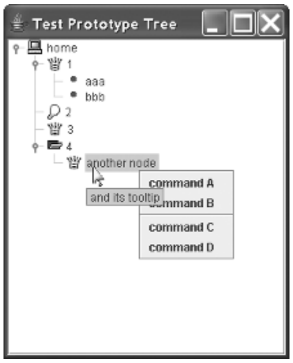


Figure 5.9 An example of a prototype tree

Each line is composed of a string id that is used by the tree to identify the given node. The special string '`root`' is used to identify the root element, and is mandatory. There follow the appearance values for that node, separated by the '=' character used by default in Java properties files.

Such values are the ordered sequence of the following data:

1. The `ext` label. This can be a sentence or even an HTML fragment.
2. The icon used to render the node. When not used, the standard icons are used instead.
3. The contextual menu that is activated by right-clicking with the mouse on the given node. Note that contextual menus are inherited from a parent node by its children. If we have only one type of contextual menu for all the nodes, we therefore just specify the required menu in the root element, and this is then used by all its descendents. If another node needs to show a different contextual menu, we then have to declare it in the properties file for the given node, as node '`a1`' does in Listing 7.1.

4. The list of child node ids that builds the tree structure recursively, or '`-`' if the node is a leaf.

In this way the nodes and their appearance are defined. There are other attributes that can be defined as well, to control the global tree appearance, These are specified in lines 10–16 in Listing 7.1. It is possible to define properties such as whether the root handles should be made visible, or whether the tree nodes should be draggable. Finally, Swing tree properties can be added to the file.

Whenever an item of information is missing – for example omitting to specify a particular icon, so that the node will be rendered using the standard icons for leaf, open and closed folder nodes – we use the '-' character.

For brevity, we don't show the implementation here. For readers interested in it, the `JProtoTree` class uses two inner classes: a custom tree cell renderer and a custom tree model.

The constructor simply instantiates a specialized tree model based on the input properties file, then queries it to change the tree's appearance. This use of the tree model is incorrect – the purpose of the Swing modified MVC architecture is primarily for separating appearance from data. Loading the model with appearance data is thus conceptually incorrect. The tree is created using default tree nodes that have an array of strings as the user object. Such an array contains the appearance information extracted from the properties file.

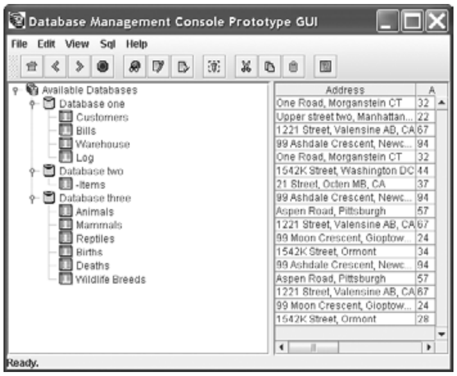
The `ProtoTreeModel` inner class reads the properties file and creates the related tree model, which will be used by the enclosing tree class. In particular, the `createNode` method is used for populating the tree recursively using a deep-first strategy.

The inner class `ProtoCellRenderer` is a subclass of the `DefaultTreeCellRenderer`, and is used to customize the node appearance as prescribed in the input properties file. The main method of the `JPro-toTree` class shows a sample use of the class, and requires a properties file named '`tree.properties`.'

Another example of the use of a properties file for defining tree appearance is shown in the sample code for this chapter, in the `DBTree.properties` file. The corresponding instantiated tree is shown in Figure 5.10.

Note that in this prototype we have modified the appearance of few special nodes only, while allowing all the remaining nodes to comply with the general tree rendering rules. These distinguish between open and closed folders – any node that has at least one child – and leaves, those nodes that have no children. We then modified the appearance of these three types of nodes in turn, allowing us to use a cheaper standard component for the working implementation.

The sample syntax shown in the previous listing can be seen as a simple, although rough and very simplistic form of a Little Language specialized for rapid prototyping. Little Languages are described in Section 12.5.



Address	A
One Road, Morganstein CT	32
Upper street two, Manhattan...	22
1221 Street, Valensine AB, CA	67
99 Ashdale Crescent, Newc...	94
One Road, Morganstein CT	32
1542K Street, Washington DC	44
21 Street, Oclen MB, CA	37
99 Ashdale Crescent, Newc...	94
Aspen Road, Pittsburgh	57
1221 Street, Valensine AB, CA	67
99 Moon Crescent, Gioptow...	24
1542K Street, Ormont	34
99 Ashdale Crescent, Newc...	94
Aspen Road, Pittsburgh	57
1221 Street, Valensine AB, CA	67
99 Moon Crescent, Gioptow...	24
1542K Street, Ormont	28

Figure 5.10   Another example of a prototype tree

We also supply a simplified version of this utility class for the SWT (Standard Windowing Toolkit) library in the source bundle for this chapter.

To recap, utility classes can be useful for quickly building tree samples from scratch for use in rapid prototypes, whether for Swing or SWT programs.

A visual container prototype utility class

This section introduces a reusable class for creating prototypes of directly-manipulatable, two-dimensional containers, such as file system folders in Windows or the Macintosh operating system. We will see these working in Chapter 15. Figure 5.11 shows an example of such a prototype component.

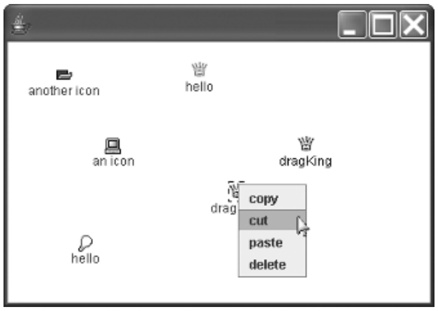


Figure 5.11   An example of a container prototype

Users can drag the icons within the container and right-click on them to show their contextual menus. This kind of component is not provided by the standard Sun libraries, but it can be employed usefully in GUIs, especially OOUIs. The `SandboxExample` class provided with the code for this chapter shows a sample use of this component for creating rapid prototypes.

5.6 GUI refactoring

Having explored the various options available to Java developers for building effective GUI prototypes, we now focus on iterative GUI development, in which the code is not meant to be thrown away, but instead is refined and improved continuously by means of small steps that do not alter its functional behavior. These are called *refactorings*.

Fowler's classic work on refactoring (Fowler et al. 2000) describes a set of changes that improve the internal structure of code without changing its external behavior. Refactoring is 'a disciplined way to clean up code that minimizes the chances of introducing bugs.' Refactoring changes the design of a system without modifying its observable behavior. We discuss refactoring in this chapter because it is instrumental to iterative GUI development. We refer to Fowler's refactorings as 'classic,' to differentiate them from the higher-level, GUI-specific refactorings introduced in this section.

The refactoring we introduce here is performed as a sequence of classic (low-level) refactorings that focus on enhancing the structure of GUI code while preserving its external behavior. Some of these GUI-specific refactorings might slightly modify the GUI's appearance, however. When this happens, the changes are always focused on standardizing the GUI design and making it systematic throughout the application.

One important point is *when* to refactor. (Fowler 2000) suggests refactoring *the third time* we happen to do something similar – this is called the 'rule of three' and is credited to D. Roberts. This means duplicating things at first and living with the duplication temporarily. For example:

- The first time we implement our panel.
- Later it happens that we find ourselves implementing a panel that is very similar to the one we have just implemented. The rule says we should leave the two panels separate.
- The third time we encounter the same situation we proceed to apply the required refactoring – see *Parameterize panel* on page 198.

Some classic refactorings

In this section we briefly introduce some classic refactorings commonly used in GUI development – for a complete discussion, refer to (Fowler et al. 2000). In the next section we discuss the most common GUI-specific refactorings.

The refactorings described here are simple and specifically concern GUI code restructuring. Refactoring techniques apply to any piece of code, of course, not just GUI code. We describe four refactoring techniques that can come to our rescue when restructuring GUI-oriented code. The first three, *Move Method*, *Duplicate*

Observed Data and *Extract Method*, are simpler and are all used in the final technique, *Separate Domain from Presentation*.

Move method

This is one of the most useful and frequent refactoring techniques, and consists simply of moving a method from one class to another. Of course you should check whether the method is declared in the superclass or in some subclasses of the current class.

After moving the method to the other class, the old method could be emptied and transformed into merely a delegating one – that is, one that invokes the corresponding new method in the other class – or it can be removed altogether. In the latter case all references are made directly to the new method in the other class.

There are no clear-cut criteria for applying this pattern – it depends on many factors, such as the semantic coherence of the code, its coupling with other classes, and so on. Move Method, and other refactoring techniques as well, are needed for example when enforcing a given structure in existing code by moving methods to different classes. This is discussed in Chapter 7.

Duplicate observed data

Business domain methods need to access business data hosted by GUI widgets – also referred to as *screen data state*, and discussed in Chapter 8. A solution is to duplicate the data, so that one representation lives in the content layer, and the other in the business domain[3], and keep them synchronized through an event-based mechanism. In the example in Figure 5.12 the latter mechanism is provided by means of the `java.util` implementation of the Observer design pattern (Gamma et al. 1994).

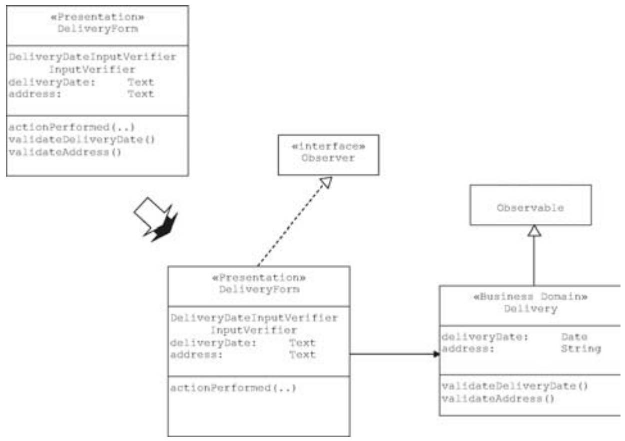


Figure 5.12 Duplicate observed data

3. The layers' names may vary according to the architecture of choice.

There is a facility in the `java.util` package that helps with the implementation of such a pattern through the `Observer` interface and the `Observable` class, as used for example in the Sandbox application in Chapter 16.

As we saw at the beginning of this chapter, such an approach is employed in the MVC pattern and in its variant adopted in the Swing framework.

Extract method

This is another very common refactoring technique. It consists of grouping code statements into a new method, and is often used with GUI code. For example, in a frame or dialog initialization, when the visual container is filled with components and these are initialized, one can organize this code into a number of methods. This is shown in the following example, in which all initialization code of a `JFrame` subclass has been organized in few self-explanatory methods.

```
private void initGUI(){
  createToolbar();
  createMainPanel();
  populateDataTable();
}
```

This organization makes the same code more readable and easy to understand without modifying its externally-observable behavior. Clearly, new methods should be devised depending on the function the code performs, and not on how it is implemented. The objective is to clarify the code rather than make it more complicated with the addition of more methods.

Separate domain from presentation

This is perhaps the most obvious refactoring in the large for GUI code. It is a 'macro' refactoring technique, taking many small steps to be accomplished, and cannot be performed automatically, but is all-important in GUI development. Its result is to separate business logic and data from the presentation, as shown in Figure 5.13.

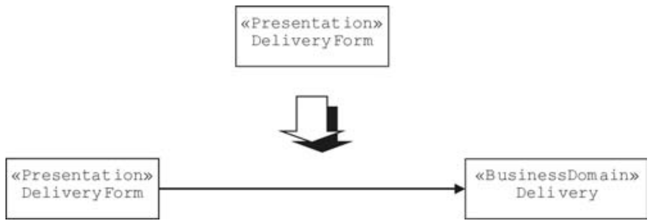

«Presentation» DeliveryForm
«Presentation» DeliveryForm
«BusinessDomain» Delivery

Figure 5.13 Separating domain from presentation

Figure 5.13 uses UML stereotypes to show the functional layer – related to the functional decomposition shown in Chapter 1 – to which the class belongs.

This technique deals with the guiding principle of separating presentation code from domain logic. We have already discussed this principle and its implications: here we present a refined version of the corresponding refactoring technique. In fact, the rules stated in (Fowler et al. 2000) are:

1. Create a domain class for every window
2. Study the data shown in the GUI windows. If there is any data that is used only in the window, leave it in the presentation layer. If some data is not shown, move it into the domain class for that window using Move Method refactoring. Finally, if any data is used both in the presentation and in the application layer, use Duplicate Observed Data refactoring to split them into two separate classes, as discussed earlier.
3. Separate the domain logic inside a presentation class using Extract Method refactoring. When the domain logic is clearly separated into one or more methods within the presentation class, move these methods to the corresponding domain object.
4. Finally, when all the code for the domain logic is separated from the presentation, 'polish' the resulting domain logic classes with further refactoring.

This approach can be further refined depending on the kind of libraries on which your application relies. For example, if your GUI uses the Swing framework, instead of relying on the first rule (create a domain class for every window) you can take advantage of the modified MVC model adopted in the Swing toolkit.

Some GUI-specific refactorings

Before introducing some of the most common refactorings used in iterative GUI development, we need to introduce the concepts of Composable Unit and Content Assembly, which we discuss in greater detail in Chapter 6.

In medium or large applications there could be a need to aggregate code following some defined abstractions. These 'units' are fully-fledged autonomous entities that handle their own data, control behavior, content, and so on. They are sort of 'mini-GUIs' within the GUI itself, aggregated following the Composite pattern. We call them *composable units*.

These aggregations can be useful for a number of reasons, such as code organization and code reusability. A composable unit is a formal building block of the GUI represented within our architecture. For example, if we adopted an MVC architecture for composable units – that is, an adoption of MVC in the large, different than using it at widget-level as in Swing or JFace – then an EmployeeMVC would be a triplet of a Model, View and Controller that together would form a single, formalized unit of reuse within our application. Following this architecture, whenever we need a panel that represents an employee, we just instantiate the related MVC triplet.

Many approaches are possible, apart from technical-oriented ones such as MVC, as we discuss in Chapter 6. The OOUI approach shown in Chapter 15 also illustrates this. Of course reuse and other functionalities can also be provided more informally, without resorting to a fully-fledged approach like an architecture based on composable units.

Content assembly is the procedure of assembling widgets using a given layout manager. The simplest way to compose widgets and composite aggregates of widgets into working panels and windows with OO technology is to use panel subclasses and put the assembly code into their constructor. This scheme works well in the majority of cases, but there could be situations in which this intuitive approach can be problematic. Other techniques are possible, and we discuss them in Chapter 6. Here, for ease of discussion, we refer to the case of content assembly implemented via subclassing. Nevertheless the following refactorings can be applied as well when other content assembly approaches are used, such as specialized factories or builders.

Extract explicit panel

We are now ready to discuss some explicit refactoring in the large for desktop application GUIs based on OO technology.

A common situation is that in which we design a panel for some purpose and then realize we may want to make a part of it an explicit, separated panel, perhaps because we may need it for reuse somewhere else in the class. This situation is shown in Figure 5.14, in which an explicit panel, `AddressPanel`, in extracted from the `PersonPanel` implementation.

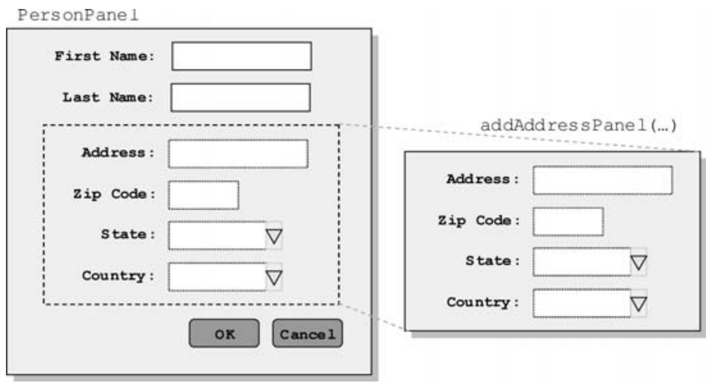


Figure 5.14 Extracting an explicit panel

After the refactoring, `PersonPanel` now invokes `AddressPanel`, while the GUI design remained unchanged. Usually an explicit panel is implemented as a private method within the same class as the container panel. Even this simple refactoring can be complicated to achieve in practice because of the intricacies of layout management.

Extracting a panel can be tricky, because we want to have a flexible panel that can adapt to different use scenarios – when it was implicit, there was no need to provide this flexibility. Let's consider visual composition, for example. We might in some cases want our extracted panel to align seamlessly with the containing panel *without knowing about it*. Think about the address panel in Figure 5.14. When we add it to another panel, we expect all its fields to be nicely and seamlessly aligned with the other fields in the containing panel.

We have two basic strategies for providing widget layout flexibility in our newly extracted panel:

- *Black box support*. The panel is provided as a unique visual container, and it is up to the layout manager to adapt it to the rest of the containing panel. In practice this might mean providing your own implementation of a layout manager that deals with this aspect.
- *White box support*. The explicit panel exposes its internal structure to the outside world so that its component pieces can be aligned with the widgets in the containing panel. An example of support for this kind of approach can be provided by means of attributes, as discussed in Chapter 12.

You can of course provide an approximated alignment without resorting to the complex mechanisms outlined above – for example alignment values for form-based GUIs, or even no alignment at all. This will result in slightly poorer visual symmetry, but it will simplify development.

Extract stand-alone panel

We might go one step further and make our explicit panel a stand-alone class. This encompasses moving the code of the private method implementing the explicit panel in a separate panel class, together with the address widgets. This allows us to reuse the content for addresses in different places of the application, as shown in Figure 5.15, in which the same address panel is used in two different contexts.

Extracting stand-alone panels is performed routinely when implementing GUIs through panels, instead of windows or other containers. Focusing on panels promotes reuse and simplifies GUI changes, even if it may seem unnatural at first. It can be a needless complication in simple applications, however – see the discussion of the *Smart GUI Antipattern* in Chapter 7.

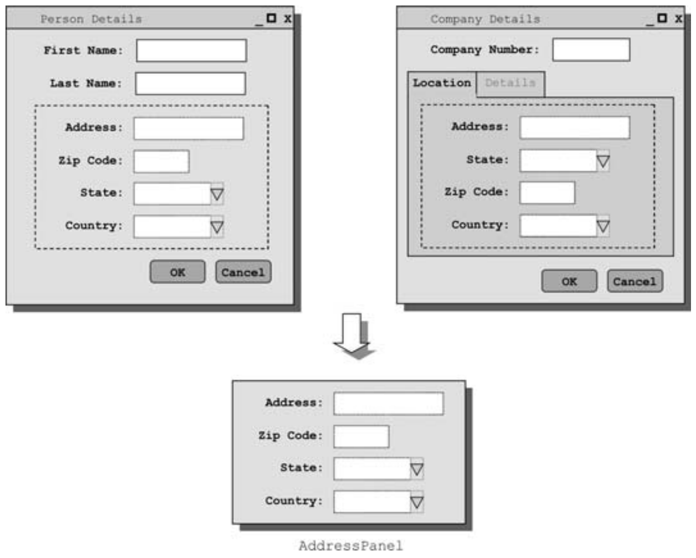


Figure 5.15 Extracting a standalone panel

Of course, extracting a stand-alone panel guarantees only content reuse – that is, the graphical aspects – and some simple, local kind of control and business code. For full reuse, we may need to escalate to a composable unit, as discussed in the next section.

Extract composable unit

Transforming a standalone panel into a composable unit means adding all the required code to make the new unit a coherent reusable block, comprising interaction and control, data IO, and domain code. Architectural behavior must also be provided – for example, composable units may be needed to implement some interfaces, or bind to a register facility. Figure 5.16 shows an example of the extraction of a composable unit from a stand-alone panel and its support code, scattered among other classes.

Merge panel

Refactorings of this type aim at visually merging a panel – either a stand-alone or an explicit panel – into another existing panel. Merge panel is the twin of the extract refactorings discussed previously.

Add parameter to panel

While building a GUI iteratively, we may find that we need to add a degree of flexibility to the code to avoid duplicating it. Suppose we implemented an address stand-alone panel that is embeddable in other panels or windows. In a new

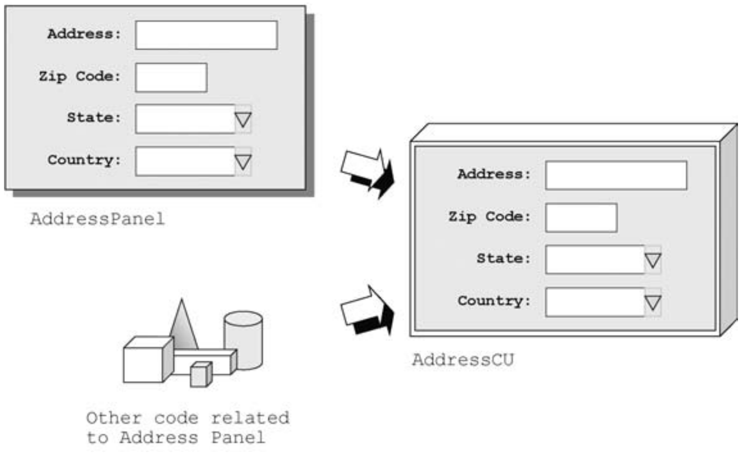


Figure 5.16 Extracting a composable unit

window we are coding, though, we then find that there is an address to display, but it should be laid out differently than would our reusable address panel.

This is a frequent problem in many places in a GUI. When new objects need to use our 'reusable' visual components, many unforeseen subtleties arise. Working in a continuous iterative way as we do, we don't worry too much about adding all the required behavior up-front, but instead add it as required, trying to keep our code as simple as possible.

We then:

1. Decide on the abstractions to be provided by our parameterization.
2. Implement these abstractions by means of a number of refactoring steps.

In our example, we may add to our address panel a `setVerticalLayout(boolean)` method that by default is false, to preserve backward compatibility with all existing clients, which accommodates this special case without revealing internal details of the address panel implementation.

Remove parameter from panel

As with methods, sometimes specific parameters are not used at all in a panel implementation. In this case it is good practice to remove them from the implementation.

Parameterize panel

Most of our development efforts focus on avoiding code duplication. Sometimes we end up having two slightly different panels that share a great deal of code, such as business logic, content, control, and so on, but that differ in detail. A simple solution is to extract a stand-alone panel and provide some means of configuration – usually using an accessory method – that implements the differences between the two approaches, as shown in Figure 5.17.

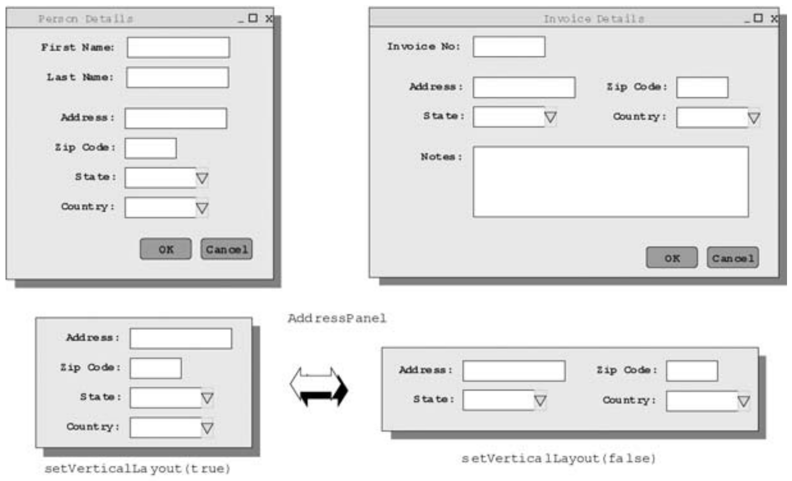


Figure 5.17 Parameterizing a panel

A trivial use of this refactoring is that in which you have the same panel duplicated in different parts of the GUI, and you want to extract a single implementation. In this case there is no need for parameters, because the designs are the same (although they might differ slightly in unimportant details).

This refactoring technique, like the similar Add Parameter to Panel, tends to create procedural code within panels to handle configuration behavior. This can be limited by using classic refactorings such as Replace Conditional with Polymorphism and Convert Procedural Design to Objects, as discussed in (Fowler et al. 2000).

Parameterize Panel and Add Parameter to Panel are similar in theory, but are used in different contexts in practice. You find yourself using this refactoring when developers were not able to exert tight control over GUI design, in cases in which GUI design was done by others – GUI designers, analysts, and so on – or when the initial design was implemented with a GUI builder that made panel extraction and reuse difficult. In these cases we would use Parameterize Panel for factoring out common panels into a single implementation.

Replace parameter with panel

The more parameters we add to a panel, the more complex it gets. We may end up with an overly intricate panel that would be better off split into two or even more separate panels. This is the dual of the Parameterize Panel refactoring technique. It should be used when a panel represents conceptually different aspects that would be more meaningfully represented with different stand-alone or explicit panels.

The address panel implementation has become too complex because it implements two different panels in one class: a simple and an extended address panel. A better solution would be to separate them into two different panels, as illustrated in Figure 5.18.

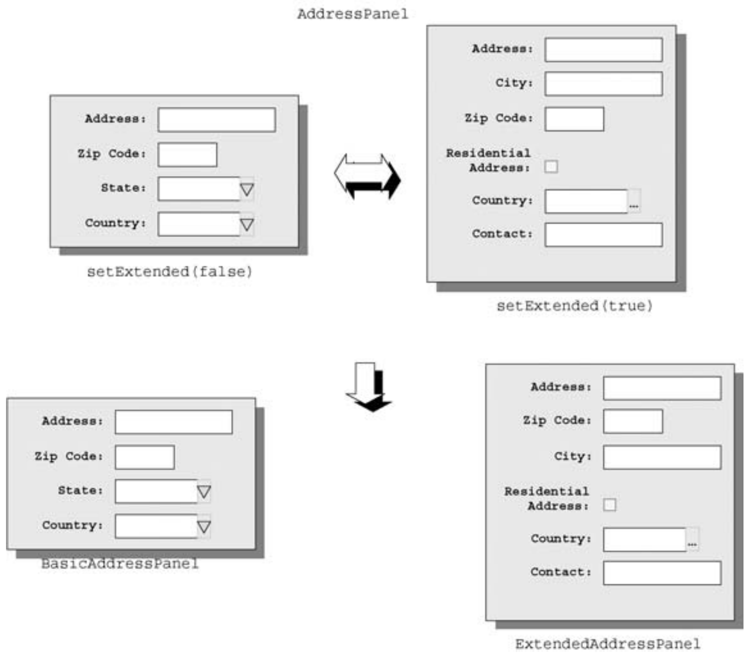


Figure 5.18 Replace parameter with panel

Here, as in all the refactoring techniques presented in this chapter, depending on the content assembly technique we use, panels will be implemented as visual panel subclasses, methods or builder strategies (see Chapter 6). Various classic refactorings can be applied to minimize code duplication between the two newly-created panels, depending on the implementation chosen.

Rename panel

Like methods or classes, the names of panels, as well as windows and other explicit visual composites, are of great importance in defining the specific conceptual identity of visual areas. If the A3GUI approach is used during analysis, then the identifiers of the areas found can be used to name the corresponding panels' implementations. Even more important than A3GUI, renaming should be done following domain-driven abstractions and a domain-driven Ubiquitous Language (Evans 2004).

Failing with style

We conclude this section with a discussion of general strategies for managing implementation errors – that is, software and systemic errors, not business-related ones. We include it here because it is an often-overlooked aspect of GUI development that should be tackled from early on in the development cycle.

Despite not being a refactoring technique, defining a clear, explicit failure strategy early in development is important for providing a coherent, usable GUI from the earliest iterations. By the term *failure*, we mean some software error or unexpected situation that hinders the execution of a program. We focus here on situations in which it is possible to continue execution, provided that the program is allowed to make some assumptions in order to proceed. That is, 'hopeless' situations for which we have no alternatives are not taken into account. Clearly, if an application is unable to find any resource bundle, messages cannot be shown at all and the GUI is unable to run. In these cases we have no option but to fail to provide the required amount of information – as described in the discussion that follows on security and error messages.

The chances are that any application will break one day, no matter how skillful we might be. This issue needs to be addressed explicitly, because providing a coherent failure strategy will affect not only the developers, but ultimately end users as well.

There are two broad strategies for dealing with failure:

- *Fail first*. As soon as there is an unexpected situation, we halt execution, providing a clear explanation of what happened. This makes it easier to detect the problem and fix it.
- *Fail later*. This approach tries to carry on program execution as far as possible. Suppose, for example, we detect that a required remote connection is down: we can signal this to the user, but still continue execution.

In practice, the problem is that the best strategies for failure are conflicting. For developers, a fail first strategy is advisable because the program does not enter into unforeseen behavior, and the problem is more easily detected. Filling code with default, specific behaviors degrades its readability, forcing us to constantly ask ourselves tortuous questions of the form 'Ok, if the program can't find the resource bundle, then it connects to the server, but if the server is down…' and so on.

On the other hand, end users don't want to be bothered by problems that might be handled without sacrificing the current session data. When driving I wouldn't like my car to stop because it is signaling something like 'Running out of air conditioning fluid. This might seriously damage the air conditioning system,' and refusing to start until a mechanic is provided.

Despite the fact that the optimum failure strategies for end users during runtime and the chosen failure strategy of developers during development are conceptually separate, it happens in practice that although maintaining two completely opposite strategies in the same application is hard, it is not impossible.

When default behavior becomes non-trivial, it can be useful to resort to a specific class to represent it, to decouple it from the rest of the GUI, enhancing code readability and eliminating tangled conditionals dispersed in the code. This allows default strategies, spanning client to server and database tiers, to be represented at a high level of abstraction.

The nastiest situations arise in practice from the repercussions of unforeseen data such as null values or empty lists on the subsequent execution of the application. When encountering an unforeseen value such as a null query result, applications are usually programmed to make some assumptions in order to provide a minimum degree of ruggedness, for example by displaying an empty results table, instead of throwing an embarrassing `NullPointerException` in a pop-up message dialog.

The problem is that from that point on the application slips into uncharted waters, while still being fully responsive to the unwary user – that is, its actual behavior is no longer clearly defined. When an unrecoverable error happens five minutes later, it might be quite hard for developers to track down the sequence of events that led to it.

Extensive testing will hopefully catch most of these situations, but without an *a priori* strategy the code will contain a cluttered tangle of `if (a!=null){` statements and endless, convoluted chains of default behaviors.

A simple remedy to this is to provide a clear, global strategy for failure and its implementation with OO technology as early as possible, such as a set of instances of the Special Case pattern (Fowler et al. 2003) that explicitly represent special cases by defining subclasses for handling special cases only, such as `EmptySearchResults`, `UnspecifiedAddress`, `NullObject`. These classes will know what to show on the screen without forcing conditional control to be scattered throughout the GUI, will properly log themselves, and will prevent the application from being crippled unpleasantly in some unforeseen situation.

Error messages

Making an application fail is a well-known form of security threat. A surprisingly high number of Web sites are relatively weak in this respect, and can sometimes be made to fail by reusing the data obtained from empty or non-meaningful queries, for example. Web applications are supposed to be much simpler than fully-fledged GUIs, so the threat is even more serious for rich clients and other client applications. Alas, GUI developers usually overlook security threats, because they assume that a restricted end user population, as is often the case with Java GUIs, will shield them from malicious use.

Fortunately, client GUIs are less restricted than Web applications, and a local log file will be able to provide technical information to the developers, and not to the end user – who shouldn't be bothered by these details, or even worse, might potentially use them against the system. Separating technical error messages from end user messages in two different distribution channels (log files and the GUI) simplifies the error notification architecture, while ensuring higher levels of security.

In some scenarios it could even be desirable to provide users with low-level technical details of errors, for example an application that is intended to be used by developers such as an Eclipse plug-in.

5.7 Introduction to user interface testing

When developing iteratively it is essential to maintain the code tested, launching unit tests after every change. More rarely, we might change the GUI design too, perhaps refining an existing feature or adding new ones. In this case we may want to test the application for usability as well as for technical soundness. While being two different practices involving different skills, both GUI test and usability testing are essential for an effective final result.

Agile approaches offer a new and refreshing 'take' on testing. The approach of giving test responsibility to the developers themselves from the early stages of development is a radical departure from 'old school' QA approaches, in which an unspoken adversarial climate often arises between developers and testers, complete with different cultures and career paths and a perception of testing as an authoritarian practice that takes place after completion of development.

Testing can be seen as a long-term investment in code – the additional investment is repaid in the small cost of further code modifications in the medium to long term. Simple forms of testing can escalate into testing practices that influence the structure of production code heavily.

Tests should be written to cover newly-written code and existing code that has been modified. Unit testing is perhaps perceived by developers as the most valuable form of testing, because of its fine granularity that allows a high coverage of the code base. No matter which type of test you use, automatic tests should be launched as part of a continuous build environment.

Test-driven development

Perhaps the single most important advice about testing, which has been validated empirically by decades of development practice, is to *test early*. The sooner, the better. Taking this to the extreme, we have test-driven development, which prescribes that tests should be written even before the code itself. This strategy works well when developers are motivated to build effective tests, but can otherwise result in a development overhead that produce vapid, ineffective tests.

Test-driven development (TDD) focuses on developers writing unit tests before writing code. It improves the design by providing goals, guidance and early feedback to developers, reduces coupling, and improves cohesion. It may involve major use of refactoring and other practices such as specifications and testing by example, as well as Agile techniques such as providing automated regression tests written in collaboration with customers.

Tests also provide a measure of a project's success and a realistic indication of overall progress. All of this nice magic comes to a price, of course. The price is higher development costs, a change in mind-set requiring greater motivation from developers and managers, and a generally more labor-intensive, responsible development style.

It is common for developers to focus the implementation and even the architecture on easing testing or other implementation aspects. While this is common practice for software that does not interact with end users, for desktop application GUIs this practice needs deeper thought. The real question behind this approach is how much the development should influence the final product – in our case, the GUI design, its performance, and its overall usability.

What's first – GUI design or implementation?

In the early Middle Ages in Western Europe towers, a very important means of defense in those days, were built with an iterative process in which scaffolding was attached to the tower itself as construction progressed, greatly simplifying the building process. Traveling across Europe you can still see these old towers, which can be recognized by regular patterns of holes in their walls that were used to insert scaffolding logs.

A tourist might dislike this effect, as it is a temporary construction trick that affected the overall result right through to today. Moving from the building techniques of the past to current software engineering practice, a frequent question is how much a GUI design should be influenced by its implementation. We saw in Chapter 2 and at the beginning of this chapter that cost-driven design is an all-important practice, but even with this approach, usability and end user-centered considerations always have the last word over implementation details.

In real situations, especially with developers not familiar with GUI design issues and concerned mostly with implementation aspects, such as providing a robust, cost-effective and easily maintainable GUI, this might not be the case. To them, implementation is *the* priority, with GUI design considered a sort of a nice-to-have, slightly dangerous luxury.

Such developers would probably consider holes in medieval towers to be part of the design, not a side effect. Structural integrity is a quality achieved by means of the building technique employed, and not a spurious, secondary effect. Cost-effectiveness is an important part of a construction technique. Others may argue that GUI design is the final product, and development must serve the final result only, possibly constrained by cost-driven considerations. You can imagine how such topics were debated in past millennia for civil engineering and architecture.

Such considerations are also important in software engineering practice, because implicit assumptions made by developers can drive the project towards unforeseen and dangerous situations. Contrary to server-side development, GUI builders also face customers' judgment. Imagine that you are an architect and your client, a wealthy entrepreneur, is paying you a substantial sum to design and build his next factory, a place where people will spend most of their daytime and which should be optimized to provide the best possible working conditions. Now imagine your feelings when during a design review the top managers and the boss ask you about the weird holes into the walls shown the drawings of the new building… equally, you don't want any holes showing in your GUI.

These apparently abstract considerations boil down to very practical situations when developing real-world GUIs. Think for example of the habit of many developers of keeping the GUI layer as thin and simple as possible. This makes extensive unit testing much easier, bypassing the GUI 'skin,' and confines presentation details outside the 'real application' automatically. Unfortunately this approach becomes burdensome as the complexity of the GUI increases, especially with regard to complexity of interaction with end users – think for example of complex, extensive interaction and control behavior.

To use a metaphor, it's like trying to build an easy-to-maintain and robust Formula 1 race car. It is hard to design your car for other objectives than speed and performance. Providing additional equipment and mechanisms for easing car maintenance could decrease performance and, as the competition gets tougher, be a costly luxury.

The bottom line is to design and develop as much as possible focusing on implementation details, as long as this strategy doesn't clash with usability and the overall, *user-perceived* effectiveness of the final GUI.

5.8 Software testing of Java GUIs

This section describes details of GUI implementation testing, and some techniques that help to build a GUI that is easy to test.

Exhaustive software testing of a GUI can be complex and expensive. You can trade technical complexity for cost, and let human beings test your GUI, or you can automate part of the testing to save time and money, but this may prove to be complex and limited, at least with current technology, a problem not confined to Java. GUIs and their building blocks are built for users. Only as an afterthought are they made available for automatic manipulation, and even when they are, it is not easy to declare interactions and expected behavior.

Expressing interaction properties of any complexity in a formal language in a simple and widely-adoptable way is a long-held dream of the GUI engineering community that is yet to prove feasible in reality.

This section provides a complete perspective of GUI testing. Practical examples are provided in some of the later chapters of the book.

How to test – GUI software test approaches

Referring to Figure 5.19, we divide our code into three broad categories for GUI software testing.

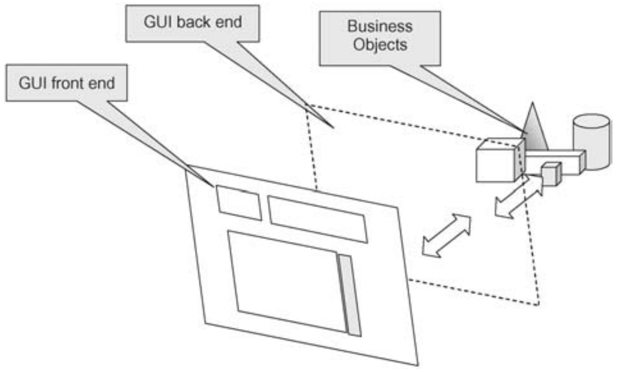


Figure 5.19 Partitioning code for software GUI testing

These are:

- *GUI front end code*. This is where widgets and all the graphics code lies, including the content layer. It is important to note that copies of the business data are stored within widgets as well, referred to as the *screen data state* – this is described in Chapter 8. We assume a general lifecycle as follows: some business objects' data is copied to widgets' data, and after specific user interactions via the GUI back end code, data is passed from or to the business objects.
- *GUI back end code*. This code oversees at the binding between GUI and data. In MVC terminology it is the controller code. This part of the code also contains the business rules and other control code. In the particular MVC flavor implemented in Swing, this code is contained within the widgets themselves.
- *Business objects*. These are the domain-dependent business data our GUI is representing, referred to here as the business domain layer. We assume that developers have tested these objects autonomously, using libraries such as JUnit, so we will not discuss their testing here, and take their integrity for granted.

  Some toolkits like Swing allow business objects to be used as GUI models directly, but for various reasons developers sometimes do not use this feature – that is, business objects are copied in and out of the MVC's model objects – so that we keep the MVC's controller and model explicitly separated for clarity.

The commonest way to test the implementation of a GUI is to get a person to test it. This is the kind of testing all of us have done many times in our lives. Figure 5.20 shows this situation.

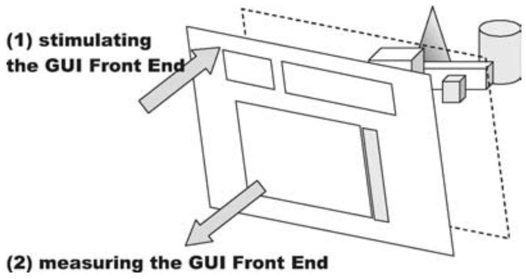

(1) stimulating the GUI Front End
(2) measuring the GUI Front End

Figure 5.20 Manually testing a GUI

The tester stimulates the GUI – pressing buttons in a certain order, typing in values, and so on – and sees whether the expected result is obtained. This kind of testing has all the problems we can imagine for testing:

- It is not repeatable. Even if a written test script is used, someone has to perform all the steps requested by the script.
- It is not 100% safe. Humans may make errors, both in manipulating the GUI and in interpreting the outcomes.
- It is expensive, because testers need a lot of time to perform extensive testing.

It has also some benefits, the major one being that unexpected problems can be found easily. Some tools allow the recording of test sessions and other limited forms of automation, as we will see, but for fully testing a real-world complex GUI, human beings are still necessary.

For effective automatic testing of a GUI, some form of modification of the implementation is needed. Special software access points must be added to the GUI code to allow it to be tested without (or with limited) human intervention, to allow the kind of interactions described above.

An example can help to explain this: think about a text field within a panel that a developer may want to manipulate and then make the resulting value available to the program. Access to the specific widget might require some form of OO visibility relaxation, such as making the field protected, for example, or some other form of runtime access.

A practical discussion of testing is provided in Chapter 8, focused on form-based rich client applications, although limited to a concrete case only.

Adding a layer of indirection between presentation and the rest of the GUI implementation works well for GUIs with a low level of interactivity. The higher the interactivity bandwidth with the user – that is, the more interaction and control behavior in our GUI – the more work is needed to maintain the additional decoupling. Ultimately, some form of testing through the GUI is always needed: for end-to-end tests, for testing interaction logic, or for (automated) acceptance tests.

Framework-dependent code in GUIs can be confined to the presentation and content layers – that is, the GUI toolkit in use. By adopting a Rich Client Platform, however, container-dependent code grows through the addition of business rule validation, data binding, multithreaded operations, and so on, and unit testing code needs to pass through this container-managed code. This situation resembles the testing of application-server contained Java server code.

The three most frequent approaches to GUI software testing are fully manual, semi-automatic and fully automatic. Each approach has its own benefits and drawbacks, and the best result is obtained when using two or all three approaches together:

- *Fully manual*. A test team ensures the robustness of the GUI by testing it directly. Documents such as refinements of analysis use cases describe detailed scenarios of use and their expected outcomes.
- *Semi-automatic*. Testers use some form of tool to automate some tests, usually lower-level ones. They launch scripts and inspect the results in the GUI.
- *Fully automatic*. Developers implement test cases provided by the test team. Such tests can be run together with the other unit tests as part of the code for the GUI.

The characteristics of these testing approaches are briefly summarized in the following table.

Table 5.2 The characteristics of testing techniques

Type	**Initial setup cost**	**Run costs**	**Precision**	**GUI coverage**
Fully manual	Medium / Low *(writing test cases in plain language)*	High	High	Low
Semi-automatic *(human tester with recording device)*	Medium /High *(learning/ purchasing tool, …)*	Low	Medium *(depends on tool)*	Medium/ High
Automatic	High / Very High *(tweak existing code, write code test cases)*	Low	High	Medium *(some GUI-only interactions cannot be tested fully)*

Designing for testing

A number of techniques can be employed to simplify automatic unit testing of GUI code. These techniques range from high-level design strategies to practical details. A number of design strategies can be employed to simplify API access to GUI code:

- Presentation Model is a design technique used in Smalltalk VisualWorks that aims to decouple toolkit-dependent code completely from the rest of the application. The data and the behavior of the GUI are isolated from the content. The class that represents the Presentation Model contains data that is displayed in a visual container such as a panel or a window and needs to be maintained in sync.

- Model-View-Presenter (MVP) is a variant of the Model-View-Controller (MVC) pattern discussed in the next chapter. This design pattern allows for a certain level of decoupling between the toolkit-dependent code and the rest of the application.
- Provide programmatic access. This approach aims to make as much as possible in a GUI reachable by API methods, so that automatic unit testing can be used to include behavior such as GUI events, interaction and control, and other parts of GUI implementation that are usually not accessible to unit tests. This is a fairly intrusive technique that requires many methods to be added to support automatic testing.

What to test – test coverage criteria

The following table shows the most useful types of tests available for 'unit'-testing widgets, that is, without interactions with other areas. For example, when ticking in a check box, are panels of related properties disabled? The italics show the tests than can only be run through the GUI back-end layer – that is, tests that are not available through GUI interaction.

Table 5.3 Data-bound widgets 'unit' tests

Data Type	**Widgets**	**Typical Tests**
List-Of	Combo box, table, List	0 elements, 1 elem., Random N elems., *Null value,* *1 Null elem.*
(Formatted) Field	Data formatted fields (Date, currency, etc.)	Empty value, Random valid* value, *Invalid* value,* *Null value,*
Ad-hoc	Ad-hoc component (for example color chooser, and so on)	Ad-hoc property, Empty value, Random valid* value, *Invalid* value,* *Null value*
Group of Boolean values	Check box, radio button	0 elements selected, *1 Null elem.,* *Null value*

(* Indicates values whose validity as defined by business rules, if any)

This type of testing can be extensively automated, including testing for special cases. We discuss some testing frameworks and tools that can be used for this purpose in Chapter 11.

In my experience many of the problems with robust GUI development today are due to non-optimal use of testing tools. The market offerings for GUI testing tools for Java are still fragmented and oblige developers to use a careful tool selection process, often using more than one test tool, depending on need.

An ideal GUI testing tool for Java

Every test tool currently on the market has some nice, unique feature that would be good to have in a comprehensive product. Perhaps this will never happen, but the characteristics listed here may be useful when choosing an existing tool.

- The ideal tool should be simple and lightweight, built with customers as reference users for acceptance tests, thus ensuring usability, simplicity, and so on.
- It should use a high-level scripting language, easily embeddable and usable by developers and non-developers alike.
- It should have a basic set of elementary GUI test functions that apply equally to SWT or Swing GUIs, and specialized libraries that provide both higher-level and toolkit-dependent behavior.
- It should provide hooks to the JVMPI interface, so that stress tests can be automated, abstract-to-concrete pick-selection, allowing the use of a widget logical identifier to find a component, then use GUI low-level events to fully simulate human interaction and integration with unit testing libraries.
- Essential features should include: proved in large, complex projects, provided with some IDE support, well documented with non-developers in mind, possessing a recording/playback facility, a lively support forum, and so on.
- Most importantly, it should be designed with a testing philosophy in mind. Instead of being a set of loosely-assembled diverse features, it should support testers, developers and customers throughout the product lifecycle.

Is this asking too much?

5.9 Usability testing of Java GUIs

Usability testing is an all-important form of testing, related to the semantic and emotional impact the GUI has on end users, the consumers of the product and those for whom it was built.

Usability testing of user interfaces is very different than the GUI implementation testing described in the previous section. While the latter can be thought of as the equivalent of testing text for grammar and spelling errors, usability testing is the equivalent of testing for poetical resonance and pathos. It involves a completely different set of skills and is a subjective form of evaluation, because it depends on the user population that will use the application. Results obtained in this way should not be generalized to other situations and users outside those in the test population.

Usability testing is important. An application that is difficult or aesthetically unpleasant and punishing to use will frustrate end users and increase the proportion who will ask for support or who will fail to complete application tasks effectively. This can ultimately cost more than the software's development.

We do not discuss usability testing in detail here – many books on this subject exist, such as (Nielsen 1993), (Rubin 1994), or (Snyder 2003) for paper prototypes. We do briefly discuss a practical approach to usability testing, leaving the interested reader to more specialized resources.

Usability tests are carried out with real users and using a specific number of defined tasks. They comprise the following activities:

1. Determine the goal of the testing. Possible goals include:
   - Testing the ease of understanding and ease of use of certain features.
   - Verifying empirically the way real users perform specific tasks in particular situations .
   - Collecting some form of data for an empirical assessment of specific GUI aspects, such as the average time to accomplish a task, how often a critical operation is achieved successfully, and so on.
2. Defining the user population and the user profile for the intended tests.
3. Finding suitable users that correspond to the profile, or picking representative users from the client's organization.
4. Defining the tasks that users will perform in the testing environment.
5. Preparing the application, or the prototype that will cover the tasks, for usability testing, including data and other simulated support, such as remote communication delay times.
6. Defining the boundaries of the prototype (see Figure 5.2 on page 177) and testing the application or prototype internally before using it for usability testing with real users.

7. Conducting usability tests with users on given tasks:
   - Usability testing is a delicate form of testing. Giving guiding instructions or letting the tester struggle fruitlessly for half an hour with a particularly cumbersome feature can both make testing a waste of time.
   - A single test usually lasts half to one hour.
   - Testing consist of letting the user use the application to perform the planned task while recording details about their experience unobtrusively.
   - Special attention should be given during usability testing to issues such as choosing the most realistic test context.
8. Collecting results from the tests, prioritizing the issues found.
9. Applying the feedback obtained. This implies modification of the GUI design to address the most important issues discovered during testing.

In the development of an experimental plug-in for Eclipse I created a special additional plug-in to observe the user at work, producing a sequence of screen shots that provided a record of the user's behavior while solving the proposed tasks. Such material, together with handwritten notes taken during the testing sessions, is extremely precious in understanding the usability shortcomings of the application with a specific user population.

Don't forget to use some form of 'informed consent' agreement signed by your users prior to testing, explaining the purpose of the tests, the amount and type of data being collected, and other privacy concerns, such as the fact that all user data is collected in an anonymous way.

Several different roles are involved in the creation and running of usability tests:

- Those who design the GUI, comprising some developers, and those who created the prototype.
- Usability testers, who conduct the tests and takes notes.
- End users, the subjects of the testing.

Many problems can be isolated by the use of simple prototypes. Once spotted during usability testing with a prototype, such problems can be tackled at an early development stage, saving money and time. The most frequent problems are:

- Navigation and ease of accessibility of features within the GUI.
- Lack or unsatisfactory implementation of business requirements.
- The terminology and concepts used in the application.
- Visual issues such as widget layout in form-based applications, and so on.

5.10 JRE runtime management

This section discusses the profiling and tuning of application runtime resources, which is an often overlooked aspect of GUI development. We discuss this topic in this chapter because a simplified, focused form of profiling can be carried out during iterative development, and this can save valuable time and energy in the medium to long term, much as can continuous testing.

Performance is a concern in any non-trivial Java application. Making a Java application perform well is a matter of design, implementation and profiling skills, as we will see.

Introduction to profiling

Profiling an application allows a developer to glean useful metrics, such as the memory use of a given object and the execution times of specific methods. This can provide detailed and valuable insights into how an application is performing. Even with a good design and the best developers, issues related to performance or memory management can be introduced, particularly in programs that consist of multiple layers and deep object graphs.

We introduce JRE profiling here in a general way, abstracting from the many tools that are available for it. The general concepts can be applied to any tool. It is important that every developer is confident with even a simple profiler, at least for detecting blocking runtime issues early in the development process. The good news is that profiling for desktop application GUIs is easier than profiling server applications, and after a little practice results will be easy to achieve.

There are two main approaches to profiling:

- *Preemptive*. Profiling is done to prevent problems occurring. We want to keep preemptive profiling as simple and cheap as possible, because if it becomes too difficult we will abandon it. For this reason preemptive profiling should be fully automated and as rapid as possible.
- *A posteriori*. This is done after something wrong is discovered in an application and we need to understand the problem. Usually it is a deeper and more comprehensive analysis than preemptive profiling, and it is performed manually by expert developers.

The JRE provides a standard interface for profiling agents, JVMTI. The old profiling interface (JVMPI) is supported only in Java 1.5. Both these interfaces are native (through JNI) and provide a two-way interaction with the JRE.

Developers usually discover profiling when a problem is found in the application, and this often happens close to the release deadline – or even later – when integration tests are run extensively. This can result in quick and dirty solutions that might spoil an otherwise carefully thought-out design.

Systemic and application-level concerns

Measuring, inspecting, and acting on threads and JRE runtime memory allocation is different than working with GUI events and widgets' data models. Using a debugger and executing tests are application-level activities, while profiling operates at the 'systemic' level. Systemic is a concept borrowed from biology, and is used to indicate something relating to or affecting an entire living organism or one of its subsystems. To make an analogy with newspaper editing, when we perform tests of any type, it's similar to editing an article for grammar, while when we do profiling, it's like examining the paper on which the newspaper is printed.

It can be hard to track a problem from its systemic, low-level effects back to its application-level causes. For example, an improper use of the GUI event queue can cause unnecessary production of GUI events, which will in turn appear at a systemic level as an excessive thread overhead, or in some cases thread contention.

Luckily, Java technology allows for fairly transparent access to the JRE's inner workings, allowing developers to inspect the fine-grained detail of runtime objects, threads, and resources.

Profiling techniques

Two main techniques exist for inspecting runtime performance in Java code, which are often used together:

- *Bytecode instrumentation*, also known as 'bytecode injection' or 'bytecode insertion.' This technique transparently modifies the `.class` bytecode and inserts special code to capture events like method entry, method exit, object allocation, and object freeing, while the code is executing.
- *Profiling agent sampling*. The JRE-native interface to profiling agents allows for interactions with a running JRE. Examples of such interactions are querying for current threads and their status, obtaining a memory heap dump, or invoking the garbage collector.

At the application level we don't have to know how these features are accomplished when we use a profiling tool.

Profiling can also be used for understanding the working mechanism of portions of an application whose source code cannot be accessed, or that it is extremely hard to understand. This is often the case with third-party libraries or with large and tangled code bases. In this cases it can be useful to inject special data into the application that has a recognizable pattern, and track its transit inside the application at the systemic level, much as chemical or radioactive tracers are used in medicine. When code is truly obfuscated, however, even this technique can prove ineffective.

Common problems

A number of problems can be detected by profiling, discussed in the following subsections.

Memory leaks

Memory leaks are characterized by an unstable memory allocation that will eventually halt the JRE with a `OutOfMemoryException` error. Memory leaks are characterized by the following equivalent effects:

- The heap decreases after every garbage collector (GC) invocation: the average heap size appears graphically as a downward linear graph. The steepness of this graph reveals the rate of memory leaking after each GC invocation – see Figure 5.21 below.
- On average the GC successfully discards fewer and fewer objects at every invocation.

As Figure 5.21 shows, the free memory heap size in a Java application should oscillate around an average value, which can be roughly shown as a straight line. The application execution will create new objects, while the garbage collector will periodically remove those that are no longer referenced. In the case of a memory leak, the average heap size appears as rising graph.

The greater the slope of this graph, the easier it is to isolate the location of the problem at the application level. This happens both because it is easier to notice large changes in time, and also because for a developer it is easier to spot differences to other object allocation trends that remain roughly constant.

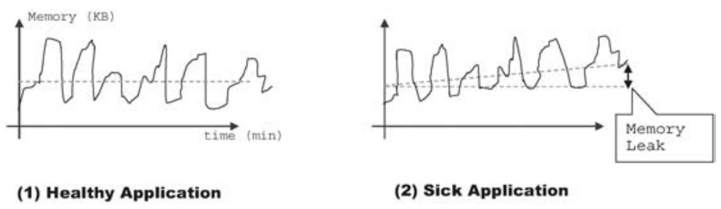


Figure 5.21 JRE heap memory allocation profiles

The strategy for finding memory leaks is similar to the strategy used for finding other performance problems. Start from the effects and backtrack, from the invoked method to the method invoking it, and so on, until the source of the problem is detected. Most profiling tools have special performance data view to make problem detection easier.

`OutOfMemoryException` errors can be thrown for reasons other than memory leaks. An infinite recursion loop, or just too small a heap size, can exhaust the JRE's available memory.

Typical occasions when memory leaks can occur in desktop application GUIs include screen disposal that is not performed thoroughly by means of an explicit `disposeResources()` method. This is specially true of Swing applications, as many developers refuse to write such methods, convinced that resource disposal is performed automatically by the library[4].

Consider the case in which you have an `Observer` instance registered to a `Subject`, for example a subclass of `Observable`. Now you dispose of the screen, which is perhaps implemented as an SWT `Dialog`. The `Observer` instance does not get disposed, because a reference to it is kept into the `Subject`'s list of listeners.

Another common situation is that in which some utility class such as the help manager is used, and we register an object contained in a screen to such a utility class – for example, through JavaHelp's `CSH.setHelpIDString(widget1, "widget id");` method. This reference now keeps the object alive, as well as all other objects it refers to within the disposed screen.

Sometimes it can be time-consuming to track down the location of a performance problem. It may be the case that a memory leak is so negligible that you would need hours of interaction to spot the cause of the problem. To speed up the detection process, you can artificially exaggerate the problem. In the case of memory leaks caused by an incorrect resource deallocation, for example, it might be useful when working with Swing to install a Look and Feel with very memory-expensive graphics and resource consumption, so that after just a few interactions you can spot easily where the problem lies.

CPU hot-spots

Profiling can help you to identify methods that consume the most CPU execution time[5]. This is achieved by isolating the points in the code where the program spends most time, starting from the effects – the location where the time hot-spot is detected – and backtracking from method to method, starting from the method

4. This is true only as far as graphics resources are concerned, and provided that developers follow common use patterns.
5. Generic execution time also includes other running threads and the time resources consumed by the profiler process itself. As a first approximation, they can be thought of as equal.

currently executing to the one that invoked it, and so on, to detect where the problem lies.

Threading issues

Threads that are competing for locks exhibit the phenomenon of thread contention. Luckily, a careful design and implementation will prevent this sort of issue in Java GUI code. Desktop application GUIs in Java are usually built on a simple single-thread scheme, as adopted by Swing and SWT. Simple design criteria ensure no threading issues as long as some basic rules are observed.

In Swing applications, for example, two main rules shape thread design:

- Manipulate Swing components – that is, invoke methods on Swing widgets – only from the event dispatch thread (EDT). This is because JFC/Swing is not thread safe, contrary to AWT.
- Lengthy tasks should not be performed on the event dispatch thread, because this will freeze the whole application. Instead, use the `SwingWorker` class to fork a new thread, allowing the application to remain responsive. Return to the EDT only when the results from longer processes are available.

Access to the EDT is achieved by means of the `SwingUtilities.invokeLater()` method. Too many such method invocations can hinder performance. The `SwingWorker` class supports the coalescing of `Runnables` – that is, many small `Runnable` instances merged into one – to ease this problem. Simple test classes can verify automatically that all widget manipulation is performed within the EDT[6], and that the EDT is not clogged by too many `Runnables`. For example, you could periodically create a `Runnable` to be inserted in the EDT that measures the elapsed time for its execution since its insertion into the EDT.

Thread problems with SWT are immediately obvious – in contrast to Swing, SWT does not allow widget manipulation outside the EDT at all: instead, a runtime exception is thrown. In cases in which data from another thread needs to be provided to a SWT widget, the method `display.asyncExec()` provides a similar function to Swing's `invokeLater()`, allowing a separate thread to communicate with widgets.

Garbage collector activity

Excessive garbage collector (GC) activity should be a primary concern when optimizing performance. An application may exhibit excessive object creation or object retention due to bad design. This will cause the GC to be launched more

6. See for example the Spin project at http://spin.sourceforge.net/.

frequently than it should, slowing the application's execution and thus its interaction with the user. The following situations can pose an excessive burden on the GC:

- *Excessive object turnaround*. Creating too many short-lived objects will cause the GC to be invoked more often than needed. This is the case in a loop that creates many temporary objects, for example.
- *Excessive object retention*. Storing objects that are no longer needed reduces the available memory and thus forces more GC invocations.

In interactive GUIs, excessive GC activity may affect the application's responsiveness. Imagine that you are using an application that occasionally and unpredictably freezes for few seconds, then returns to normal responsiveness. This is very frustrating. A reason for this bumpy type of interaction could be an excessive heap size, requiring much time to parse during GC activity, or some other form of poor GC tuning.

In tuning the GC for interactive GUIs, the focus is usually on minimizing *pauses* – the times when an application appears unresponsive because garbage collection is occurring – instead of maximizing *throughput* – the percentage of total time not spent in garbage collection, averaged over long periods.

J2SE 1.5, differently than 1.4, chooses the GC algorithm automatically depending on the type of machine on which the application will run. For more information about JRE's GC tuning, read the excellent Sun documentation for the J2SE version of interest.

In enterprise applications and rich clients by far the predominant source of latency and lack of responsiveness is caused by remote communication and the way it is designed. No matter how well the work of the garbage collector is streamlined, or how well local threads handle complex operations on the client, the latency of remote communication is almost always orders of magnitude greater than these client-side enhancements.

In this common case, optimizing data communication over the network will have an enormous impact over the overall performance perceived by the end user.

Continuous profiling

Continuous profiling is an automated, simplified version of application profiling that focuses on isolating the most serious systemic problems as early as possible, such as:

- Memory leaks
- Thread deadlocks and contentions

Continuous profiling doesn't reveal the exact line of code where the problem lies. Instead, it generates an alarm signal for developers to investigate a serious problem while an application is still in production, using a smaller and easier-to-examine application. Developers will have fresher knowledge of the implementation at this stage and will be able to spot the issue more quickly than they might a few months after product release. Finding the problem and solving it in parallel with on-going iterative development augments the chances of providing a good solution without a last-minute rush.

Continuous profiling demands automation, at least of input stimuli, to emulate end-user interaction. At least two main scenarios must be simulated in an automatic and repeatable fashion: stress and average use of the application. These tests are applied to the application and appropriate performance measurements – elapsed time, available memory after a GC invocation – are verified. When using JUnit, for example, JUnitPerf, a collection of specialized test decorators, can be used to measure performance automatically.

Premature optimization is the root of all evil[7]

Continuous profiling should focus only on isolating systemic problems that may seriously hinder the application, or stop its execution completely. Any further optimization should be postponed to *a posteriori* profiling sessions, if any. This ensures that the application will work well without last-minute nasty surprises, and without wasting precious time optimizing code that may later be heavily modified or discarded.

A posteriori profiling

This is the commonest form of profiling, performed when a serious, blocking problem is threatening development and it needs to be isolated and fixed, usually in a short time-frame. Common issues for this kind of profiling are:

- Finding the slowest methods. Optimizing performance is a frequent theme in Java GUIs, especially in cases of non-trivial tasks and limited memory resources. As with CPU hotspots, this is achieved by backtracking from invoked methods to the invoking ones to discover where the bottleneck lies.
- Detecting where most garbage collection activity is concentrated. Mysterious abnormal GC activity degrades the performance of an application, making it almost impossible to use on some machines. The culprit is usually an area in the code where there is excessive object creation and subsequent disposal, possibly within a loop.

7. Despite being traditionally credited to Donald Knuth, this popular quote is of uncertain origin. Others credit it to Edsger Dijkstra.

Several other common profiling and optimization issues can be verified during *a posteriori* profiling. This is unfortunately the most common case in practice. We are pressurized to find memory leaks and thread contentions only after they bring the application to a halt, and in the worst possible time-frame: close to the product release date.

A posteriori profiling is also concerned with careful fine tuning of application performance, if necessary. This can be different than the profiling work we have discussed previously, which is only aimed at avoiding blocking problems during program execution.

A posteriori profiling often needs the data from the particular context in which the problem surfaced in order to solve it. An interesting aspect of Java profiling and debugging technology is the ability to perform these operations remotely using a dedicated communication protocol. This allows developers to 'plug into' a client's JRE at a specific point during execution and inspect its current internal state, thus studying a problem within the actual scenario that caused it.

5.11 Summary

This chapter discussed the various techniques that are collectively used when developing desktop application GUIs iteratively using Java technology. Although we focused on J2SE and desktop GUIs, much of the advice provided here is applicable to J2ME applications as well, and, with some modifications, to Web GUIs too.

We discussed the important issue of ordering activities to provide a more practical approach to iterative development by focusing on the most important issues first. We then presented the different alternatives available for producing scaled-down, inexpensive representations of real GUIs using Java technology.

We also provided an introduction to refactoring practice and testing for software soundness and general usability, both much-needed techniques when developing iteratively. The chapter concluded with an introduction to the often overlooked practice of profiling for runtime resources, another useful tool for producing sound and usable GUIs.

The next chapter delves into software design details for building professional Java GUI applications.

6 Implementation Issues

As we saw in the first part of the book, user interface design is not a matter of taste, or at least, shouldn't be. On the contrary, while the exterior appearance of a GUI should adhere to standard design guidelines, in practice the inner software design is more or less left to the developer's goodwill, with the tacit assumption that the implementation is okay as long as it works.

This chapter discusses some of key issues in the implementation of Java GUIs, such as how many closely-intertwined objects can communicate in a modular fashion, which criteria are traditionally followed for organizing the code of complex GUIs at design time, and how user interactions and the way the GUI reacts to them are represented and managed. General problems are introduced and the most effective solutions to those problems proposed. Such solutions usually imply adopting one or more OOP design pattern and other techniques[1]. The chapter organization follows the functional decomposition of the general model introduced in Chapter 1. The chapter is structured as follows:

6.1, Revisiting the abstract model discusses various issues related to the implementation of GUIs and the abstract model presented in Chapter 1.

6.2, Content discusses common design solutions for implementing the content layer, such as content assembly and navigation.

6.3, Business domain illustrates the main issues related to representation of the business domain in GUIs.

6.4, Data input-output discusses general design issues concerning the data I/O layer, data communication and code security, and the Data Transfer Object (DTO) design pattern.

6.5, Making objects communicate introduces the Observer pattern and its variants, and discusses the pitfalls of event-based designs and other related issues.

6.6, Separating data from views discusses the main design strategies used for separating data from its visualization, discussing MVC and its various flavors.

1. Most of the patterns described here can be found in (Gamma et al. 1994). For a discussion that is more specific to Java (but with a smaller selection of patterns) see for example (Cooper 2000). GUI-specific and original patterns are also discussed.

6.7, Interaction and control introduces the three main design strategies for implementing this functional layer in Java – scattered, centralized, and explicit design.

6.8, Some design patterns for GUIs introduces other patterns and design strategies that are useful in more than one functional layer.

6.9, GUI complexity boosters lists some implementation issues that dramatically complicate software development for Java GUIs.

6.1 Revisiting the abstract model

Chapter 1 presented a generic, abstract GUI model, in which functionalities are decomposed in layers, as shown in Figure 6.1.

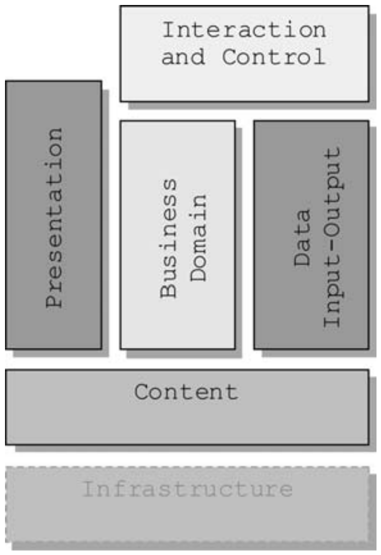


Figure 6.1 An abstract model decomposition

The functional layers in the figure are:

- *Business domain*. The representation of the domain of interest, without references to GUI details. This layer can be modeled using a domain-driven approach (Evans 2004).
- *Content*. The 'structure' of the GUI: widgets, panels, windows, and navigation among different windows. Layout is also included, to ease the understanding and manipulation of widgets.
- *Data IO*. The interface with the rest of the software that supports all interaction with the GUI other than the user's. This layer defines the

communication data in applications that need to exchange information with remote servers.

- *Infrastructure*. Low-level support, GUI frameworks, runtime environment, utilities.
- *Interaction and control*. Low-level events and control logic. This layer contains controls such as disabling the commit button in a form when a required field is empty. Despite being business-dependent (like any form of software) this type of control is also generic and can be factored out as a separate layer, leaving the domain model more focused on business logic and less on GUI details.
- *Presentation*. Graphical details dependent on the given presentation technology. Pixels, colors and the like are confined in this functional layer.

Functional organization – that is, storing and organizing things depending on their use – is a criterion we all use extensively in everyday life: for example, we don't look for our car keys into the fridge. This model suggests a comprehensive organization of GUI implementations based on function, together with a minimal organization of relationships in layers[2]. The main purpose of this model is to provide a useful trade-off between generality and practicality. For example, navigation is considered part of the content layer, and not of the presentation layer. This is because it is easier to define navigation during prototyping and early design, together with GUI content. As with any classification, the decomposition into layers proposed in this abstract model highlights some aspects and ignores others.

One of the most useful benefits of the model is in decoupling responsibilities. For example, having a clearly-separated business domain layer helps when applying all the experience and tools object-orientation has provided over the years. Analysis and design patterns, refactorings, Domain Driven Design and more become available for non-trivial GUIs. All of this power and its related complexity may not always be needed, of course. In such cases some of the layers can be merged, until a unique, comprehensive 'blob' of presentation, data and business logic is obtained, such as the one-layer architecture discussed in Chapter 7[3].

Having a general functional model also helps to move more easily between technologies. This is especially useful for the Java world, in which many competing technologies and tools can be used interchangeably. Several libraries exist for data binding, multithreading, or GUI testing. They can be mixed effectively as long as a sound decoupling between different functional aspects can be enforced.

2. See Chapter 7 for a definition of a software layer.
3. Also known as the 'Smart GUI antipattern' (Evans 2004).

Some issues are common to all the layers of the abstract model in Figure 6.1:

- *Adaptation*. Desktop application GUIs adjust themselves to context data such as the locale or the graphics resources available. From an implementation viewpoint, these external factors work like *parameters* to the GUI. There are many forms of adaptation that may affect some or all of the abstract layers in Figure 6.1, as we will see later.
- *Requirements*. Requirements may apply to any aspect of the GUI and need to be addressed explicitly throughout the software lifecycle. Some requirements may be specific to only one functional layer, such as for example a business rule, or cross several layers, such as details of the data handled in a given screen.
- *Testing*. The various kinds of testing discussed in Chapter 5 affect all the functional layers.
- *Preferences and configuration data*. GUIs need to accommodate a wide range of situations. Each layer may have a set of configuration data and user preferences. Preferences are set directly by users by means of a preference panel, as discussed in Chapter 4. An example of preference data, which mainly affects the presentation layer, might be selection of a special look and feel for visually-impaired users. Configuration data is more implementation-oriented, for example defining the time interval between which clients ping their server, or the JRE memory configuration, and is set manually by users, or in some circumstances by an administrator.

Testing the various layers

Testing follows the general model proposed in Figure 6.1. Because the content layer is the base for the other functional layers, testing it is also useful for testing all other layers. The infrastructure layer and its code – GUI toolkit, third-party libraries, and so on – usually don't need to be tested. A common approach to unit testing is to limit testing to a functional area of interest. Depending on the layers in Figure 6.1, different tests are possible:

- *Business logic tests*. Testing domain logic should be done in business logic terms, not through the GUI. It is pointless indirection to translate business logic tests into GUI interactions that in turn invoke business domain objects. Client business logic tests are usually a subset of the comprehensive test suites found on server software. Integration and acceptance tests will of course check all the functional layers in an application via the GUI.
- *Content tests*. To perform these tests, the content layer implementation should provide a means to access data and widget properties. As a basic facility,

content units such as widgets, panel and windows should be made accessible, usually by means of a registry[4] and unique ids.

- *Data IO*. Data tests are predictable and can be largely automated or generated. Some possible tests are:
  - Data binding from data transfer objects (DTO) to widgets. Testing for null values, for empty collections, and so on.
  - Results of commands, especially from server to client. Client to server testing is performed as part of interaction testing.
  - Sequences of commands and other control data.
- *Infrastructure*. This layer is composed of support frameworks, GUI toolkits, and other third-party libraries outside the application developer's control. Although it should be possible to take the infrastructure's soundness for granted, sometime this may not be the case. When developing for a new release of a rich client platform, for example, or isolating the causes of some unexpected behavior, infrastructure testing can be useful.
- *Interaction and control*. Trigger interactions and assessment of the results can be performed thanks to facilities in Java GUI toolkits that simulate input and allow widget properties to be probed. Such tests are the cornerstone of automatic GUI testing. By building on them, it is possible to represent complex interactions and define GUI acceptance tests. This kind of testing is fundamental in agile methodologies such as XP.
- *Presentation*. Presentation testing is rarely done, because presentation is more closely related to general usability testing rather than to specific unit tests. If graphics plays an important role in the software (such as a GUI toolkit or some visual tool) it may make sense to provide presentation tests. Such tests might look for expected pixel patterns in the resulting GUI, or prescribed colors, and so on.

The principle of Single Functional Responsibility

The *Single Functional Responsibility* principle is a simple yet useful design technique, and its associated code documentation, that can be used in the development of any GUI. I derived this technique from my practical experience of applying R. Martin's principle to GUI development (Martin 2002). This is a formulation of the cohesion principle in designing classes, and states that a class should be designed to have only a single responsibility.

This principle can be mapped to the functional layers in Figure 6.1 by striving, when it is meaningful, to have classes that belong only to a single functional layer. When this is not possible, sometimes we might apply this approach to the

4. See for example (Fowler et al. 2003).

fine-grained level of methods as well. By designing fine-grained methods to belong to a single functional layer, code can be kept decoupled and different responsibilities organized by functional layer, additionally to domain-specific responsibilities (managed by the single functional responsibility principle). This can be seen as an addition to the general single functional responsibility principle.

A simple technique for applying this principle in code is to tag methods and classes with metadata. One simple tagging approach is to tag the method (or class) with the main functional layer from Figure 6.1 to which it is thought to belong.

Assigning a single functional responsibility to a method or class is a good discipline that tends to keep code more decoupled. This is less important when adopting a layering technique that is based on functional decomposition, as metadata tagging becomes redundant because the functional responsibility of the class is then defined by package or layer identity.

Suppose you are developing a widget library in which pixel spacing must comply with specific guidelines. To test this, you could prepare a `testPixelCompliance()` test fixture and tag it as `@Presentation`, meaning that you mean to test the presentation layer:

```
@Presentation Public void testPixelCompliance() {
```

The use of metadata could enable automatic processing and other features beyond mere code documentation, even though the main intent is to support a clean OOP design. It's possible to build on this approach, describing complex architecture information with metadata and their attributes. This is discussed in Chapter 7 in the context of evolving architectures.

Isolating presentation details

The presentation layer in the abstract model of Figure 6.1 is composed of those graphical details that are not strictly related to content functionality, data, and other non-graphical aspects. A common design strategy is to enforce the same type of separation as is provided in the reference model. This is often done at the level of infrastructure libraries and basic GUI toolkits, in that presentation details are intrinsically *extensive* values (that is, they are common to all widgets and screens in a GUI) and centralizing them in a separate implementation module eases their application throughout the whole GUI, transparently from application code.

In particular, it is useful to isolate the implementation of the following details from the rest of the implementation:

- High-level visual details such as graphics design (that is, the visual aspects in a look and feel).
- Interaction behavior. Some forms of user interaction styles can be separated from the implementation, such as whether a button is triggered by a single or

a double click. Although a powerful feature, it is one that is seldom used: achieving this kind of separation would provide a complex implementation structure that will complicate the overall implementation, providing little practical utility.

- Fonts are an example of a presentation detail whose management is usually defined separately from the rest of the GUI implementation, to provide separated management and centralized access.
- Colors and color themes are usually factored out in separate modules to provide easy customization of GUI appearance.

Separations of this type are achieved by Swing thanks to its pluggable Look and Feel design, but SWT, AWT, and many other modern GUI toolkits also enforce some variants of this modularization. It provides important advantages: for example, GUI appearance can be customized independently of the application, maybe by setting a large font in the current profile and so changing the font in all other applications. GUI can also be made 'skinnable' – different visual styles can be applied without modifying application executables, even by the user. User preferences can be adjusted transparently to application code, which is very useful for supporting visually- or kinetically-impaired users.

6.2 Content

This section discusses two key engineering issues for the content layer: content assembly and screen navigation.

Content assembly

A practice common to all desktop GUIs is to place a set of widgets on screen. The widget's layout depends strongly on the chosen layout manager, the object that is responsible for abstracting layout details. The positions of widgets are abstractions of real X, Y screen locations handled by the layout manager. We refer to the procedure of preparing widgets and assembling them in a visual container to be shown on screen as *content assembly*.

The simplest way to implement content assembly with OOP is to subclass a container class (that is, panels, or windows) and provide widget initialization and layout code in its constructor, or in another method, as in the following idiom:

```
panel.add(new OkCancelPanel());
```

An alternative way to implement content assembly is to provide factory methods that create the required, pre-assembled panels, as in the following code:

```
panel.add(Factory.createOkCancelPanel());
```

When there are many variants of panels, or when the application is extremely large and complex, one can resort to a 'little language[5]' to describe the layout and widgets involved:

```
panel.add(Builder.create("btn:ok, btn:cancel"));
```

Content assembly is almost always a statically-defined behavior (that is, content doesn't change at runtime). In a few cases content assembly can be adapted to external parameters – this is discussed in Section 6.8. For usability reasons content assembly should be made variable at runtime only in few situations, for example when changing the layout of widgets in reaction to some event, such as showing advanced search features in a search dialog.

Who assembles content?

From a pure OO approach, content should be created by objects that own the corresponded data and that represent the business domain. Such 'responsible' objects are in charge of representing domain knowledge and also of how such information should be represented on the screen. From a practical perspective, though, enforcing such an approach extensively can be complex, because in following it, it is easy to mix business logic with presentation details.

Assembler intermediaries can be called in to take care of the visual details needed to represent domain information objects. But such intermediary objects, such as a `BankAccountPanel` class that represents a `BankAccount` instance visually, should be treated with care by developers, in that they carry sensitive information and are not just a mere implementation requirement.

Content details are extremely important, because they convey essential semantic information to users. When this information is business domain-dependent, it should not be overlooked or, even worse, automated. Consider for example the position of an **Account number** field within a bank account form. While this is just another string in the class `BankAccount`, it may be the most frequently-accessed information within the account form, and as such deserves a prominent position within the form. It may also need some additional real-time searching facility, assuming that users search accounts usually by their number. Other content, conversely, obeys domain-independent rules, such as **Ok** and **Cancel** buttons at the bottom of a dialog. Such content can be assembled automatically, or, at least domain-independently of the rest of a screen. In this case content assembly can be implemented as support code, separately from domain-sensitive GUI design behavior.

5. See *Domain-specific and Little languages* on page 466.

Explicit navigation

Navigation, the flow of control from one window to another, is a major part of the user experience in GUIs with many screens, such as form-based applications. It is usually hard-wired into the code, like content assembly. Navigation behavior is usually assigned to event listeners, which invoke methods such as `actionperformed()` and the like.

In a few cases, though, navigation need special attention. In cases in which user adaptation is needed, for example if screen navigation depends upon the current user's role, using the Adaptation pattern will suffice, as it leads to the implementation of a suitable `NavigationManager` class.

In other cases, when navigation changes often, for example during development, or in a complex navigational mapping scenario, a centralized, explicit navigation scheme can be useful. Such a mapping of event $\rightarrow$ screen transition can be implemented with a hash map, or with a more elaborate structure in which other support information is represented together with transition rules.

6.3 Business domain

Representing domain logic within a GUI is always a tricky engineering issue. In client–server applications, domain logic should be limited as much as possible to the server tier. 'Pollution' of the GUI implementation with business code brings classic problems such as duplication of code – the same business rule code duplicated on the client and on the server – and code rigidity. On the other hand, confining business logic to the server can transform a rich client into a dumb HTML-like application, degrading responsiveness and overall usability. It is still useful to isolate business-related code, because it is likely to be one of the most volatile parts of a GUI, even for applications that don't exhibit repetitive remote connections, or for very simple ones.

In the following the term *business rule* is used to refer to a very specific form of domain-specific behavior in which domain logic is represented in the form of a declarative rule suitable to be handled by rule engines or an embeddable little language interpreter. By representing domain logic with business rules, developers can leverage the wide literature and tools available.

Many solutions have been considered for implementing a business domain effectively on a client application. Some of the possible strategies are:

- Including a rich domain model representation in the client application. Usually this is a subset of the wider domain model that resides on a server application, or is dispersed on various servers. This is the case for example when using Web services from multiple organizations.

- Adopting a software architecture for separating domain logic from the rest of the GUI implementation, as we will see in Chapter 7.
- Using an explicit representation of business rules that can interoperate with the rest of the GUI code and that can be deployed from a server at runtime as needed. This solution may involve the adoption of a little language, such as a script language specialized for business logic. This is a technically non-trivial solution that makes sense in large applications with many, mission-critical, and dynamic business rules.

  A cheaper alternative is to formalize business rules with a lightweight OOP framework that is embeddable into the rest of the client application and provides some form of 'zero-deployment[6]' mechanism.
- Separating business logic in packages or classes that are shared with the server code base. This eliminates code duplication, but necessitates a new application build and deployment to clients after changes in business rules.
- As a minimum solution, using the principle of single functional responsibility to identify explicitly portions of code that are intended to capture business behavior. This ensures simple traceability of business logic code within the application, but alone does not enforce decoupling and modularity.

Domain logic can creep into GUIs in unexpected ways. Suppose you have a form that shows customers loans. Your client wants customers with a debit rate higher than 10% of their annual income to be signaled by the GUI with a special warning icon, and to require extra confirmation when such customer's data is manipulated. A rule `isCautionCustomer()` is then clearly part of the business domain layer, even if it is used only on the client.

It is important to address explicitly the representation of business logic within the application design. Lack of awareness can easily lead to tangled code that becomes increasingly hard to maintain as the application evolves. For large applications it is also important to maintain common policies among developers to keep the code uniform and coherent.

In practical situations, some form of tolerance is often used to simplify the software design. Defining mandatory fields[7] directly in client code, for example, is a violation of the separation between presentation and business logic, because if such data should subsequently become no longer mandatory, the client code would need to be modified.

6. Zero deployment, also known as 'dynamic deployment,' is a general term for a number of techniques and technologies aimed at simplifying software installation and update, both for users and for developers (Marinilli 2001).
7. Mandatory fields are those widgets that must be filled in to complete data entry in a form.

6.4 Data input-output

Data IO is the conceptual layer that defines all possible interactions of an application with software external to the GUI. Depending on the application, interaction might be with a local database, with a remote server, or with a set of Web services. Figure 6.2 shows such a situation.

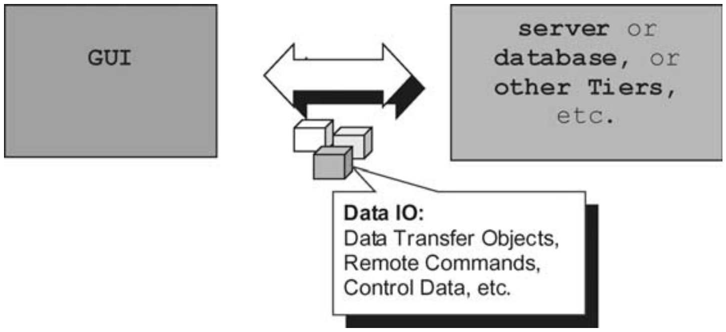


Figure 6.2 Interacting with external software

The main benefit of a well-thought-out data IO layer is the decoupling of the GUI from the rest of the system. This provides many benefits, such as clear conceptual and practical borders, technology independency, and greater flexibility, but also affect the whole application. Data IO is often overlooked as a detail, 'backyard' facility. But GUI performance, even GUI navigation and structure[8], depend directly on the data IO layer.

A good design for this layer should always consider a comprehensive data IO design strategy. For example, how will external communication evolve, and what will future requirements be? What's the driving force behind IO? Typical design criteria could be performance, flexibility, security, and interoperability.

A comprehensive data IO design strategy

While approaching the design of the data IO layer it is a good idea to work out an explicit design strategy. The main design criteria are:

- *Performance*. If the runtime responsiveness of a client–server application is a requirement, then DTO structure and the serialization format must be chosen carefully.
- *Flexibility.* Allow for ease of modification.

8. For example, windows are often mapped directly to DTOs.

- *Technology independence*. If independence from technology is required, the data IO layer should be designed accordingly. A common case is the ability to provide different presentation technologies for the same application cost-effectively. Depending on the type of technology, this can be achieved with various levels of reuse. A Web application and a rich client, for example, can share DTO and service information, such as commands and responses, while such sharing may be less for a J2ME applet, which might need much custom DTO. Designing DTO explicitly with flexible reuse in mind may be worthwhile.
- *Security*. Even if communication protocols provide authentication and security, it is always important to think about security up-front for security-sensitive applications.
- *Network* topology. Particular network topologies might favor some form of DTO structure rather than others. As a basic example, for communication performed with network portions using unreliable protocols/connections, small and simple DTOs should be designed.
- *Scalability*. An application deployed on a large number of clients, or exhibiting peak-like use patterns – say several thousands of users submitting transactions at the same time – should have a specific DTO design.
- *Interoperability*. Will the application provide its communication format to others?
- *Infrastructure services* such as security and authentication. These services are provided 'for free' by the underlying technology and should be taken into consideration as part of a data IO design strategy. Is there really a need to provide a home-grown, custom 'ping' protocol facility when adopting HTTP, for example?

It is good practice to focus only one main criterion. This will drive a clearer design and avoid dangerously ambiguous statements such as 'our application will perform the fastest possible remote communication while ensuring maximum levels of independency from data formats.'

Some design patterns

A number of design patterns are commonly used when implementing the data IO layer. These patterns are used for designing distributed systems, such as Proxy, and Broker. This section discusses the Data Transfer Object pattern, because it is specific to GUIs.

Data Transfer Objects

A Data Transfer Object (DTO) is an object used for holding business data in transactions between client and server. A single method call is used to send and retrieve the DTO, which is passed by value. In this way DTOs are used to reduce bandwidth: by substituting them for a number of remote calls to exchange data between client and server, data is clustered in coarse-grained chunks. Needless to say, DTOs should be kept as simple as possible, to speed up their translation to other formats such as XML. For this reason, when remote communication can be a bottleneck in an application, DTOs should contain other objects only when strictly necessary.

Remote communication design

The way a client application communicates with the external world over the Internet affects its user interaction style and the overall user experience. When designing the details of communication between a client application and its server counterpart, a number of options are available:

- *Asynchronous/synchronous communication*. Asynchronous communication is preferable when the communication channel is intermittent or unreliable, as in wireless communications, and also when synchronous communication might take too long. Synchronous communication is used in desktop applications as well because of its familiar conceptual model, similar to method invocation: issuing a request to the server and waiting for the response.
- *Bandwidth constraints*. From a bandwidth consumption viewpoint, desktop application GUIs are a blessing when compared with Web applications, in which all the presentation information must be sent with the data. In some cases, however, such as for wireless devices, bandwidth can still be an important issue. In such cases a proprietary binary format, or some form of object serialization, can be a necessary choice over other more common protocols such as HTTP.
- *User population*. Users affect the way a client–server communication channel is designed. The number of users concurrently using the application, the nature of the transactions, user habits, and various other details all influence the choice of communication design.
- *Scalability*. If you plan to deploy a client over thousands of installations, communication protocol should be able to cope with the likely scenario of thousands of concurrent communications.

Multithreading issues aren't considered here, as they are taken for granted.

Most of the time client–server communication will take advantage of the HTTP protocol, especially for desktop applications. HTTP is extremely useful in that it shields developers from a whole array of network-related issues, such as avoiding additional communication ports, proxies, and firewall administration. Most important of all, though, is its ubiquity.

Seamless deployment

Some form of remote connection is needed to install an application and keep it up-to-date, as this feature is now expected. CD ROMs or other physical means are usually expensive to create and distribute when compared with on-line deployment, and in a world of continuous releases, are useful only for major installations.

Seamless deployment, the ability to install an application directly from the Internet and update it as required during its lifecycle, is a *must* for modern desktop applications. In this book we treat it as a basic infrastructure service, such as fresh water or electricity. Without powerful and seamless deployment support, modern client applications could not exist. Such a feature can be achieved with a variety of technologies:

- OS-dependent ones like those provided by Microsoft on Windows machines.
- Fully Java-based ones, such as Java Web Start and JNLP.
- The Eclipse deployment facility (with a different feature set).
- On-line installer files.

What is important is that the installation is as automated as possible, even though for first-time Java users this will mean downloading JRE's 7 MB-plus and that, after installation, the deployed clients can be controlled remotely for the provision of updates[9]. Java technology also provides remote debugging and profiling, so that the idea of 'standalone clients, remotely connectable' is now largely obsolete.

Familiarity with these new technologies is important, as they affect the way the application is built and conceived, and affect the user's perception of the software. For example, they allow business domain code on the client tier to be updated seamlessly and inexpensively as required, or additional functionalities installed while the application is running.

Security issues

Security is seldom considered at the start of design when developing desktop application GUIs. Usually there is more to security than a secure transmission channel. An important part of security for client applications is ensuring the authenticity of the other party – clients to trust their servers, servers to authenticate clients.

Desktop application GUIs need to add another link to this chain of authenticated transactions: the user. The mechanism of user name and password is a widely-used form of authentication. Authentication mechanisms are needed for applications

9. See (Marinilli 2001) for a general discussion of Java deployment, even if slightly out of date for some technologies.

that transfer sensitive data to external entities, and also for accessing local resources. For example, a security XML file might be stored in one of the application's JAR files and used for collecting the addresses of trustworthy servers. Ensuring that it is never tampered with and fake (and dangerous) addresses inserted is vital.

Fortunately, security is addressed at various levels in all the technologies on which Java applications rely. HTTPS can be used to ensure secure communication channels, while fine-grained Java security policies or signed JAR files can be used for almost any aspect of the Java platform, or for local resources authentication.

Given the additional complexity that such technologies pose to development, developers often postpone these aspects to subsequent releases, even if the required details can be added relatively easily to the build environment, such as automatically signing sensitive files with certificates, and obfuscating executable code.

When using iterative development on a project on which security is a major issue, security should be implemented from the initial releases[10].

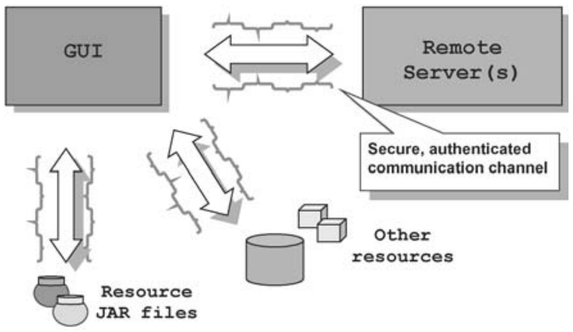


Figure 6.3 Securing communication

The following high-level steps are involved in securing desktop application GUIs:

1. Identify sensitive assets within the application.
2. Create a security architecture that considers security throughout the whole software lifecycle.
3. Detect and document possible vulnerabilities. This usually implies the entities shown in Figure 6.3.
4. Assess the risks and plan a risk strategy.

10. See Chapter 11, *Security tools* on page 412.

How users perceive security and privacy impacts their experience of an application as well. Such a perception is not limited to the GUI. This is a large topic that goes far beyond the scope of implementation issues.

One thing has been taken for granted so far – that the code base of the application, the `.class` binaries stuffed into the JAR files installed on the local machine, is safe. Unless you actively take care of this issue, the chances are that your executable code is absolutely open to all sort of attacks and malicious behavior. The very first step in securing an application at all levels therefore lies in securing its binaries first.

Securing the code base

Java code can be decompiled very easily. This exposes not only your intellectual property, algorithms, and architecture, but also the management of license keys, where an application is distributed with some form of license control, and virtually any other aspect of the application, including encrypted remote communication and authentication protocols.

Using a good code obfuscator[11] is not enough, because a good protection strategy begins with the design of the code itself. There might for example be situations in which you want to leave some classes open to your users, maybe because they are supposed to extend or interact with them, or times when you rely on class names for some reason, such as logging a class name along with an error message. In such common situations obfuscation cannot be a last-minute matter, but should be an integral part of the whole design.

Securing the code base goes beyond obfuscation to target the way an API can be exposed to malicious eyes, or unforeseen breaches left open through inattention. Imagine for example what could be extracted from a running application with a debugger.

In most cases, perhaps, nobody would be interested in your code, so a standard security policy would be fine – and *always* better than nothing. In cases in which security *is* an issue, because your code contains some secret algorithm, or just because competitors would love to see how you have implemented a specific feature, you need to resort to a thoughtful security strategy to protect your code base.

Such a strategy should focus on sensitive Java packages – those that need to be absolutely secure – and also on other code with a lesser security priority. The signature of methods and the structure of classes belonging to these sensitive

11. Chapter 11 describes a selection of available tools.

packages need to be planned explicitly and carefully designed, to expose the least possible information to malicious eyes.

The default approach to security is to include obfuscation in the build environment as a routine task, even if it is limited only to some packages, together with unit testing and continuous profiling.

6.5 Making objects communicate

This section focuses on a foundational aspect of GUIs implemented with OOP: the basic communication infrastructure as implemented with event-based communication mechanisms.

The following sections discuss the various OOP implementations of event-based communication that are part of the Interaction layer in the abstract GUI model shown in Figure 6.1 on page 224. Such implementations are not perfect, as they suffer from typical OOP shortcomings, such as too low a level of representation and an excessive cognitive burden on developers[12]. They are nevertheless one of the most successful applications of OOP to practical software engineering.

The event-driven object communication mechanism is a cornerstone of modern OOP GUI implementations. We first introduce the Observer pattern, then, after looking at some uses of its concepts in Java GUI technology, conclude by discussing two conflicting forces in any software design, object communication and decoupling, from a software design viewpoint.

The following section refers to OOP design patterns. A design pattern describes a proven solution to a common design problem, emphasizing the context of the problem and the consequences of the proposed solution. OOP design patterns have a number of benefits:

i. They are proven designs: they are the results of the experience, knowledge, and insights of developers who have successfully used these patterns in their own work.
ii. They are reusable: when a problem recurs, there is no need to invent a new solution.
iii. They are expressive: design patterns provide a common vocabulary of solutions that can be used to describe complex systems succinctly.

12. These shortcomings become significant in medium-sized and large systems with complex designs. A cognitive abstraction effort is often needed to mentally visualize and to correctly manipulate abstract concepts such as events from just reading the source code or the available documentation.

iv. Design patterns reduce the time for designing, describing, and understanding software. Clearly, wisely applying design patterns helps in writing better software, but it does not guarantee software quality.

The Observer pattern

GUI implementations typically suffer from the problem of trying to make many loosely-coupled classes communicate. The Observer pattern defines a one-to-many communication method by means of a publish-and-subscribe mechanism. Objects that are interested in changes in a *source* object's state, referred to as *observers* or *listeners*[13], register for later notification by *subscribing* to the source object's changes. Later, when the source changes – for example, if a new item is added to a collection – all its registered observers are notified. The source object does this by invoking a conventional method on each of the observers, passing a representation of the given event as a parameter.

Note that it is the source object that is responsible for triggering the notification event, by scanning its list of registered observer instances and invoking the method associated with the given event on each of them.

Although many possible variants of this pattern are possible, we will focus on the scheme shown in Figure 6.4.

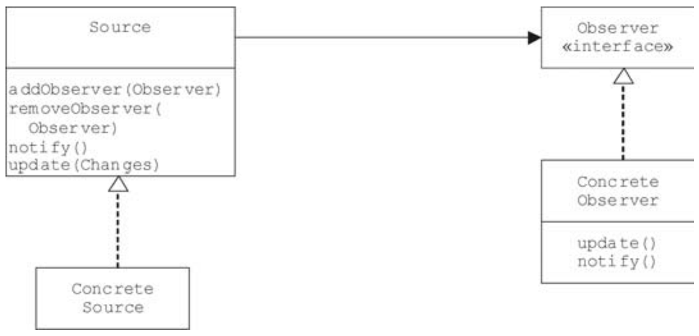


Figure 6.4 The Observer design pattern

13. Both terms are in common use, and we use them here as synonyms.

Figure 6.5 shows an example of the runtime behavior of an example of the Observer pattern represented as a sequence diagram.

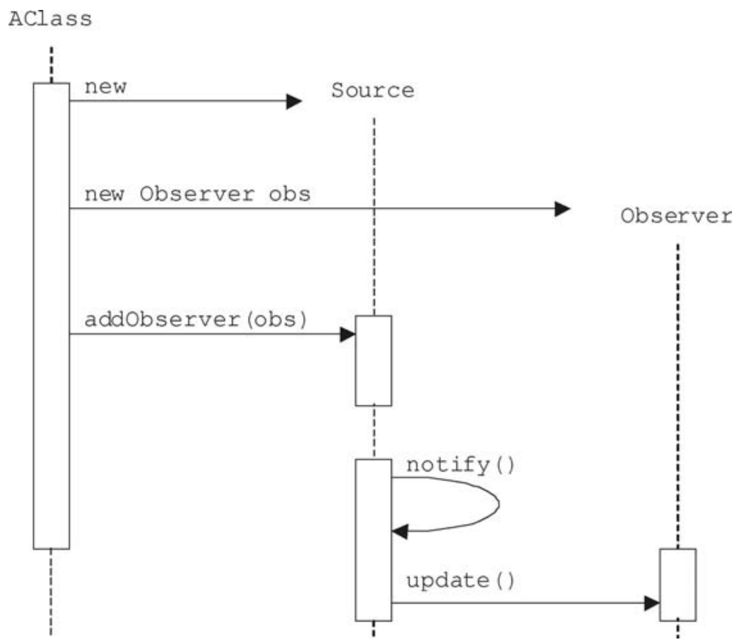


Figure 6.5 An example of runtime execution of the Observer design pattern

Each event can be described by an object that encapsulates useful information about what happened, typically the event source and other event-dependent data. Each source object can have multiple observers registered on it in a one-to-many communication mechanism that is defined at runtime by observers subscribing to the source object. Like any modern high-level GUI toolkit, the Swing library makes extensive use of specialized events – that is, specialized classes that handle particular kinds of events, such as `KeyEvent`, `ListSelectionEvent`, `CaretEvent`, and so on. SWT also uses an additional low-level simplified event representation.

The listener class needs to provide the related methods for handling the event, for example using Swing events:

```
public class ListenerClass implements ActionListener
```

Any instance `listenerClass1` of the listener class registers itself with the event source, for example:

```
eventSource1.addActionListener(listenerClass1);
```

This design is an example of another useful strategy in OOP design, that of favoring object composition over class inheritance. For example, compare the difference between using object composition instead of subclassing when defining the action triggered by a button widget. In the case of object composition, you will be setting an action listener object (that is, an Adapter object implementing the `ActionListener` interface) while in the other case you would be obliged to extend the `JButton` class. Clearly the first approach is much more versatile and flexible.

The event-based approach is widely used in GUIs, because it provides various benefits:

- It is simple to understand and use, while general enough to accommodate a large number of practical cases.
- It implements a mechanism of broadcast communication. Observer objects need to adopt the event description defined at design time (the classes defining the event), so that source objects don't have to know anything about their observers apart from a reference to each of them.
- Observers don't need to know anything about each other, and in practice they don't. This minimizes the visibility references among objects, although this could be a problem in some cases, because of reduced compile–time dependencies between different parts of a program.
- It increases extensibility and encourages code reuse, while easing the maintainability of code.
- It makes the coupling between source and observer object instances more abstract.

Perhaps the greatest shortcoming of event-based mechanisms regards the control flow indirection they bring to code. The Observer pattern can be thought of as a scheme in which control flow (procedural runtime execution) bounces back and forth from the source object to all its observers whenever they invoke methods. This implies that reading the source code alone is not enough to work out the actual flow of a chain of events. Developers need to run a sort of simulation of runtime execution in their heads to understand the control flow. The real situation can be determined only through careful, time-consuming debugging. Heavy reliance on event-based mechanisms makes the actual behavior of an application at runtime hard to understand.

Swing events

The Swing framework has adopted a variation of the Observer pattern since JDK 1.1 in which listeners listen for events using the same mechanism as for Java Beans – not surprisingly, as Swing widgets *are* Java Beans.

Figure 6.6 shows the classes involved in this approach. Compare this with Figure 6.4, which shows the classic Observer pattern.

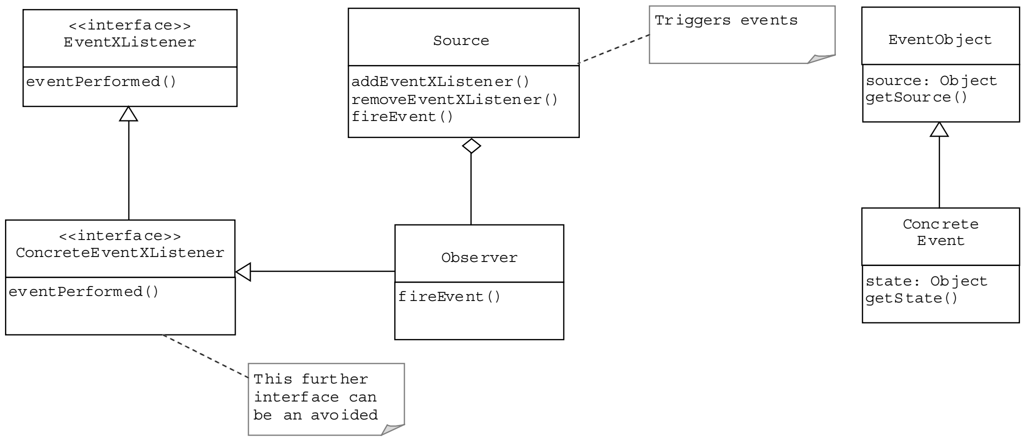


Figure 6.6 Swing events

Given the fact that Swing widgets are also Java Beans, they may use another event-based mechanism specific to Java Beans: `PropertyChangeListener`. We will see an example of the use of this variant of the Observe pattern later in this chapter.

SWT events

SWT's event architecture is similar to Swing's, although Swing-like high-level events are implemented as a convenience for the application developer. In fact, a low-level, simplified event mechanism is used by SWT classes for implementing the typed events SWT event mechanism. All subclasses of the `Widget` class can have an observer added to them by using the method: `void addListener(int, Listener)`, where the `int` parameter defines the event type. All available event types are supported by constants within the SWT class (such as `SWT.Selection`, `SWT.Collapse`, `SWT.Deiconify`, and the like).

Design-time class decoupling with events

Suppose you are going to develop a multi-player video game for the Java 2 Micro Edition. The video game will show a large 2D world in which a number of entities 'live' and interact. Players control one of the actors through a Java-enabled wireless device, while some entities are controlled by the game server. Figure 6.7 shows what this might look like.

Suppose one of the requirements of the implementation is that users should not be forced to download newer versions of the client software from time to time to play the game, as newer features or classes are added, as downloads might involve expensive communication apart from normal client–server data exchange. The code has to be designed to work with new classes added in newer versions of the game. Older versions of the game should work with newer ones as far as possible.

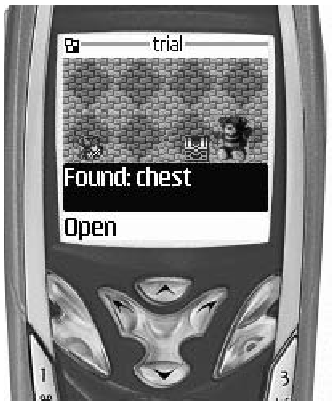


Figure 6.7 Using design-time class decoupling through events in a J2ME application

This is a situation similar to the design of an OOP library, for which you design utility classes that will be used by other programmers in the future. When you design a reusable library, you don't know which client class will use it, all you can do is to try to minimize the constraints imposed on clients that will use the code.

Suppose a player runs a teddy bear instance in the video game using Version 1.0 of the code, downloaded few months ago. When it encounters another game character released in Version 1.1 two weeks ago, the code in Version 1.0 of the game must be able to deal with it[14].

The simplest and most effective solution is to define an event-based decoupling mechanism among entities. When the teddy bear class is designed, all the possible entities it might encounter during its lifetime, and the possible reactions, are unknown, but what a teddy bear can ever do in the virtual world is known. By formally defining its possible interactions with the external world by means of Java code, you can make it available to future classes to interact with. For example,

14. This is a design issue: in fact, thanks to dynamic class loading, the J2ME client running Version 1.0 can load the new Version 1.1 class, but without proper software design they cannot interact.

you might decide that a teddy bear instance can sleep, run, and possibly do more in future releases. The class diagram would then be like that shown in Figure 6.8.

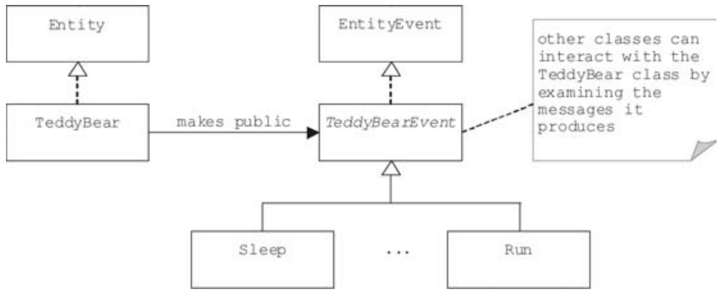

Entity
EntityEvent
TeddyBear
makes public
TeddyBearEvent
other classes can interact with the TeddyBear class by examining the messages it produces
Sleep
...
Run

Figure 6.8 Decoupling class interaction

When an entity wants to interact with the rest of the world it will prompt an event to interested parties – that is, it will issue a coded representation of a change in its internal state.

You could design one or more types of event for your video game, or even a fully-fledged hierarchy. The essential point here is about communication. Subjects make public predefined messages to whichever instance is interested, without having to know anything about the observers. Such event messages are published to the rest of the world, and interested classes know how to deal with them. This kind of communication mechanism guarantees a powerful, dynamic decoupling among interacting classes.

When developing ad-hoc components it is common to create new, specialized kinds of events. Extensive use of event-based communication mechanisms among classes is demonstrated in the examples in the chapters that follow. In this chapter the QuickText example application (Figure 6.18 on page 263) shows a simple example of the Observer pattern at work in detail.

Event Arbitrator

Events are so useful for implementing modern GUIs that they easily become one of the predominant aspects in the runtime execution of a Java GUI, and one of the main sources of difficulty in understanding the actual execution of the application. This provides an additional degree of complexity, especially for readability and extensibility – adding a new class implies adding extra code to connect the new class with the event mechanism.

An Event Arbitrator is a class that listens to a number of events and redirect or manipulates them according to some objective. It is used to simplify or provide structure to the software design, enhance performance by rationalizing event distribution instead of broadcasting to many listeners, and to centralize event flow. An Event Arbitrator can:

- Forward events, by shunting specific events to interested parties for some particular situation.
- Absorb events, for example for debugging purposes.
- Aggregate events, for example by aggregating low-level events into higher-level events.
- Manipulate events to provide some useful service.

An Event Arbitrator does three things:

1. Receives events it is in charge of, called *input events*. It needs to register as a observer to the objects that fire those events.
2. Arbitrates events, processing them to provide a specific feature.
3. Possibly transmit other events, or those it received, to interested parties, following some given organization criteria. In some cases it can also provide some collateral effect, such as modifying global variables, as well as issuing new events.

The most common form of Event Arbitrator works synchronously with its input events, so that the reaction to the received input event is performed sequentially to its reception. Other Event Arbitrators work with more sophisticated arbitration schemes and require extra care when handling threading issues.

The following subsections discuss the most common applications of this pattern in desktop application GUIs.

Aggregating events

A particular case of the Event Arbitrator strategy is for aggregating events from various sources, exposing them in a simplified, centralized fashion to interested parties[15]. In this case the Event Arbitrator acts as a single source of events, hiding other detail events fired by other objects. The Arbitrator class registers for all the detail events, so that clients need to register only with it.

Suppose we are designing an address composable unit (CU), that is, an assembly of simple widgets that act like a unique macro-component representing addresses, as shown in Figure 6.9. We want to hide detail events of the internal widgets and its clients. When using the `AddressCU` class, other client objects only need to register for `DataChangedEvents`.

15. This case is also called Event Aggregator by Martin Fowler.

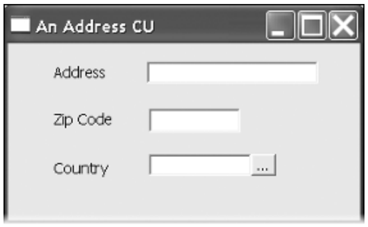


Figure 6.9 *The SWT Address CU*

Aggregated events can be of the same or different types as the detail events listened for by the Arbitrator. In this example the `AddressCU` works like an Event Arbitrator and fires new high-level events, as shown in Figure 6.10.

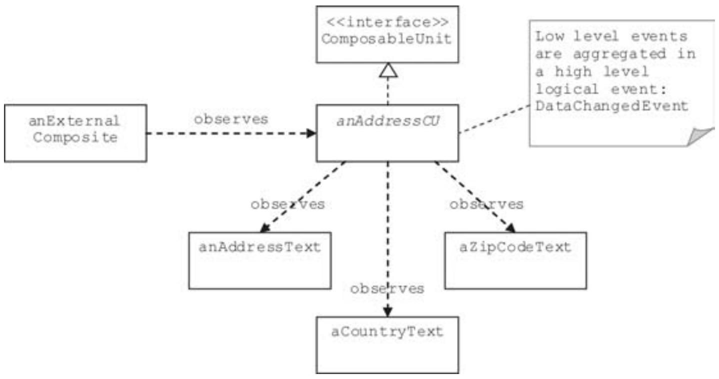


Figure 6.10 *The SWT address CU as an event aggregator*

Forwarding events over hierarchies of closely-related objects

It is often useful to organize event flow in hierarchical fashion, with a master event listener asnd many slave listeners that receive events forwarded by the master. This organization can be nested using the Composite pattern – that is, the master can contain other masters. This is the case with HMVC controllers, introduced in a later section, and with various other designs that we discuss below.

Sometimes many domain-specific objects enclosed in a container object must be handled in a GUI, possibly a subclass of a standard class such as a panel, or the root of a complex text document. The container forwards events to its contained objects, implementing a hierarchical Event Arbitrator.

The example ad-hoc component discussed in Chapter 16 implements a 2D desktop-like container into which items can be dragged, dropped, and manipulated by the user. In order to achieve maximum flexibility and decoupling, the container doesn't know anything about the nature of the contained items apart from their basic behavior, and forwards mouse events to them in a hierarchical fashion, thus implementing an Event Arbitrator. This might also be the case in a complex CAD GUI, in which a given scene is made up of a large number of small objects organized following a recursive Composite structure. Container objects will behave as Event Arbitrators on contained objects, enforcing some sort of domain-specific event-forwarding criteria.

An important property of Event Arbitrators, and especially for hierarchical Event Arbitrators, is that they should never allow loops in the graph induced by the event flow. The Composite structure of a hierarchical Event Arbitrator should form at least a direct acyclic graph (DAG), even if a simpler tree structure is much easier to manage and fits most practical cases. In the simple tree case it's sufficient to avoid any cross-references among objects in the Composite structure. Having cycles in the flow of events will of course lead to `StackOverflowExceptions`, as the same event is forwarded indefinitely.

Misuses of event-based messaging

Like every good thing, you can have too much of the Observer pattern. A common problem with the overuse of this pattern is Observer chains longer than one, for example when an Observer A observes another Observer B that in turn observes another object, and so on[16]. Control flow becomes very hard to figure out in such situations, and unforeseen behavior is likely. In some particularly unfortunate cases events can go into *resonance* – that is, an event X can cause a chain of events in which a new event X is triggered, causing another chain of events to be fired all over again, and so on.

In some case this incorrect behavior is not apparent from application execution, other than users noticing weird delays in particular circumstances, and log inspection or debugging are needed to work out what is really going on in the application. A possible solution is to use an Event Arbitrator, although this should be used carefully: adopting this pattern alone does not guarantee a cleaner design or a solution to unwanted event-based side effects.

Understating event-infested code

When inspecting someone else's code, it can be difficult to work out the actual chains of events. I personally remember a few cases in which there was such a

16. See for example a discussion on this aspect on Martin Fowler's Web site, http//: www.martinfowler.com.

massive use of events that fully understanding the runtime execution control flow was very hard. In one case an event-based composable unit strategy was adopted at a very fine level of granularity, making even the simplest local communication a matter of event messages. With time I resorted to a simple sketch diagram while inspecting code, and for the unfortunate reader who needs to decipher a tangled web of events, describe it here.

Depending on the situation, you might be interested either in who fires events, or in who is observing them. A simple variant of the standard UML sequence diagram can help to identify potential hot-spots in event chains. This diagram can be drawn by hand as you navigate the code, and can be applied to other event-based designs as well, such as those discussed in the next section.

Start by inspecting the code to see which objects register for changes in a source/subject. A simplified version of this diagram that takes into account only classes, and not objects, is much simpler to draw, but nevertheless useful. Whenever you find an object observing another object, draw an arrow from the subject to the observer representing the change propagation event, as shown in Figure 6.11.

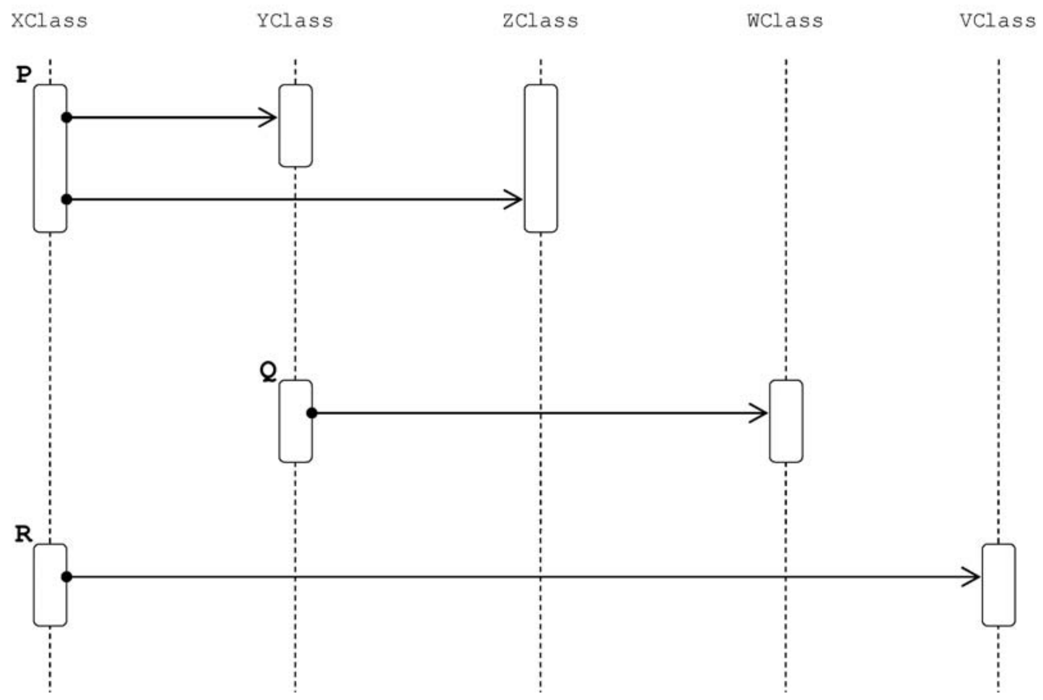


Figure 6.11 Informally describing events

This diagram[17] represents an object of class `XClass` that has two observers for a property P. Instances are invoked whenever P changes the listener methods in `YClass` and `ZClass`. An instance of `YClass` is also observed by an instance of `WClass`.

At the end of the code inspection this diagram will tell you roughly the possible chains of events in the code. The next step is to individuate those classes that have two or more boxes – for example, `YClass` and `XClass` in Figure 6.8. Focus your attention on these classes, for example adding debug breakpoints, because they are likely candidates for odd behavior.

You may think that problems arise only from combinations of ingoing events and outgoing events, such as `YClass` in Figure 6.8. This is not always the case, however – in some situations a subject that is common to more than one set of observers can create unexpected problems as well, like `XClass` in Figure 6.8. For example, this can arise when a change to a property A in a class also modifies another property B, and these properties are observed by two different sets of observers with a common class C. This could lead to unexpected side effects in C when the observer is notified.

A good guideline is to have chains of observers no longer than one, to avoid such possible problems and to keep runtime behavior easily understandable. For complex event schemes, consider using one or more Event Arbitrators to simplify and handle the resulting complexity, carefully designing the desired event flow.

Alternatives to event-based communication mechanisms

Message-oriented approaches are an alternative to event-based communication. Such approaches focus on sending messages asynchronously on a common message bus, where interested parties register to receive messages without any knowledge of who could be sending them. Event-based communication instead obliges the connection of the source – the object that fires the event – with the destination to be established explicitly.

How does this affect the design of client GUIs, with communications performed on the same machine and within the same JVM? Message-based communication for desktop application GUIs is most useful for interaction in-the-large, where the problem is to have an object X be visible to an object Y, rather than as a substitute for low-level interactions such as key presses and 'item selected' notifications. The easy solution to this, although not such a nice solution from an OOP design view-point, is to use some form of static visibility to access the required object references.

17. The diagram in Figure 6.8 is not a standard UML sequence diagram, nor does it have similar semantics. To avoid confusion, it uses different graphical details than UML sequence diagrams.

This might be done by implementing some form of object registry in which all required objects can be located by accessing a static service or Singleton class, or by making key objects available from a number of Singletons and then extracting the required references from them.

To avoid this drudgery you could resort to message-based communication, or, for the bravest, to a well thought-out OOP design – which is almost invariably the scarcest resource in real-world, deadline-tight, fast-paced production environments.

Message-based communication can be useful at an application level, in medium to large GUIs built by large teams for whom employing communication-specific infrastructure code makes sense. Outside such scenarios, message-based interaction is still attractive, because it provides a simple mechanism for communicating among different classes within the same application with a level of automated threading support. Queues usually run transparently in different threads, so that developers can focus on sending messages via specific queues to communicate with other classes and perform operations. The presence of a centralized bus also makes other automatic forms of control over the messages themselves possible, such as enabling or postponing actions. This approach appears natural to developers used to working with server-based messaging technologies.

Despite the possible benefits of message-based communication systems, they are not so popular among Java GUI developers. Technologies such as JMS[18] exist to enable message-based communication in distributed heterogeneous environments, but they are beyond the scope of this discussion.

As with any technology, message-based communication can be misused, bloating the volume of messages published on the message bus, or even worse, using it as a short-cut for serious design effort.

6.6 Separating data from views

GUI implementations usually need to define many classes and other support resources. As the complexity of the GUI increases, the code size increases dramatically. Therefore some code organization, usually at class or package level, is needed in all but the most trivial cases. The most common organizational criterion focuses on the separation between data and presentation.

This section introduces the Model-View-Controller (MVC) pattern (Buschmann et al. 1996), a popular design that enforces the separation of presentation and business code. In reality the MVC approach has proved far from perfect, as witnessed by the many variants that have been developed to try to cope with its shortcomings. Nevertheless, MVC still proves a popular design strategy for GUI code.

18. Java Message Service (JMS) API

Model-View-Controller

The MVC approach builds on the Observer pattern for connecting data *models* and their graphical representations (called *views*) by means of specialized entities called *controllers*. MVC was introduced and popularized by Smalltalk (Burbeck 1992) along with the Observer pattern. A variation of MVC has been adopted in the Swing library for separating business data from its GUI representations[19].

The model is the part that represents the state and the abstract data of the given component, separately from its visual representation. The model oversees the state and manipulates it as requested from outside. Following the Observer pattern, the model has no specific knowledge of either its controllers or its views. The view is thought of as being the graphical representation of the model's data. It handles the visual display of the state represented by the model. The controller manages user interaction with the model, providing the mechanism by which changes are made to the model.

One of the critical points in the large-scale adoption of MVC derives from the deeply-coupled relationship of controllers with models and views. In non-trivial scenarios controllers tend to become deeply intertwined with models and views.

In the Swing implementation of MVC, both the controller and the view are gathered in the same class, the user interface component, while the model is implemented as a separate entity, thus enforcing the separation between presentation and business logic. Each controller–view pair is associated with only one model. However a particular model can have many controller–view pairs.

The MVC design is also widely used for Web-based architectures. A simpler and less sophisticated version of MVC is used for server-side Web GUIs, briefly mentioned in Chapter 9.

Adopting an MVC approach provides the following major benefits:

- *Design clarity*. The list of a model's public methods describes a model's behavior clearly. This trait makes the entire program easier to implement and maintain.
- *Design modularity*. New types of views and/or clients can be created and plugged into existing models at design time just by adding new view and controller classes. MVC works well even when only enhancing existing classes – that is, when supporting incremental development. Controller and view implementations can be modified independently from the model.

19. Following SWT's design philosophy of being lightweight and performance-driven, there is no built-in support for MVC, which is delegated to the JFace library.

Older versions of views and controllers can still be used as long as a common interface is maintained.

- *Multiple concurrent views on the same model*. The separation of model and view allows multiple views to use the same business data model. Views could be even different classes, for example a tree view and a table view on the same data model instance. Despite being one of the most interesting features of MVC, the possibility of many concurrent views on the same model is rarely used in common GUIs.

Hierarchical MVC (HMVC)

The HMVC pattern decomposes the client tier into a hierarchy of parent-child MVC layers. The repetitive application of this pattern allows for structured architecture, as shown in Figure 6.12.

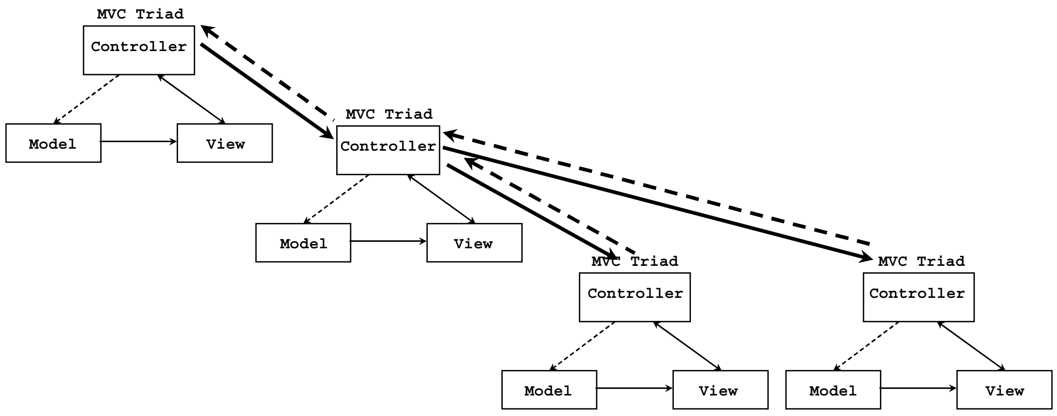


Figure 6.12 The HMVC pattern

Views hide the presentation technology from the model and the controller. GUI-related events are intercepted within the view, and eventually a request is made to the related controller in the form of an HMVC event. If the controller cannot handle the request on its own – each controller is responsible only for its own view and controller – it dispatches the request to its parent controller[20], and so on recursively.

20. Following the Chain of Responsibility design pattern.

This hierarchical structure relies on controllers, which are in charge of responding to HMVC events for navigation, such as changing screens and so on, and updating visual data.

HMVC, when applied to non-trivial applications, adds additional complexity to the implementation in the form of burdensome machinery – events and messages are exchanged through dispatchmethods that follow a hierarchical structure – and cognitive workload, as it can be hard to track down bugs and work out the current behavior of an application with many nested HMVC composable units. As a result, HMVC is not one of the most used variants of MVC for client desktop application GUIs.

Model View Presenter (MVP)

Model View Presenter (MVP) is a variant of MVC that attempts to loosen the coupling between the view and both the model and the controller in classic MVC. This tight relationship complicates MVC adoption and makes it hard to use in practice, resulting in MVC's various variants and workarounds.

In the MVP approach the actors have the following characteristics:

- The view in MVP is mainly responsible for graphical output. It also performs user-input gathering of low-level events like keystrokes and mouse events that are redirected to the presenter via events. Views communicate with their model via events. This limited responsibility of views in MVP makes this approach useful for reducing the amount of behavior to be tested without the view – and hence without testing through widgets and GUI toolkit classes. This in turn allows the testing to be accomplished without GUI testing tools, possibly using simpler unit testing tools.
- The presenter holds direct references to both the view and the model and is responsible for manipulating the view and the model to keep them in synch. The presenter does this by reacting to the events forwarded by the view itself.
- The model is similar to the classic MVC model. It is a business domain class that has no connection with GUI-related code, and also no connection with the presenter.

MVP experienced a new popularity with the advent of Test-driven development (TDD) and test-intensive practices, where the view is kept as simple as possible so that the application code can be tested, without full coverage, by writing standard unit tests focused only on the presenter and model.

Figure 6.13 shows the differences between MVC and MVP designs. Dashed lines represent event notifications, while solid lines denote object messaging (that is, direct method invocation).

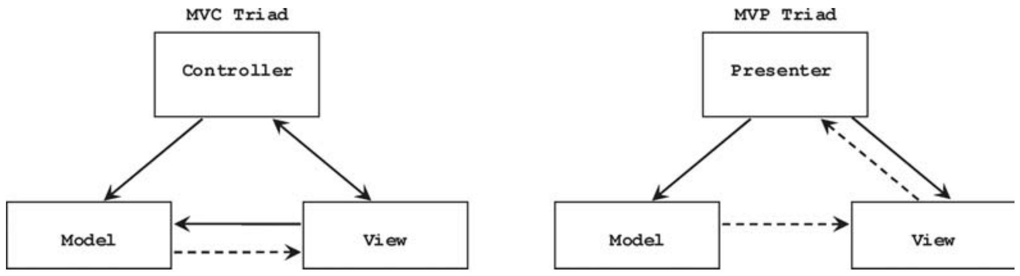


Figure 6.13 Differences between the MVC and MVP approaches

Concluding notes on MVC

The MVC design strategy enjoys a wide popularity among developers and in GUI-related frameworks, especially for Web user interfaces, where the level of interactivity and the overall complexity are lower than desktop application GUIs. One might wonder why it has been so successful, given that it produced a number of secondary issues that the various MVC variants have been created to solve. A simple answer is that MVC is an intuitive, practically-proven arrangement that works better than alternative solutions in real cases.

MVC, or one of its many variants, is already provided by all major presentation technologies and frameworks: adding another MVC layer on top of the one provided by the toolkit (as in Swing for example) usually adds complexity without providing any important benefit to the design[21].

In practical cases the MVC approach or one of its many variants, used alone, provides a minimal, localized decoupling between presentation and non-presentation code. The kind of decoupling provided by MVC may be improved by adopting some other complementary approach, such as a layering scheme[22] or a composable unit structure. This is especially true for non-trivial GUIs, when the implementation architecture is more important.

MVC is often used as a means of design, while it should always be treated as a *solution* to a given problem – it should be used as a design *means* rather than a design *end*. If there is no serious problem, perhaps there should be no need for its solution, and thus no need for MVC. In other cases MVC is used as a solution to a different problem, for example in an attempt to provide a structural organization to a design. This is not bad in itself, but should be achieved with a more comprehensive strategy, including layering, defining Java packages and so on, rather than just 'applying the MVC pattern' to a bunch of classes.

21. This is another example of the 'going against the flow' antipattern mentioned at the end of this chapter – in this case adding too much of a given solution to a design!
22. See Chapter 7.

For more information about MVC, see (Burbeck 1992), and as an example of its numerous variants (Potel 1996), while for some of the problems it raises and the possible remedies, see (Reichert 2000). See (Sundsten 1998) for an article introducing the Swing version of MVC, or (Fowler 2000b) for a comprehensive overview of Swing's MVC flavor.

Adapters

The flavors of MVC discussed so far employ the Observer pattern for synchronizing views, and models to provide the ease of use and flexibility modern GUIs need. This comes at the price of increased complexity, even for simple situations in which a fully-fledged MVC architecture is not really needed.

Building such a powerful, complex, and expensive design into a basic toolkit would force all users to employ it and to pay its price in terms of complexity and performance. This was the dilemma faced by Eclipse's architects when deciding how to provide data models on top of raw SWT widgets. To avoid over-engineering the JFace library, which provides utility features on top of SWT, including data support, Eclipse's architects employed a different design than MVC to separate data from presentation – they used Adapters.

The `org.eclipse.jface.viewers.Viewer` class implements a general Adapter for SWT widgets and handlers of data objects. A concrete example is the `TableViewer` class. This class adapts an SWT table widget with a *content provider* object. Such an object is responsible for providing content data, taken from a data object. The content provider therefore acts as a mediator between the viewer and the domain-specific data object itself.

This scheme is not a traditional MVC design as we discussed it, because it doesn't couple data with view – if you change the data model object, neither the viewer nor the content provider will automatically notice the change. It is nevertheless a simple and effective way to decouple data from presentation. It is even better than full MVC designs, such as Swing, in this respect. In a full MVC implementation, to have a table model for a `JTable` requires domain-specific data classes to extend a Swing class or interface such as `DefaultTableModel` or `TableModel`. With the SWT approach based on Adapters of content providers and raw widgets – called *viewers* in JFace – data can be provided by any Java class, without any constraint or dependency on SWT/JFace classes. A drawback of this simple design is that developers are in charge of managing coherence between data objects and views.

A traditional, event-powered MVC design is of course possible using SWT and JFace, and has been implemented in some of the standard libraries, such as the GEF[23] viewer classes.

23. The Graphical Editing Framework (GEF) is a Java library for creating ad-hoc components on top of SWT.

6.7 Interaction and control

One major source of complexity in modern GUIs is the high level of interactivity derived from sophisticated GUI designs. Features like undo/redo, or highly responsive GUI designs, need a sound implementation architecture.

Interaction here means the explicit representation of user interactions with an application, and the GUI's reactions to user interactions. A very simple GUI doesn't need to represent user interaction explicitly, it only needs to react to simple user input such as a button press by just executing the associated code. More elaborated GUIs can react in more sophisticated ways, for example by triggering a set of reactions throughout the user interface itself.

Control means an explicit form of management of interactions. Handling complex, changing interaction rules during the lifetime of an application can be a major source of architectural degradation if not addressed properly in the design from the beginning.

Representing user actions with the Command pattern

Handling user commands is a common problem when building GUIs. This book illustrates a number of solutions, most of them based on the Command design pattern. Such a pattern essentially transforms requests (commands) into objects: the request is contained within the object itself. This involves encapsulation of the code associated with the request or, more specifically, the code that actually performs the command.

Figure 6.14 shows the Command pattern directly instantiated for the Swing library.

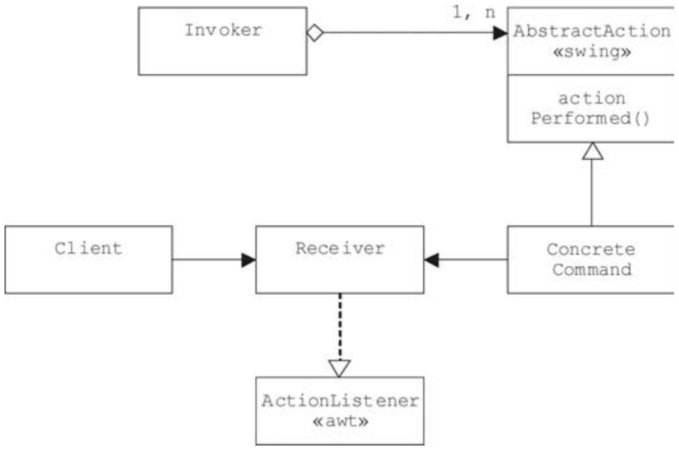


Figure 6.14 The Command design pattern

For the Swing library, the invoker can be a `JMenuItem`, a `JButton` instance, or similar. The `ConcreteCommand` is the command instance that is set up by the `Client` class, usually the main frame or the director. The `Receiver` is the class that actually carries out the action's execution. It implements the `ActionListener` interface for Swing and its analog for JFace's actions, implementations of `org.eclipse.jface.actions.Action`.

At the price of a little additional complexity, the main benefits of using actions are:

- The whole implementation is more natural than with command code centralization, taking advantage of OOP polymorphism over centralized, procedural mechanisms such as chains of conditions for executing commands.
- Behavior specific to a single command is kept logically localized within an `Action` subclass.
- Undo and redo features stem naturally from this approach.
- The class organization that derives from this approach is clearer and more systematic than that using a centralized mechanism for commands. This is especially true for large and complex applications.
- This pattern has been officially adopted in the Java API, in both Swing[24] and SWT.

Such an approach also has some drawbacks. It produces many small classes (the commands themselves), scattering command code among them. This implies additional complexity that must be addressed at design time, essentially in the form of communication between classes and overall management.

An unorthodox use of Swing actions

The Swing implementation of the Command pattern in this book make two different uses of Swing's `Action` subclasses. The difference lies in where the command code is located.

- Those `Action` instances that delegate command execution to an external class are referred to as *shallow* actions, acting as mere containers of data related to the given command, such as icon, mnemonic key, command name. These classes work like an expanded version of the action command string used in the AWT framework, holding GUI data passively, but not the command logic itself, which is stored somewhere else. Shallow actions do not therefore implement the Command pattern, even if they subclass the `Action` interface of the Swing library[25].
- In contrast, *deep* actions those classes that fully implement the Command pattern – that is, normal action classes. In this case the behavior of the

24. A brief introduction on the use of Swing actions can be found in (Davidson 2000).
25. See also the use of *retargetable* actions in the Eclipse framework.

given command is coded into the `Action` subclass, as the Command pattern suggests.

The shallow use of actions has been introduced in this book only for practical convenience. In simple GUIs, or where we don't want to use the Command pattern but still want to use a framework that adopts it, like Swing, it is handy to have `Action` subclasses delegating the execution of their command to a centralized point. This is shown in the sequence diagram in Figure 6.15.

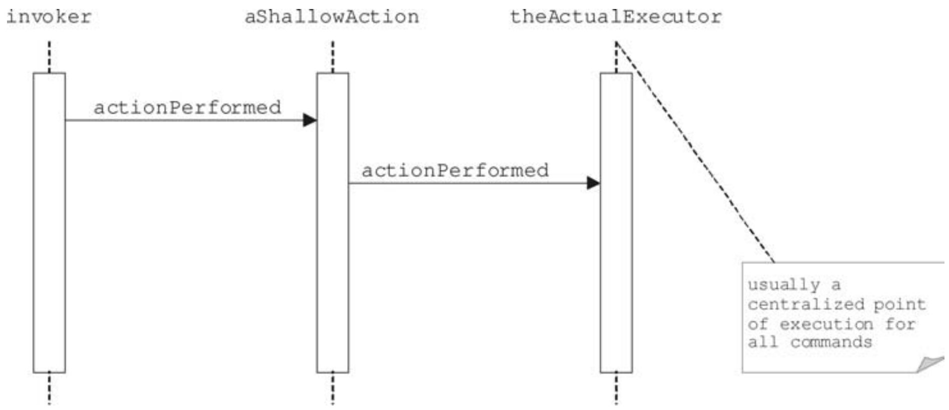


Figure 6.15 Shallow actions at work

Here the `actionPerformed()` method merely invokes the `actionPerformed()` method of the registered class. This simplistic delegation mechanism supports only one invoked class.

An example of fully-fledged 'deep' actions can be seen in Chapter 16. The code provided in Chapter 15 uses the unorthodox, shallow use of the `Action` class introduced here.

Command composition

A frequent solution for making command menus available to users is to aggregate commands hierarchically. Every object in the GUI is responsible for the commands it supports. In an iterative sequence similar to the Chain of Responsibility pattern (Gamma et al. 1994), commands are aggregated in pop-up menus suitable for use in menu bars or contextual menus.

Chapter 16 contains an example of such a behavior for container objects, which negotiate with their contained items the list of available commands to be incorporated in a common menu. This mechanism allows for maximum flexibility in an OO way, in that every object only knows its available commands, while keeping clearly-defined responsibilities among different classes.

Control issues

Some common issues arise when implementing control in professional GUIs. We have seen in the first part of the book how alert or error messages can disrupt the usability of an application. A good GUI provides coherent metaphors and low-level interaction rules that avoid the possibility of inconsistent interactions as far as possible. This translates into software that constantly manages parts of the GUI to enforce the abstract rules that govern it.

Depending on the complexity of the controls to be implemented, different design strategies are possible:

- *Scattered control*. Control is implemented on a local basis, attaching observers to the areas to control and executing reactive code as required. Control code is scattered throughout the GUI implementation and is thus hard to maintain. This approach is quite simple to adopt, but is useful only for limited control needs.
- *Centralized control: the Mediator design pattern*. As a rule of thumb, when more than three objects need to be controlled in a window, we need to escalate to another design strategy: centralizing the control behavior in one place. This has several benefits: tangled event listeners and references derived by the extensive adoption of the previous strategy are limited, and control is centralized in one place. This technique scales to a non-trivial number of controlled objects, even though references to controlled objects become a problem, together with handling the control logic code.
- *Explicit control state*. When things get really complicated even the Mediator pattern shows its limits. In these few cases, very articulated control logic can be represented in explicit classes. These classes represent the concepts behind the control logic and interact with the rest of the GUI. In this way screen control state is not represented within a Mediator class, but is shared among explicit objects.

While some control behavior strongly depends on business logic[26] other control logic is essentially domain-independent. This latter form of control can usefully be extracted in reusable, general-purpose code. We can tell whether specific control logic is business-dependent or not by answering the following question: if the business rule changes, would the given control logic on the GUI change?

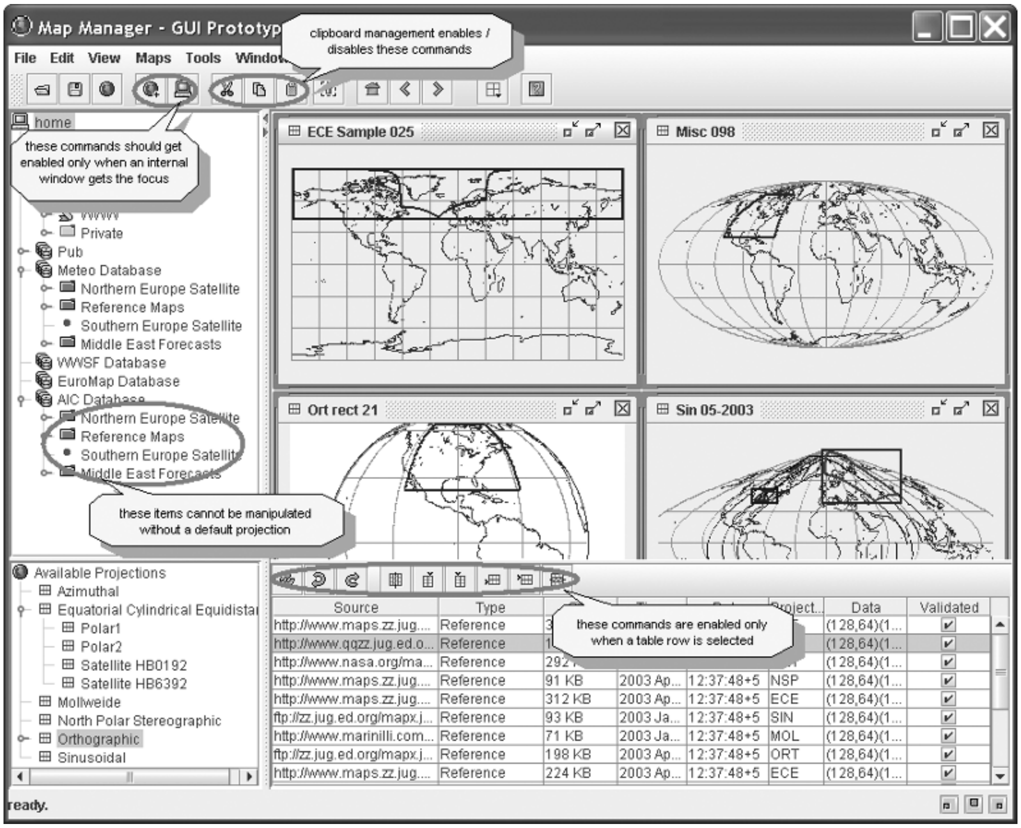
Distinguishing between business and non-business control rules is also useful because it is frequently the case that changes in business rules also impact the GUI. Separating them from the rest of the code helps maintenance and implementation clarity. Non-business control behavior rarely changes after the initial design

26. Such as data validation – see Chapter 8, *Validation* on page 332.

phase, so it can be treated differently than business-dependent controls. An example of a non-business control might be the following: in a GUI in which users can inspect item properties, whenever they modify data for an item and close the property dialog, the application asks whether the modified data should be saved or discarded. Such control can be performed automatically for any kind of item, independently from the business domain.

When control layer behavior that comes from actuating a domain's business logic rule in the GUI is used extensively within an application, for example in a highly interactive application with a formalized business domain, it can make sense to capture this behavior in a domain-based interaction control framework.

Figure 6.16 shows examples of interaction control rules governing in an example GUI.


Map Manager - GUI Prototyp
File Edit View Maps Tools Windo
clipboard management enables / disables these commands
these commands should get enabled only when an internal window gets the focus
home
Private
Pub
Meteo Database
Northern Europe Satellite
Reference Maps
Southern Europe Satellite
Middle East Forecasts
WWSF Database
EuroMap Database
AIC Database
Northern Europe Satellite
Reference Maps
Southern Europe Satellite
Middle East Forecasts
these items cannot be manipulated without a default projection
ECE Sample 025
Misc 098
Ort rect 21
Sin 05-2003
Available Projections
Azimuthal
Equatorial Cylindrical Equidista
Polar1
Polar2
Satellite HB0192
Satellite HB6392
Mollweide
North Polar Stereographic
Orthographic
Sinusoidal
these commands are enabled only when a table row is selected
ready.

Source	Type				Project...	Data	Validated
http://www.maps.zz.jug....	Reference	3				(128,64)(1...	✓
http://www.qqzz.jug.ed.o...	Reference	1				(128,64)(1...	✓
http://www.nasa.org/ma...	Reference	292				(128,64)(1...	✓
http://www.maps.zz.jug....	Reference	91 KB	2003 Ap...	12:37:48+5	NSP	(128,64)(1...	✓
http://www.maps.zz.jug....	Reference	312 KB	2003 Ap...	12:37:48+5	ECE	(128,64)(1...	✓
ftp://zz.jug.ed.org/mapx.j...	Reference	93 KB	2003 Ja...	12:37:48+5	SIN	(128,64)(1...	✓
http://www.marinilli.com...	Reference	71 KB	2003 Ja...	12:37:48+5	MOL	(128,64)(1...	✓
ftp://zz.jug.ed.org/mapx.j...	Reference	198 KB	2003 Ap...	12:37:48+5	ORT	(128,64)(1...	✓
http://www.maps.zz.jug....	Reference	224 KB	2003 Ap...	12:37:48+5	ECE	(128,64)(1...	✓

Figure 6.16 Examples of GUI control rules

Such a control behavior isss the essence of any credible user interface, one that presents sound metaphors, that needs minimal memory load on users, minimizes errors, and so on. The Mediator design pattern is commonly used in the implementation of this layer of control.

The Mediator pattern

A Mediator object (Gamma et al. 1994) provides a common connection point, centralizing the behavior of a number of disparate classes.

The use of the Mediator pattern in GUIs typically consists of the organization of relationships and interactions between visual components, their data models, and related events, all in one controller class. Such a class enforces a form of domain-dependent logic, so is specifically tailored for a given application – that is, it belongs to the Application layer. Figure 6.17 shows the Mediator pattern class diagram.

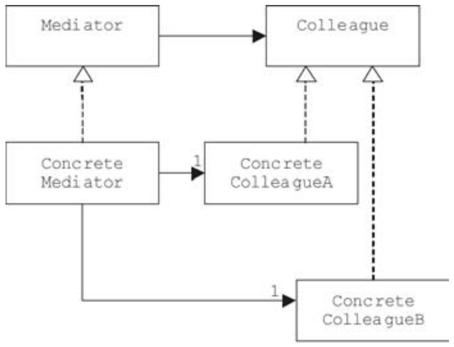


Figure 6.17 The Mediator design pattern

The Mediator pattern is useds in many of the examples in the third part of this book.

The `AbstractDirector` class represented in the UML diagram in Figure 6.18 is a simple and limited example implementation of the general behavior of a Mediator class used in some of the example applications.

Mediators can also work as Event Arbitrators, tidying event management for actions and other controlled objects. This is one of the advantages of centralized control over scattered.

Any director class manages a number of actions. Apart from keeping them coherent (enforcing business rules on them), other possible uses are to act as an Event Arbitrator, releasing actions to interested classes, and also possibly taking care of executing actions by funneling (aggregating) various `ActionPerformed` events in

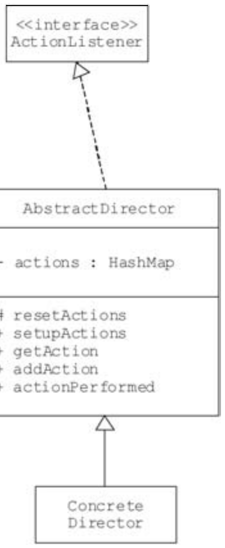


Figure 6.18 The AbstractDirector class

the director's `actionPerformed()` method. Such actions directly implemented by the director usually need many references to various objects and involve a complex web of references if they are to be executed outside the director class. This latter arrangement can prove useful:

- In architectures in which commands are centralized at a unique point, as could be the case when using shallow actions.
- When the nature of the action itself makes it simpler to handle this way – for example when one action needs to manipulate other actions or other classes that are already visible to the director.

Mediators can manage any class, not only actions. The example class in Figure 6.18 considers only actions, because in general they are the commonest case. Subclasses can add similar functionality for other classes as well.

Thread management

Apart from control design patterns, thread management can also be considered a form of dynamic runtime control.

Thread handling is essential for professional GUIs, and is the backbone of any interaction and control implementation. From Chapter 2 we know that response time is an important parameter for the user's perception of usability. GUIs that freeze while executing a command, or that have unexpected concurrency problems, are unusable no matter how well-designed they are. As we will see later, multithreading is necessary, but not sufficient in itself, to achieve a responsive GUI. Poor object lifecycle management, and the overhead it poses to the garbage collector, might also induce a 'jagged' user experience[27], even for a multithreaded GUI.

The basic issue with multithreading support in Java GUI derives from the fact that GUI toolkits are single-threaded. This applies equally to SWT and Swing toolkits. The underlying OS platform detects low-level GUI events and places them in the application event queue using the toolkit's event classes and other toolkit-specific formats. The toolkit is acting as an Event Arbitrator, isolating a platform-specific event model from a Java-specific one.

Multithreading is needed in several cases in GUIs:

- Most importantly, to keep the application responsive, a key characteristic from the users' viewpoint.
- By far I/O time is the commonest case of long-running task in client-server applications.
- Whenever an asynchronous task must be performed, for example when a background computation starts but the user must still be able to interact with the GUI.
- For faster initialization. Applications can resort to a separate thread to instantiate details asynchronously from the application's start-up process.
- To better take advantage of existing and future hardware power. People always faithfully hope that newer, more powerful hardware will magically and dramatically speed up their applications' performance. This is unlikely to be the case if their GUIs keep doing all their work sequentially. Employing multithreading wisely is an investment in higher performance on more powerful machines.
- For object creation. Creating expensive objects in parallel with other tasks whenever possible will enhance GUI performance and improve responsiveness. This use of multithreading couples with object lifecycle management, which is the subject of a later section.
- In the general case of multiple, concurrent tasks that need to be performed interactively, for example a memory manager thread that runs with a low priority.

From a usability viewpoint it is important to communicate what is going on inside the application during task execution. This is usually accomplished by displaying progress indicators coupled, via events, to the running task thread.

27. During garbage collector activity the application freezes.

Software bugs due to concurrency issues can be an annoying problem, because they are difficult to track down, in that they are not always repeatable. They can also occur in completely unexpected ways, as they depend on the particular user interaction with the GUI. So don't use threading differently than suggested for GUI applications (use threading for example by applying the Active Object pattern, or for performance optimization) or in situations where there is no apparent need for it.

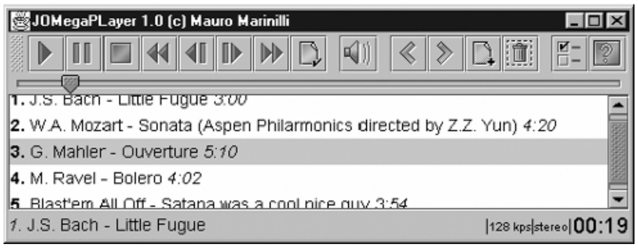
A common way to organize threads on single-threaded architectures built using Swing and SWT is to use objects that represent tasks that are executed within a specialized support class or within a larger framework. This scheme is simple to use and accommodates a vast number of practical cases. When using the Eclipse RCP, it is straightforward to use the thread management provided by the framework, while for Swing one can use the `SwingWorker` class.

Chapter 5 contains a more technology-oriented discussion on threading in connection with profiling. Later in this chapter we introduce the Active Object design pattern that is the design approach used for multithreaded support in both Swing and Eclipse.

The next section discusses another approach to organizing design-time control issues in GUIs.

A state-oriented approach to GUI control

In some cases the level of complexity of control needed in a GUI justifies the adoption of some kind of formalized, explicit representation. Figure 6.16 shows the GUI of a fictitious MP3 player. Such a GUI enforces a non-trivial set of interaction control rules. A *mode* is maintained to represent the different operational states (playing, paused, stopped, and so on), and this information affects the functionalities available in the GUI – such as which buttons are enabled, what information is displayed. See the disabled buttons in the application toolbar in Figure 6.19, for example.



1. J.S. Bach - Little Fugue *3:00*
2. W.A. Mozart - Sonata (Aspen Philarmonics directed by Z.Z. Yun) *4:20*
3. G. Mahler - Ouverture *5:10*
4. M. Ravel - Bolero *4:02*
5. Blast'em All Off - Satana was a cool nice guy *3:54*

1. J.S. Bach - Little Fugue |128 kps|stereo| **00:19**

Figure 6.19 An application with an internal state representation

A useful common abstraction is the use of *states* to describe the GUI's internal situation. States are defined when designing the GUI, and can be organized temporally in a state transition diagram that shows how the GUI's state changes when specific events occur. The granularity of each state definition depends on the GUI design[28].

States are also useful for clarifying the implementation of an application in which states were not explicitly defined in the GUI's design. Most of the time software designers don't need to formalize the possible states of a GUI explicitly, either for a single window, a part of the GUI, or the whole system. There are cases, though, where they may be confused by the abstract working of the theoretical GUI design, or its equivalent analysis documents. In these cases it's a good idea to try to write down a list of the GUI's possible states at a suitable level of abstraction, as well as their possible transitions. This exercise will make analysis clearer, even if there is no need to represent states explicitly in the code.

These considerations bring us directly to the Memento pattern. After a brief introduction to this pattern, we will see it at work in a practical example that implements an explicit control state.

The Memento design pattern

Sometimes the state of an object needs to be manipulated as a whole. Doing this straightforwardly may disrupt OOP encapsulation, leading to weaker code.

In the Memento pattern one class, called the Originator, is made responsible for creating the Memento object, usually transferring a portion of its internal state into it. Another class, called the Caretaker, requests the Memento from the Originator and uses it. Figure 6.20 shows the class diagram for Memento.

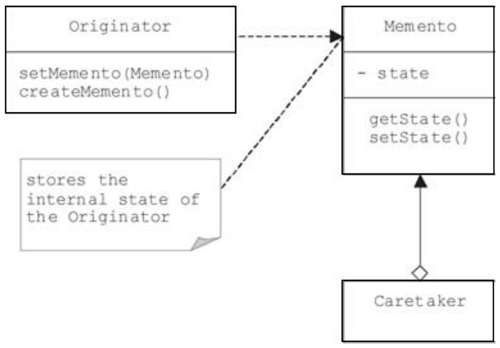


Figure 6.20 The Memento design pattern

28. The natural generalization of this approach – providing specialized classes for each meaningful state and a common interface for any generic state – leads directly to the State pattern (Gamma et al. 1994).

The Caretaker object is responsible for the memento's safekeeping, although it never examines the contents of a memento instance. Memento objects are inherently passive. They are used to encapsulate carefully-planned portions of the Originator's internal state for some specific purpose: a common case is to make it persistent.

The Memento design pattern can be used to represent and manipulate both the data state and control state[29] in a GUI. Consider for example a point of sale rich client application. In no case must the application lose data about a transaction, even when the connection is down and the user needs to close the application. In such cases the application can make the memento that represents transaction data persistent, so that it can be sent to the server as soon as the connection is restored.

We are now ready to see a practical application of these ideas to representing the control state of a GUI.

The QuickText application

This subsection describes an example application that uses several design strategies and some code tactics that are oriented towards simplicity and performance.

In any GUI there is usually a practical need to access data from different places. Such data can be variable over time, or needed just once in a session. An example of the former could be the row and line values of the caret cursor in a text editor, for example. We want to associate some control behavior to these values, for example to issue a beep when the end of text is reached, and to show them in a status bar component. This is a classic example of the use of an event-based mechanism – that is, some variant of the Observer pattern. In such cases it can be useful to adopt a mental habit of centralizing the required information in a meaningful way by providing abstractions over the current state of the GUI. Generalizing this idea, we might consider a class that represents the GUI's internal state, or at least what we need of it, which can be accessed by all interested classes. Some portions of the state could therefore be made observable.

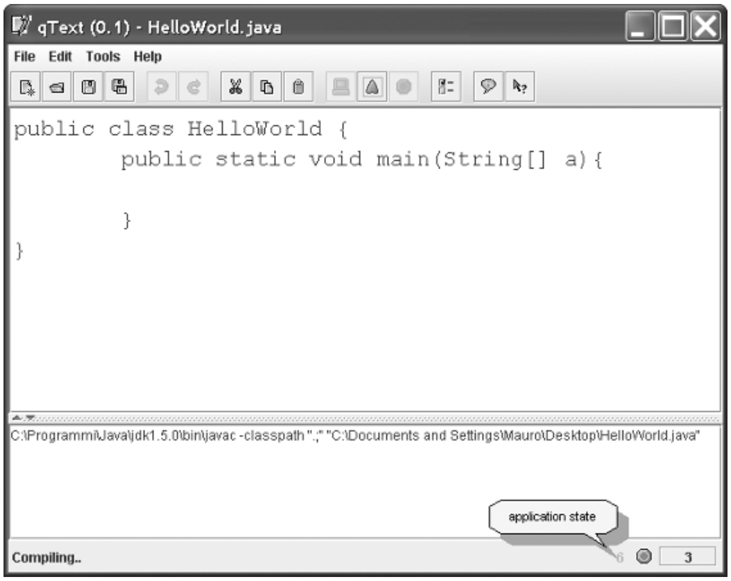
In simple situations this approach can be pushed to the extreme, accommodating in a common class both dynamic information, requiring an event-like communication mechanism, and less variable data such as system properties and preferences. Such a class could also enforce business logic rules for the GUI state as a whole.

Here is an example that can be useful when a basic approach to modeling a GUI's state suffices. The idea is to model the dynamic part of a GUI's state as a set of `Boolean` flags. Changes in these values are of interest to other classes. Examples of state flags in a text editor application could be used to indicate things like

29. See Chapter 8 for more details about these definitions.

whether the current file is saved, or whether a spell checking error occurred. Here we are only interested in a proof of concept, so the implementation is minimal to just show the basic ideas at work. This model can be used with other more elaborate abstractions for handling more complex situations.

Figure 6.21 shows the QuickText application, a simple text editor for compiling and executing Java code that serves as a background for the implementation techniques introduced here. Java code is entered into a text area, below which there is a console showing the compiler/JRE command-line messages.



```
public class HelloWorld {
        public static void main(String[] a){

        }
}
```

```
C:\Programmi\Java\jdk1.5.0\bin\javac -classpath ".;" "C:\Documents and Settings\Mauro\Desktop\HelloWorld.java"
```

Figure 6.21 The QuickText application

A listener has been added to the text model (the document), so that whenever its content changes, the **Save** and **Save As** buttons are enabled. The status bar at the bottom of the main window reacts to this event as well as the commands, by showing an icon on the right-hand side of the state code[30]. A green icon signals correct compilation, while red means that errors occurred during compilation or execution. Finally, the current caret line number is shown at the bottom right-hand corner.

30. The state code is shown only for debugging – in a production application it would be invisible.

Whenever the text file is saved, the file is assumed to be unmodified and the application returns to its initial state. The set of GUI state flags are implemented as integer values, as you can see from the number shown in the status bar at the bottom-right in Figure 6.22, the decimal equivalent of binary 011.

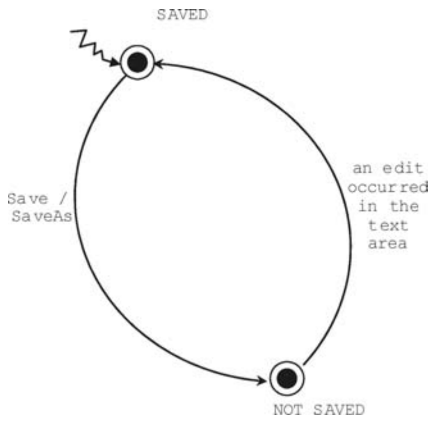


Figure 6.22 Text modifications as control state changes in the QuickText application

In this simple application only two classes are interested in state changes, as shown in the class diagram in Figure 6.23: the director, which coordinates all actions, and the status bar component. The director is in fact not really needed in this arrangement, as single actions can listen to state changes without passing through a common director class.

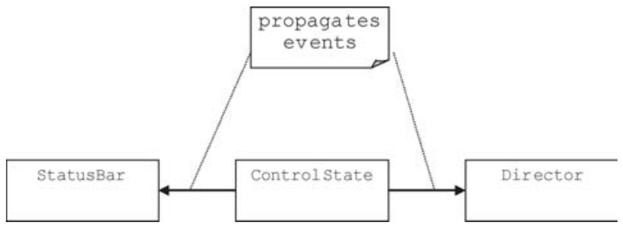


Figure 6.23 State changes

The `ControlState` class holds the current GUI state, and is responsible for exposing changes[31] to interested parties using the Observer pattern. This class implements a variant of the Memento pattern, shown in Figure 6.23 above.

31. Instead of writing an event class, the sample application uses the `PropertyChangeEvent` class from the `java.beans` package to represent state change events.

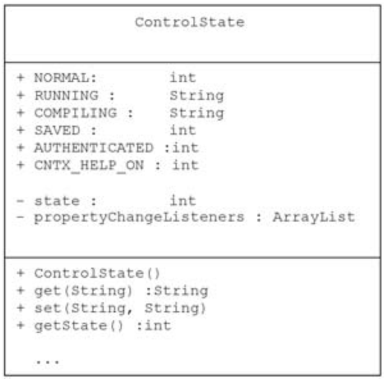


Figure 6.24 The ControlState class

Only the required part of the control state has been made accessible through the event mechanism. Minimizing event coupling is important to avoid needless complexity and unforeseen behavior. Another class, `Props`, stores the application's properties, without using event notifications when a property changes. Application properties must be queried when required, such as the application name property used by the main window for its title.

For simplicity the event implementation does not use any event representation when triggering a change notification message. Swing events use specialized classes to represent event data that is sent with the event notification. In this example, listeners retrieve the state when receiving the change notification message, for example displaying the state value in the application's status bar whenever it changes.

The `ControlState` class implements the GUI state with one or more integers and a bit mask. Interested readers can see this in the implementation of methods `isState()`, which tests whether a given flag is true, `addState()`, which sets a given flag to true, and `subtractState()`, which sets a given flag to false. Flags are implemented as Java constants, powers of 2. Bit masks provide a simple and extensible data representation mechanism. For the QuickText application the possible control states are shown in Table 6.1.

An example of bit mask use in GUIs is provided by the SWT library, in which component properties (called *styles*) are represented with sets of Boolean values.

The `Director` class implements the Mediator pattern in a very simple way. Whenever the application control state changes, the `Director` class is notified. The director then queries the GUI state and enables the **Save** and **Save As** actions according to the value of the `STATE_MODIFIED` flag. The `Director` class is also responsible for creating, managing and updating the internal state of all the actions used in the application and executing them using the 'shallow' action approach.

The status bar component is another listener to changes in control state flags – like the `Director` class, it implements the `PropertyChangeListener` interface. It registers itself for changes in the GUI state, and the method `propertyChange()` reacts to state flag changes.

Table 6.1 Possible control states in the QuickText application

Type	Data
`RUNNING`	An external JRE process is currently executing code
`COMPILING`	An external Javac process is currently compiling code
`CNTX_HELP_ON`	Contextual help is on
`NORMAL`	Start up, default value
`SAVED`	Current text has been saved

The QuickText example application also demonstrates an alternative solution for localization. Instead of using property files or other dynamic support for locale-sensitive data, it employs Java constants, for performance reasons. This is demonstrated in the `Msgs` interface provided with the source code of the application.

6.8 Some design patterns for GUIs

This section introduces designs typically used in OOP GUI implementations, some formulated explicitly for the first time, others well-known design strategies for desktop application GUIs.

Adaptation

Developing a professional GUI can be a complex task, with many requirements to be met. Adding some form of adaptation to GUI code can help to decouple different concerns and conceptually-separated responsibilities effectively. Typical of such requirements might be different behavior depending on runtime information such as different user roles or locale, or the resources available on the client machine.

These situations can be resolved in the same way:

i. Clearly define the adaptation mechanism.
ii. Elicit the context data the adaptation mechanism will need.
iii. Assign the runtime-dependent behavior to a separate manager class.

Providing a separated implementation avoids cluttered code by decoupling extraneous issues from existing code, making the whole application more modular.

The design goal of Adaptation is to make the application absorb the additional complexity without degrading the quality of the final implementation. This general approach can be applied to any functional layer.

Some examples of adaptation

Suppose a program contains the following code:

```
textField.setText(“controle el valor”);
```

We can make this message text locale-parametric as follows:

```
textField.setText(ResourceManager.get(“control.value”));
```

This confines the responsibility for locale-dependent messages to a specialized class, `ResourceManager`, and only the minimum amount of information must be provided for it to do its job of providing localized message strings.

Figure 6.25 shows an example of localization. Localization is not only a matter of locale-dependent text messages, but can imply a deep adaptation of the whole GUI, from widget layout, dimensions, and more, as discussed in Chapter 4.

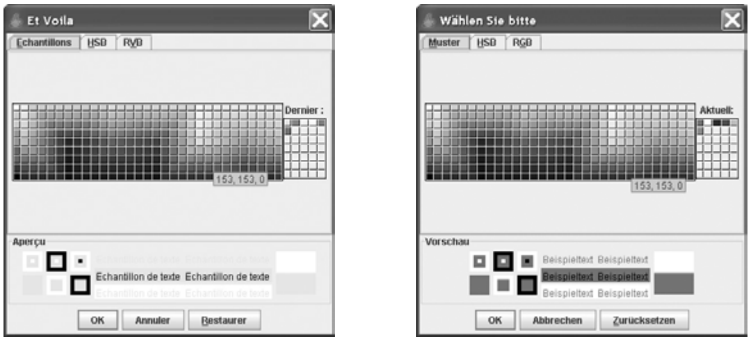


Figure 6.25 Examples of locale-based GUI adaptation

Another example of adaptation might be authorization code. Suppose one requirement in a GUI prescribes that sensitive information like employees’ wage details must be available only to certain user roles. This rule could be added to every widget that required authorization. Suppose pressing a button in the GUI shows the salary for the selected employee:

```
if (RoleManager.getCurrentRole().equals(BossRole.class)){
    button.setEnabled(true);
} else
    button.setEnabled(false);
```

Everything works well until the management want to change the authorization policy because someone complained that they don't want anyone else to see their wage details. The new requirement now states that:

i. Senior managers can still see other employees' wages.

ii. Middle managers can know that wage details are available to their superiors, but cannot actually see them.

iii. All other employees don't have to know that there is such a button in the GUI at all.

You could change the `if–else` code above to accommodate managers (`button.setEnabled(false)`) and all other employees (`button.setVisible(false)`). But what if managers complain and you have to change this authorization policy yet again? You would need to go into the code again and modify all this conditional behavior, which is likely to be scattered in many places throughout the GUI's screens. Authorization code shouldn't be intermingled with presentation code, and should be made more flexible to change. After all, these are business requirements, much as localization is a translator's job, and they should not burden programmers. It would be better if authorization could be handled to some administrator or customer representative rather than being relegated to developers.

This problem can be seen as an application of adaptation to runtime data. We want a GUI to adapt to the current user role. As the role is only available at runtime, a form of dynamic parameterization is required. The nice thing about adaptation is that it is somebody else's worry. Developers only need to enforce it, while decisions will be taken somewhere else, away from code.

You could provide the following implementation:

```
AuthorizationManager.prepare(button, this);
```

where authorization is relegated to a specialized manager, much like localization, and you provide the subject (the button) and the context (in the present example, the parameter `this`) where the subject appears. The authorization manager then retrieves the current user role and performs all the appropriate authorization policies on the subject.

Adaptation is common in any professional GUI. Some examples are:

- *Localization*. Here the context parameter is the current locale. This is a classic form of parameterization that is handled explicitly by the Java API.
- *Authorization and other role-based adaptations.* Here the context parameter is the current user role. Some commands, screens, or single widgets may depend upon the user role.
- *User profiling.* We may want to save preferences, customizations, and other information on a single-user basis, so that different users using the same application installation find their own settings and specific data.
- *Business-specific parameters*. Country, international branch, or some other domain-specific concept are examples of parameters for which adaptation rules might be dictated by specific requirements.
- *Resource-dependent constraints*. An example is a client application that runs in two different remote connectivity scenarios: modem lines and broadband connections. To provide a good GUI design, the commands available might need to be adapted to the remote connection type.

Adaptation techniques make sense when the need for adaptation is common to a sizeable part of the GUI. If only one or two panels need a limited form of adaptation, and no extensions are planned in future, a simple local solution would be cheaper to implement yet still effective.

Building a comprehensive API for parameterization could be a complex task for most real-world applications, with few real benefits[32]. What is important though is to be aware of the problems adaptation may generate. Some guidelines for effective adaptation are:

- *Clearly define parameters and carefully separate them*. Define exactly what the parameters are in your application and their reason for existing. It is important to keep parameters conceptually separate. Implementing this conceptual separation involves enforcing orthogonality in code (Hunt and Thomas 2000). The effects of different adaptations should provide cumulative, predictable results. If for example an application already provides localization and role-based adaptation, and you add business rules parameterization, you expect these three aspects to coexist gracefully without unexpected side effects.

32. Common parameterizations such as localization are already provided by standard APIs. Other forms of adaptation can be achieved relatively easily without requiring a comprehensive, unique framework.

- *Avoid explicit parameters scattered in the code.* These are hard to modify and make code fragile. All context information should be sent to the manager object responsible for the parameterization, such as the following code example:

  ```
  if (user.getRole().equals(ROLES.ADMIN)){
  // admin users-only code here
  ...
  }
  ```

- *Define a common strategy and enforce it.* If some aspect is parameterized using an XML file, for example, no code should deal with that parameter in a different way, for example by means of local conditional clauses.

Advanced adaptation

Adaptation is normally performed at runtime depending on context information. Other more complex forms of parameterization can exist, although these are needed only in special cases.

Adaptation may become a source of complexity if differences between individual adaptations are too wide to be housed in the same application. In such cases different code bases should be considered. This could be the case for example with the development of a single application that supports a multinational insurance company. Laws, cultures, practices, and other differences in each country could make it too complex to bundle such aspects into a single application code base. Shipping such a huge single application would make little practical sense. In such cases solutions other than dynamic adaptation should be considered, such as a software family-based approach – building a common framework that comprise all the common aspects, and creating the required adaptations using different custom builds of the application, or simply building different applications with a common organization, software reuse policy, development infrastructure, and so on.

Another example of non-dynamic adaptation is parameterization of an application at build time for security reasons. For example, you might want to generate an application installation on demand to work only with a given license key. In this case the license key is the parameter.

Using A3GUI for parameterization

A3GUI (Abstract-Augmented Area for GUIs) was introduced in Chapter 2 as a flexible approach for expressing generic information about a GUI. The idea is to identify areas of a GUI – a single widget, a panel, or a complex screen – and attach useful information to these abstractions. Augmented areas can also be used to express parameterization, even in cases in which there is no direct link with a screen area, such as business rules parameterization.

This approach lends to a declarative parameterization style in which localities in the GUI are identified by A3GUI identifiers, and their correct instantiation[33] is done somewhere else, as we saw before when discussing the `AuthorizationManager` example. Suppose you must implement the security of a very sensitive banking application. In certain parts of the GUI a number of controls are enforced in reaction to specific GUI events, such as modifying sensitive fields, pressing buttons, or displaying specific screens. Security practices may change over time – some parts of the GUI may become sensitive, while other areas may have existing checks loosened – so you need to make your implementation flexible.

Areas can occur in different places in the same application. Suppose the panel shown in Figure 6.26 is currently an extra-sensitive part of the GUI. Whenever such a panel appears in any part of the GUI, its behavior is dictated as follow:

- Depending on the current user and the current time of the day[34], it is possible to modify the **Amount** field. During holidays and at night, when there is no central human control, some potentially dangerous transactions are not allowed.
- In certain other circumstances, such as combinations of the context data mentioned above, other behavior is needed, such as making the panel invisible to the current user.

By providing a unique A3GUI id for the panel, you could associate the current security level, stored in a signed encrypted file for example, with that identifier, without scattering ad-hoc controls in the application's code. This would centralize the GUI's security implementation in a specialized and reusable manager. Such areas can be defined at analysis time, during GUI design, or later. A3GUI ids can be composed following the GUI containment hierarchy, to provide an exact identification for a given panel instance in a given screen, or used generically for all occurrences of relevant panels.

In cases in which a total A3GUI identification of the whole GUI is not needed, for example when parameterized properties don't change so often, ids for specific widgets or panels can be provided directly in the code. This keeps the application modular, but avoids the complications needed in the general case.

A useful technique for providing unambiguous context information for the adaptation design strategy is to take advantage of the visual composition of widgets into screens. This technique can also be used for requirements other than defining Adaptation contexts. It is discussed in the next section.

33. Here we mean the instantiation of an A3GUI area, that is, a portion of a screen that is adapted depending on specific parameters. Implementing this in Java implies instantiating a number of classes.
34. As measured on the server, for security reasons.

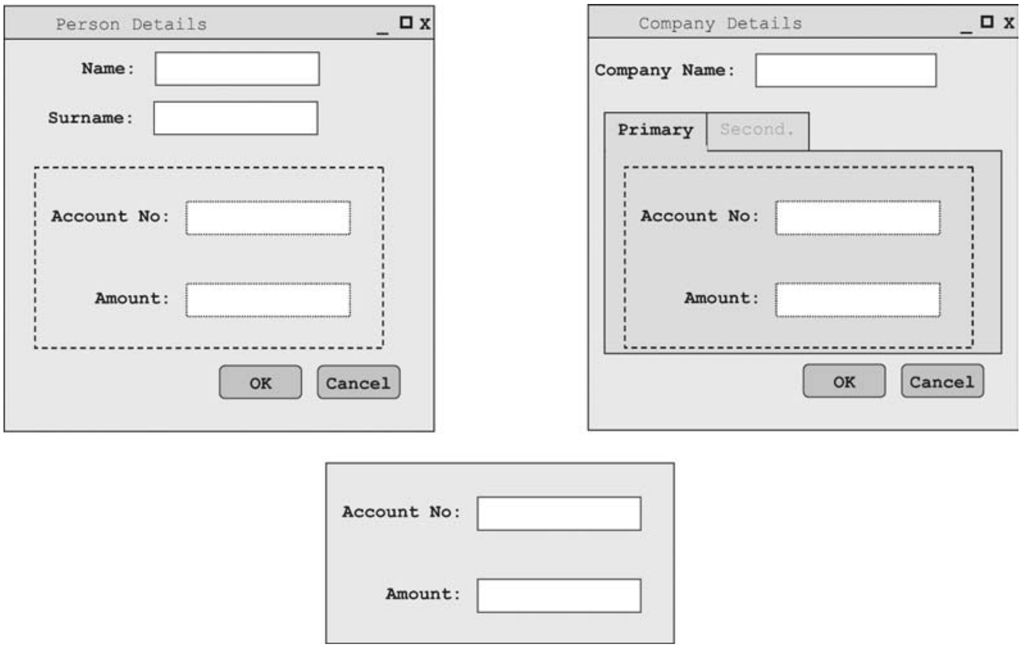


Figure 6.26 A sensitive panel

Composite Context

GUI content composition is heavily based on the Composite design pattern. In some cases it is necessary to identify specific areas of the screen. Identifying components, whether elementary widgets or composite aggregates, could be needed for various reasons:

- Suppose you want to provide every meaningful widget in a GUI with its own unique identifier, for testing, ease of look-up, and so on. The problem is that the widget may be nested in different panels, but you want it to have a unique id throughout the entire GUI. One possible solution is to use the Composite pattern for the ids as well, recursively attaching all components' ids to create a global, unique id for the widget, no matter how many instances there are of the same class.
- You used the Adaptation pattern, but you need a formal context id to represent the adaptation context in a simple way.
- You employed a composable unit strategy in your GUI in which all CUs are registered in a common registry for look-up. You need to provide an infrastructure service that will supply unique ids for CU instances automatically.

The Composite Context pattern describes a mechanism for providing identifiers for widgets, panels (that is, composites), and screens. The idea is to use the hierarchical organization of the visual composition to provide unique (or local) ids for widgets, panels, and composable units. Figure 6.27 shows an example dialog in which the ids of some widgets and CUs are shown.

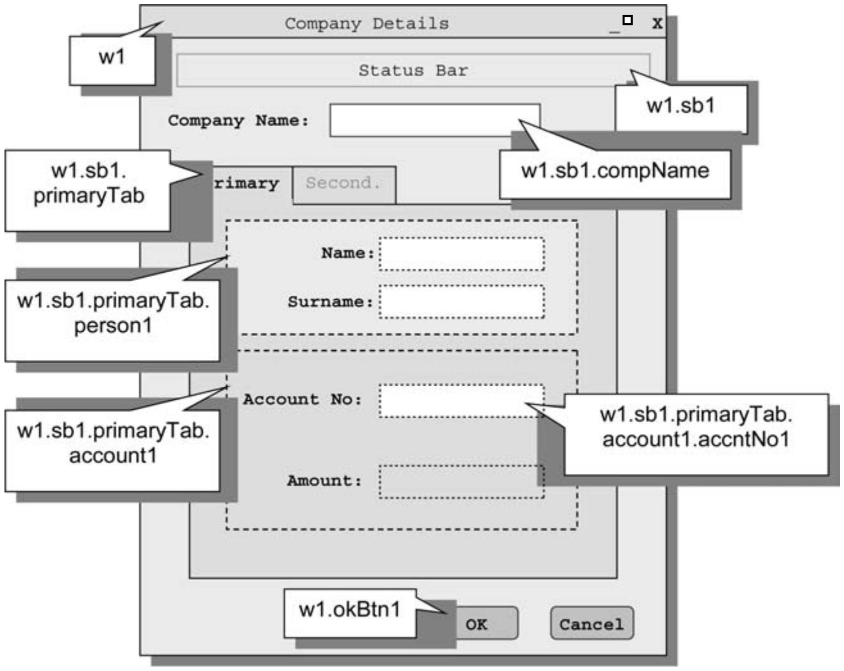


Figure 6.27 Composite Context at work

Referring to the figure, the screen `w1` contains a status bar whose standalone id is `sb1`, and when composed within the screen as shown in the figure, has an identifier `w1.sb1` that reflects the actual visual composition of the screen. The same widget composed in another screen would have a different identifier.

Composite Context can also be used to provide ids for A3GUI areas in analysis and design phases. In this case identifiers are simply applied by hand by analysts or developers, following the hierarchical approach proposed above.

The hierarchical mechanism provided by Composite Context can be used to provide information other than just identifiers. Support information, or an automatic mechanism for generalizing ids, can be provided as well – for example, for querying all items contained in a composite, or for simplifying the support XML files with inherited values.

Active Object

The Active Object design pattern[35] focuses on the creation of objects whose state develops asynchronously. As a consequence, the state, details about the operation in progress, or the final result, need to be shared among different threads. In the case of GUIs, the interested threads are the event dispatch thread and a worker thread that is executing a long-running task such as a remote transaction. Another thread is used by the scheduler object, which takes responsibility for executing tasks requested by clients transparently to them.

The Active Object pattern consists of three phases:

1. *Method request construction and scheduling*. In this phase, the client invokes a method on a proxy class, which in turn packages the task and forwards it to an executor in the form of a method request. This maintains references with the method itself, as well as any other data required to execute the method and return its results. A reference to a `Future`[36] instance is returned to the client that will provide the result when available.
2. *Method execution*. After the client requests the execution of a task, it continues its normal activity. Within its own execution thread, the scheduler determines which method request can be executed, depending on its synchronization constraints. When a method request becomes runnable, the scheduler executes it, usually passing responsibility for its execution to a servant instance.
3. *Completion*. In the final phase, the results are stored in the `Future` reference for the client to access them. The method request and the `Future` instance are no longer needed and are ready for garbage collection.

Both Swing and SWT toolkits provide framework support for this pattern. Given its importance in supporting smooth interaction with users, it is used in all the examples provided in the third part of the book.

A Swing example of Active Object

A simple implementation of a long-running task using the `SwingWorker` class is provided as an example of the Active Object pattern in the code bundle for this chapter. The task is activated by pressing the **Paint Nicely** button shown in

35. For more details, see http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf, an updated version of the original chapter in (Vlissides et al. 1996).
36. The java.util.concurrent.Future interface represents the result of an asynchronous computation. Additional methods are provided to check whether the computation is complete, to wait for its completion, cancel the computation, to retrieve the final result, and more.

Figure 6.28. This starts an instance of the class `FancyPaintWorker`, which paints the area in the window without freezing the rest of the application.

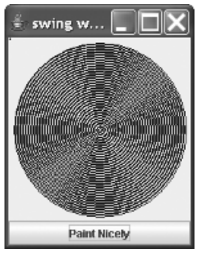


Figure 6.28 An example of SwingWorker

Object lifecycle management – a general mindset

Any non-trivial Java desktop application GUI handles tens of thousands of objects, especially if implemented using Swing. Large applications might handle hundreds of thousands of objects or more. No matter how powerful and well-tuned the garbage collector, it will always have a lot of work to do. Taking care of the lifecycle of the objects we create is important for providing a simpler life for the garbage collector, and thus providing smoother interaction for the user. We don't want an application to freeze from time to time, out of the blue, while the hard disk whirs mysteriously. Neither do we want an application to take a long time to start or launch specific features because it needs to create many expensive objects at the same time.

From a GUI design viewpoint users generally prefer to have a partially-functional application that starts up quickly, even if some secondary portions of the application are initialized later, than wait longer to have the whole application up and running at once. This general design strategy is more important in medium to large applications that handle many objects, but the habit of taking care of the lifecycle of objects is nevertheless always a healthy one, even in small applications. To condense the experience of many projects into one line, *instantiate lazily and dispose eagerly*.

The many virtues of lazy instantiation of objects in desktop application GUIs have already been discussed. A less well-known strategy concerns object disposal. Disposing of objects as soon they are known to be no longer needed is important, in that it helps the work of the garbage collector and keeps the memory profile of an application trim. This is also true for Swing applications, where there is no need for explicit object-disposal policies, although disposing of particularly expensive objects manually helps garbage collection, thus smoothing GUI interaction.

An object's lifecycle can be optimized as follows:

- *Object creation*. When is an object needed, and is it possible to postpone its creation until required using lazy instantiation?
- *Object disposal*. As soon as it is known that an object is no longer needed, it can be disposed of explicitly, easing garbage collector work and enhancing the application's responsiveness. Object disposal in Java is achieved very easily by setting the variable that refers to (and holds) the object to `null`.
- *Multithreading support*. Sometimes the instantiation of many objects, or of a few expensive ones, can be performed in parallel with other tasks, thus speeding up performance.

Here are some common scenarios of object lifecycle management that can be used for enhancing application performance:

- *Application start-up*. Forking threads to allocate resources and the essential start-up configuration of an application as soon as possible is the best introduction an application can give to its users. As we know from Chapter 2, the overall user experience is often dictated by the first impression they have of an application.
- *Lazy instantiation of hidden panels in tabbed panes*. This is a relatively simple and useful optimization, especially on large forms with many complex tabs. Only the first tab is populated, and the others are created lazily when opened. For forms with mandatory tabs – that is, tabs that must be opened to complete a task – it is better to let them be populated asynchronously after the first tab is completed and is occupying the user's attention.
- *Partial initialization of widgets*. Some widgets can be shown empty to the user initially, and while they are interacting with the GUI, initialize themselves asynchronously. Examples might be a table that populates itself asynchronously, or an ad-hoc panel showing a graphic chart that requires a lot of data to be drawn, so is shown initially as a grayed-out area. In such cases it is important to exert robust control over any unintended interactions the user can cause on partially-instantiated widgets.
- *Dataset paging in large lists, trees, and tables*. For large data sets, paging is the only viable design solution for loading only visible data and discarding previously-seen values. This technique consists of fetching only a number of pages of data at any time. The minimum number of items fetched is usually 1.5 to 2 times the current visible view size, or a constant, reasonable value derived from that. Hence, if you have a table showing 100 rows, you would fetch 150 or 200 rows, to fill the view and allow for some leeway for scrolling. An efficient mechanism for coarse-grained scrolling is needed, so that it is possible to jump directly from, say, the 100th element to the 5000th. The topic is clearly more complex than this, and is a perfect fit for a utility library such

as those available for Swing. An example of lifecycle management for large trees is described in Chapter 15.

Special care is needed over handling events and interactions on objects that are not yet instantiated, such as data models for expensive widgets, to preserve the robustness of the GUI.

Such techniques should be used conservatively and without over-doing it. Trying to optimize code preemptively is always asking for trouble, and ultimately needs extra care to avoid weaker implementations.

Value Model

The Value Model design pattern originated in Smalltalk to allow sharing values between different actors. It can also be used as a form of Adapter between an actor interested in a single value and a large object that stores the value along with other data.

Sometimes this design is used for coupling data models with widgets in Java applications. An Adapter class could be used to bridge the different method signatures of a widget with a data model, for example an SWT `Text` with the `street` string property in an `Address` class, and change property events used for notification when the value changes. This arrangement requires an Adapter for each widget – if another widget `PostalCode` needs to be synced or bound to another string property of the same instance of the `Address` class, a new Adapter is required. We can solve this by using a Value Model object that complies with a standard signature, using the value property of type `Object`. An Adapter object is still needed to convert widgets' method signatures to the `ValueModel`. Figure 6.29 illustrates this design.

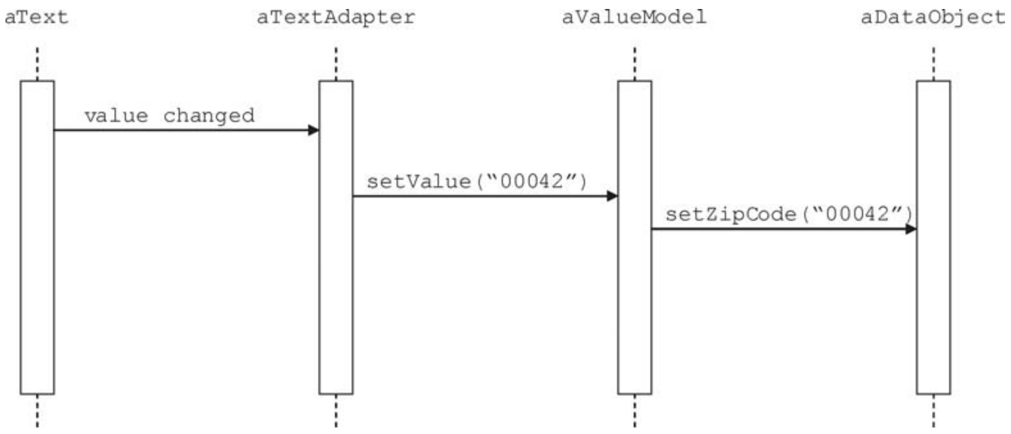


Figure 6.29 How Value Model works

This design can be thought of as an application of the Chain of Responsibility and Adapter design patterns. Communication the 'other way around,' that is, when a property value changes in a data object, is ensured by events. Note that this mechanism implements the binding between a widget and a generic data `Object`. When values are copied back and forth this binding can be made automatic on data changes, or it can be enforced explicitly. The latter choice eases debugging, because synchronization events are easier to track at runtime than data changes (see Figure 6.30).

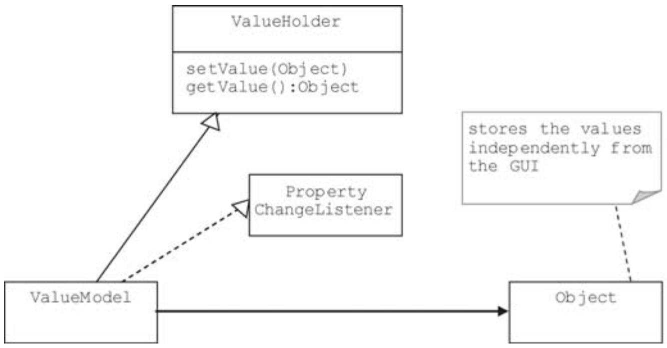


Figure 6.30 Value Model class diagram

An example of use of this design is discussed in Chapter 8 for binding widgets to data objects in form-based rich client applications.

6.9 GUI complexity boosters

This chapter concludes by discussing some issues in GUI development that significantly raise the level of complexity in an implementation.

J. Coldewey describes typical 'complexity boosters' in the development of distributed applications[37] as issues that dramatically complicate software development. More circumscribed, common sources of complexity can be found in GUI development as well. Even when such sources of complication cannot be avoided, as is unfortunately often the case when developing professional GUIs, being aware of them is nevertheless important. To use a colorful metaphor, you can imagine these issues as being like items on your workbench with a 'Danger!' label on them, to remind you to handle them with extra care.

37. These are distribution, multithreading, multi-platform, extreme performance, and paradigm gaps (such as Object-Relational Mapping), as mentioned in (Fowler et al. 2003).

The main sources of sharp increases of complexity in developing GUIs are:

- *Extensive control*. When you want to control explicitly a large and increasing number of disparate objects, such as a word processor with hundreds of commands that need to interact.
- *Going against the flow*. Some solutions are naturally supported by GUI toolkits, while others are just unnatural. Deciding on a personal, arbitrary design approach may prove a useless and expensive choice. An example might be deciding to avoid the Command pattern for representing commands in favor of a homegrown approach. This is a general situation that can apply to GUI design as well as software architecture. It can be paraphrased as *avoid unnatural choices*.
- *Ad-hoc solutions*. This can be seen as a special case of the previous point. Creating alternative solutions to those provided in standard toolkits, such as developing special ad-hoc widgets, can be a necessity in certain situations, but it remains an expensive choice.
- *Flexible layouts*. Despite the fact that we take dynamic layout managers for granted (also known as 'liquid layouts'), these are an explicit cause of complexity in GUI development.
- *Internationalization and localization*. Providing a GUI ready for multiple languages is a common source of complexity. This in turn involves flexible layouts – fixed sized screens and widgets are a certain recipe for internationalization troubles, as we saw in the GUI design perspective in Chapter 4.
- *Multithreading*. This issue comes into play in different ways depending on the GUI toolkit and runtime platform of choice. Multithreading is also at the root of remote IO. Even the simplest communication in fact requires an explicit management of threading control. While standard IO needs are serviced with toolkit facilities, other specific threading issues must still be developed in-house.
- *Remote IO*. Everything works fine in a GUI until you start to interact with the rest of the world, either via a single remote server or several Web services. Suddenly your code gets messed up with try-catch clauses, multithreading, and remote connection code spread all over its once neat implementation. Note that multithreading is usually involved with interactive remote access, depending on the support provided by the underlying infrastructure, as mentioned in the point above.
- *Distribution of business domain code among client and server*. This is typically the case with rich client applications, in which business logic needs to reside on the client side as well as the server to allow the client to perform meaningful operations off line.

- *Runtime constrains*. Requirements like a maximum memory footprint of 128 MB, or a maximum limit of thirty seconds for obtaining the list of all registered passengers in a flight reservation application, will impact deeply on a GUI design and its subsequent development, constraining the possible choices of a Java application.

Issues such as these are often closely intertwined, so that they exhibit a 'burst-like' behavior. For example, suppose you decide to provide internationalization in an application, but also need to adopt flexible layouts throughout the GUI – for some locales, such as Asian ones, this might involve installing a custom input method implementation, which could in turn require multithreading support for better performance. I use to call these situations, in which a single feature prompted a domino-like effect, 'complexity bursts.' Good engineers should be aware of them, ideally before triggering the burst, and be daring in evaluating the real benefits of the features to be added.

6.10 Summary

This chapter introduced several techniques useful in the design of the implementation of professional GUIs using Java technology. We have seen how the main implementation issues, presented as the functional layers in the model in Figure 6.1, can be addressed, and how objects communicate by means of the Observer pattern and its various variants, as well as the main problems in over-use of event-based communication mechanisms. The chapter also discussed the three main strategies for implementing control: scattered, centralized, and explicit representation of screen control state information.

Other common design strategies for building professional user interfaces were discussed, such as Adaptation, Composite Context, Active Object, Objects Lifecycle Management, and Value Model. It is useful to recap the main design strategies discussed:

- The principle of Single Functional Responsibility for a clear definition of functional responsibilities.
- Content Assembly and its various implementation strategies for handling widgets' layout.
- Explicit Navigation for managing explicitly the navigation among the various screens in an application.
- Some issues for representing business domain logic effectively in an application.
- Devising a comprehensive data IO design strategy.
- Addressing security concerns in an application explicitly, and including them in the overall architecture.

- The Data Transfer Object (DTO) pattern for exchanging information remotely.
- Designing remote communication with server applications.
- The Observer design pattern, its various flavors, and the high-level designs built with it: MVC and its main variants, HMVC and MVP.
- Representing user commands, composing them, and taking advantage of existing support frameworks for your design needs.
- Representing reactive control behavior with implementations that depend on the complexity required by the GUI design.
- Other design strategies commonly used in GUI implementations, such as Adaptation, Composite Context Active Object, and Value Model.

Figure 6.31 recaps the main design strategies used specifically in GUI designs, represented by the functional layer on which they mainly focus. These are only the commonest solutions used in modern OO client software designs, and the techniques listed in the figure are not exhaustive.

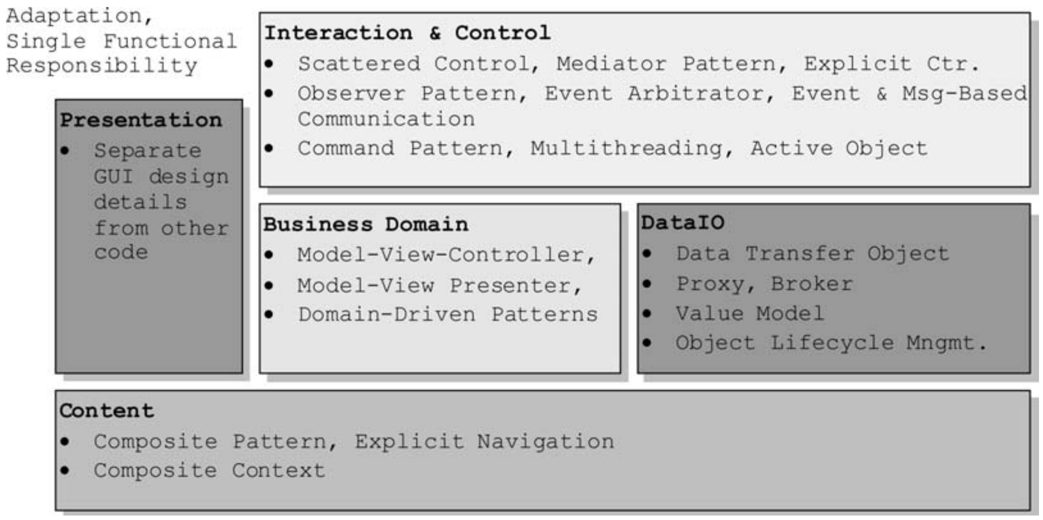

Adaptation, Single Functional Responsibility
Interaction & Control
• Scattered Control, Mediator Pattern, Explicit Ctr.
• Observer Pattern, Event Arbitrator, Event & Msg-Based Communication
• Command Pattern, Multithreading, Active Object
Presentation
• Separate GUI design details from other code
Business Domain
• Model-View-Controller,
• Model-View Presenter,
• Domain-Driven Patterns
DataIO
• Data Transfer Object
• Proxy, Broker
• Value Model
• Object Lifecycle Mngmt.
Content
• Composite Pattern, Explicit Navigation
• Composite Context

Figure 6.31 Common solutions by functional layer

The next chapter discusses the main issues involved in the definition of the overall software architecture for applications, covering the most popular solutions found in real-world GUIs.

7 Code Organization

This chapter discusses the main trade-offs and issues related to the organization of code and other implementation artifacts for application GUIs. Experimental or unproven approaches, or solutions that don't fit within existing Java GUI technologies, are not considered. The client tier – that is, the portion of software that is deployed on a client machine – is the principal focus. The chapter includes some implementation details for an example layering scheme. It focuses on J2SE/J2EE, but the design strategies discussed here can be applied to J2ME applets, as shown in Chapter 10, as long J2ME's resource constraints are observed.

The chapter is structured as follow:

7.1, Introducing software architectures discusses some general issues of software architectures and related software design strategies for GUI applications.

7.2, Some common GUI architectures introduces some of the most useful software architectures for GUIs.

7.3, A three-layer organization for GUI code goes into the details and the trade-offs of the layering scheme.

7.4, Two examples of a three-layer implementation shows examples of the application of the layering scheme, one to a simple project and one to a large one.

7.5, The service layer describes the details of the proposed implementation of the service layer for the three-layer architecture.

7.1 Introducing software architectures

Layering is a well-known technique for reducing dependencies between parts of a software system. An element in a particular layer is only permitted to access elements in the same layer or in layers below it. Strict layering, which is more laborious to enforce, prescribes communication only with the layer immediately beneath the current layer[1].

1. This induces a directed acyclic graph (DAG) structure in which nodes are layers, and arcs are dependencies, ensuring that, among other things, there are no circular dependencies.

Organizing implementation artifacts is a vast field for which a large amount of literature is available. It is also an important topic that will affect any project during its lifetime, even though it is not a guarantee of quality.

Layering is an attempt at structuring code by minimizing dependencies, removing duplication, and possibly attempting some form of reuse. The most important issue when deciding a strategy for organizing GUI implementation artifacts is the definition of clear responsibilities for each layer. That is, what is the main issue we want to address with our architecture? Some of the main practical strategies are:

- *Data flow.* This decomposes an application based on the flow of data within the application. The runtime data flow is a vivid concept, easily defined and shared. This strategy is used particularly in data-centric applications such as form-based GUIs. Chapter 8 has an example of a runtime data model in the context of rich client applications.
- *Domain-driven*. This strategy aims at enabling a portion of a rich (OO) domain model to operate on the client machine. To achieve this, a number of services and infrastructures are built to host the domain model as well as possible[2]. Developers focus on domain-driven issues first, carefully decoupling them from any technical 'plumbing' or graphical details. This strategy works well with GUIs for complex application domains, where non-trivial domain representation is needed on the client side and some form of a rich domain model is available (or planned) on the server side.
- *Functional*. This approach focuses on the function of each module, decoupling concerns on a functional basis. Business concepts usually drive the Composable Unit[3] strategy, if any. Instead of domain concepts, a commoner strategy for Java GUIs is to use implementation-driven concepts. An example could be using a variant of the functional decomposition proposed in Chapter 1 to define the layering scheme.
- *Reuse*. The main objective of this layering strategy is to simplify future reuse of software. 'Reuse' is a magical word that implies different things to different people. We could reuse concrete code, code patterns, or abstract approaches and skills – for example, we might want to use a full OO technology because we can then leverage our existing design patterns experience. This is one of the most often-used organizational strategies, even if the results are not always guaranteed to be fully reusable.

2. That is, leaving the domain model in its purest form, free of GUI or low-level details.
3. See Chapter 6.

These are just some of the strategies for breaking down GUI code: other are possible. Our underlying assumption is that there is no one single 'killer architecture,' not even in the relatively well-defined and circumscribed domain of Java GUIs. Instead, each implementation organization has its advantages and drawbacks, as we will see. Successful adoption of one architecture over another also depends upon the development team's skills, the problem at hand, the chosen technology, the timeline, and other non-technical factors.

Choosing the right strategy is the most difficult point in using a layering scheme (Fowler et al. 2003). Lack of deep knowledge, conservative attitudes, or just plain sloppiness are possible causes for naïve GUI architectures. This is often the case with Swing, due to its high-level feature-rich design. Imagine for example that you are called to help a project in trouble. Its developers show you a complex tabbed form, containing hundreds of Swing widgets, then explain that a PAC (Presentation-Abstraction-Controller[4]) variant was used as a layering on top of Swing widgets 'to make things clear.' The implicit underlying concept here is 'We don't care about that Swing mess.' The PACs are nested in reusable panels, so there are nearly thirty PAC triads in the form, plus all the associated specialized event machinery, all of this sitting on top of Swing widgets that add hundreds of other objects (Swing's MVC support, decorators, and so on). At that point you realize what causes the room's lights to dim every time the application is launched…

Taming references

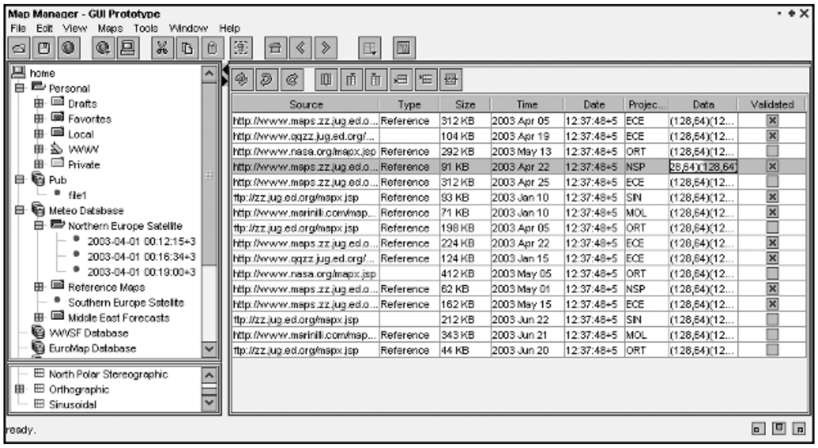
Suppose our team has developed the application shown in Figure 7.1. We adopted a clean architecture, unit-tested all our code, and did everything we considered beneficial. Next week we are going to release the application – guess what… we are late with the planned schedule.

We then observe a strange problem in the data in the exploration tree on the left-hand side, and need more time to track the source of the problem. For some mysterious reason our caching mechanism, which we thoroughly tested, is not responding well. We run more unit tests, but the problem seems to be caused by the final integration with the server application. We don't want to spend time on an ad-hoc basis, such as profiling by hand, or spending too much time trying to

4. The Presentation-Abstraction-Control (PAC) pattern was defined in (Buschmann et al. 1996). This pattern defines a hierarchy of cooperating agents for structuring interactive software systems. Each agent is responsible for an aspect of the application's functionality and consists of three components: presentation, abstraction, and control. These components separate the human and computer aspects of the agent from its business domain-dependent core and its communication with other agents.

replicate the problem, or involving time-consuming end-to-end test sessions. What we really want is a reusable, lightweight test that will also be useful in the future.

We think this is a great opportunity to set up some partial integration testing, even if we are late with the schedule. When we try to simulate data from the server to the client realistically, however, we stumble on a number of unforeseen issues. A number of services, both server-side and local facilities on the client application, need to be started up front to prepare the scene for even the simplest test. Testing the exploration tree in this way involves also starting the table component on the right-hand side, otherwise we would be obliged to step back to a previous CVS snapshot, and this in turn needs other parts as well. Ultimately we find we need to launch the whole application!



Source	Type	Size	Time	Date	Projec...	Data	Validated
http://www.maps.zz.jug.ed.o...	Reference	312 KB	2003 Apr 05	12:37:48+5	ECE	(128,64)(12...	☒
http://www.qqzz.jug.ed.org/...		104 KB	2003 Apr 19	12:37:48+5	ECE	(128,64)(12...	☒
http://www.nasa.org/mapx.jsp	Reference	292 KB	2003 May 13	12:37:48+5	ORT	(128,64)(12...	☐
http://www.maps.zz.jug.ed.o...	Reference	91 KB	2003 Apr 22	12:37:48+5	NSP	28,64)(128,64)	☒
http://www.maps.zz.jug.ed.o...	Reference	312 KB	2003 Apr 25	12:37:48+5	ECE	(128,64)(12...	☐
ftp://zz.jug.ed.org/mapx.jsp	Reference	93 KB	2003 Jan 10	12:37:48+5	SIN	(128,64)(12...	☒
http://www.marinilli.com/map...	Reference	71 KB	2003 Jan 10	12:37:48+5	MOL	(128,64)(12...	☒
ftp://zz.jug.ed.org/mapx.jsp	Reference	198 KB	2003 Apr 05	12:37:48+5	ORT	(128,64)(12...	☐
http://www.maps.zz.jug.ed.o...	Reference	224 KB	2003 Apr 22	12:37:48+5	ECE	(128,64)(12...	☒
http://www.qqzz.jug.ed.org/...	Reference	124 KB	2003 Jan 15	12:37:48+5	ECE	(128,64)(12...	☒
http://www.nasa.org/mapx.jsp		412 KB	2003 May 05	12:37:48+5	ORT	(128,64)(12...	☐
http://www.maps.zz.jug.ed.o...	Reference	62 KB	2003 May 01	12:37:48+5	NSP	(128,64)(12...	☒
http://www.maps.zz.jug.ed.o...	Reference	162 KB	2003 May 15	12:37:48+5	ECE	(128,64)(12...	☒
ftp://zz.jug.ed.org/mapx.jsp		212 KB	2003 Jun 22	12:37:48+5	SIN	(128,64)(12...	☐
http://www.marinilli.com/map...	Reference	343 KB	2003 Jun 21	12:37:48+5	MOL	(128,64)(12...	☐
ftp://zz.jug.ed.org/mapx.jsp	Reference	44 KB	2003 Jun 20	12:37:48+5	ORT	(128,64)(12...	☐

Figure 7.1 A buggy application (Squareness)

Although the previous situation is not such a nightmare, it is a common experience that at a late stage of development and without careful planning and continuos discipline, application modules start to look 'tangled' together, despite our initial commitment. What happens in reality is that dependencies are added very easily to code, so easily that you don't even realize, but are much harder to get rid of.

Application GUIs are composed of many interacting parts. Some of these parts are outside our control, such as GUI toolkits or third-party libraries. As the application grows and parts are added over time, such interactions tend to intensify, driving the implementation toward degradation of the decomposition strategies we initially devised. Intertwined references are not only a problem for testing, they are also a problem for the stability of the whole implementation – degrading the system's 'orthogonality' (Hunt and Thomas 2000), they hinder maintenance, parallel work, future reuse, and so on.

Layering helps to avoid mutual references, at least at the level of specific layers. Various solutions have been proposed for this: the well-known OO design principle of 'designing to abstractions' is one – abstraction being usually interfaces, but in some cases also abstract classes. A useful principle for untangling layer references is to use the 'dependency inversion principle' (DIP) discussed in (Martin 2002). Although there are many other principles around for clean OO structuring, we focus here only on the simplest and most useful.

The Dependency Inversion principle

The dependency inversion principle (DIP) (Martin 2002) states that:

- High-level modules should not depend on low level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

This principle suggests designing to interfaces rather than to concrete implementations. This newly-added layer of static indirection helps in the overall decoupling. 'Dependency inversion' refers to the effect that results from applying this principle to software layers, as shown in Figure 7.2.

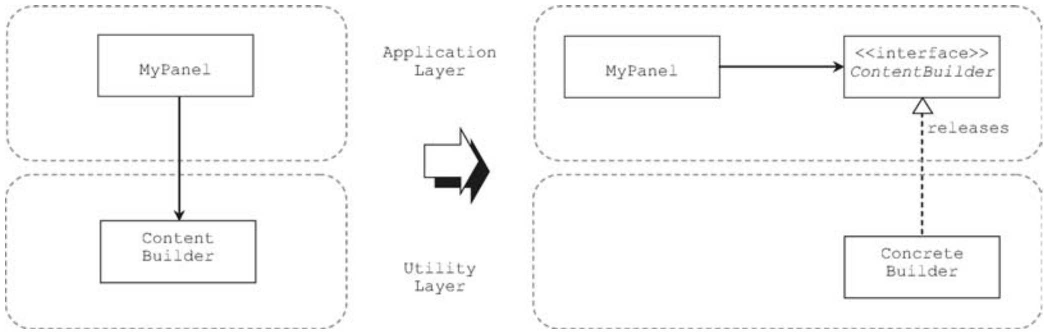


Figure 7.2 *Inverted layers*

A class `MyPanel` directly invokes a utility class `ContentBuilder` for content assembly of common widgets such as buttons, panel structure, and so on. This single invocation makes the whole application layer dependent on the utility layer. Whenever something is changed in the utility layer we need to modify all its clients.

A better solution would be to decouple the two layers by means of a Builder abstraction (usually an interface or an abstract class). This inverts the dependencies: the utility layer now depends on the application layer. `JLFBuilder`, a concrete `ContentBuilder` specialized for the Java Look and Feel design guidelines, will implement the details of building common widgets without strong dependency on its clients.

This technique does not protect designs from the annoying problem of modifying an interface that is already implemented by many clients, of course. This problem can disrupt a design if we are not in control of client code, for example when releasing a library to clients. A careful design of abstractions such as interfaces is extremely important in such cases.

The Observer design pattern discussed in Chapter 6 uses DIP in separating `Subject` from `ConcreteSubject`.

As a variant of this technique, you can expose all your abstractions in a separate package, allowing implementation classes to extend them, so that violations of the DIP principle become evident as package dependencies.

Separated Interface

The Separated Interface pattern, as described in (Fowler et al. 2003), can be seen as a generalization of DIP for decoupling two packages by providing interfaces (or, when meaningful, abstract classes) in separate packages. The separate package can belong to the client package that uses the interfaces, or to a third separated package when more clients are possible. There are usually a set of 'concepts' – the interfaces – separated from their various concrete implementation classes. This in turn requires factories for generating concrete implementations of such interfaces, which might themselves require other separate interfaces. To provide the right implementation for separated interfaces, we can use static references (that is, at compile time) or adopt a more flexible form of code configuration, such as the plug-in approach discussed in Chapter 13.

Composable units

Disciplining references is just one aspect of organizing OO code for non-trivial GUIs. Another aspect concerns aggregation criteria, that is, principles for organizing an implementation into useful clusters.

A simple form of code organization is the notion of reusable stand-alone units, following the Composite design pattern. These 'composable units' can be thought of as micro-GUIs, because they encapsulate content, presentation, data IO, business domain, and interaction and control in a set of classes that are thought of as a single reusable unit that can be composed together with similar units to build user interfaces in a modular fashion. Clearly a support infrastructure needs to be provided to enable this approach.

Composable units can either partially or fully coincide, or be orthogonal to the layer architecture of an application. For example, you can have a three-layer architecture and use MVP triads as composable units. That is, MVP triads form the autonomous macro components that compose the application: each triad spans the three layers into which the implementation is decomposed.

There are two common, practical main strategies for composable units:

- *Implementation-oriented abstractions*, such as MVC designs and their many variants. In these designs the focus is on implementation-oriented abstractions.
- *Domain-oriented abstractions*. These can be defined at various levels of formality. For example, we can define an Address composable unit as a set of classes responsible for rendering and managing data about addresses in a GUI. A design can range from simple, informal aggregations to fully-formalized component-based decompositions.

Composable units are useful for aggregating code in medium to large projects. They favor reuse, clean organization, and a systematic approach to decomposition. The drawback of endorsing a composable unit formal design is that the design process becomes more complex, and with a more formalized (that is, heavy) infrastructure. Therefore consider the use of a composable unit-based design only if the project is medium to large, or if many developers are involved, possibly in multiple locations, so that an objective, formal code aggregation criteria is needed.

Composable units are *intensive* code aggregations, that is, they only focus on limited parts of the implementation. Layering architectures, in contrast, are *extensive* aggregations, in that every class in the application belongs to a layer, with no exceptions. These differences are important in practice, because a layering architecture always guarantees a total decomposition of the implementation code, while composable units, being specific to a time and location, do not enforce full code coverage and are consequently harder to put into effect[5].

Microsoft's Composite UI Application Blocks architecture is an example of a composable unit strategy. This approach uses the concept of a shell application,

5. The concept of *intensive* and *extensive* entities is borrowed from physics. Intensive measurements are specific to some point in space, such as pressure. Extensive measurements tend to be constant in a whole environment, such as temperature.

for example a Windows form application that provides services to composable units, within which one or more composable units – called 'SmartParts' – can interact. Such units are the building blocks of rich client applications delivered as plug-ins, here referred to as 'Modules.'

Once a good architecture is devised for a GUI, and perhaps some form of clustering has been decided on, either extremely formal like a composable unit strategy, or a more informal approach such as panel reuse, we are still left with one last issue: facing the reality of development.

Evolving order and appropriate architectures

Changing an architecture during the implementation of a GUI is an apparently bizarre idea. However, client applications have common design aspects, and are usually smaller and more manageable than server-side applications. These aspects make architecture evolution more feasible for desktop application GUIs than for large server applications that have many external dependencies. Changing architectures is not a new idea – it occurs for example in Agile approaches, and also in the Evolving Order pattern for domain-driven designs (Evans 2004). But given the nature of desktop application GUIs, one could question the real effectiveness of such a potentially costly activity.

The Book of Five Rings contains all the art and science of the famed duellist and undefeated Japanese Samurai Miyamoto Musashi, who died in 1645[6]. Among the many insights in this book, one interesting thing concerns the acquisition of knowledge about an opponent. Musashi said that if you know your enemy, and you know yourself, you can't lose a duel – although that is clearly different from always winning. Transferring this idea to software engineering, we may say that if we know ourselves (our organization, people skills, technologies, and so on) and the problem to solve (the enemy) we cannot fail – at worse we can wisely give up a task that is too daunting. Then why do so many projects fail or produce poor results from so much effort?

Whole forests have been chopped down to explain this, and we don't want to event try here[7]. Luckily we can say something about architectures and the structuring of implementation artifacts, especially as regards desktop application GUIs.

The reality is that most of the time we don't know the problem exactly (including the domain, its context, technologies, the potential of our own organization). This is even truer when problems change over time, through relentless iterations, changing requirements, and so on. Our initial assumptions based on past

6. For an on-line version, see http://www.samurai.com/5rings/.
7. As the reader can see, I am following Musashi's advice here too.

experience and reasonable evidence might look completely misplaced after a few months in this challenging environment. To what extent will our old architecture design be able to copy effectively with the current problem?

If the architecture has to fit the given scenario (domain, extra-functional requirements, team composition, implementation technology) and the scenario is changing, the architecture should change as well. In order to have a 'fit' architecture – that is, one that matches the engineering task as a whole without uselessly wasting resources – we need to provide an architecture that, if necessary, can evolve smoothly.

Individuating patterns

It is possible to modify the structural configuration of an application using sequences of basic refactoring steps. The required input is a number of patterns that need to be present systematically in the code. It is possible to refactor GUI code at a macro level under limited circumstances, such as finding precise patterns in the code automatically, a simple transformation path to the new structure. This is normally possible only for rather specific scenarios within GUI implementations.

Using this assumption, one could see some aspects of a software architecture as a set of systematic low-level patterns found in the code. It does not really matter which particular pattern is present, as long as it is used extensively and systematically throughout the whole application.

When a given architecture is applied systematically and extensively, it is possible to refactor GUI code at a macro level to modify existing patterns into new ones, thus evolving the software architecture. In fortunate cases these refactorings can be automated, for example perhaps factoring out all Command pattern instances from the view layer in a three-layer architecture, making them part of the Application Model in a four-layer architecture[8].

The ability to modify an architecture inexpensively and predictably during development makes the whole issue of guessing the perfect architecture at project start-up less crucial. This in turn allows for more flexibility and a greater degree of adaptation to the problem at hand as it evolves during development iterations.

7.2 Some common GUI architectures

Having discussed some of the main issues of software architectures for GUIs, we are now ready to see some of them in action. This section discusses some common architectures seen in real-world projects. It briefly reviews some of the most used

8. See the next section for details about these architectures.

architectures for client applications in order of the complexity of the problem at hand. *A three-layer organization for GUI code* then discusses one particular architecture in detail, providing a practical implementation of it.

On the principles of continuous architectural refactoring and iterative development, one could think of these architectures as various evolutionary stages of the same application in its lifecycle, even though refactoring of a working architecture should be done only when needed.

The smart GUI antipattern

The shame of every GUI book is the 'smart GUI antipattern' (Evans 2004). This consists of shoveling all implementation into one class and forgetting about sophisticated layering techniques or fine OO design. Building a GUI this way has a small start-up complexity, but one that tends to grow sharply. Nevertheless, this could be just what we need in some situations. Simple projects with form-based GUIs and with little business behavior are good candidates for this approach.

By adopting such an approach, developers don't have to worry about decoupling, reuse, and domain-driven wisdom. The approach is to provide a class for each dialog or screen, and incorporate all the business logic, interaction, control, threading, and client–server communication we may need into this class. The only thing to worry about is providing some form of defined protocol for:

- *Presentation*. No matter how we have built them, screens must comply with standard GUI guidelines and other constraints.
- *Data IO*. Having everything encapsulated in a single (possibly bloated) class, all we need to care about is data transfer with a remote server. By providing a data transfer protocol with the rest of the world[9], we ensure a minimal protocol that can be used to decouple the poorly-implemented client from the server.

This approach couples well with the many visual builder tools available, doesn't require sophisticated developers, and results are almost immediate. With this organization, GUI testing options are limited to testing through the GUI (both automatic and manual), and some limited unit testing.

This architecture was the default choice in the early days of mainstream distributed computing (the early 1990s) when OO technology was still to become widespread and the server side of an application usually consisted of a relational database.

9. Perhaps using data transfer objects (DTOs), as discussed in Chapter 8 in the case of rich client applications.

A semi-smart GUI architecture

This approach is a slightly more structured variant of the previous architecture. A layering scheme attempts to factor out content – widgets and layout – and a data IO layer that defines the remote connection with a server. The rest of the application remains as a 'blob' of business rules, event listeners, and everything else that is needed.

In cases in which this scheme is just a temporary stage towards a more well thought-out architecture, the 'blob' layer will be thrown away and replaced by a more structured implementation. The important aspect of this layering approach is that the front-end interface (GUI) and the data transfer backend interface are implemented in separate, and therefore more easily evolvable and reusable, layers. This is as shown in Figure 7.3.

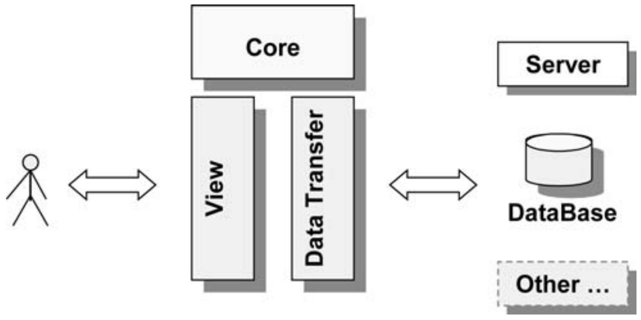


Figure 7.3 A semi-smart application architecture

This organization can be applied to cases that lie between the smart GUI antipattern and more demanding requirements that might await future iterations. Suppose for example that you are asked to port a Web application to a rich client, leveraging an existing J2EE server application. Suppose either that this is an exploratory project, or that you simply don't feel confident with a more complex architecture. You could then organize your GUI into three layers:

- The *presentation layer* is a passive container for widgets, layout, and other graphical details. No interaction or control is provided here. Widgets are manipulated passively by the core layer. Visibility is provided by package-level visibility or accessor methods. The responsibility of the presentation layer is only to contain widgets and organize their layout (for example content assembly). This allows panels to be assembled using visual builders, possibly by less skilled developers.

- The *data transfer* layer contains the POJO classes (or their XML equivalent, or the like) that enable communication with the external world.
- The *core* is the 'glue' layer that controls and drives the other two. This layer will be the most complex of the three, and potentially the least maintainable in the long term.

This approach has the same drawbacks and advantages of the semi-smart architecture, while providing a smoother curve regarding evolution costs.

I like to call this approach the 'ice cream cookie sandwich architecture,' in which you are only concerned about the two 'cookies' – the end user interface and the server application – and not about the 'ice cream' – the business logic and control – which remains a fluid, monolithic implementation.

A three-layer architecture

Following this popular approach, code is organized into three layers: one for the presentation (the GUI code), another for the application (a suitable representation of the business domain), and one for common utility services, as shown in Figure 7.4. This layering scheme focuses on reuse. Factoring services into a common layer modularizes the rest of the implementation and enables the reuse of the service layer.

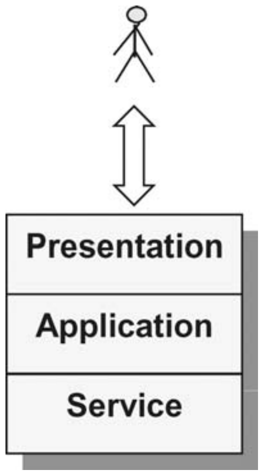


Figure 7.4 Three-layer application architecture overview

Some variants are possible, especially regarding the 'thickness' of the presentation layer. Some versions also have a quantity of control logic built into the presentation layer, while others prefer to maintain only simple control in the presentation as long as it is needed to spare the presentation layer from low-level GUI details.

Developers can use visual builders for defining content. Application layer code may contain some spurious reference to GUI classes, possibly concerning their higher-level aspects[10]. The service layer contains infrastructure code such as database connections, client server communication, and so on. Some layering strategies also put graphics utilities in the service layer, while others prefer to keep such toolkit-dependent code with the presentation layer.

The main differences between this approach and a semi-smart architecture are:

- In the semi-smart GUI approach, the view layer is thin and passive, while in three-layer architectures the presentation layer undertakes more responsibility, decoupling the application layer from widget details.
- The service layer gathers infrastructure behavior and works as a façade for a number of services, both in the presentation, such as providing the message localization service, and in the application, such as packaging a server request or handling a database connection. The service layer works also as a dynamic indirection facility at runtime.
- The purpose of this architecture is to provide a simple yet beneficial organization that allows for a robust application layer representing a business domain to operate within the application separately from presentation details, both statically, for example code references, and conceptually. For example, we might have exported existing business classes from a server application and want to reuse them in a client application.
- The 'glue' code is provided by the presentation layer. Most of the time this is just limited to a `main()` method that launches various components shared between the other layers.

A possible incarnation of this architecture is detailed later in this chapter.

A four-layer architecture

Four-layer architectures are a well-known extension to the three-layer approach, in which the application layer is split into an application model, responsible for decoupling the view layer from the domain model, and a domain model, in which view-independent domain representations manage information (see Figure 7.5).

This organization allows for better decoupling of the domain model from the rest of the view layer details. Depending on the variant chosen, the application model

10. This is the case with Swing applications in which model classes need to extend abstract toolkit classes, or interfaces where SWT+JFace domain classes don't have any dependency on GUI toolkits.

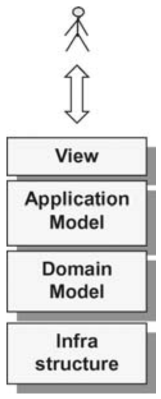


Figure 7.5 Four-layer application architecture overview

can factor out both commands and operations, not just GUI-related commands, so that the domain model remain a foundational layer for domain-specific knowledge (Evans 2004).

The application model layer will typically contain all the commands offered to users by the application: the Command pattern is used in both Swing and SWT for this. A 'thin' variant of the application model would typically contain no state apart from that needed by GUI commands.

In cases in which a high level of sophistication is needed, the infrastructure layer can also handle the application's internal communication infrastructure, or other services typical of server-side applications, such as JMS support, advanced caching mechanisms, and the like.

As we are only discussing desktop application GUIs, we are implicitly assuming that all four layers will be deployed on the client side. This architecture can also be deployed with the upper two layers (the view and application models) on the client and the other two layers on the server. This latter scenario is more common for Web clients, in which there is no need for off-line capabilities and the whole domain model can comfortably operate on the server side.

7.3 A three-layer organization for GUI code

It is now possible to look at the details of a specific implementation of the three-layer architecture discussed in the previous section. This section discusses a reuse-based decomposition of a client implementation that is based on three parts:

presentation, application, and service. These three parts are composed mainly of Java classes, possibly with other resources such as images, support files, and so on.

This scheme has its strengths and weaknesses, as we will see. Our objective is to discuss this type of GUI architecture in some detail, rather than suggest that it is some sort of 'silver bullet' architecture.

Overview

The presentation layer is what we see on the screen. Users interacting with dialogs or watching a splash window at application start-up are dealing with presentation objects. The other two layers are the 'behind-the-scenes' of the software:

- The application layer is where the application domain's objects are gathered, the business objects or domain logic.
- The service layer provides a wide range of standardized utility services.

Figure 7.6 illustrates this.

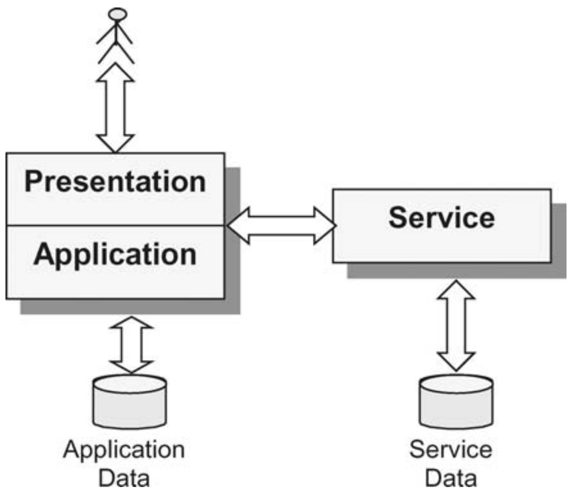


Figure 7.6 Three-layer GUI architecture overview

There is always a presentation layer in a user interface. It is made up of components that are usually inherited from `javax.swing.JComponent` (or, in SWT, from `org.eclipse.swt.widgets`), plus other classes that represent user input, or that are responsible for interaction and control. The user interacts with the presentation layer mostly with mouse and keyboard. This layer separates users from the application's logic.

The application layer lies immediately behind the presentation layer, tightly coupled to it. It is made up of Java classes that implement the logic and the business objects that are represented graphically in the presentation layer. If we have a clock window, for example, the application layer will contain a `Date` object that is tightly coupled with a `DateViewer` widget in a panel with some buttons, and so on, all of them in the presentation layer.

The third layer comes into play when we want to reuse some aspect of the code. Let's suppose we want to add more features to the clock. We want to offer international language support, with on-line help, and the option of customizing the clock's appearance depending on a user's tastes, and in such a way that users customizations are persistent across sessions. Thinking of these services as a separate layer helps in reusing them more systematically.

The following table shows how data is managed by the various layers:

Table 7.1 Relationship between layers and data

Type	**Data**
Presentation	Depends on the underlying GUI technology
Service	Authorization
	Configuration
	Help
	Localization
	Security
	User Profiles
	Etc.
Application	Application-dependent

This basic architecture is not intended to be all-encompassing, but rather to impart a minimum organizational infrastructure to GUI code, without being too pervasive. Developers can adapt it to their own production environments and needs.

Some of the benefits of this layering scheme are:

- *Division of work*. In the early stage of a development cycle, somebody will work on the GUI, designing and validating it with users, while perhaps someone else will take care of the business objects specific to the application domain, database issues, and so on. The two groups might even work in parallel after an initial period. This architecture helps to divide responsibilities neatly and so better organize the work.

- *Integration with existing toolkits*. This approach fits nicely with the Model-View-Controller (MVC) architecture and with similar object-oriented mechanisms that are in widespread use in Java programming, even though it can be used with simpler libraries such as SWT or AWT as well.
- *Flexibility.* One of the main practical advantages of such an architecture is its neutrality – it can be used for both medium-sized and small GUIs.
- *Common reference*. Like any kind of structured organization, this architecture is also useful for reference. Throughout the product lifecycle (and in this book as well) we can address functional parts with the same name. This helps developers working in teams to standardize their cooperative efforts. It also gives us an overview of all the challenges and problems designers and developers will face during the product lifecycle.

Some of this scheme's drawbacks are:

- *It needs a clearly-defined separation between presentation and application*. If this is not maintained, the architecture can easily degrade.
- *Extra care in testing is required.* The service layer can be a problem for testing. Mock-ups are needed for expensive services such as remote connections, databases, and so on, and special care is needed with Singletons that initialize statically.
- *It gives poor insulation for complex domain models*. The model doesn't scale well for projects with a complex domain model. In these cases a four-layer architecture is strongly recommended.

The following three sections look more closely at the three individual layers.

The presentation layer

The structure of the presentation layer is repetitive: the user interacts with some widgets, clicking with the mouse, filling up text fields, and so on. A control manager[11] is normally used to support the widgets, supervising all the widgetry and keeping it coherent – for example, disabling fields in a form until all required data is valid.

Figure 7.7 shows a high-level conceptual view of the presentation layer with a centralized control state that implements the Mediator design pattern.

11. See Chapter 6.

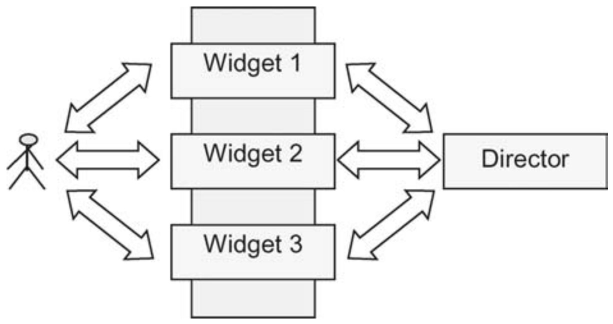


Figure 7.7 The presentation layer

In Swing this layer includes all the views in the MVC, plus related support classes such as table decorators, together with the objects that represent user input and control. In SWT it includes the content widgets.

The application layer

If we use Swing or other MVC-based frameworks, all the MVC models needed for the views in the presentation layer can be gathered here, as shown in Figure 7.8. As well as these, other objects are needed for the particular domain with which we are working. The application layer is the most variable of the three, because it implements the logic of the application domain (possibly accessing remote services) and also commands that use that logic. The application layer might also include other details apart from a representation of the business domain, such as commands, domain-specific support functions, and so on, and for this reason we prefer to call it the *application layer.*

MVC models can be used as the interface with the application layer. This is a simple choice and helps to decouple the two layers clearly – but 'pollutes' the application layer with classes that are needed to extend the GUI toolkit's interfaces for data models.

Even if your GUI does not use the Swing library, or JFace on top of SWT, this architecture still turns out to be useful – as shown in Chapter 10 in the context of J2ME GUIs.

The application layer, also known as the 'business domain model,' is domain-specific. You can find a comprehensive and insightful discussion of its design in (Evans 2004).

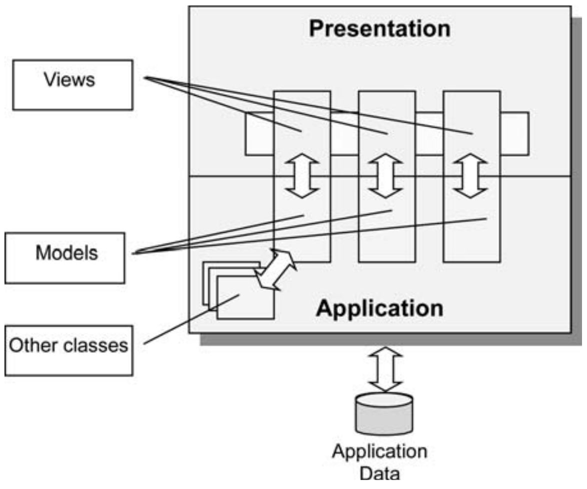


Figure 7.8 One flavor of application layer

The service layer

The service layer implementation described here has just one class, `Service-Manager`, as its interface with the other two layers. It usually performs all the initializations – loading configuration files, initializing external devices, and so on – and offers infrastructure services to the application and presentation layers. A common service offered to the presentation layer, for example, is localization support of widget appearance. Figure 7.9 illustrates a possible structure for this layer.

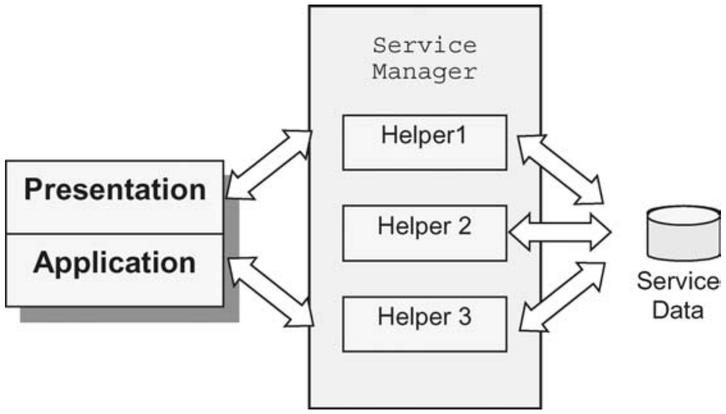


Figure 7.9 The service layer

Providing a single point of access is useful and intuitive, and can be used in a wide variety of situations, but it may pose problems in non-trivial applications. Code supervision should be enforced to prevent invalid services being moved into this layer, such as specialized Factories, for example. A configuration facility can be used to plug in new services, such as support for special hardware devices.

Even in simpler GUIs that have a minimal service layer implementation a service layer could be useful, because it enforces a standard, systematic yet simple structure on the code.

7.4 Two examples of a three-layer implementation

Two examples of the three-layer architecture serve to illustrate it:

- A simple example describes the architecture from a technical, programmer's viewpoint.
- The second example zooms out to see how the model can be used to organize a more complex and formal project.

These examples illustrate the practical application of the architectures described in the previous section, and will be discussed in detail in the following chapters.

We know from Chapter 3 that we can think of components – that is, subclasses of the `Component` class – as being divided into three groups, in order of complexity and development cost:

- Standard components, such as `Tree`, `Panel`.
- Custom components, such as `DataBaseTree`, `MyCoolButton`, obtained from the specializations of standard – Sun, Eclipse, or third-party – components.
- Ad-hoc components, such as the graphic equalizer in a music player, for example, that have no counterpart in standard components and must be developed from scratch.

An MP3 player

Kenrick and Rajeev are two friends in their first university year of study in computer science. They are developing some Java classes for playing MP3 files, for fun. One day Kenrick comes to his friend, very excited. He has found out from a Web site that a Java shareware distribution being launched on CD-ROM in a week's time. They will therefore have only a very short time to ship their product, and although their MP3 decoder classes work nicely, there is no GUI at all at the moment.

Rajeev has some experience with the Java Swing library, and decides to develop the GUI with the help of this book, while Kenrick will add the file streaming and other essential features to the Java audio subsystem classes they have already developed.

Rajeev is amazed by the possibilities Java can give their GUI, such as portability and a pluggable look and feel, but at present he has no time for advanced GUI features. He decides to build a simple GUI for the first release, leaving 'cool' features for future releases. The paper mock-up is straightforward, and the GUI design is inspired by similar products already on the market.

Rajeev gets into the implementation details of his GUI, devising the following top-level containers:

- A main frame.
- Four modal, unrelated dialogs, one for information about the current track, another for application settings, one for the help, and one for choosing files.
- Two accessory pop-ups – a pop-up window for the volume control, and a simple pop-up menu triggered by the right mouse-button on the track list.

Rajeev decides not to use any particular UI approach, mainly because he has never used one before and feels that he has no time to learn new material at present. He sketches the main window on paper, defining all its components, shown in Table 7.2, together with their development complexity.

Table 7.2 The main frame components for the MP3 player

Component name	**Java class**	**Type**
ToolBar	`JtoolBar`	Standard
TrackList	`Jlist`	Custom
TrackSlider	`Jslider`	Standard
StatusBar	`JstatusBar`	Ad-hoc

The next step – to define the required services – is straightforward: Rajeev decides not to use any standard service at all. He then defines the user interaction, basing it on the following commands.

- Track-related commands:
  - Play
  - Stop
  - Rewind
  - Fast forward
  - Pause
  - Show track properties, which displays the track properties dialog
  - Step back, which rewinds the current track by five seconds
  - Step forward, which steps the current track forward by five seconds

- General commands:
  - Preferences, which shows the dialog for changing the application's options
  - Help, which shows a simple dialog with a text area describing authors and product
  - Set volume, which shows the pop-up window with the volume slider control
- Track-list commands:
  - Next track
  - Previous track
  - Add to track list
  - Remove from track list
  - Select from track list, which plays the selected track in the list

The other user interactions are the right-click on the track list, which displays a menu with track list-related commands, the double-click that launches the 'Select from track list' command, and the keyboard accelerators.

A simple director class will manage all the GUI coherence – for example, the track list navigation arrows should be disabled when it is not possible to use them, such as at the beginning or the end of the list.

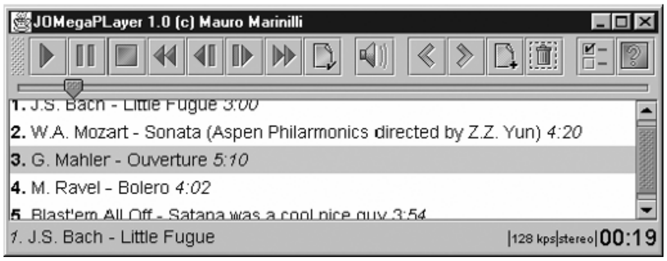
Once finished with the director class, the presentation layer and the service layer (here empty) are defined. The last step is to define the model classes in the application layer. There are three MVC models: the track list, the current track elapsed time, and the volume slider models. Another application class represents the tracks to be played.

Rajeev decides to incorporate the elapsed time model in the track class, because this simplifies the handling of the two views. The same object – the current elapsed time for the track – is observed by two views: the interactive slider and the read-only digital display at the bottom-right of the main frame. This type of model – the slider model and the ad-hoc digit display model within the status bar – is quite simple. The track list model is just an ordered collection of `Track` object specializations of the `ListModel` class.

Rajeev adds the other application classes that have been refined by Kenrick meanwhile, resulting in the GUI shown in Figure 7.10. Kenrick simply couldn't believe it.

With his remaining time, Rajeev refines the `StatusBar` component and tests the GUI with the help of their friends.

This example shows how a principled, top-down general implementation organization also helps in the development of small applications, not only at a technical level, but also in team organization.



```
1. J.S. Bach - Little Fugue 3:00
2. W.A. Mozart - Sonata (Aspen Philarmonics directed by Z.Z. Yun) 4:20
3. G. Mahler - Ouverture 5:10
4. M. Ravel - Bolero 4:02
5. Blast'em All Off - Satana was a cool nice guy 3:54
1. J.S. Bach - Little Fugue                |128 kps|stereo|00:19
```

Figure 7.10 The MP3 player's GUI (Metal1.2)

An electronic circuit simulator and editor

In the next example we take another perspective, without going into technical details, to see how a three-layer architecture can be used for managing large projects.

A joint venture by a university and a private software firm is set up to develop a graphical electronic circuit simulator and editor in Java. The plan is to begin by wrapping an existing, reliable simulation tool that is based on a command-line user interface that has been developed by university researchers over the past ten years. In future releases the code will be ported entirely from C to Java, so that the application will become 100% pure Java and totally cross-platform.

This highlights an interesting and important aspect of the Java community – not only its end users, but also developers, architects and designers. Java technology is widely used by open source and not-for profit organizations, where the Java characteristics of portability, inexpensiveness, and inherent multi-vendor sourcing, make it the perfect development choice in these cases.

The goal for the first release is to lay out the GUI, while the backend will interface with the existing legacy command-line application. Users are both engineers and academics, and the software will be released in two versions:

- A basic one as freeware, which will provide the same functions as the existing command-line application.
- A 'professional' edition that will be the starting point for future enhancements.

The software company will hold the copyright for the source code, possibly expanding the software to handle more features in the professional edition. They will manage the overall development, while leaving the university partner to work with the legacy code they know well.

We won't discuss the management of the project here, although this can be handled within the three-layer architecture as well: we just focus on the code development effort.

After the development group is created, three teams are formed, one for each functional layer:

- The presentation team will interact with the other two teams and with end users.
- The application team will interact with the other two teams, and with the researchers who developed the command-line simulator for the domain analysis.
- Finally, the service team will interact with the other two internal teams.

Development begins with an estimation of key numbers, such as how many and which type of services the GUI will need. Then, after several meetings with users of the command-line version of the application, the presentation team proposes the first GUI mock-up. In the meantime the application team starts discussions with the developers of the command-line simulator, while trying to define the software architecture best suited for a graceful migration to Java in forthcoming releases.

The first version of the GUI is shown in Figure 7.11. It took less than half an hour for an expert developer to build the mock-up using a visual GUI builder[12].

The idea behind the proposed GUI is to centralize circuit building and manipulation in one ad-hoc component, the circuit editor, shown on the right-hand side of the figure, while the left-hand side provides a data navigation facility that allows circuit elements to be inspected. The designed interaction is from the data inspector to the circuit editor and back – that is, selecting an element in the circuit editor automatically selects the data in the inspector, and vice-versa. A palette of circuit elements is used to add items to the circuit editor and the underlying circuit model interactively.

The circuit editor is chosen as a key component in the whole design, so the presentation team divides its responsibilities in two sub-teams: one responsible for the circuit editor and its related classes, the other for the remaining classes in the presentation layer, together with the organization of the feedback interviews with users.

The second presentation sub-team plan thirty or so commands (action classes), with one director and two auxiliary classes, allowing an initial approximate cost

12. See Chapter 5.

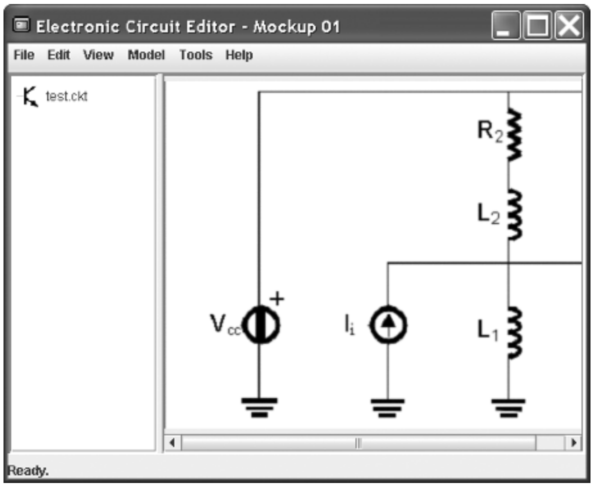

Electronic Circuit Editor - Mockup 01
File Edit View Model Tools Help
test.ckt
R2
L2
Vcc
Ii
L1
Ready.

Figure 7.11 The first version of the mock-up

estimation for the whole development. The design of the exploration area requires an extra meeting with end users.

After a feedback session with users, it emerges that the command-line interface, which is provided as an on-demand pop-up dialog in the first version, is so important and frequently used that needs a more central place in the GUI. The presentation team therefore decides to incorporate it permanently in the GUI. This change won't affect the application team that is working on the business classes.

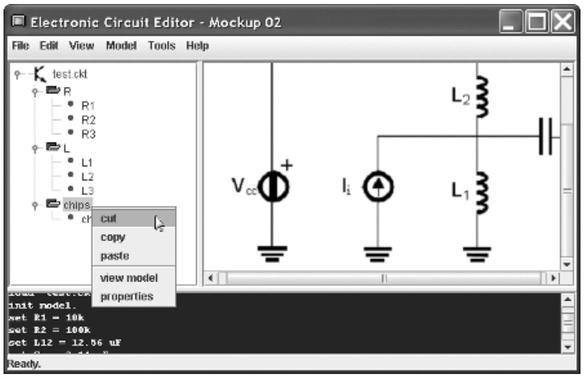
The project needs the following services: standard internationalization, interactive help, and basic persistence. The service team does not need to implement these functions immediately, but needs to publish the interfaces other teams have to use right from the beginning.

The two presentation teams continue the design phase. The first team has a somewhat easier job, in that they already have a clear idea of what to develop in the circuit editor component. The second team still has to define some design issues. For the data inspector, two hypotheses are viable:

- A table-like solution, possibly implemented with a specialized `JTable` component that adopts a high-density approach to data visualization[13].
- A hierarchical browser style using a specialized `JTree` and opting for a limited information layout strategy.

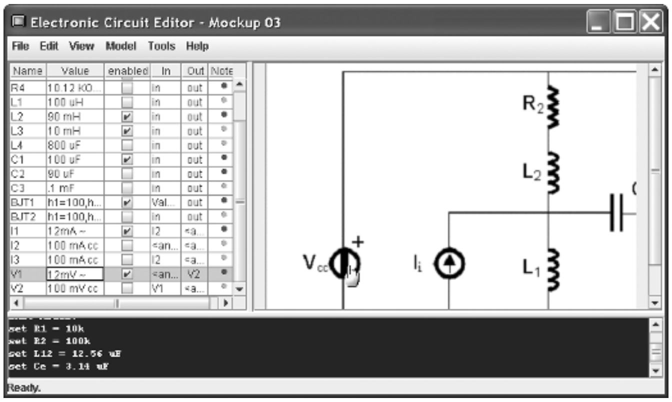
13. See Chapter 2.

The former solution with a tabular inspector is shown in Figure 7.12, and the latter solution has been sketched in the mock-up in Figure 7.13.



```
init model.
set R1 = 10k
set R2 = 100k
set L12 = 12.56 uF
```

Figure 7.12 The second version of the mock-up


Electronic Circuit Editor - Mockup 03
File Edit View Model Tools Help
R_2
L_2
V_{cc}
+
I_i
L_1
Ready.

Name	Value	enabled	In	Out	Note
R4	10.12 KO...	☐	in	out	●
L1	100 uH	☐	in	out	○
L2	90 mH	✓	in	out	●
L3	10 mH	✓	in	out	○
L4	800 uF	☐	in	out	○
C1	100 uF	✓	in	out	●
C2	90 uF	☐	in	out	●
C3	.1 mF	☐	in	out	○
BJT1	h1=100,h...	✓	Val...	out	●
BJT2	h1=100,h...	☐	in	out	○
I1	12mA ~	✓	I2	<a...	●
I2	100 mA cc	☐	<an...	<a...	○
I3	100 mA cc	☐	I2	<a...	○
V1	12mV ~	✓	<an...	V2	●
V2	100 mV cc	☐	V1	<a...	○

```
set R1 = 10k
set R2 = 100k
set L12 = 12.56 uF
set Ce = 3.14 uF
```

Figure 7.13 The third version of the mock-up

After some discussion within the development team and some brief usability testing with users, it becomes clear that the table-like solution is less usable and more difficult to manage, especially for large circuits with a greater number of elements. For budgetary reasons no other ad-hoc component will be implemented for this release, for example for the exploration area.

Meanwhile, the application team is working on the overall analysis, starting from the business classes and leaving the model classes, the part that is more changeable – at least at the beginning of the development cycle – to last.

The service team is working in parallel on the services the application needs: internationalization, persistence of the application settings, and help support. These features will be implemented using standard libraries (such as those provided with this book) so the implementation cost is almost zero.

After another meeting with users and university staff, it transpires that the GUI is mature enough to be considered definitive, at least for this release. An internal meeting is also held between the presentation and the application teams to define the interfaces (classes) that describe the boundaries across which the team's code will communicate. This is often just a matter of defining the models of the MVC architecture. In this version of the GUI there are three main models, corresponding to the three views used:

- A `DataTreeModel,` which subclasses the `TreeModel`.
- A `CircuitModel` that represents the electronic circuit managed by the circuit editor component.
- A `CommandLineModel`, for the command line component, which is a specialized `JTextArea` component.

While the `DataTreeModel` and the `CommandLineModel` are relatively easy to write, the first because it is just an implementation of the standard Swing tree model, and the second because of the intrinsic simplicity of the command line component, the `CircuitModel` is a new component, and hence needs more effort.

The interfaces required are agreed, and from now on the four groups have defined their responsibilities more clearly. The two presentation teams will use dummy model classes to refine the prototype, while the application team is still busy with domain analysis. The only possible problems could be in the definition of the `CircuitModel` class, which could change in the future. The service team finishes its job and its members join one of the three remaining teams. The two presentation teams work to refine the prototype, adding dummy delays and other real-world constraints, the second team constantly validating the user interface with end users.

In time the application team finishes the implementation of the three models, together with the remaining classes. After local tests, the three set of classes (application, presentation, and service) are merged in one application, while the three teams – the two presentation teams and the application team – continue to work separately. The end of the integration produces the first alpha release of the whole application.

This example shows a possible division of work for development teams on non-trivial GUIs and how this architecture can be applied on medium-scale projects.

7.5 The service layer

The service layer is essentially a reusable library that implements a number of services offered to both application and presentation classes. The key value of the service layer lies in its specialization. Centralizing general-purpose, service classes in a top-down manner provides a number of benefits. This section looks at the implementation details of the service layer, together with a simple implementation.

Overview

The services offered by the service layer are centralized in a `ServiceManager` class whenever this is meaningful. Expanding on Figure 7.6 on page 301 gives us the architecture details shown in Figure 7.14.

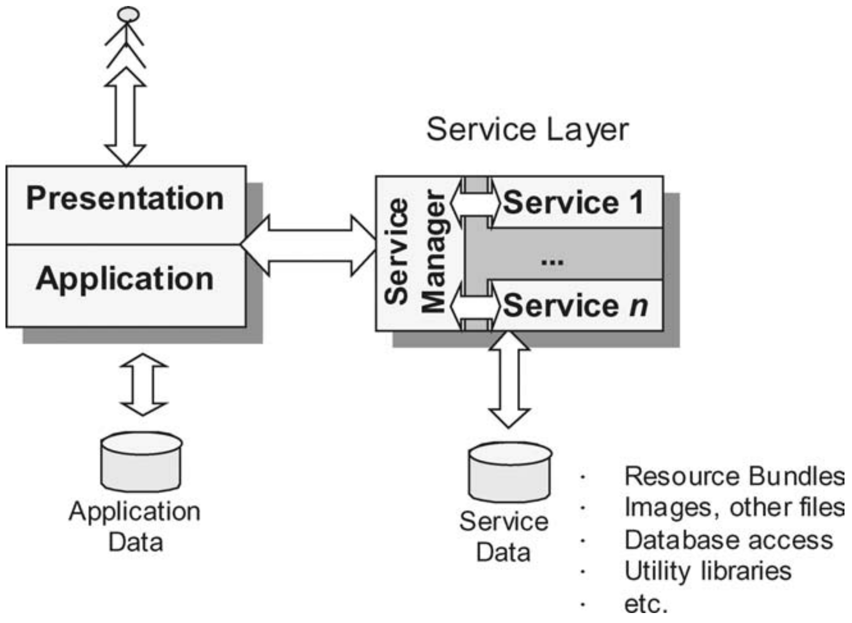


Figure 7.14 Architecture overview for the service layer

We keep the organization of the proposed service layer as simple as possible, providing a static structure with no dynamic discovery or plug-in of services. Figure 7.15 shows a possible set of services.

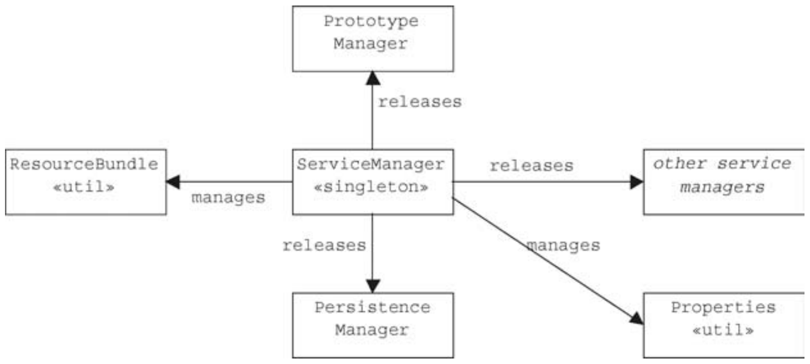


Figure 7.15 Service layer class diagram

The `ServiceManager` class is a Singleton that acts as a one-stop access point for many of the services provided in the service layer. From a software design perspective, such a class is a useful container for the many utility services needed by a graphical application.

The `ServiceManager` class also provides useful features at development and testing time. For example, when resources aren't found, a dummy (non-null) default resource is always supplied to keep the application running: a dummy image is built programmatically by the `initializeDefaultResources()` method. In this way a default image is always available, even when no resources are provided. For example, a picture (`EMPTY_ICON`) is returned by the `getImage-Icon` method whenever the requested image is not found. Similarly, the `getMsg` method doesn't abruptly break execution when a resource string is not found, returning an empty string instead.

The private constructor initializes each specialized service class by using lazy instantiation. Only when a particular service is needed is it instantiated on the fly. The same technique can be used during application shut-down.

Service interfaces should be kept as simple and homogeneous as possible, following the Segregation Interface Principle[14], especially when shared among diverse developer groups. General advice for developing public APIs should be

14. The Segregation Interface Principle states that clients should not be forced to depend on methods that do not pertain to them and that they won't use. Such external methods clutter the design and should be made available separately. For more details, see (Martin 2002).

used when designing the public interface of the service layer[15]. The enforcement of the principle of Single Functional Responsibility also helps to keep services 'fit' (that is, focused only on a well-defined, coherent responsibility) and more understandable.

Loading services

Loading external resources is a common functionality in an application. Images, property files, and other data should be loaded in a coherent way, because a Java application can be launched in different contexts.

More common situations could be:

- *Development time execution*. During development, testing or debugging, the application is in a protected environment in which some parts (resources or code) could be missing.
- *Standalone runtime execution*. This is the standard way to run a Java application.
- *Java Web Start runtime execution*. Given the particular implementation of Java Web Start technology, some mechanisms for loading resources cannot work.
- *Java programs run as applets*. In this case the loading mechanism is simplified by the applet container.

It is good practise to centralize external accesses to the local file system and the Internet. This strategy could prove useful also for security control and other issues. It will be far easier to change the loading mechanism used in the program if this is centralized in a service class. If the application is planned to be deployed in different scenarios, a pluggable specialized `ResourceLoader` could be provided. Nevertheless, in all common situations the loading mechanism provided by the `ServiceManager` implementation in the code bundle will suffice.

Localization services

Localization is essentially the loading of files that translate text messages or other resources shown in an application appropriately for different countries and cultures. These files can store references to the relevant resources, such as images and text strings, that need to be localized. For simplicity we will deal here only

15. Countless book discuss good OO design, such as (Martin 2002) mentioned above. For a more specific discussion, see for example 'Evolving Java-based APIs' by Jim des Rivières, available at http://eclipse.org/eclipse/development/java-api-evolution.html

with properties files, although more elaborate schema are possible, such as XML files.

Localization properties files are often edited and managed by different people, such as programmers or translators. To prevent problems, the development team can agree on simple guidelines for their format. The localization files provided here are compliant with a simple standard.

There are a few simple properties that should be enforced for localization files, as well as other configuration and development files:

- It should be possible to navigate back to the point in the source code where a text string in a resource bundle is used. *Traceability* is essential for maintaining the files in a complex development environment.
- The files should be owned by only one member of the development team. In this way responsibilities are clearly defined.
- The files should be kept to a reasonable size. Excessively large files are difficult to maintain and manage, while too many small files could be excessively resource-consuming at runtime.

We use a simple convention for resource files in this book that ensures these properties, and that has proved quite robust in large projects:

- Key strings are composed of tokens separated by a period. They begin with the fully-qualified class name, excluding common paths, and eventually include inner classes. A brief explicatory label is used.
- A two-character code is used for special-purpose labels, to distinguish the type of label. In this book we use the following suffixes:
  - '`tt`' for tooltip text
  - '`ad`' for accessible descriptions
  - '`mn`' for mnemonics
  - '`im`' for images
  - longer, ad-hoc suffixes such as '`title`' where required
- Finally, some additional information can be inserted in the heading comments, such as the current version, the authors, and so on. This would typically follow your company standards.

You can adapt this type of convention to suit your project's needs and development organization as required. If the planned application is not complex, or the development team is limited in number and stable over time, you can consider dropping any convention on properties files altogether.

An example of a resource bundle file following this convention is provided in Listing 7.1 below.

Listing 7.1 the `message.properties` file.

```
#
# Naming convention adopted here:
# (especially useful for large projects)
#
# 1.package path (excluded common base like "com.marinilli.b1") to
the first class that uses it, then the classname followed by "." and
# 2.optionally 2-char code (tt=tooltip, mn=mnemonic, ad= accessi-
ble desc., im=image), followed by "." and
# 3.finally a short description of the string
# (in case of simple text label the 2-char code is omitted)
#
# (c) 2000-2006 Mauro Marinilli
#

c6.util.ui.memory.MemoryCheckBox.label=Don't\ show\ this\ message\
again

c4.common.LoginDialog.title=Log\ In
c4.common.LoginDialog.login=Login\ Name:
c4.common.LoginDialog.login.mn=l
c4.common.LoginDialog.login.tt=insert\ user\ name
c4.common.LoginDialog.pword=Password:
c4.common.LoginDialog.pword.mn=p
c4.common.LoginDialog.pword.tt=insert\password

c4.common.AboutDialog.title=About\ J-Mailer\ Pro
c4.common.AboutDialog.close=Close
c4.common.AboutDialog.info=Info..
c4.common.AboutDialog.logos.im=CompanyLogo.gif
c4.common.AboutDialog.about.im=AboutLogo.jpg
c4.common.AboutDialog.version=Version\ 1.00.0.0002
c4.common.AboutDialog.text1=&copy;\ 2002\ All\ Right\ Reserved.
c4.common.AboutDialog.text2=blah\ blah\ blah\ blah\ blah

```

This listing refers to the message labels in two classes:

- `com.marinilli.b1.c4.common.LoginDialog`
- `com.marinilli.b1.c4.common.AboutDialog`

Remember that once your application is deployed, zipped JAR files eliminate most of the redundancy seen in message keys, both in properties files and in compiled classes.

Note that the ‘\’ character is needed to enable portability only when developing on multiple platforms, for example mixing Unix, Microsoft, or Apple Macintosh machines.

Resource bundles and other configuration files are also useful for non-technical staff that need access to messages and other GUI appearance material. For example, designers can use them to finely tune the final GUI.

Persistence services

Persistence services are provided as a way to save data persistently from session to session. Memory components[16], for example, are implemented by using persistence services. From J2SE 1.4, a limited form of persistence is provided by the library `java.util.prefs`. We will use such a library in the example application in Chapter 14 and its utility libraries. This example code is provided with the rest of the source code bundled with the book.

The `PersistenceManager` is organized as a Singleton, following the design of the other specialized service handlers. Its private constructor loads a specialized properties file from the local disk that stores class-persistent data in a text format. This method can be used for static variables and common object instances. Saving static fields requires you to serialize the whole instance. The `get()` and `put()` methods, and their various versions specialized for handling elementary types, work on the class-instance persistent cache. This is written and read as a properties (text) file. In contrast to binary serialized objects, text can be read and manipulated by humans easily.

Interested readers can experiment to see what is written in the `persistence.properties` file in the application directory, set to '`.app`' by default in the system root. This mechanism is quite useful for storing GUI options and other persistent data so that it can be manipulated with a text editor outside the application. For more details about class and instance persistence, see (Marinilli Persistence 2000).

For more details, see the implementation of the `PersistenceManager` class provided in the code bundle. Apart from class persistence, the proposed `PersistenceManager` class supports instance-level persistence by means of the `loadInstance` and `saveInstance` methods. This type of persistence is handled using normal serialized files, one for each object. The proposed implementation can be modified to use external libraries or other persistence means such as remote servers, databases, and so on.

Factory services

One example of an additional service that can be provided by a service layer is a facility for creating new objects from prototypes. This is an implementation of the classic Prototype software design pattern (Gamma et al. 1994). This service

16. See Chapter 4.

can be used by the application for creating new objects from templates, or directly by the user, as in the Library example application in Chapter 15. In this latter case, users can modify the templates themselves using the same GUI that is used to modify normal objects, so providing a handy reuse of an object's property screens for developers.

This general service can be employed in a variety of different contexts. The implementation of the `PrototypeManager` class supports a cache for storing class instances. Such a cache, implemented by the `repository` instance, is made persistent by means of the standard persistence services provided by the service layer.

To avoid unnecessary commits when the cache has not been modified, a 'dirty' bit is employed. The `firstTimeInitialization` method performs the initialization, building a number of default prototypes that are used when the program is first launched. These default prototypes can then be overwritten by new instances. The `PrototypeManager.defaultValues` property defines where the default values – used only at start up – are stored. The (private) constructor loads the array of default prototypes from disk. The static method `createNewFrom` creates a new instance, given the prototype object. It tries to fetch the prototype from the cache, and returns a new instance from it. In this way new types that are not provided at design time can be managed by the program at runtime. Finally, accessory methods like `remove`, `get`, and `put` can be used to manipulate the cache directly.

The `PrototypeManager` class is available in the code provided for this chapter.

Other services

Different kinds of services may be needed, depending on the application domain. Designers often include graphical utilities with the sort of general services discussed above, especially for medium to large projects. This approach can easily lead to a fully-fledged graphical-utility static class (that is, a collection of static methods) that solves common graphical problems such as component resizing, dynamic container introspection, and so on. A potential problem with this approach is that novice developers tend to reinvent the wheel, providing features that are already present in the standard GUI library, but often of a lower quality, because of a lack of knowledge of the technology used.

Common services are often those related to application IO, such as client–server communication or database connection management. A database manager can centralize connection pooling and other related features. Specialized managers providing adaptation can also be gathered here.

Providing new services

A few words about the boundary between the application domain classes and the service layer are relevant here.

Consider for example the domain of GIS applications, such as the Geopoint example in Chapter 3. In this context the API for geolocalization and managing physical values on the Earth's surface can be included as a general-purpose service within the service layer. Although wrong from a theoretical viewpoint (such an API is not a general-purpose one), it could be an appropriate decision if the company is specializing in geolocalization products and the API will be used in other applications as well.

The Swiss army knife syndrome

Improper design of the `ServiceManager` class can easily lead to a do-it-all service class with many disparate utility methods patched together. In (Brown et al. 1998) this scenario is called the *Swiss Army Knife Antipattern.*

A common solution to this problem is to differentiate the classes that provide the different services, eventually providing a more elaborated architecture within the service layer itself. This is another point of friction in the proposed implementation of the three-layer architecture in real-world scenarios.

7.6 Summary

This chapter presented advice about code organization, and proposed a three-layer architecture suitable for Java GUI development in detail. Such an architecture is composed of three layers, as follows:

- *Presentation*. This layer contains all the GUI-related classes and resources, essentially `Component` subclasses, and graphical resources such as images.
- *Application*. This layer gathers the business-specific classes and the remainder of the MVC classes whose view components were included in the presentation layer.
- *Service*. This layer consists of a standard reusable library of services that recur in all non-trivial GUIs. It can sometimes be expanded to handle special services typical of the current application.

We discussed the proposed architecture, providing two practical examples that highlighted the main advantages such an architecture provides.

Key ideas

Here are some of the more interesting ideas seen in this chapter:

- The main criteria and issues related to code organization for desktop application GUIs, especially for layering.
- Providing a code structure organized around service classes has a number of benefits, such as code reusability, a systematic software design, better communication among developers, and so on.
- Service classes can be reused easily to provide sophisticated services inexpensively. The factory services implemented by the `PrototypeManager` class is a good example of this.
- For simple applications the service layer approach can still be used, compacting the services offered to reduce the additional runtime overhead.

Aside from general discussion and practical examples of implementation organizations, we also have looked at the three layers of a proposed layering scheme in detail.

8 Form-Based Rich Clients

In this chapter we will explore a very popular class of GUI applications: rich clients (also known as 'smart' or 'fat' clients). While we focus our discussion on desktop applications, most of the concepts and approaches discussed here can be applied also to J2ME applications, as described in Chapter 10.

This chapter focuses on form-based GUIs, on their design, and on issues such as input validation, and distribution of code between client and server tiers, while other chapters complete the puzzle by providing related advice on this popular class of GUIs – Chapter 11 discusses various tools and technologies for Java GUIs, while Chapter 13 focuses on Java rich client platforms.

The chapter is organized as follows:

8.1, Introduction clarifies various details related to rich clients.

8.2, Reference functional model applies the abstract functional model discussed in Chapter 1 to rich clients.

8.3, Runtime data model introduces a general, simple, informal model for runtime data representation that is used in the example application.

8.4, The cake-ordering application, the XP way proposes an example form-based application created using this methodology.

8.1 Introduction

Figure 8.1 shows a screen shot of a fictitious application for Java-powered cell phones. This illustrates a type of Midlet[1] (see Chapter 10) that is provided by the central transportation authority of a major city to its clients. The user is admitted into the transportation system by means of an ingenious system – when requested by the user, the cell phone screen displays a special machine-readable pattern that is interpreted by fixed devices.

The application also works as an 'e-wallet' – the user can purchase transportation credits electronically – and receives broadcast messages about transit and other transport-related news. These messages are optional: the user needs to pay their

1. A small Java application that is intended to be executed within a managed container conforming with CLDC and MIDP profiles for mobile devices.

carrier provider for the cell phone traffic, so it can be disabled by users who do not wish for the service. This type of application works mostly off line – displaying the machine-readable pattern and thus allowing the cell phone owner to pay for the ticket – but needs some on-line type of connection from time to time, to update the user's credit and to show news messages.

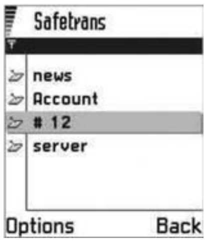


Figure 8.1 A J2ME rich client

This type of application cannot be implemented as a WAP (or Web) page, because it needs to be able to function off site. At the same time, by working off site and using the same appearance as the cell phone's software environment, it can camouflage itself alongside the other cell phone applications, and so appear more natural to its users.

The application must be easy to download, for example from a Web page, and be easy to use, because repetitive users are going to use it more than once a day. Being written in Java allows it to be deployed inexpensively to a wide range of devices. This is an imaginative – and imaginary – example of a particular breed of client GUIs, straddling the ground between complex Web pages and lightweight traditional applications.

Defining rich clients

There are many definitions for rich and smart clients, some of them elegant and useful, but none yet suit the uniqueness of the Java platform. We will introduce Java rich clients (RC) gradually, starting from their differences from Web applications, and next from the other GUI client environments.

First of all, let's look at the properties that define a rich client for the purposes of this book:

- They offer richer user experience than other media. If rich clients are to be useful, they should be superior to existing alternatives, namely Web-based applications. This is true for both end users and developers.

- They offer a *network-centric* approach. The ability to connect to remote servers smoothly, also for first-time deployment, together with the ability to operate off line, are two important characteristics of rich clients.
- They are *local environment-savvy*. In contrast to Web applications, RCs might have access to local resources, and can fit into the overall GUI experience of existing applications and operating systems. Also in contrast to Web applications, rich clients have to be designed with a specific target environment in mind, both technically and as regards the user experience.

It's useful to briefly review the reasons why we might need to develop a rich client application:

- When the application is targeted at many different platforms, or when ease of deployment and administration is favored over end-user experience, Web-based applications should be preferred over other client strategies. In situations in which a computer is used by more than one user, by occasional users, or when users access the application from more than one machine, then Web applications are also preferable.
- When end user productivity and overall experience are important, or when off-line capabilities are needed, rich clients should be preferred over Web applications. This latter decision should be based on an assessment of the real need, however, rather than 'nice-to-have' features.

Java rich clients

A wide variety of rich client technologies exist, such as Microsoft client technologies, Macromedia Flash, and many others. There are at least three factors that make Java different: a fully-fledged object-oriented approach, multi-platform execution, and a lively, highly collaborative developer community. While the first two factors can be problematic if not mastered, these three aspects together offer a unique blend of features.

Before going further, it is important to highlight the fact that Java desktop clients suffer from one major drawback due to Java's multi-platform characteristics. The hurdle is the Java Runtime Environment, which requires at least 7.2 MB to run (J2SE 5.x, using special optimized installers). This can be a problem that should be considered in advance before opting for Java client technology.

On the other hand, OO technology has been around for decades, and developers can rely on a host of well-proven techniques and patterns, of which this book takes advantage heavily. As domain complexity rises and application scale grows, OO technology has proven a valuable although labor-intensive approach. In addition, thanks to the availability of SWT, the portability of pure Java GUIs can be sacrificed for tighter integration with the underlying OS environment on some platforms.

GUI design for rich clients: the Third Way

First came traditional desktop application GUIs, which thrived unchallenged for decades. Then came Web browser-based GUIs, with their document-like structure, hyperlinks, lots of scrolling, and so on. Now, an interesting design cross-breed is slowly making its way onto our desktops. Perhaps you have already noticed some desktop applications that have hyperlink-like buttons, panels that resemble Web pages, and other features. This convergence is taking place on the Web side as well – think of 'rich' Web applications such as Google's services, and those offered by similar Web sites.

Unfortunately, this middle ground is still largely uncharted territory, in which GUI design habits and conventions are not yet established. There are no GUI design guidelines, nor even established informal idioms, and this 'third way' of lightweight rich clients remains a wild land that GUI designers enter at their peril.

This is a pity, because Java rich clients have the potential to offer a unique end user experience that falls between traditional desktop applications and Web applications, by providing a unique, platform independent 'Web desktop' feeling, backed by traditional software engineering technology and libraries proven by decades of industrial development.

8.2 Reference functional model

Returning to the functional decomposition we presented in Chapter 1 and discussed in Chapter 6, in this section we will specialize it further for rich clients. Figure 8.2 repeats the model from Chapter 1.

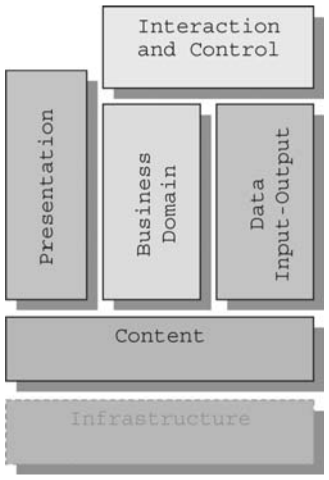


Figure 8.2 A functional decomposition for rich clients

It is worth reiterating that this is just one possible, general functional decomposition of a rich client application, merely a reference to theoretical concepts. On the other hand, when organized as a practical decomposition, rich client implementations help in the discussion of common issues at a higher level of abstraction.

This discussion applies to all Java GUI technologies, libraries and toolkits – the model is even valid for Web applications and non-Java GUIs.

To recap briefly, the various functional layers possible in a rich client application, following the model in Figure 8.2, are:

- *Content*. This is the 'base' of the GUI, composed of widgets, screens, and navigation. For convenience we can also represent widget layout information here.
- *Presentation*. This is an orthogonal layer to the others, containing graphics appearance and low-level presentation details such as look and feel information, icons and the like.
- *Business domain*. This is the logic of our application. Very simple rich clients do need very little client logic, so this layer would be almost empty in their cases
- *Data IO*. This layer contains behavior and data needed for exchanging information with the outside world. It mainly gathers data related to client–server communication and data binding information.
- *Interaction and control*. This is the topmost layer and 'glues' the other layers together. It is responsible for enforcing data validation by invoking rules from the business domain's layer, together with low-level interaction and control, such as disabling buttons or executing commands.
- *Infrastructure*. This layer is the foundation of the entire application, and includes the Java platform, the GUI toolkit (such as Swing or SWT), and other infrastructure frameworks such as a rich client platform like Spring RCP or Eclipse RCP. We won't focus on this layer here.

This functional decomposition will be a guide when discussing the many details and issues related to rich client application development. It applies not only to code, data and resources, but also to testing and business analysis as well.

Distributing behavior between client and server

One of the most obvious bonuses of developing a Java rich client lies in the synergies possible with server-side Java code. One such advantage is the ability to use the same code on both client and server. Another advantage lies in using the same proprietary protocols between client and server, such as RMI. The latter situation is less often available and it is fairly straightforward to implement: a wide range of tutorials are available on the topic.

A less-discussed although important issue concerns the distribution of common code between client and server, from a client application perspective. (Although we are discussing it here for rich clients, this point is valid also for other applications that need to connect remotely with server-side applications.)

One of the achievements of multi-tier systems is the ability to keep business code away from clients. This was a major step in implementing business logic that is dispersed on remotely-installed clients. Changing deployed business logic with 1990s technology was about as hard as updating the firmware of a space probe that had already landed on Mars. With today's deployment technologies, it is possible to bring business code back to client machines while maintaining centralized control, allowing rich clients to work off line as well, and to provide a richer user experience with more natural and responsive GUIs. See for example the discussion about data validation later in this chapter.

The important point is to *control* business logic, not on which tier it is physically located. With technologies such as JNLP and Java Web Start, for example, it is possible to update business code seamlessly, and also to force clients to update to a specific required version of the business code before running the application.

Unfortunately, even today's technology is still far from perfect, and distributing code between client and server is still one of the 'complexity boosters' for GUI applications. Bringing business code to the client complicates design, because apart from deployment issues, we need to design remote communication, with coarse-grained interfaces, DTOs, and all the required machinery of its implementation[2].

Business logic on the client also generates a number of issues concerning caching of data and code. Code caching can be performed with deployment technologies, but data caching it is still up to the application developer. Considering the transportation Midlet example, we need to set up a mechanism for synchronizing the user's local e-wallet, contained in the application, with the account held on the central server. Client validation logic might also need to refresh some parameters periodically – for example, we might want to update the currency exchange rate used to give customers an approximate transaction value before issuing them.

Rich clients need business logic locally, usually for data validation and other calculations on user-input data. Suppose we implement a loan calculator screen as part of a larger financial application, for example. This could instantly calculate

2. See for example the discussion on Data Transfer Objects in Chapter 6.

the market interest rate of financial data as soon as we insert some amounts, without incurring the overhead of time-consuming connections.

For more details about designing client/server communication, read the technical discussion in Chapter 6, or one of the many sources available on line and in the literature, such as (Fowler et al. 2003).

Common problems

There are several common problems related to rich client development. The most common solutions, in the form of OOP design patterns or simple best practices, were briefly discussed in Chapter 6, and will be shown in the various examples in this book. Figure 8.3 shows the choices needed when developing rich clients, together with the functional layer to which they belong.

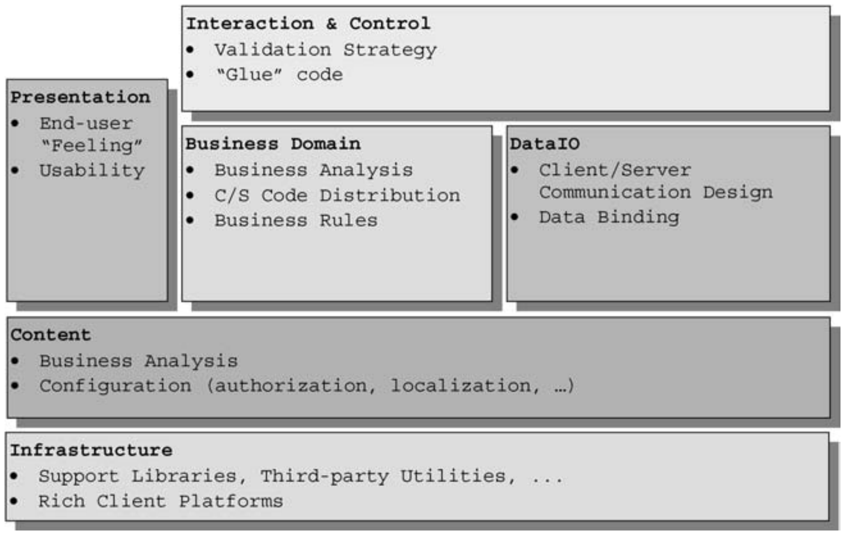


Figure 8.3 Common decisions for rich clients

The next section discusses a key aspect of rich client applications: the business domain data they handle.

8.3 Runtime data model

As well as the functional model discussed in the previous section, another useful model exists that is particularly apt for form-based, rich client applications. This model abstracts the data concepts that recur in all data-centric GUI applications. Data is the 'lymph' of rich client applications, and its management is essential for well-designed software.

We have data in a GUI that is first represented in a buffer within widgets, the *screen data state.* This data can be transferred in *business domain objects* (BDOs) for further processing, and eventually copied into other objects – or equivalent structures, such as XML files – for remote transfer as data transfer objects (DTOs).

Figure 8.4 shows this simple model of data representation within a GUI. Screen data state is the software interface to the user data with the remainder of the client application.

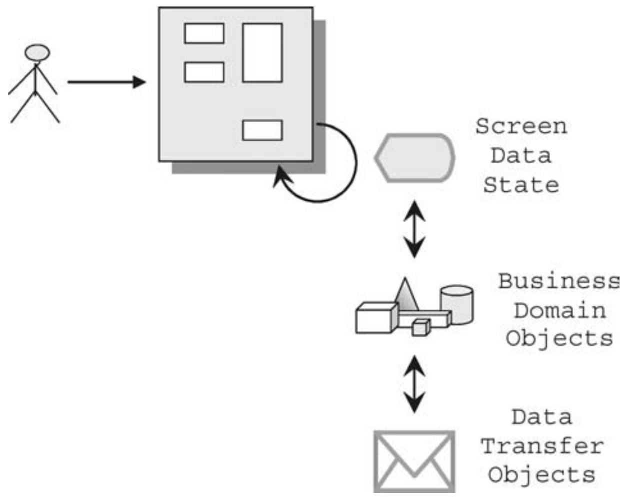


Figure 8.4 Runtime data model for rich clients

While they represent the same data for different functional purposes – end-user input-output, business processing, and remote communication respectively – these concepts help to clarify the implementation design.

The concept of screen data state can lead directly to application of the Memento design pattern to the data backed by content widgets. This approach is useful in a number of cases. Suppose we are building a form in which a great deal of data needs to be input by users to complete a transaction. A requirement states that at any point users can close the screen, and all the data they input previously should reappear when they re-run the application in a new session to complete the transaction. A simple way to achieve this is to serialize the screen data state Java Bean locally and resume it as needed.

As shown in Figure 8.4, we refer to three different representations of the information manipulated by an application at runtime. These are essentially copies of the same data, represented in different parts of the implementation. The synchronization of these various representations is dictated by the GUI design between screen data state and business domain objects (for example at form commit) and by technical constraints over transferring information between BDO and DTO. For simple applications these three types of data can collapse into just one representation: you can use the same class as widgets' data state, domain data, and transfer object, all at the same time. Such a trick is quite limiting, though, and might work only on very simple implementations.

Part of the J2EE community refers to DTO as *value objects*. This is misleading, because VOs have a different, more general meaning – their identity is based on their state, rather than on their usual object identity. VO examples are numbers, dates, strings, and currency values (Evans 2004).

(Fowler et al. 2003) distinguishes three types of data, depending on the runtime lifecycle. *Screen state* is the data that is deleted when a screen is disposed, so it chiefly corresponds to the screen data state. *Session data* lasts for a whole session, while *record state* is persistently recorded from session to session. Our approach here focuses on practical abstractions over implementation models, and not on the strict lifetime of data. Hence, for example, screen data state can survive even after the window is disposed – for example, because we dismissed a modal dialog, but we still have to access to its data – and business objects can be created for specific business logic computation, then dismissed along with a dialog.

A common problem with data representation in Swing widgets comes from an incautious use of the default model classes. These models come with predefined data structures that may not adapt well to the given application needs. I am still amazed by how often I have seen the results of a database query copied into some form of default table model subclass, thus uselessly duplicating data and wasting precious client execution time. Developers doing this have applied the idioms learned in simple tutorial applications, in which a couple of rows are loaded into a dummy data collection, to real-world situations. They can be apprehensive of going outside what they learned in tutorials and tackling the complicated internals of Swing's widgets. This problem is less frequent with SWT and JFace, but it is still possible.

While SWT can be used without a predefined screen data state implementation – such as JFace – Swing can be used *only* with its MVC models. This means that

Swing models are the only available option for implementing the screen data state buffer.

Validation

We want to assess the validity of user input as soon as possible. From an implementation viewpoint, validating data early is good, because code can be more robust and simpler. From a usability viewpoint, the closer to its input invalid data is notified to users, the easier it will be for them to understand the problem.

The simple runtime data model introduced in the previous section is useful for discussing validation. Validation is basically a form of interaction and control based on user input and business logic. From an implementation viewpoint, it can be seen as defining *security perimeters* over the data within our application. When designing an application, we can:

- Decide to confine invalid data to screen data state only, so that business objects and DTO remain secure.
- Decide to evaluate some data as BDO.
- Delegate the evaluation directly to the server.

These security perimeters are enforced by means of widget interactions, data filters, notifications to users, and the like. If we adopt extensive testing, this security check becomes less critical, but validation and notification still remains important, because it has an impact on end users. If data has already been validated in the GUI, it relaxes the need to further validation later, for example on the server.

As with any design, the more quality we pour into it, the more it will cost. The cheapest form of validation is of course no validation. We offload all responsibility to the server, which eventually returns notification to the client, for example the list of fields that didn't match some business rule. This kind of interaction slows down a GUI terribly.

On the other hand, as soon as we start to perform non-trivial business validation on the client, we observe an increase in development complexity on the client side. This is because we now need business domain classes on the client side, causing a whole new host of implementation and design issues.

Let's expand the runtime data model in Figure 8.4 to better illustrate validation, to give that shown in Figure 8.5. This figure represents examples of different forms of validation that can occur during a rich client session.

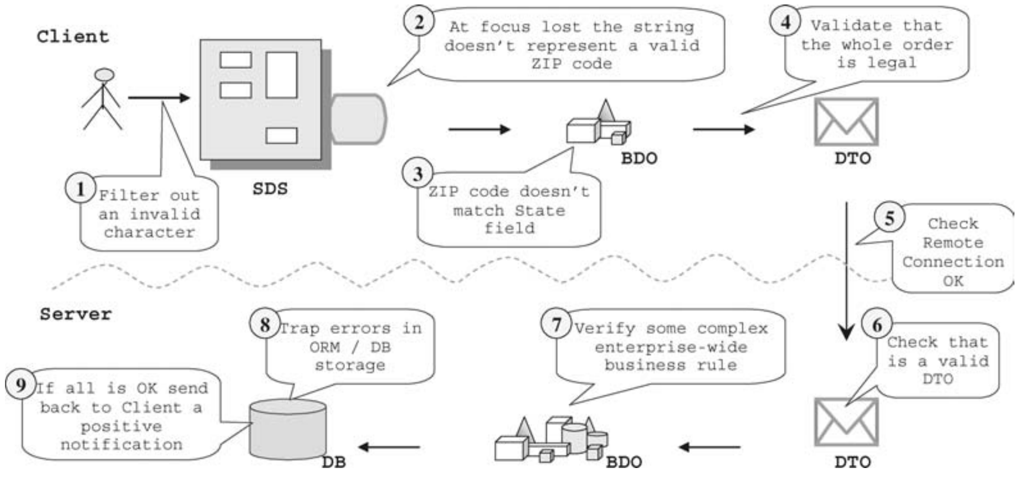

Client
SDS
BDO
DTO
1 Filter out an invalid character
2 At focus lost the string doesn't represent a valid ZIP code
3 ZIP code doesn't match State field
4 Validate that the whole order is legal
5 Check Remote Connection OK
Server
6 Check that is a valid DTO
7 Verify some complex enterprise-wide business rule
8 Trap errors in ORM / DB storage
9 If all is OK send back to Client a positive notification
DTO
BDO
DB

Figure 8.5 The journey of valid data from client to server

The figure shows the following examples of validations, in chronological order:

1. The simplest validation – that with the narrowest data scope[3] – can be done on low-level events. For example, filtering out all invalid characters in a Zip code text field.
2. Single values can be validated only when user data entry is completed. This is usually performed when the field loses the focus. Here the validation scope is a single field value.
3. More complex or wide-ranging validations also need other values. For example, to assess if a Zip code matches a State field, even in an approximate way. This type of confirmation needs a number of values to be assessed.
4. Before packaging data for transfer to the server, an overall validation can be performed, limited by the data and business logic available on the client.
5. Further validation can be performed on the server connection, such as time out, reliability, and so on.
6. When the DTO arrives at the server, a preliminary corroboration should always be performed, checking for (i) client authenticity, (ii) an eventual session consistency, and (iii) other forms of basic validation.

3. Here scope does not refer to the lifecycle of objects, as in the EJB specification, but just to the amount of data needed to evaluate a specific business constraint.

7. After the DTO is transformed back in business domain objects, plus further data that is available only on the server, a complete business validation is possible – possibly repeating the evaluation of client-side business rules.
8. Another validation is performed when persisting the business data. Exceptions and errors are trapped and treated as a (non-business) negative validation result.
9. Finally, the validation result is returned to the client for notification to the user.

The situation in Figure 8.5 is only an example, of course. Real-world situations can be simpler or more complex. In some situations business validation is performed entirely on the server, while in other cases data is requested from other servers to perform some form of business validation on the client – consider for example a transaction that requires results from various Web services that we might want to validate on the client.

No matter how complex a validation rule may be, it is always composed of the following six elements:

- The triggering event: *when* a validation rule has to be started. Activation events could be:
  - A low-level GUI event, such as a selected checkbox, a keystroke, and so on.
  - Focus lost or other forms of entry field completion. In the good old days mainframes used the **Enter** key to validate user input data, and completion was so much simpler to detect as a result.
  - Screen completion. When committing a dialog, either for submission, disposal, and the like. This type is also referred to as 'deferred mode,' while the previous two are referred to as 'immediate mode.'
  - At client–server connection. Before connecting – usually an expensive operation – data can be safely assumed to be ready for evaluation.
  - Other events as well may trigger a validation process, depending on the given situation.
- The scope: *what* needs to be validated. That is, the data needed in order to assess the result. Scope can be:
  - Simple low-level interaction data. Imagine a function that takes as its input a single character and validates it against some built-in criteria.
  - A single field value. This usually needs a simple validation
  - Some values. This case is mostly handled in the business domain layer.
  - Internal and external data. For example, an application may need to invoke a remote Web service in order to retrieve data about a user's identity, to be used together with input data to validate the current operation.

- *Where* the rule is to be evaluated, whether on the client or on a remote server.
- The business rules that logically define the rationale behind the validation. Even a simple rule, like 'only digits,' is rooted in a business rule.
- The notification to the end user. This is an essential feedback that can take different forms, depending on the kind of conventions we have established:
  - If we establish the 'silent success' convention, nothing will be provided in the case of valid values.
  - In other designs we could provide feedback, such as a change in the value's formatting, to provide response to the user that the value was input correctly.
- Further reactions to the evaluation outcome. For example, other fields might change value, or other control rules could be triggered, enabling/disabling other widgets, and so on. Such forms of reactive validation have the purpose of ensuring a specific level of quality on the application data, hence defining the security perimeter.

Note how the data scope depends upon the domain situation at hand, while the 'when' of validation is decided by the designer.

The distinction made in *Control issues* in Chapter 6 between business domain-dependent and non-business domain dependent logic also applies to validation.

For example, disabling the **Submit** button in a form until at least one field has been modified – to save the bandwidth required to submit unchanged data to the server – is a form of validation that is not dependent on the given application domain, and can be implemented as part of a reusable, domain-independent infrastructure framework.

Figure 8.6 shows the interactions that occur among the various layers in the functional decomposition when a validation rule is triggered.

The interaction and control layer is notified by the triggering event, and assembles the scope data needed to evaluate business rules. The results are used to prepare notifications, depending on some reaction policy, for example 'always delete invalid data if focus lost.' Further reaction is still possible.

This is just a theoretical scenario. In real-world cases we would probably require validation at the business domain layer only for complex, abstract business rules. Trivial rules like 'field must be numeric-only' don't need such a complex organization.

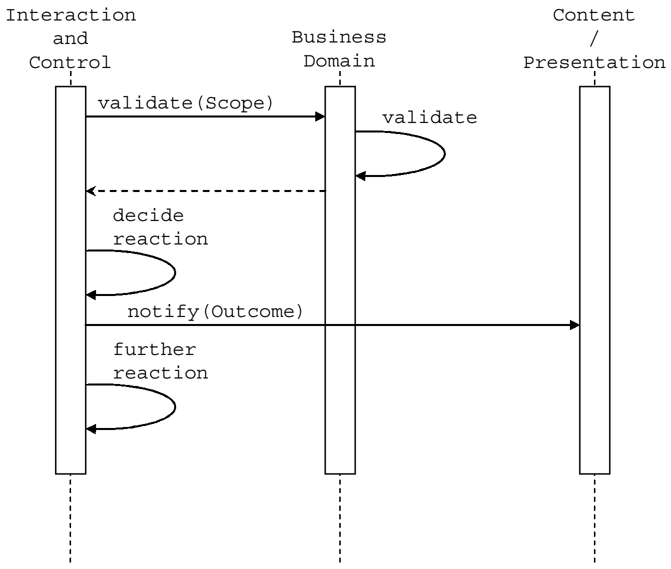


Figure 8.6 Sequence of interaction among functional layers

When seeking a validation framework to include in a project – to build, or to extend an existing one – we need to look for the following requirements in addition to the usual ones of good documentation, reliability, and so on:

- *Integration*. The framework should integrate well with other infrastructure code – data binding, and especially with business rules. Even if not mandatory, the ability to work with both Swing and SWT, for example through specialized presentation decorators for notification, would be an advantage.
- *Flexibility*. It should allow a wide range of validation strategies and designs. The ability to allow for a high-quality design of end-user notification is essential.
- *Extensibility*. It should be simple to extend, to provide unforeseen behavior or for handling particular cases (for example, validating JDNC components). For large projects, the ability to provide extensibility for higher-level validation behavior, within a more complex framework, is also important.
- *Non-intrusiveness*. The framework should not require special widget subclasses and other invasive design constraints. Ideally the framework should fit the interaction and control layer, as shown in Figure 8.2, without invading other functional areas.
- *Transparent remotization*. Whether we validate on the client or on the server shouldn't change the way the framework is used and the way in which it

works. When validating remotely, latencies must be taken in account automatically. This feature allows for an easy distribution of business domain code along distribution tiers.

An important question is 'Do you really need explicit validation support?' In simple cases, and for early development iterations, we can provide a good GUI without extra machinery. Criteria for adopting explicit validation support in our implementation are:

- We already have a formalized representation of business rules, even if on a functional decomposition basis only. That is, business rules are still represented informally but systematically in our code, for example as a collection of static methods.
- We are going to build more than a few screens that need a non-trivial amount of control.
- The development team is not homogeneous and/or we need to enforce fine-grained systematic development patterns. We are outsourcing part of the development, or we want to impose uniform development in a large team.

The user side

Our discussion so far has been mostly technical. Unfortunately, the trickiest issues in validation lie in usability. To begin with, reactions should be designed uniformly. While an improvised implementation of validation (if thoroughly tested) may pass unnoticed by the end user, an improvised validation interaction design certainly won't.

In practice, we might be forced to validate some data entirely on the client, because we have all the required information there, while in other parts of the same application we may need to send it to the server and wait for the outcome. Our GUI design should manage this heterogeneous form of validation in such as way that it provides a predictable and reliable experience to end user. This implies, among other things, that we must provide a clear indication in the GUI of those areas where validation is going to be performed remotely. Notification plays an important role in ensuring a high-quality user experience. If the user doesn't know why the focus cannot leave a field, or why the filled-in values have errors, the whole experience can be frustrating.

Too high level of reactivity can be confusing, such as fields that change value in reaction to other events too often, and it may be expensive to provide uniformly throughout the whole application. Such details are never a purely technical decision. Customers are always eager to automate data entry as much as possible, even by devising less-than-usable GUI designs. Our job is also to say 'no' to customers when we have to.

Even little hints can make an enormous difference for the user. Figure 8.7 shows some hints that reveal the data affordances[4] allowed by two fields, a date and a currency field. This speeds up interaction by preventing users from making erroneous inputs.

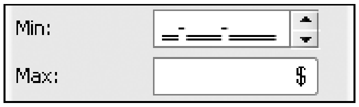


Figure 8.7 Providing visual hints as a form of preemptive validation

Visual or interaction clues help users to understand *why* data is not valid: they are part of the notification strategy. If the user knows that a given text control is a currency field and they entered an asterisk, they could probably figure out the reason for a validation error.

Another common example of signaling explicit validation constraints preemptively is the use of mandatory fields, which are illustrated by the dialog in Figure 8.9 on page 342.

Given the double nature of validation – the user side and the software side – its careful design is important from early development iterations. During development it is useful to think about intermediate validation strategies, and to evaluate them with users. Changing sensitive interaction issues such as validation from iteration to iteration can be frustrating for end users.

The overall validation policy should be defined in terms of a general objective. Completion time or data integrity could be possible objectives. In specific situations we might want the user to be able to work as smoothly as possible, always providing correct values when possible, while adjusting entries to match meaningful values, providing default values when possible, automatic completion with previous input, and so on.

In other cases the input data may be too important to make use of default values or automatic adjustments, and we might have to slow down the user's interaction, using an analog of speed bumps in the GUI, and implement a stricter validation policy. We might even want to be intentionally cryptic for security reasons. For example, while filling in a bank account transfer operation with codes that don't

4. See Chapter 2.

match, we might want to issue a generic error instead of being more specific and by so doing, reveal sensitive data.

The general objective should be defined in terms of the user population. Most of the time Java rich clients are built mainly for repetitive users. In such cases visual and interaction smoothness must be balanced against the clarity and completeness of information.

We provided general advice for form-based GUIs in Chapter 2. Here we briefly recap the main points from a validation perspective, referring to Chapter 2 for more details.

- Provide clear, consistent, and visually non-intrusive validation signals:
  - Provide a hint to the data affordances allowed by a field, as shown in the example in Figure 8.7.
  - Signal mandatory fields, using an asterisk in the field label, a border decoration or the like. Don't provide signals for optional fields.
- Guide user input as much as possible, but without getting in the way of users' work. Consider the investment in creating your own widget support.
- Provide meaningful feedback. Validation errors, warnings and status should be devised early in the development process, so that inconsistencies across the application and costly modifications can be avoided. Enabling or disabling portions of the GUI can be a valuable form of interaction. Try to minimize the cognitive load on users by providing local feedback, instead of messages like:

  ```
  Wrong value on <address> field
  ```

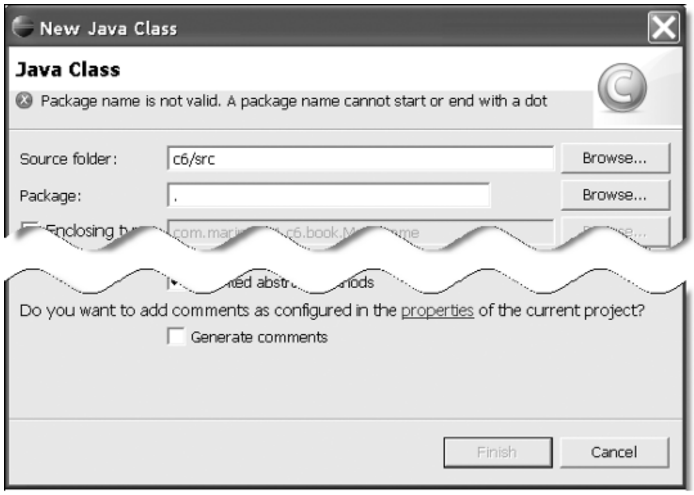
  Don't forget that you have full control of the GUI in a rich client!

Validation should also harmonize with the local visual conventions. This will save development time and provide a uniform experience to end users. For example, when building an Eclipse plug-in, we should always use Eclipse's built-in validation design, as shown in Figure 8.8.

In the Eclipse GUI guidelines, validation is performed as much as possible in immediate mode, and feedback is provided at the top of the dialog. This notification mechanism, simple to implement for developers, is fine for repetitive users, the main target of the Eclipse user population, but can be a little uncomfortable for occasional users, who might have to scan the GUI to find the package field that happens to cause the problem.

When to validate and notify

Ideally data should be validated as early as possible, so that users have the lowest possible cognitive burden in associating notification with their input. Validating and notifying data as early as possible means that, when the data scope is limited to values already known, the best moment to validate a piece of information is immediately after the user has completed data input.


New Java Class

Java Class

Package name is not valid. A package name cannot start or end with a dot

Source folder: c6/src Browse...

Package: . Browse...

Do you want to add comments as configured in the properties of the current project?

Generate comments

Finish Cancel

Figure 8.8 Eclipse GUI as an example of local validation style

Three strategies are usually used for starting validation and subsequent notification:

- *During user input*. As the user enters data, it is validated constantly, even if incorrect values are typed before completion of input. This is the strategy chosen by Eclipse – see the dialog in Figure 8.8, for example. This requires constant invocation of validation methods, which can be a costly overhead in complex Swing forms. If validation involves expensive calculations (in time or resources) this method should be avoided. From a GUI design perspective, immediate validation should be unintrusive as possible – we don't want to disturb the user with bells and whistles while they are entering data that is only temporarily invalid.
- *At user input termination*. This usually corresponds to the user leaving the field (on the 'focus lost' event). This is cheaper, but it might involve some subtleties, as explained in *The hidden pitfalls of validation* on page 342.
- *Deferred to a given event*. Examples might be form submission to the server, or when the user presses a button. The farther the notification from the input, both in time and space, the harder it is for the user to correctly interpret it. For this reason deferred notification is usually best performed with local, extensive clues.

How and where notify

A number of approaches can be used throughout a rich client application for notifying users of the outcome of input validation. Also in this case, consistency is a very important factor for an effective notification strategy. Some common forms of notification are:

- *Final summaries*. In deferred notification it can be useful to recap a form's data before submission. This approach is rarely used in rich client applications, however, as developers can use more powerful and direct communication.
- *A notification area*. This consists of a fixed area of the screen that is devoted to communicate with users, for example a status bar. This technique makes it possible to convey more information than with other approaches, and can be a good choice for applications designed for occasional users. In some cases navigation cues can be useful, for example signaling the notification information (text message plus icon) of the current focus. The major drawback of this approach is the load of keeping the notification area updated, and shortening the link between the focus and the notification area.
- *Local notifications*, done with:
  - *Icons*. Icons are a simple yet effective notification means that is better suited for repetitive users. Be aware that people with visual deficiencies may have problems with icons that are too small.
  - *Text messages*. Labels beside fields can convey notification information and constraints. The problem with this approach is that it can be expensive in terms of screen real estate.
  - *Colors and other adornments*. Special borders, background colors or other visual signals can notify the user of the outcome of input validation.
- *Control*. Disabling widgets after validation can be useful, but it needs an expressive notification support, otherwise occasional users can find it hard to understand the reasons for specific reactions to their input.

To simplify our discussion, we assume that validation and notification are close in time. In some cases this may not be the case.

Figure 8.9 shows some examples of local notifications.

For brevity Figure 8.9 shows various notification styles together, which explains its confusing and inelegant appearance. The figure shows examples of various local notification strategies:

- Providing an icon, in the case of the **Postal Code** field.
- Using label adornments such as color, or font style, to signal mandatory fields or invalid values, and so on.

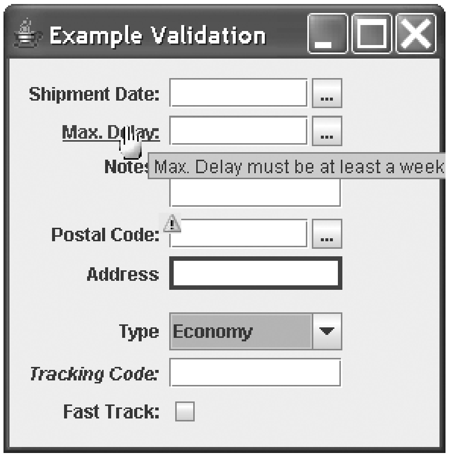


Figure 8.9 Local notification examples

- Using tooltips or hyperlink-like labels to provide more information to occasional users about why their input value is not valid.
- Supplying borders or background colors to communicate notification result, or that a given field is mandatory.

Carry out at least a basic but effective usability test with your end users, even if you are adopting a supposedly harmless validation design, perhaps provided as the default by a third-party library. Colors or other adornments such as borders can be hard to notice – 1 in 12 people have some sort of color deficiency[5] – or be too visually overbearing for repetitive users.

The hidden pitfalls of validation

Just because a specific GUI technology allows a specific feature, it doesn't necessarily mean that it should be employed in a GUI design. This applies equally to validation. In this section we discuss a concrete case of misuse of technology in implementing validation in a rich client application.

There is an interesting quirk in Swing's validation support – one of several – that highlights some of the complexities involved in handling low-level events, specifically focus-lost events, and validation at the same time. Swing provides the class `InputVerifier` to allow developers to validate user input before the focus leaves a widget. Using this mechanism in the case of invalid data, we could restore an old value, or we could force the focus back to the field to allow valid

5. Source http://www.iamcal.com/toys/colors/stats.php.

data to be input. When we choose this latter approach, though, we have an additional problem – users remain trapped in fields that have invalid values.

Suppose we have a date field that forces users to type in a valid date before leaving the field. In the case of an invalid date, users cannot leave the widget. Clearly this is clumsy GUI behavior. Users cannot close a dialog, as this would involve moving the focus from the date field to push the **Cancel** button, or to click on the **x** icon, or do almost anything else before entering a valid value.

To solve this problem in a general way, a specific method was added to the `JComponent` class: `setVerifyInputWhenFocusTarget()` defines when a component can circumvent an `InputVerifer` block on focus. Thus, by setting the `verifyInputWhenFocusTarget` property of the **Cancel** button to `true`, users can close a dialog even when trapped in a field with an invalid date value.

Unfortunately, this still doesn't solve the issue, for a number of reasons. A particularly nasty one is the following. Imagine that an indecisive user clicks on the **Cancel** button without releasing the mouse button and moving the mouse pointer away from the **Cancel** button. The **Cancel** button will not be triggered and the dialog will not close. The focus is now transferred to the **Cancel** button, thus escaping the block on invalid values and crippling the entire validation schema.

The bottom line is to avoid being trapped in complex situations. If we observe a steady increase in complexity in our implementation without any real benefit, we should question our previous choices and be prepared to lose some work instead of heading towards an increasingly convoluted situation.

In this example the mistake was to block user input of invalid values in the first place. Things can get even more complicated such an application is released without extensive testing. Users are now accustomed to a blocking validation approach, and it becomes a political issue to change the data validation interaction throughout the whole GUI. This is a typical case in which developers might blame perverse end users, or the toolkit, rather than themselves.

8.4 The cake-ordering application, the XP way

This section introduces an example of a simple rich client GUI developed with iterative design practices that follows the Extreme Programing (XP) methodology introduced in Chapter 1. Our intention is to illustrate common issues that arise when following an iterative development approach to GUI development.

Our customer is the Cake-o-Matic company. This company makes custom cakes to order that are then delivered to clients. They order their cakes through a call center. Phone operators use a rich client application to place orders with the factory and to organize delivery. The application's end users are call-center operators – frequent, but possibly unskilled, users – who will use the GUI while interacting by phone with Cake-o-Matic's clients.

Setting up the first Iteration

The first iteration will take two weeks and will implement the first story, the placement of a simplified delivery order on the server. In an Agile approach we focus constantly on the specific practises discussed in Chapter 1. Figure 8.10 shows the simple application decomposed in its functional parts. For the purposes of this example, we adopt the theoretical functional decomposition as the layering architecture for code and tests as well.

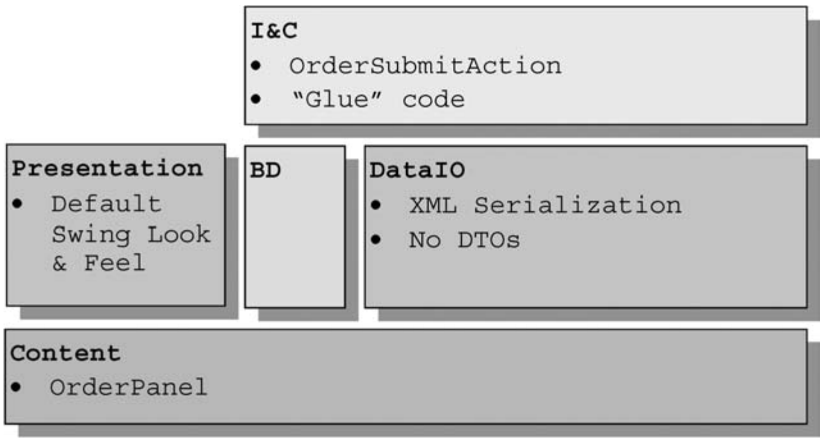


Figure 8.10 Functional parts for a first iteration

Following an XP approach, the first iteration of the application will be the simplest thing that could possibly work. There will be no business domain layer at all, we provide no custom presentation, and data IO will be limited to plain XML serialization. The runtime data model will be even simpler. There will be no DTO, nor a business domain model. We avoid all validation.

An occasional misunderstanding is that following an Agile approach means focusing on implementation, then deriving the GUI design accordingly. This is a completely wrong assumption. GUI design should always drive the implementation. Iterative development is just a mechanism for implementing GUI applications effectively. Considering development costs during design, as in the cost-driven design approach described in Chapter 3, is part of GUI design, not of its subsequent implementation.

Defining the testing strategy

It remains to define the overall test strategy for the project. We know from Chapter 5 that we need to find the best mix of two main options:

- Testing through the GUI (TTG). This is slow and somewhat coarse-grained when compared with traditional unit testing. We need TTG at least because it's needed for automated acceptance testing – another useful XP practice.
- Testing bypassing the GUI. This is faster than testing through the GUI, and we are going to adopt it extensively in our project.

As we pointed out in Chapter 5, TTG is useful for writing acceptance tests – ideally created by the customer, but we can provide them with some support – for testing interaction and control, although we won't have this at least until the second iteration, for end-to-end coarse testing, and for all those tests that necessarily involve the GUI. Non-GUI testing (such as plain unit testing à la xUnit) it is required for Agile approaches such as XP – continuous refactoring and integration, TDD, and so on.

In a real world scenario, with such a simple application, manual TTG would probably make more sense than using the various testing tools we are going to use in our example. To illustrate the chosen development approach, we opt for a fully-automated testing strategy. Chapter 5 discusses more considerations for general cases.

To highlight the differences in the two approaches to testing:

- Unit testing can be performed thousands of times for each integration.
- TTG is much slower, running at users' speed – open a dialog, click here, insert text there, wait for the result to appear here, and so on.

The first approach would be much better than TTG, but unfortunately it is not exhaustive enough for GUI testing. We may want to use TTG selectively for automatic acceptance tests, interaction tests – for example testing our interaction and control rules with the most extreme data/interaction sequence we can think of. We may also want to use it for generic end-to-end testing, such as inserting some data, submitting to the server and check the final result, and profiling – an often overlooked aspect of GUI testing that can be fully assessed only with a TTG approach. (Imagine for example a GUI in which we want to perform hundreds of transactions and probe the inner state of the client JRE, for example to look at memory allocation or other properties on the server side.)

To achieve a pervasive testing mechanism, one could resort to extending the underlying toolkit to provide automatic test behavior. Given its flexibility, Swing is particularly well-suited for this technique. For example, we could register a custom `ComponentUI` factory for testing presentation details, or extending one of the many default factories such as that for `Formatter` classes.

Content first

Content is the base of any rich client application. Beginning iterative development from content implementation is a simple approach – we just translate use story GUI prototypes directly into code, thereby lowering risk as long as we validate our work with the customer, and providing a basis for all subsequent work. Working with the customer results in the sketch GUI shown in Figure 8.11.

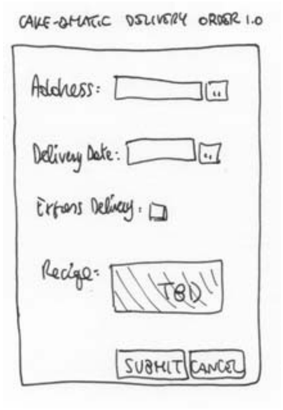


Figure 8.11 GUI prototype for the delivery order panel

Our first step is to implement it. Using a TDD approach, we start with our first test case[6]:

```
public void testWidgetsExists() throws Exception {
  DeliveryPanel dp = new DeliveryPanel();
  Assert.assertTrue(PrivateAccessor.getField(dp,"recipeDesc")instan-
ceof JTextField);
  Assert.assertTrue(PrivateAccessor.getField(dp,"deliveryDate")instan-
ceof JTextField);
  Assert.assertTrue(PrivateAccessor.getField(dp,"expressDelivery")
instanceof JCheckBox);
  Assert.assertTrue(PrivateAccessor.getField(dp,"pickUpDelivery")
instanceof JCheckBox);
  Assert.assertTrue(PrivateAccessor.getField(dp,"submitButton")
instanceof JButton);
  Assert.assertTrue(PrivateAccessor.getField(dp,"cancelButton")
instanceof JButton);
}
```

6. This first step is probably too big: our objective here is not to introduce XP practices (there are many books about that) but to show common issues when adopting XP practices for effective GUI development.

This test fixture was written with JUnit and the JUnit-Addons suite, which provides the `PrivateAccessor` class for testing private members and methods. Clearly this test fails, because class `DeliveryPanel` does not exist yet. Nevertheless, it is defining the content of the application, because it mentions the class `DeliveryPanel`, its intended widgets and their types. For brevity, we will skip testing the `DeliveryPanel` class.

After a brief discussion between our two programmers, they decide to go with private members for widgets. As with every beginning, we are full of good intentions and we definitely want to keep the widgets for our forms private.

This choice gives us the chance to discuss the possibility of testing properties on private methods and members. This is not regarded as an orthodox approach, otherwise the class `PrivateAccessor` wouldn't be part of an optional package. From encapsulation and data hiding dogma, we know that modifying the inner (private) behavior of a class shouldn't necessitate rewriting its tests. On the contrary, if we need to access such private members only for testing, would it be wise to make them publicly available through accessors, thus cluttering `DeliveryPanel` without providing any real application-level behavior? The two programmers could go on discussing this issue in length: "Application and testing behavior are at the same level… this way production code can be in a separated package from tests… once you have the accessor methods you might want to use them for other purposes… but this violates Agile methods' simplicity principle…" and so on. This is important, because such a dilemma underlies implicitly the importance we want to give to unit testing within our GUI development. A whole school of thought asserts the need to make unit testing drive any implementation, especially complex ones like GUIs. We will return to this important and often overlooked aspect of GUI development later.

Our work now focuses in implementing the `DeliveryPanel` in order to make the test pass.

We quickly write the following class implementation:

```
public class DeliveryPanel extends JPanel {
  private JFormattedTextField deliveryDate;
  private JTextField recipeDesc;
  private JCheckBox expressDelivery;
  private JCheckBox pickUpDelivery;
  private JButton submitButton;
  private JButton cancelButton;
}
```

To our surprise, we still don't pass the test. Fields need to be instantiated. We therefore provide a constructor:

```
public DeliveryPanel() {
  deliveryDate = new JFormattedTextField();
  pickUpDelivery = new JCheckBox();
  recipeDesc = new JTextField();
  expressDelivery = new JCheckBox();
  submitButton = new JButton("Submit");
  cancelButton = new JButton("Cancel");
}
```

This time we get the green light. Now basic content is secured. What's next?

We would like to have some concrete, reassuring feedback – that is, we'd like to have something that we can see working and make us feel good. We want to put our widgets onto the panel. This will also serve as a basis for writing acceptance tests.

Laying out widgets is not that difficult, but how should we test it? The easy way would be to prepare a clever unit test fixture, but this cannot test our layout visually. Testing for properties within the layout class leads to brittle test code. If we change the layout class or some layout parameter, as is more than likely in future when we refine the details, we would need to change the test as well.

On the other hand, we can use a GUI testing framework to define an all-*visual* test. The developers opt for this choice. They use JFCUnit, an extension of JUnit – although in this example tools are not important, as we are focusing on concepts. We would like to express the visual properties of the widgets in Figure 8.11 in a form of automated test. We therefore define the following JFCUnit fixture:

```
public void testWidgetsVisible() throws Exception {
    NamedComponentFinder finder = new NamedComponentFinder(JCompo-
nent.class, "cancelButton");
    JButton cancelButton = ( JButton ) finder.find( dp, 0);
    assertNotNull( "Could not find cancel button", cancelButton);

    finder.setName( "submitButton");
    JButton submitButton = ( JButton ) finder.find( dp, 0);
    assertNotNull( "Could not find Submit button", submitButton);

    finder.setName( "recipeDesc" );
    JTextField recipeDesc = ( JTextField ) finder.find( dp, 0 );
    assertNotNull( "Could not find the recipeDesc TextField", recipe-
Desc);
   assertEquals( "recipeDesc field is empty", "", recipeDesc.getText(
));
...
}
```

We can make a couple of observations from this. First of all, we are merely asserting that the widgets are found in the container panel, and that they are empty. Then, there is a lot of machinery for interfacing with the Swing toolkit. This time we went closer to reality, as this test is stronger than the previous ones. We observe that the previous test (JUnit) took 0.1 seconds to execute. This one takes 0.95 seconds. Clearly, we expect a proportional increase in time consumption as the number and the size of tests grows. But we can also observe that the tests are largely overlapping.

When we launch the test, it fails, complaining that it cannot find widgets, despite the fact that we created a frame in the fixture's `setup()` method and showed the panel properly.

The reason is that Swing requires widgets to be named, using the `setName()` method, for the finder facility to work.

So our `DeliveryPanel` constructor becomes:

```
public DeliveryPanel() {
  deliveryDate = new JFormattedTextField();
  deliveryDate.setName("deliveryDate");
  pickUpDelivery = new JCheckBox();
  pickUpDelivery.setName("pickUpDelivery");
  recipeDesc = new JTextField("");
  recipeDesc.setName("recipeDesc");
  expressDelivery = new JCheckBox();
  submitButton = new JButton("Submit");
  submitButton.setName("submitButton");
  cancelButton = new JButton("Cancel");
  cancelButton.setName("cancelButton");
}
```

The Model View Presenter (MVP) pattern is an approach to better decouple the view from the rest of the implementation. With the valuable advent of extensive, early testing, MVP also became useful as a way to test a GUI by bypassing its graphical 'skin.'

To build really effective testing tools and GUI technologies with TDD in mind, we wouldn't need systematically to adopt MVP as a workaround for easy testing: unfortunately this is not only the case with current GUI technologies. Decoupling and structuring GUIs is of course beneficial and a valuable best practice, apart perhaps from simple cases, and MVP can be very useful.

One question remains unanswered: how to balance the two types of testing approaches in the most effective way for Java applications?

Testing overhead can be seen as a long-term investment in code. Simple forms of testing can escalate into testing practices that influence the structure of production code heavily, but ultimately what makes things work well is a deep understanding and faith in the approach, rather than a list of 'gotchas' such as 'we need to do extensive testing in our project.'

Getting back to work

We need TTG only for interaction and control, and content and presentation – which rarely needs to be tested, as discussed in Chapter 5. Business domain, data IO and some parts of other layers can be tested with non-GUI practices.

Our team decide to continue using both approaches. Non-GUI testing will be used as much as possible, because it is much more fine-grained, faster to develop and run, and code coverage can be assessed easily. TTG, on the other hand, is needed for testing the whole application and for GUI-only details.

Now that we have tested our simple content, we can write the acceptance test for this simple user interface. To do so, we would like to have an easy-to-use tool, because writing acceptance tests is often done by the customer under the XP approach. For this simple example we can still use JFCUnit. We could arrange for the customer to record the acceptance tests, for example in an XML file, or we can record them in association with our customers.

JFCUnit shows its limits here. Acceptance tests don't look good as JUnit-like Java code, and they look even worse as bloated XML files. We need to resort to another tool, as discussed in Chapter 11 in the GUI test tools section.

Our next move in moving from the GUI to the server is to focus on data. Rich clients are discrete data-driven applications, so we now focus on designing the data structure.

Data second

The next step is creating the data that will back up our content. Here we have a number of choices available: we choose the one that seems the simplest and provides enough feedback. We define the data in a 1:1 fashion from the data we represented in the content layer, then we implement the server support for it, deliberately ignoring any additional behavior or data so far. Our objective is to demonstrate content data moving back and forth between the client and the server.

We want to define a plain Java Bean – sometimes called a Plain Old Java Object, or 'POJO' – that holds content data within the GUI. We can think of it as screen data state (SDS), thus replacing the default Swing models built into the panel so far, or as a data transfer object – it doesn't really make much difference at the moment. We opt for an SDS, part of the content layer. We define its structure by means of a test, and we work on creating the class to make this test pass. Speeding up our iterative development a little, we don't show these steps.

We are provided with the class in Figure 8.12.

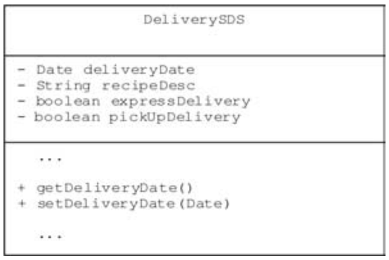


Figure 8.12 Delivery panel data transfer object

It's now time to bind the data to the content. We decide to use the JGoodies data binding framework, which is based on the MVP and Value Model patterns[7] and serves as a basis for the Spring RCP implementation.

Before we even think about the way to implement this binding, we should first focus on expressing it in a unit test. We want to bind the content to a POJO using a support library. Of course, we could write copy methods that copy values into the data holder and vice-versa, or even automate them in a general utility class that uses reflection, for example. We prefer to use this approach to show a more realistic but simple scenario.

No matter how we do it, we need some specialized behavior that performs the binding and some test on it. Let's work first on the route from user to data. This is a tentative test fixture:

```
DeliverySDS sds = new DeliverySDS();
DeliveryPanel dp = new DeliveryPanel();
dp.setDeliveryData(sds);
testWidgetsExists();
```

We can now work on the `setDeliveryData` method in `DeliveryPanel`.

Our idea is to bind the data holder object to the content while honoring the layered architecture in Figure 8.2, which we decided to use as the reference architecture for this example. For DTO, we will use the same SDS class. This separates the two layers (data IO depending upon content). We decide for simplicity to use a plain `Object` instead of the SDS class: this will ease subsequent refactorings, but we now lack compile-time checks, especially when field names are changed.

7. See Chapter 6.

Anyway, we are confident in our tests for this kind of control – that's why we use them so extensively, after all. This results in the following code in the `DeliveryPanel` class:

```
public setDeliveryData(Object data) {
  this.deliveryData = data;
  PropertyAdapter pickupVM =
      new PropertyAdapter(deliveryData, "pickupDelivery", true);
  Bindings.bind(pickUpDelivery, pickupVM);
 ...
}
```

In the previous code we only provided one example of field binding, to keep the code short. The JGoodies binding framework needs an adapter object (which works as a Value Model) for adapting a given Swing widget's model to a generic POJO, the `deliveryData Object`. The `true` value as a parameter passed in the constructor of the `PropertyAdapter` object will make it actively reactive to change events. In this way we have bound the content panel to the related SDS class.

What are the implications of the fact that the data holder now *is* our SDS object? What happened to the default Swing models built in each widget we created with an empty constructor? Here is the implementation of the method `Bindings.bind()` for Boolean values in the JGoodies framework:

```
public static void bind(JCheckBox checkBox, ValueModel valueModel) {
  boolean enabled = checkBox.getModel().isEnabled();
  checkBox.setModel(new ToggleButtonAdapter(valueModel));
  checkBox.setEnabled(enabled);
}
```

That is – via a newly-created Swing model that is specialized for representing values as of the Value Model pattern – data is directly bound to the SDS. We still have Swing models, but they are connected dynamically to our data source. In order for this mechanism to work, though, we need some event-plumbing machinery in the data holder, such as methods for adding and removing listeners and firing events. The simplest way to achieve this is to make `DeliverySDS` extend the `Model` class provided by this data binding library.

Before moving on, one of the developers started discussing the use of the accessor method for setting the data holder. He found it disturbing, arguing that a `setScreenDataState()` method, used to substitute the model, was useless or even dangerous now that such an automatic binding is established, and that only the constructor method should be provided. After a brief discussion they refactored the panel class in order to have only a data constructor. This choice may turn out to be too inflexible, but they decide to provide the data class at construction time only.

The `DeliverySDS` class now serves as a screen data state via the Swing adapters provided by the JGoodies binding framework. We don't yet have a business domain class.

An alternative and simpler solution that avoids using third-party data binding libraries would be manually to synchronize the SDS and the panel, thus making the `DeliverySDS` class a plain POJO extending `Object`.

Commands third

In our use story, after users fill in the order form, they have to submit the order to the server. We need to add commands for this to our GUI. For content, we add the two standard buttons **Submit** and **Cancel**. We use a content factory, part of the content layer of a utility library, that provides standard buttons out of `AbstractActions`:

```
public static JButton[] createButtons(AbstractAction...
  action) {
```

It is responsibility of the interaction and control layers to integrate all the individual parts. In our case we bind the buttons to the related commands[8]. In the interaction and control layer we define the class `SubmitOrderAction` and the class `CancelOrderAction` for the commands.

For further details on the Command pattern, see *Representing user actions with the Command pattern* in Chapter 6.

A first version of the submit command is shown below, skipping TDD tests for brevity:

```
public static class SubmitOrderAction extends CtrAction<DeliveryPanel>
{
  public SubmitOrderAction(){
super("Submit");
  }
  public void actionPerformed(ActionEvent e) {
Application.server().submitOrder(panel.getData());
  }
}
```

This code assumes that we have a class `Application` that centralizes utility access – so far, only the remote access to the server application – and initializes the whole application. We also have the class `CtrAction` that subclassed `AbstractAction`, for generic support of commands that need to operate on

8. The Command design pattern centralizes graphics and control together, which works against our extremely articulated layering scheme. See (Gamma et al. 1994) or (Buschmann et al. 1996).

content objects. For more details of this, see the source code provided with the book.

Clearly, being our first step, the naïve implementation of the **Submit** command shown in the code above is limited:

- Error handling is not defined – what happens, for example, if the server is down?
- The GUI doesn't support asynchronous submissions: the application will completely freeze until the results are available to the client. Despite the fact that we might want to enforce this behavior, because users could not do anything useful anyway in this interval, we always need to provide some basic form of asynchronous behavior for long operations, at least to allow the user to abort the process if desired.
- No feedback is provided to the user: the return value from the remote connection is not used. It is likely that a mere `boolean` value for success/ failure will not be enough to describe a remote operation outcome in the future, but we are working iteratively and we need to keep things simple at this stage!
- There is no basic low-level Control. If we have an asynchronous remote invocation, we need to disable the submit action, thus disabling all the bound widgets, otherwise the user could click the **Submit** button again, potentially invoking the same transaction many times.

Despite the fact that all these issues are important, perhaps the most delicate one regards multithreading. This aspect, specific to Control, needs to be addressed early in the implementation, because it can be costly to upgrade a non-trivial code base with some tens of remote commands or more.

We need a mock object[9] for the proxy server class that simulates latencies and server failures, plus some good tests that will assess whether our code is performing as expected with respect to multithreading.

Defining the tests is not difficult as long as we have a library for unit testing that support time constraint test decorators, provided that we expect the instruction after the command execution to be executed without any substantial delay. The interesting point arises when devising the simplest functional multithreading scheme.

9. Mock objects replace real objects with mock implementations that are used only for ease of testing. Mock objects are widely employed in unit testing, because they help to mask unnecessary factors during testing, helping developers to focus on the specific aspects to be tested.

We now change the natural order of iterative steps followed so far to get to this important issue quickly.

Implementing robust commands

Dealing effectively with time-consuming operations that might have non-predictable completion time, such as a remote request, requires some precautions when working with single-threaded toolkits like Swing and SWT. Here are the main issues:

- Operations that take more than a few seconds to complete should be handled in a separate thread, to maintain the GUI's responsiveness.
- Such threads should be handled judiciously. Most of the time we don't need to fork many new threads at once in a GUI, but simply to keep the main event dispatch thread (EDT) working while one or two other worker threads are performing some specialized task on behalf of the user.

  This seems to be a perfect scenario for the Executor pattern[10]. This technique essentially applies a simple indirection layer to shield the developer from thread execution details. This allows the developer to focus on writing the `Runnable` at hand (that is, the task) while leaving its execution details to a specialized class that can implement an optimized thread pool privately, a simple task queue, or a simple thread fork.
- Tasks have a common lifecycle:
  1. They are forked from the EDT on a separate thread.
  2. At a certain point in their execution they may produce intermediate results. The thread now needs to interact with the rest of the GUI. Because the task's code is currently executing on a thread different than the EDT, it needs to invoke the synchronizing utility methods: Swing's `invoke-Later()` or SWT's `asyncExec()`.
  3. Eventually the task concludes its work, producing a final result. This situation is similar to point 2. The task now interacts with the EDT via synchronous utility methods, `invokeandWait()` for Swing and SWT's `syncExec()` for SWT.
  4. The worker thread is no longer needed. All its resources are freed, and testing should ensure this. The thread might be recycled or garbage-collected, depending on the Executor's policy.
- During the whole of their lifecycle tasks can be interrupted at any moment. A server connection can time out, or a user can change their mind. Before or immediately after halting a task, we should take care to switch the GUI

10. See http://www.cse.buffalo.edu/~crahen/papers/Executor.Pattern.pdf

Control state from 'work in progress' to '[no] work done.' For example, in the delivery panel, when the **Cancel** button is pressed, the **Submit** button becomes re-enabled and the dialog's content becomes active again, as if nothing had ever happened.

This discussion applies equally to Swing and SWT applications.

In general we can think of the possible scenario:

1. A command object is invoked by user interaction.
2. The operation is started in a worker thread separated from the EDT so that the GUI remains responsive. From a GUI design viewpoint, we need to address the 'command in execution' Control state in our GUI.
3. The result is returned to the user, represented as an `OperationResult` object. Such instances are fed to a `VisualDecorator` instance, which notifies to the user of the outcome of the transaction in a meaningful way. The outcome could be nothing – in the case of success, we just dismiss the dialog – or some visual decorations, for example to notify validation errors, or some Control behavior (perhaps because we don't have enough credit in our bank account, so that a **Purchase** button – yet to be implemented – gets disabled). From a GUI design perspective, we are handling the 'command executed' state.
4. The application implementation usually returns to the state just before step 1. This is important from an implementation viewpoint. After a command is concluded, all involved objects should be released apart from notification graphics or other meaningful data. Testing should aim to check for a fully restored situation.

Handling the operation in progress

When the worker thread is ready to go we are left with the task of providing feedback to the user. This can be done in several ways:

- Enforce the 'work in progress' state of the dialog or the whole GUI by changing the mouse cursor and/or the state of the **Cancel** button.
- Disable all the content in the dialog, or put a semi-transparent panel on it, or some other visual hint that signals to the user that the window is 'busy' at the moment[11].
- Provide some simple means of controlling the time-consuming operation. Usually the same button that is used to dismiss a dialog can be used to stop its operation as well, but in this case the operation is interrupted and the

11. This solution applies to communications where the final result is needed to conclude the operation. Suppose you have a text chat application in which, once submitted, you don't need to know what happened to a message: in this case there is no need to freeze the GUI.

dialog is not dismissed. Clicking the **Cancel** button a second time would dispose the dialog[12]. Alternatively, the dismissing button in the dialog, shown in the top-right corner of Figure 8.13, will close the dialog and stop the task, possibly after a confirmation dialog.

Figure 8.13 shows these steps visually, and uses the solution of recursively disabling all the widgets within the content area of the dialog. This solution is visually less appealing that using a semi-transparent overlay panel.

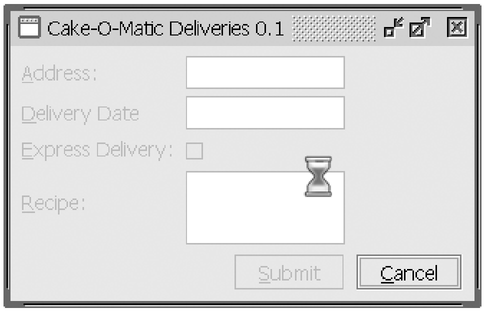


Figure 8.13 Providing visual hints for remote communication

At this point our cake delivery GUI is roughly equivalent to a Web application, in that we have content, a limited form of command execution, and no business logic.

Closing the loop with the server

For the sake of this exercise, we assume that we already have a J2EE server where all domain logic is implemented: all we have to do is to connect to it. We assume that our client will interface with a Session Facade instance – that is, an application of the Facade design pattern for J2EE in which coarse-grained session beans hide server-side fine-grained business objects (Alur, Crupi and Malks 2001), (Marinescu 2002).

The next step would be to represent this behavior in our architecture, providing a stub implementation for the server application.

12. Note that this design slightly overloads the **Cancel** button's semantics, creating a potentially tricky state-dependent interaction (see Chapter 2) in the GUI's design.

8.5 Summary

This chapter discussed some details of the development of form-based rich client applications with Java. We focused on the following issues:

- Common implementation challenges and software design choices for Java rich clients, discussed by means of the functional decomposition proposed in Chapter 1.
- A simple runtime data model that applies well to form-based rich clients.
- The design of an effective validation strategy for rich client applications.
- An example iterative development that abstracted from the particular situation/technology to discuss general and common issues in rich client practical development, such as testing approaches, multithreading management, and so on.

9 Web-Based User Interfaces

This chapter introduces user interfaces delivered within a Web browser using Java technology. This covers a wide array of options, spanning classic server-side markup-based Java technologies such as JSP, JSF, Servlets, and so on, Java applets, and other Web-based GUI technologies. We will also consider technologies for Web GUIs that are not strictly Java, such as Javascript, XMLHttpRequest, and others, because they can be generated by a server-side Java application alone. In general Java can be used in a wide number of different combinations, both on the client and on the server, so covering all the possible alternatives would make the discussion needlessly detailed.

A brief discussion of Web GUI[1] design is provided for consistency with other chapters book that discuss analogous topics for different platforms, such as wireless devices and the desktop.

The chapter is structured as follows:

9.1, An overview of Web user interfaces briefly introduces the main characteristics of Web GUIs.

9.2, GUI design for the Web introduces some considerations for GUI design for Web user interfaces.

9.3, Implementing Web applications with Java provides an overview of the the technical details of implementing Web applications with Java.

9.4, From Web applications to rich clients discusses the common case of Web developers facing the task of building a rich client interface for an existing application that supports a Web client.

9.1 An overview of Web user interfaces

Web-based user interfaces are GUIs executed within a Web browser, usually rendered using a form of markup language such as HTML, XML, or XHTML. The richness and variety of the markup and scripting languages available allow for a great number of choices, ranging from pure HTML to the use of sophisticated

1. This chapter uses the term 'Web GUI' to mean the client-side GUI application that runs within a Web browser, whether it is a Web page, Java applet, or some other technology. 'Desktop GUI' is used to refer to the user interface of applications.

frameworks based on scripting and forms of client-side control, or to code interpreted by separate plug-ins such as Flash or Java applets.

Web applications have a number of distinctive traits:

- The Web user experience[2] is hard to design down to the final pixel on the Web, given the wide differences in display size, connection types, hardware platforms, and software infrastructures of the various possible clients. This forces the designer to adopt a conservative approach and to give up the notion of cross-platform fidelity.
- Due to the nature of Web GUIs, designers have to limit their exploitation of Web technology to enforce fine-grained control, as the amount of control they can enforce is limited. Navigation buttons are always accessible to the user, and it is hard, if not impossible, to impose any form of flow control. Even browser configuration settings can limit the interactivity of a Web site, such as security options that block local storage of information. These shortcomings are an intrinsic part of the Web medium, and have the result that users expect a great deal of control when interacting with a Web application.
- GUI design coherence is usually limited to a single Web site – few GUI design guidelines apply across multiple Web sites. This is the opposite of OS GUI design guidelines, which apply to all applications executing on a client machine. This makes the Web GUI design process more delicate.
- The main reason for the rapid diffusion of Web applications lies in the ubiquitous presence of Web browsers on client machines. Among other things, this provides the ability to update and maintain Web applications without distributing and installing software on clients[3].
- Web applications are usually available only when the client is on line and the server is accessible and working[4]. That is, they cannot operate when disconnected from the server unless using an embedded client-side HTTP server or similar technique. Web clients that implement some form of active control

2. User experience denotes the overall experience perceived by a customer engaged with a product, a service, or some form of communication from a company. Such an experience includes feelings, observations, perceptions, and interactions. This definition aims to bring the concept of customer experience to the digital world.
3. Unfortunately this reason, as economically compelling and practical as it might seem, is inherently technical. It turns out in reality that porting complex applications to the Web makes economic sense, but, given the nature of Web technology when compared with conventional applications, ultimately results in greater difficulty in providing usable and compelling user interfaces.
4. Technologies like Macromedia Flash can be made to operate off line, and, using some form of local caching, it is possible to execute applets or other Web-based content while off line.

can run disconnected for some time, even though their initial deployment always requires download of the relevant page from the remote server.

9.2 GUI design for the Web

Every type of Web application domain has its own type of users and established GUI design idioms. The details of GUI design for the Web would take another book (or more) on its own. The general material in Chapters 2 and 3 applies to Web GUIs as well: this section introduces topics specific to Web GUI design.

Fine graphics details

The potential audience for Web applications is much wider than for desktop application GUIs. Appealing and creative graphical contents are more important than in classic desktop GUIs. When it comes to delivering 'rich' visual experiences, designers often resort to plug-ins. However, despite being a powerful tool in this respect, Java applets are not a popular choice among Web designers. This is essentially due to two reasons:

- The lack of pre-installed JRE support within the most widespread Web browser, Internet Explorer. Applets can also be created with the older Java 1.1, even though sophisticated GUI support is missing.
- The burden of using object-oriented technology for Web designers, graphic artists and other non-programmers.

Nevertheless, there are many examples of well-executed applets on the Web, and the sound OO framework provided by Java is well suited to tackling complex domains and implementing sophisticated user interactions.

Other technologies related to Java are also available for creating Web-based GUIs, such as specialized XML formats and other proprietary technologies.

Unexpected shortcomings

Even though graphic details in a Web GUI can be specified only at an approximate level of definition, in a few cases such low-level details can be handled better by the Web browser than by a desktop application GUI, potentially providing a greater level of control.

For example, aligning widgets along their text baseline provides a pleasant visual effect that enhances usability, as shown in Figure 9.1. Even a basic effect such as this is not available automatically in Swing (as of JSE 1.5), and developers wishing to fine-tune visual appearance must provide this kind of alignment explicitly in their code, consuming time that could have been spent on more business-critical issues.

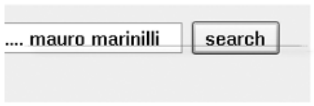


Figure 9.1 Text baseline alignment in a Web GUI

Area organization

The organization of display area in Web applications strongly depends on the type of Web site and its intended audience. *Wireframe prototypes* are non-graphical layouts of a GUI design that are popular for the design of Web pages. Content-rich Web sites need a clear and well-defined organization of contents, layout, and control flow, and specialized prototypes can be used to evaluate such a critical aspect of their design.

Figure 9.2 shows an example of such a technique applied to a generic corporate home page.

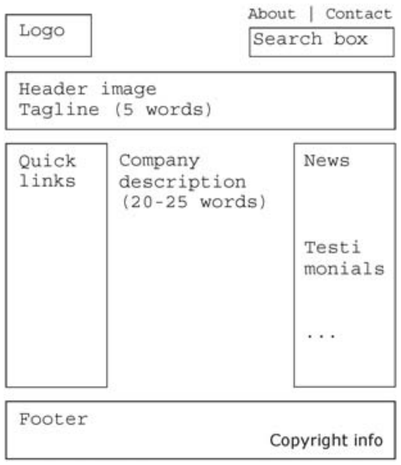


Figure 9.2 An example wire frame prototype for a Web GUI

Levels of client-side control

Current Web GUIs are starting to compete head-to-head with traditional desktop application GUIs in terms of graphics and interactive features, and also their ease of use for developers. This section briefly discusses the current landscape for Web-based GUIs, from a user interface design viewpoint, without going into details of the many available technologies. The technologies considered here work within the client browser, so that they can also be used together with server-side Java technology.

Web developers used to face the challenge of low bandwidth and control-free Web pages, which taken together provided a poor interaction experience for users. This problem is disappearing thanks to better connections – greater bandwidth, increased availability, and decreased cost – and better Web GUI technologies.

Bandwidth and interaction

A peculiar aspect of Web GUIs is the relationship between the available communications bandwidth and the perceived quality of interaction. We know from Chapter 2 that items remain in people's short-term memory for fifteen to thirty seconds at most. For computer interaction, most users tolerate long delays poorly, and become highly frustrated when interactions take more than, say, ten seconds. Connections are increasingly improving in this respect, but consuming bandwidth and time with glitzy graphics and presentations[5] never increased the usability of a Web site. A good means of providing more responsive Web applications lies in the use of client-side control technologies. These also impact on the server tier, in that a lesser number of interactions are required between client and server, allowing server code to be simplified to handle fewer and more specific requests.

More responsive Web GUIs such as this can be built using a number of technologies and approaches, ranging from plug-in-specific code such as Flash, OpenLazlo, or Java applets, to combinations of scripting and other recent technologies widely available in Web browsers.

This type of highly interactive Web GUI should not be confused with rich clients, introduced in the previous chapter. Rich clients are client applications installed locally that don't need a Web browser, can operate off line, and allow for a certain level of integration within the host client machine.

In Web applications where there is little client-side control behavior, the application forwards all the requests to the server, as shown in Figure 9.3. This is the situation with older applications, or where development simplicity was preferred over a more sophisticated GUI.

5. I am a great fan of the 'skip intro' link found in many home pages.

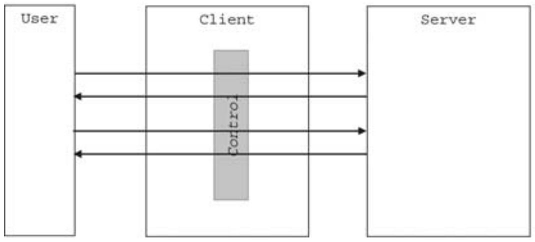


Figure 9.3 Web application with thin control

In Web applications with a control layer, requests are intercepted and processed on the client, and only in some cases are they forwarded to the server, as shown in Figure 9.4.

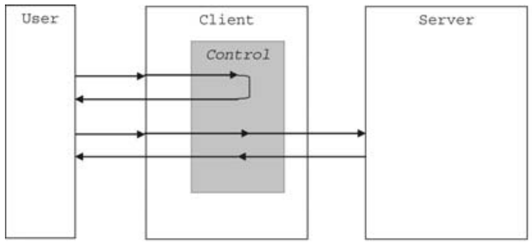


Figure 9.4 Advanced Web application with client-side control

Newer Web technologies such as XHTML, DOM scripting, `XMLHttpRequest`, and JavaScript, not to mention plug-ins such as Flash, and combinations of these technologies[6], can be used together to provide more responsive and interactive Web GUIs. For example, the use of `XMLHttpRequest` object support in all major browsers allows for background server connections without reloading a Web page, thus providing a more fluid and interactive experience than classic plain Web applications.

A substantial client-side control layer allows for some form of business logic to be hosted on the client, even if this incurs all the dangers discussed in Chapter 6, such as duplicating code for handling the same domain-specific representation, both in the client and on the server, establishing some form of control over deployed code for upgrading obsolete business logic, and so on.

6. Various Web technologies used together and wrapped in a convenient framework are often dubbed with acronyms as if they were themselves fully-fledged new technologies, such as AJAX and its variants, or the new client scripting support from Microsoft, code-named ATLAS.

Navigation issues

Because of the many different screens involved in Web GUIs, well-designed navigation becomes essential in providing a usable Web GUI. This involves designing hyperlinks and connections within and between pages in a way that is both usable and that provides an effective means to access required information or perform the task in hand.

It is important to provide clear navigation aids, such as consistent and clear graphics, systematic organization, and so on, to provide support for access to information or to perform some operation via the Web site. A common form of support is to provide feedback about the location of the current page within the Web site. Figure 9.5 shows an example of use of 'breadcrumbs' to provide navigational feedback.

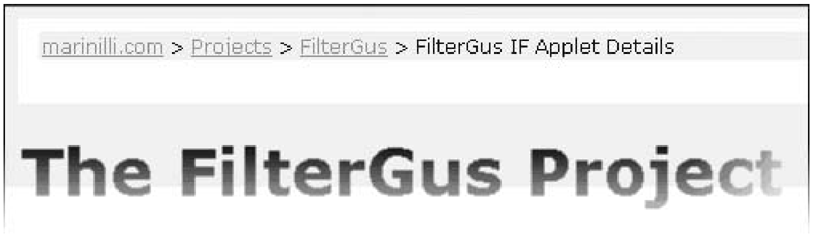

marinilli.com > Projects > FilterGus > FilterGus IF Applet Details
The FilterGus Project

Figure 9.5 Providing feedback of Web site navigation

Basic navigation links, for example back to the home page and other navigation crossroads within the Web site, should be present in consistent locations on every page, to help users avoid dead-end pages, and to take account of users that jumped into the Web site without following a planned navigation path, for example through the use of a search engine. Ideally, all valuable content within a Web site should be one or two clicks away from the home page.

The following subsections provide some examples of various GUI design strategies, organized by the main layout strategy employed. Layout strategies were introduced in Chapter 2 in the section *Display Organization*. Various combinations of such strategies are often used in real-world Web sites. For a more comprehensive discussion about navigation, see for example (Fleming 1998).

High-density information strategy

This strategy is used to provide users with a quick and informative overview of the primary choices available. It is usually employed in home pages only, because it eats up precious screen estate and tends to clutter other content information in the page. An example of such an approach for a fictitious Web site is shown in Figure 9.6.

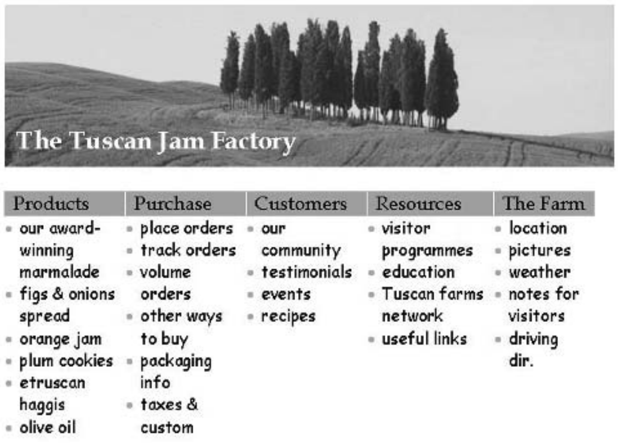


Figure 9.6 Example design of Web site navigation employing a high-density strategy

Another solution is to rely on Web page scrolling – another key difference between Web GUIs over traditional desktop application GUIs – and providing a scrollable navigation area on the left-hand side of the page, as shown in the example in Figure 9.7. Alternative solutions can be used, such as adopting hierarchical menus, or only showing the main categories.

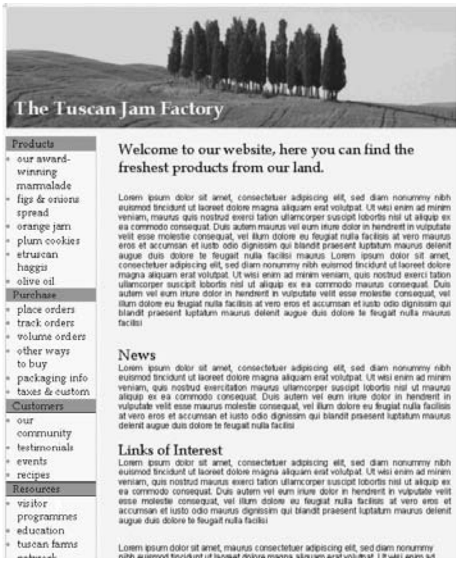


Figure 9.7 Design of Web site navigation arranged vertically employing a high-density strategy

Limited information strategy

A dual strategy for high-density visualization consists of hiding unnecessary navigation information, and is frequently used in real cases. Figure 9.8 shows a navigation menu modeled on traditional desktop application GUI menu bars.

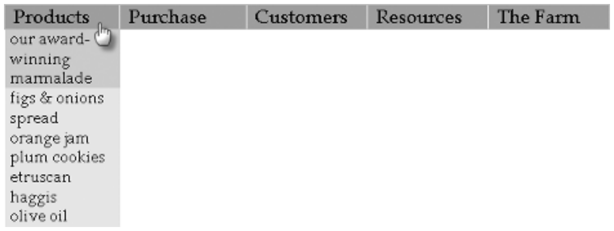


Figure 9.8 Classic application menu-like navigation

When only two levels of navigation are shown, a common design is that shown in Figure 9.9, in which the lower bar changes dynamically depending on the category activated by hovering the mouse in the upper row.

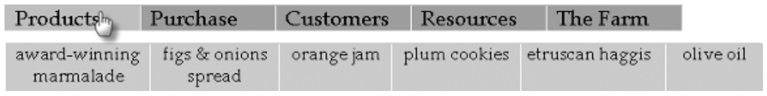


Figure 9.9 Showing navigation items on two levels

Web sites are accessed and used differently than desktop application GUIs, and usability tests are crucial for ensuring that a pleasant-looking abstract idea is really working with target users. From various empirical evaluations conducted on Web GUIs, most users seem to adopt a very aggressive approach to information seeking within a Web page, focusing almost exclusively on their current goal. The very compact navigation design shown in Figure 9.10 certainly saves precious screen estate, but more than likely some of the menus are going to pass unnoticed by many users.

Other possible solutions can range from a combo box containing the most popular link destinations – a sort of a collection of navigation shortcuts – to combinations of graphics and interactive features. The important issue is still the same: devising the best GUI design for the given user audience.

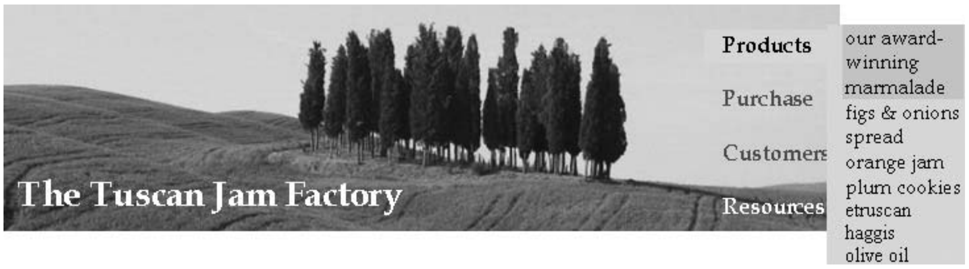


Figure 9.10 Too compact a navigation design

9.3 Implementing Web applications with Java

This section discusses the main technologies available for implementing Web GUIs with Java. Before going into details, it briefly introduces the 'big picture' in current architectures for Web GUIs.

The typical architecture of a Web application

One of the most striking differences in building Web interfaces using Java technology, rather than applications, is the fact that Web pages are defined in non-object-oriented languages. This results in an 'impedance mismatch' between the presentation technology and the remainder of the code that is implemented with Java classes and other languages, similar to that created by relational databases.

Web presentation also impacts GUI interaction, which usually results in GUIs with a lower level of user interaction than for desktop application GUIs, such as lack of features like drag and drop, undo, and so on. In Web GUIs event-based communication is usually needed less, because Web clients usually provide less interactive features for users, which is reflected on the server-side implementation. Furthermore, additional care is devoted to simplifying the generation of the GUI in the markup language of choice, and the content assembly of different areas of the GUI.

A Web GUI typically communicates with a Web server, submitting HTTP requests that the Web server forwards to other parts of the Web application. Such requests contain the session state and the data related to the current request. Although many arrangements are possible, the Web server is usually part of a Web tier in a J2EE architecture, interacting with the business and enterprise information system tier.

Design details of Java Web applications can be found in countless books, such as (Alur, Crupi and Malks 2001) for J2EE patterns, (Fowler et al. 2003) for general enterprise patterns, (Marinescu 2002) for EJB-specific design patterns and (Johnson 2003) for a general introduction to J2EE applications.

Basic Java Web GUI technologies

Java Web technology can be seen historically as a stack of technologies that have grown by accretion over the years – that is, a higher-level layer on top of an existing, less powerful one – in an attempt to provide more powerful features with lower complexity for developers. This technology stack can be roughly described from the bottom as:

- Servlet technology (from the first half of 1997) accepts and processes Web requests using server-side Java code executed in a specialized container application, the servlet container.
- JSP (JavaServer Pages) technology (from the first half of 1999) builds on top of the Servlet technology to provide easier management of dynamic Web pages.
- JSF (JavaServer Faces) technology, whose first specification release was at the end of 2003, builds on JSP technology to provide higher-level specification for user interfaces.

Given its novelty, JSF deserves a brief introduction of its own. JavaServer Faces[7] is a framework for visual components for Web applications that allows the creation of Web GUIs that run on a Java server. The GUI's rendering is left to the browser. Components are rendered separately from their logical definition – different types of table widgets can be used, or the same command component can be rendered as a button or hyperlink as needed, for example.

JSF comprises a Java API and custom tag libraries. The API represents UI components, manages state, handles events, and validates input, as well as supporting internationalization and accessibility options. JSP custom tag libraries are provided for defining visual components within a JSP page, and for binding components to server-side objects. Tag libraries can be created using various disparate Web presentation technologies.

The stack of technologies for Web GUIs for server-side Java is shown in Figure 9.11.

7. Defined in Java Specification Request (JSR) 127.

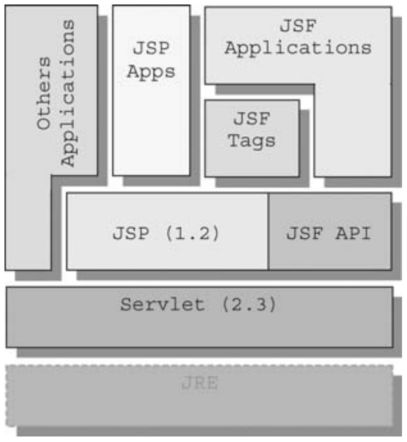


Figure 9.11 Basic technology stack for server-side Java Web GUIs

Java applets

Instead of adopting a markup-based technology to represent primary content within a browser, developers can use Java applets embedded in Web pages and interpreted by a dedicated browser plug-in that launches a JVM to execute them. Two main options are available:

- *Java 1.1 support*. This can be achieved by targeting the older JDK Version 1.1, does not require any additional installation, as JRE 1.1 comes pre-installed on all the main Web browsers, and can optionally take advantage of other libraries for more sophisticated services. Swing can also be loaded as a separate library, and a wide range of third-party libraries and client environments that support JDK 1.1 exist. For a discussion about such third-party technologies, see Chapter 11.
- *Targeting the Java 2 environment*. Writing code for the Java 2 platform makes it easier to achieve sophisticated graphical effects and in general to take advantage of its more powerful and up-to-date software runtime environment. The major drawback of this approach is the need to download and install the JRE plug-in.

Java applets are not straightforward mini-applications written in Java, as Java game developers know only too well. Code written for a general applet container must target a number of different environments – different applet containers built for different browsers all differ slightly from each other, even for different versions of the same browser – and must cope with their quirks uniformly.

Fortunately, Java applets have been available since late 1995, so a wide knowledge base is available to developers. Despite their demise in favor of more domain-specific technologies such as Flash, Java applets are still a viable and competitive choice in many scenarios. There are niches of application domains in which Java applet technology is widely used, such scientific simulation, didactic client-side Web applications in general, small applications, video games, and so on.

9.4 From Web applications to rich clients

The architectural discussion of Web applications mentioned code that resides on the server tier and provides Web content suitable for display on a client Web browser. From a technical viewpoint, Web applications with basic GUI design requirements are simpler to build than desktop applications, because of their more regular structure and the wide availability of mature support frameworks. On the other hand, the sheer range of possibilities available when building a desktop application GUI can confuse developers with a background solely of Web programming.

Accommodating a desktop application GUI as an additional client of an enterprise application poses the following challenges for Java developers whose programming background is mainly in Web technology:

- The intricacies of putting together a desktop GUI go far beyond laying out widgets on a screen, as this book demonstrates.
- Given the current stack of technologies available for building desktop application GUIs, developers are more involved in GUI design details when working on desktop applications than for Web-based ones. This poses a number of critical issues regarding usability, visual design, and others that only developers themselves can solve.
- Developers work in an environment in which the domain model and business logic is already built and working on the application's servers. Some parts of it might be able to be extended to accommodate specific needs of the desktop application GUI, but much of the domain is often given 'as is' to GUI developers. This poses problems if the business domain was weakly modeled on and/or influenced by the Web paradigm, or if details dependent on Web issues leaked into the model itself, such as a page-oriented API for obtaining query results.
- When it is not feasible to separate the Web-oriented user interface aspects fully from the business domain, the simplest solution may be to reuse some of the existing code for the rich client application. This will inevitably tend to create a Web-like GUI that costs as much as a full-blown desktop GUI, as well as being harder to maintain because of the common dependencies with

the Web-specific code. Such an approach, although sometimes unavoidable, can lead to dangerous long-term maintenance scenarios, and impact the quality of the GUI design itself, which can have extensive ramifications on customers and the real value added to the whole application.

Different development habits

Web development and GUI development are slightly different animals, for a number of reasons:

- Development and installation brings a number of issues and technical decisions that Web GUIs don't have.
- User are always ready to judge the results of your work, and expect a more compelling experience from desktop application GUIs than from Web GUIs. Desktop applications are usually preferred over Web clients because of their better user experience, especially in specific areas, such as for repetitive users, or business-critical tasks, so developers must satisfy higher expectations than merely providing a 'Web interface.'
- Rich client platforms, although catching up, are still less refined and usable than their server-side counterparts, and a unifying standard is missing[8], something like EJB on the server side, for example. This confuses developers who are familiar with the Web technology landscape and who often prefer using raw GUI toolkits and few other support libraries even when developing mid-sized projects.

Deeper software design differences also exist when the same design strategies are applied to the two different scenarios. As an example of this, consider the differences between the MVC (Model-View-Controller) design introduced in Chapter 6 for desktop applications, and its corresponding version for the Web.

The classic MVC design for the Web[9] organizes an application into:

- A model with its data representation and business logic.
- A number of views for the model, providing data presentation and user input.
- A controller to dispatch requests and handle control flow.

This design works fine over the Web, where requests to the server comprise a small fraction of the total volume of user interactions in the client GUI. There is thus no need to add the Observer pattern to track changes among the various parts, and hundreds if not thousands of MVC instances may be active at the same

8. See the discussion in Chapter 13 about standard components for rich client applications.
9. See the Struts library or, for a general reference, (Alur, Crupi and Malks 2001).

time, all needing to interact with each other (at least when the Swing toolkit is used).

It is little wonder therefore that Web developers feel a bit lost when building complex desktop GUIs, and resort to vague concepts like the 'need to centralize controllers' or to set up some form of central command management. Despite being called by the same name, client-side and server-side MVCs are very different when it comes to their details.

9.5 Summary

This chapter discussed the scenario for Web GUIs using Java technology, covering some aspects of Web GUI design in relation to the general concepts introduced in Chapter 2.

We covered the implementation aspects of Java Web GUIs and the architecture of Java Web applications. Differences between desktop and Web GUIs were highlighted for the common case of adding a rich client to an existing server application. We also mentioned differences between client and server applications in the implementation of some common design strategies, such as MVC.

10 J2ME User Interfaces

This chapter introduces the user interfaces supported by the Java 2 Micro Edition (J2ME or JME) edition. Java platforms other than J2EE/J2SE, such as the JavaCard environment, which has no explicit GUI support, won't be covered. The practical examples in this chapter are based on the J2ME Mobile Information Device Profile (MIDP). However, most of the material in this chapter, whenever not explicitly expressed otherwise, applies to J2ME GUIs in general, not only MIDP GUIs.

J2ME is introduced briefly with a technical introduction of this programming environment, followed by some details about GUI design for this profile. Practical examples are given. A questionnaire for assessing the usability of J2ME applications can be found in Appendix B.

This chapter is structured as follows:

10.1, Introduction to the MID profile briefly introduces the J2ME MID profile.

10.2, The MIDP UI API introduces the details of the API for MIDP GUIs.

10.3, Designing MIDP GUIs provides an overview of GUI design for MIDP GUIs.

10.4, Designing navigation discusses the specific of navigation in a MIDP GUI.

10.5, An example custom item discusses the customization of a MIDP GUI component by means of a practical example.

10.6, An example ad-hoc item shows an example an ad-hoc item component for representing numeric data using pie charts.

10.7, An example application introduces the Park MIDP application, illustrating a GUI design and development approach to navigation.

10.1 Introduction to the MID profile

J2ME is targeted at embedded and consumer electronics devices. It has two primary types of component – *configurations* and *profiles*. The J2ME architecture is composed of a few configurations that define the common features for a class of devices. Two configurations are currently available:

- The Connected Limited Device Configuration (CLDC), designed for devices with constrained hardware resources. Such devices typically run on either a

16- or 32-bit CPU and have 512 Kilobytes or less of memory available for client applications and the Java platform itself.

- The Connected Device Configuration (CDC), aimed at next-generation devices with more robust resources than CLDC devices.

These configurations dictate the Java virtual machine, core libraries and some APIs, while leaving the differences between each device to be described by a profile. User interfaces are defined on a per-profile basis, allowing for maximum flexibility when taking advantage of device characteristics. Tailoring APIs to a particular profile allows for efficiency and accuracy, but results in several different class packages and slightly different vendor-specific API implementations.

This chapter concentrates on the MID profile, part of the CLDC configuration. The MID profile is aimed at modeling the large category of Java-enabled wireless handheld devices. Such profiles describe all issues, such as the user interface, the application model, networking, and persistence storage, that are related to Java-enabled mobile devices like two-way pagers and cellular phones. We focus on the UI API here.

An application running on this type of Java-enabled devices is referred to as a *MIDlet*, because it can be deployed seamlessly on a wide range of different MIDP-compliant devices, just as an applet is deployed in different Web browsers. This capability is one of the most important features of the J2ME initiative for wireless devices, and has the potential to create a completely new and huge market for such software applications. The MID profile has been designed both to abstract applications from the client hardware on which they run, and to ease the development of similar applications. The latter aspect arises when we discuss the UI API, which has been modeled around the typical UI seen in today's consumer cellular phones.

Both the terms *applet* and *application* are used to refer to MIDP programs.

For more information about the J2ME platform, visit:

`http://www.javasoft.com/products/j2me/`

The code suggested here was developed and executed using various development tools. The J2ME Wireless Toolkit from Sun, for MIDP 2.0, is available at `http://www.javasoft.com/products/j2mewtoolkit/`.

Main UI concepts

The J2ME MIDP GUI API has been designed for generic handheld devices with LCD screens of various sizes, a typical minimum being 96 x 96 pixels, and running on hardware with limited resources. The richness of concepts and software architectures employed in the AWT and Swing APIs in the desktop Java world is clearly out of reach here, even if the MID profile assumes quite powerful hardware – at least when compared with other embedded devices or the JavaCard specification.

The basic functionalities provided include the capability of manipulating the device's screen to show a top-level component, a widget that occupies the whole screen, that has been decorated previously with simpler UI components, referred to as *items*.

There is always only one screen object active at a time, representing the whole contents of the current device's display: the concept of multiple windows is absent. Applications simply switch from one screen to another.

No navigation semantics have been provided, both for generality and because applications are expected to be simple and not require many different screens and menus. One common navigation semantic seen on such devices, for example, is stack-like screen navigation, in which users find their way from one screen to another, closing the current one and redisplaying the previous one as screens are 'piled' to resemble a hierarchical organization. Navigation styles are left to the application developer's implementation.

Only basic widgets and a simple user input framework are provided. The Java classes provided are designed to reduce the need for subclassing by providing a comprehensive range of built-in options.

It is important to note that, given the nature of the platform, developers must always query the current screen size for all but the simplest UI screens: assuming a fixed screen size can produce unusable UIs on MIDP-enable devices.

The lifecycle of a MIDlet

The lifecycle of a MIDlet is important, because it involves the concept of screen management directly. The native module that handles MIDlets in the MIDP-enabled device is called the *application manager*.

A MIDlet can be in one of the following three states:

- *Paused*. The MIDlet is shallowly initialized – that is, it does not use or hold any resources. The instance is quiescent in device RAM and is waiting for the application manager to be activated. This state is reached every time the application management invokes the `pauseApp` method on the given MIDlet, not only after its creation. The MIDlet screen is removed from the device display.
- *Active*. The application manager has activated the MIDlet by invoking its `startApp` method. The MIDlet must explicitly assign the screen device.
- *Destroyed*. This state is entered only once, and instructs the MIDlet to release all its resources and terminate its execution.

All the UI initialization is performed in the `startApp` method.

User input management

User input is handled both at high level, using `Command` objects, and at low level using the device keys directly. Although the latter solution is available, it is discouraged, because it can hinder the portability of an application.

The `Canvas` class provides a general abstraction for input keys that is portable across all implementations, encompassing all the keys in the ITU-T standard telephone keypad – numbers from 0 to 9, the '*' and '#' keys – as well as a set of abstract input actions called *game actions* that include the four navigation keys (up, down, left, right), a 'fire' button, and four application-dependent keys.

Commands are the most important means of expressing user directions. Commands are managed by the underlying MIDP implementation, and can be rendered as soft buttons – text labels shown above special buttons near the device screen – voice commands, or in any other platform-specific way. To help the MIDP implementation interpret a given command correctly, commands have different types, each with a precise use:

- *Back*, used to return to the previous *logical* screen.
- *Cancel*, which cancels the current screen and all data previously set in it. Cancel is the standard negation command.
- *Screen*, reserved for application-specific commands related to the current screen.
- *OK*, the standard affirmative command.
- *Help*, used to show display content.
- *Item*, associated with a particular item on the current screen.
- *Stop*, which stops the current operation. This should be implemented, to allow users to stop lengthy operations.
- *Exit*, used for quitting the whole application.

The command type is used only as a rendering hint for the MIDP implementation: developers should always specify the corresponding action in their code by implementing the `CommandListener` interface. This is demonstrated in the example code provided for this chapter.

Two levels of API

While J2ME's UI API is oriented towards easing development in most common situations, some hooks have been left for implementing ad-hoc UIs as well. This allows developers to subclass low-level general classes such as `Canvas` and `Graphics`. This is however a complex procedure, and one that may ultimately produce non-portable MIDlets, for example by relying on a key present on a specific mobile phone model but not on other MIDP-compliant devices. Nevertheless, in some situations this is the only way to go.

Figure 10.1 on page 381 shows the relationship of the `Canvas` class to other classes of the package.

Main UI limitations

The J2ME API has several limitations, mostly dictated by hardware resource availability. First, whenever possible UI implementation details are left to the device vendor, to simplify the implementation of the MIDP for a given platform. This encourages implementation differences between one device and another, not only in the UI's look and feel. Developers are urged to consider the API as a high-level and not completely accurate specification. Furthermore, the absence of any guidelines for navigation semantics encourages different approaches that could confuse the user when moving from one application to another.

These and other similar considerations highlight the API 'shortcomings when used in non-trivial UIs – and such applications will be growing in number with the trend towards more powerful devices in this industry sector.

Cost-driven design for J2ME GUIs

Cost-driven design can also be applied to J2ME GUIs, as for all the general technique discussed so far, from iterative GUI development to Agile methodologies, test-driven development and the design patterns and architectures discussed in the previous chapters. The devil, as usual, is in the details. Aiming for professional user interfaces on constrained devices should involve a constant focus on usability rather than other, secondary, issues. This is not often the case: with tight deadlines and tough technical challenges to cope with, usability concerns often slip away.

10.2 The MIDP UI API

This section describes the practical UI component classes provided in the `javax.microediton.lcdui` package using a top-down UI design–oriented approach, rather than illustrating low-level API details. These can be found in the related literature[1].

UI widgets

This section describes the top-level components of the MIDP UI library.

Table 10.1 shows the built-in components provided for developing MIDP applets.

1. See http://java.sun.com/products/midp/.

Table 10.1 MIDP UI Top-Level Components

Top-level component name	Description
`Alert`	Similar to a dialog box for showing read-only messages., composed of simple *items* (see Table 10.2).
`Form`	Shows a collection of *Items*.
`List`	Shows a list of homogeneous, selectable elements .
`TextBox`	Similar to a TextArea for editing multiline text.

In Table 10.1 the first two top-level containers, the `Alert` and `Form` classes, are visual containers of simpler UI widgets called *items*, which all extend the `Item` abstract class. The remaining two, `List` and `TextBox`, are specialized components that are designed to fill the device screen.

The standard items provided with the MID profile are listed in Table 10.2.

Table 10.2 MIDP UI Items

Component name	Description
`Label`	Shows a single line of read-only text.
`DateField`	Shows a calendar or other device-dependent date / time picker.
`ChoiceGroup`	Shows a set of boolean values.
`ImageItem`	Shows an image.
`StringItem`	Shows some text.
`TextField`	Shows some formatted text.

When creating data input screens or other GUIs, developers use a `Form` instance containing properly initialized `Items`. Menus are meant to be implemented through `List` instances, while `Alerts` are used for notification only. `TextBoxes` are used for displaying long text strings, such as SMS text messages, memos, and so on.

`TextField` and `TextBox` components provide built-in data input constraints. Developers can specify which of the following constraints the component will enforce when handling user input:

- *URL format* – only Web-compliant addresses will be accepted.
- *E-mail format* – only e-mail address will be accepted.
- *Phone number format*. This is implementation-dependent due to regional conventions for phone number formatting, network requirements – the GSM network, for example, may use '+' at the beginning of every number – and device implementations. Once a phone number format field is filled out, a device-dependent key can start a telephone call on a cellphone host device.
- *Integer value format*. In this case only digits can be entered, optionally prepended with a minus sign. Range constraints can be enforced by developers.
- *Password field format*. Inserted characters are masked as they are typed.
- *Free text*. Any character can be entered.

Other constraint types can be combined with the password constraint to create, for example, a numeric-only, password-like text field.

Figure 10.1 shows the basic static class diagram of the major classes in the `javax.microediton.lcdui` package: some classes, such as `Font` or `Graphics`, are omitted.

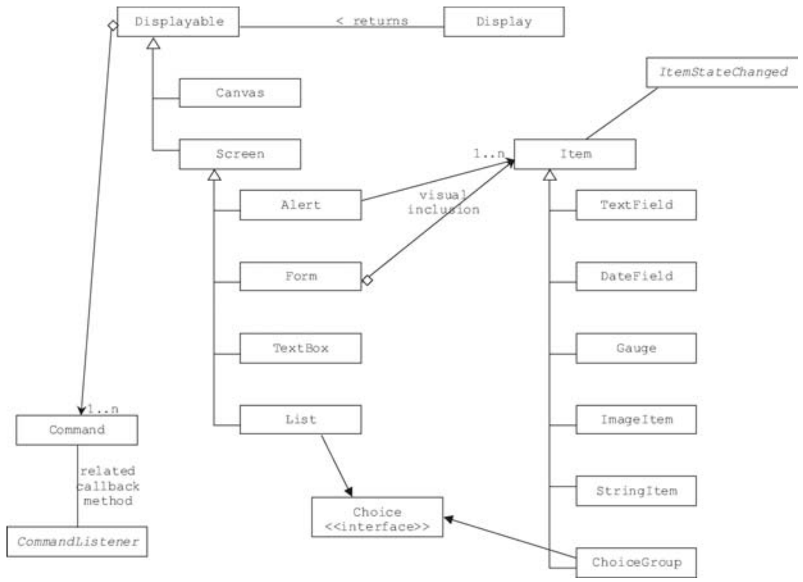


Figure 10.1 Simplified class diagram of the `lcdui` *package*

10.3 Designing MIDP GUIs

Today's J2ME GUIs range from basic cell phones to sophisticated personal assistants, and new devices are broadening the already wide choice constantly, such as WiFi-enabled 'smart' cell phones that can also be used as much cheaper, standard devices for a variety of computational tasks, ranging from work to personal entertainment.

The main difficulty in designing GUIs for such a diverse set of devices arises from the fact that J2ME technology provides only an approximate definition of the GUI's final details. Such details are rendered autonomously by the device on which the application is hosted. This point is discussed in more detail later.

The main characteristics of a MIDP application that impact on its design, both the GUI design and its implementation, are as follows:

- *Personal devices*. Differently than other computing means, wireless devices are inherently personal devices, and as such they are used differently than desktop PCs or other similar computing machines. Wireless devices are carried with the owners throughout the day, and applets can be used at any hour of the day or night.
- *Privacy concerns*. As an important detail of the previous point, users feel uneasy in allowing foreign code to execute on devices that contain as much private data as does a cellphone. This also applies to applications sharing data externally. Even if J2ME poses important limits on applet intrusiveness, users should not be expected to have to know the MIDP specification. GUI designers need to consider this aspect when designing privacy-sensitive applications.
- *Type of users*. Differently than desktop applications, mobile applications have a much wider range of possible user types. Households, retired people and teenagers can all be potential mobile users. Their education and levels of computer literacy can vary greatly, requiring more care than for desktop applications in the choice of language used in the application, and in general in the whole GUI design. Outside North America cellphones are more widespread than desktop computers.
- *Limited bandwidth and intermittent connection*. A wireless device typically has much less bandwidth available for transmitting and receiving data than a wired device. Furthermore, wireless connections are typically unreliable, so an intermittent connection should always be assumed.
- *Pay-per-use billing schemes on bandwidth/connection*. Most carrier operators charge for bandwidth consumption on a per-use basis. This is an additional psychological factor in shaping use patterns, because users might be uneasy about allowing applets to connect remotely.

- *Power consumption and related use patterns.* A wireless device is usually a mobile device as well, with batteries as its only means of power supply while on the move. Even the longest-lasting batteries offer a limited amount of power. This dictates use patterns for such devices that GUI designers must take into account.
- *Limited hardware resources on client devices*. Because of their mobile nature, the available power source, and sometimes also for economical reasons as low-end consumer devices – wireless devices have limited resources. The same reasons limit processing power.
- *Restricted input means*. Limited keyboards and compact pointing devices are available on only a small segment of devices. On others input is obtained via keypad and navigation keys.
- *Context of use*. Wireless devices can be used in the most diverse surroundings, such as in school classes, on a train, in a café. This obliges designers explicitly to study sets of use contexts for applications. Furthermore, the context can be an important input for the application, such as those making use of localization services such as the Global Positioning System, GPS[2].
- *Intermittently active sessions*. When interacting with mobile applications, users can be interrupted at any moment, either by an incoming call on the same device, or by some situation in the environment. This means that MIDP applets should be able to chunk both user attention and transactions in small quanta, and need to provide simple mechanisms for restoring session data.
- *Limited GUI output screen*. Typical displays are very small compared to other devices. This makes viewing more difficult. Combining this limitation with some classes of users, such as those with visual impairment, might transform a GUI design that seems brilliant when run in an emulator into a totally unusable and frustrating experience for some users.
- *Fragmented market for client devices*. This can result in a number of small incompatibilities in the way the J2ME MIDP specifications are implemented among different vendors, and even among different models of end device. Fortunately the industry has provided a thorough set of test criteria for a mobile device to be certified as 'Java compliant.' Vendors also tend to add proprietary APIs that are not widely portable, so that developers are often faced with choices about whether to restrict portability or simplify development[3].

2. Not to be confused with GPRS (General Packet Radio Service) a mobile telephone network standard that can be used by wireless J2ME devices for remote connection.
3. This is especially true for those market segments in which Java applets follow consumer-led paths – new devices are continuously released with a short time to market, shrunk budgets, and quick obsolescence.

- *GUI details are ultimately left to the actual client device implementation*. GUI design is ultimately dictated by the device that is executing the applet, and there might be discrepancies between different, supposedly compatible, devices. While this was a major problem with early versions of embedded JVMs, it is still a hindrance for developers.

General advice about multi-presentation applications was given in Chapter 9 in connection with Web GUIs, so won't be repeated here.

Abstract GUI designs

Accommodating a professional GUI design in a lowest-denominator platform like the J2ME MIDP is challenging. It should always involve thorough testing on the commonest target devices available on the market, not just on their software emulators used for early testing and development.

A simple solution to this challenge is to choose a target platform explicitly, usually from one of the major vendors for the target user population that has a wide choice of development resources, such as documentation, emulators, GUI design guidelines, and so on. This situation is similar to that of designing for the Web in the 'old days' when there was no single market-leading Web browser.

An incorrect solution to the problem is to try to bypass the JME (or J2ME) specification by rolling out a home-grown look and feel in an attempt to provide a consistent 'branded' user experience across various devices[4]. This usually results in a poor, possibly weird-looking, user interface that is expensive to build: low-level details need to be handled explicitly and cannot be left to the underlying device's implementation.

In cases in which the lowest common denominator is too problematic a solution to pursue – and only in these extreme cases – a better strategy is to segment the design in a divide-and-rule fashion. This is discussed in the next section.

Segmenting the GUI design

Some scenarios are clear-cut and allow two main segments to be easily identified. Consider for example a traffic congestion applet being developed for a major city traffic authority. The target user population is identified by means of preliminary questionnaires, and is roughly divided into two groups:

- Those that will access the application from the Web.
- Those that will use it from a Java-powered consumer wireless device.

4. Possible because MIDP 2.0 allows more low-level GUI details to be specified. This is especially true for those market segments in which Java applets follow consumer-led paths – new devices are continuously released with a short time to market, shrunk budgets, and quick obsolescence.

The latter group is better served by MIDP applets, because the user population will be made mostly of repetitive users that prefer to download the applet only once, instead of using other more expensive solutions. A WAP-based GUI, for example, would require city maps to be downloaded for every session, while with a rich client, only current traffic congestion data is needed.

Building a single GUI for these two group of users can prove tricky, as it is a situation in which designers cannot transfer complexity to end users, and one where usability is an essential requirement. Such a GUI would have to serve two distinct needs at once: that of satisfying both power-users and normal drivers, groups that have very different information needs. Splitting the design serves both segments better, greatly enhancing the usability of the overall application.

Two types of GUIs can therefore be designed:

- Using a high-density visual strategy. This version is aimed at expert users that need more data and a richer interaction, such as people that spend most of the day driving in the city – taxi drivers, delivery drivers, and so on.
- Leveraging a limited information style[5]. This type of user would prefer a mainly textual application, where details are limited and only basic congestion information is provided. This version will accommodate most users, so cheaper phone models can be used, leaving the 'power-user' version to deal with more powerful devices with larger screens.

For such an approach to be viable, however, requires thoughtful implementation, otherwise it can escalate into an expensive and risky development situation. Portions of code not common to the two GUI versions should be minimized, by providing a rich set of utilities and a supporting architecture implemented in a GUI-neutral way that factors out all commonalities among different GUI implementations. The objective of such a design is to minimize GUI-dependent code and maximize GUI-independent code. Building and deploying the two versions can be completely automated and a few classes can assemble the building blocks to ship the different versions of the same application that are geared towards different user segments. For larger application scenarios, this strategy leads to the Software Families software engineering approach.

10.4 Designing navigation

MIDP GUIs have to cope with small physical screens. One of the main consequences of this constraint is that GUIs will have to have more, and smaller, screens. Navigating between such screens therefore becomes all-important for the usability of all but the simplest applet.

5. See Chapter 2.

Finite State Automaton (FSA) is a formal model of computation for modeling UIs on simple devices such as wireless phones. FSA, which can also be represented as a finite state machine, consists of a set of states, with a special start state, an input alphabet, which defines the type of input to the FSA, and a transition function that maps input symbols and current state to the next state.

Given the simplicity of MIDP GUIs, the user interface structure can be expressed by means of simple diagrams like that shown in Figure 10.2.

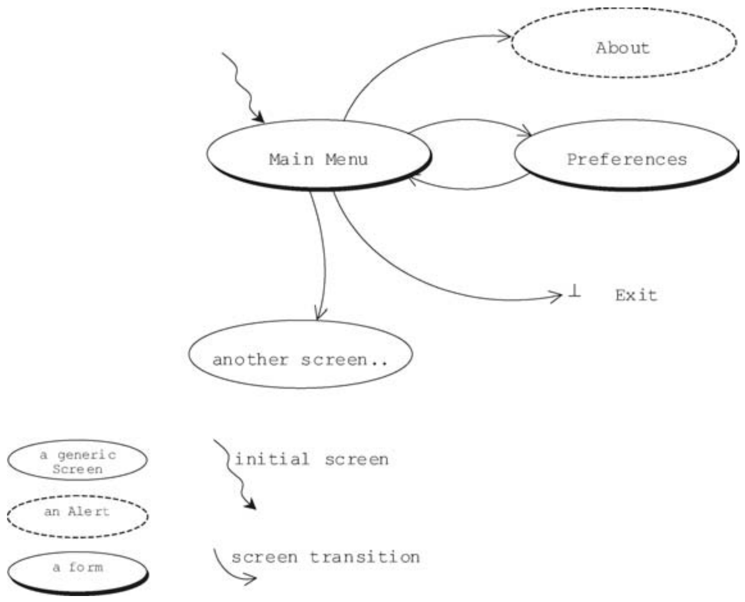


Figure 10.2 The structure of a typical MIDP GUI

The diagram describes the possible transitions among different MIDP screens in a typical mobile application. A later section shows an example of use of this diagram for the design of a simple GUI.

GUI design needs to match the limited client resources of J2ME clients – memory, processing power, screen real estate, and so on. From a usability perspective, this involves placing a lower cognitive burden on users. The widely-used strategy of localizing feedback[6], for example, becomes increasingly difficult to enforce on devices with limited screen space. Interaction design can then degenerate into a sort of long wizard with tiny pages. Consider data input in one screen that affects

6. This was discussed in connection with validation in Chapter 8.

data in another screen, for example disabling some option. With limited screen estate it might not be possible to keep these related items close, presenting a puzzling experience to the user.

10.5 An example custom item

We are now ready to see some practical examples of J2ME MIDP applications. In contrast to desktop computers and other rich computing appliances, wireless devices have a limited set of features, and this influences their user interfaces. A common error when developing MIDP GUIs is to try and achieve a cross-device look and feel – that is, a look and feel that is the same on all supported platforms. This might initially seem highly desirable, because it is supposed to help users, while giving a strong brand identification to the product. However, such efforts sometimes end with incomplete, arbitrary GUIs that can confuse end users. Users become accustomed to a mobile device's look and feel, and may be uncomfortable with a downloaded applet that behaves 'weirdly.' Figure 10.3 shows a sample custom alert box that illustrates this[7].

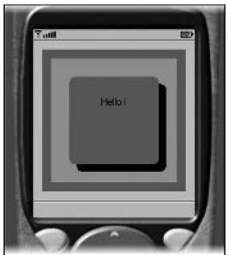


Figure 10.3 An ad-hoc alert box developed for a specific task

It is interesting to see how easy is to manipulate the display area directly. In the MID profile there is no deep and complex class hierarchy like Swing's, and taking advantage of the `paint()` method is straightforward. The custom class that creates the alert box shown in Figure 10.3, whose source code is provided in the code bundle for this chapter, extends `Canvas` and draws directly on the display's `Graphics`.

7. Such a custom component is an extension to a standard visual component provided by a reference toolkit, as discussed in Chapter 3.

10.6 An example ad-hoc item

Providing ad-hoc GUI solutions is a powerful feature of J2ME. It needs to be used with care, but can enhances an application's usability enormously.

This simple example of the correct use of ad-hoc components – from a GUI design viewpoint – takes advantage of a custom item component. The `Item` class represents the generic component that can be used in lists and forms. With MIDP 2.0 the `CustomItem` class is available for subclassing, which provides a simple way to create ad-hoc items.

Figure 10.4 shows a simple custom item for representing numeric data using pie charts. It consists of a form composed of custom items on the left, and the legend in the right-hand figure.

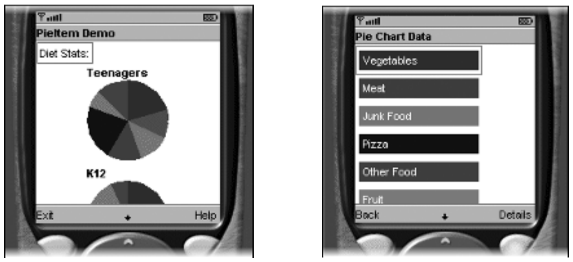


Figure 10.4 A form made of pie charts

Each element is itself a form item, so it can be manipulated with the same conventions, such as check boxes, text fields, and other standard items. This is a powerful way to employ ad-hoc designs without disrupting native platform usability. Figure 10.5 shows how users can move through items in the applet using the cell phone's navigational keys.

The implementation is organized around three classes[8]:

- The `PieItemTest` class, needed to launch the demo.
- The `PieItem` class, which implements the pie chart custom item.
- The `PieData` class, which represents all configuration data, such as colors.

The related classes are provided in the code bundle for this chapter. The `PieItemTest` class is a test MIDlet that creates a screen composed of custom pie chart items. The core of the example is the `PieItem` class. The `values` array stores

8. This is a demo implementation that has been developed only to show the visual component customization features of standard J2ME MIDP widgets.

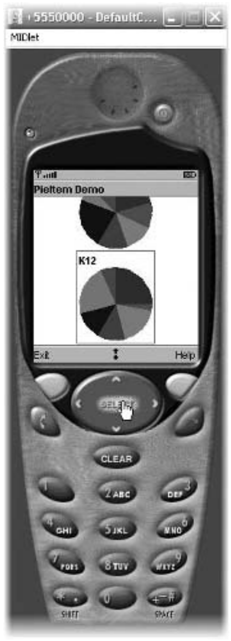


Figure 10.5 Navigating through specialized items

the data related to the current item. Whenever a non-empty string label is provided in the constructor, this is used as the item's caption (see Figure 10.4 and Figure 10.5).

The methods `getMinContentHeight()`, `getPrefContentHeight()` and `getPref-ContentWidth()` are needed by the `CustomItem` class. The `paint()` method draws the data as a pie chart.

The `PieData` class gathers all configuration data relevant to pie charts. It provides the chart legend shown in Figure 10.4. Such a screen itself employs another custom item implementation, an inner class of the `PieData` class. The `PieData` class manages configuration data for all pie chart items.

10.7 An example application

This section describes an example application that illustrates a simple mechanism for implementing Finite State Automaton (FSA).

The example application manages the billing of car parking in which users pay for parking via an applet. As only its user interface is of interest, the applet's other details are only sketched. For simplicity it use only few screens, as shown in Figure 10.6.

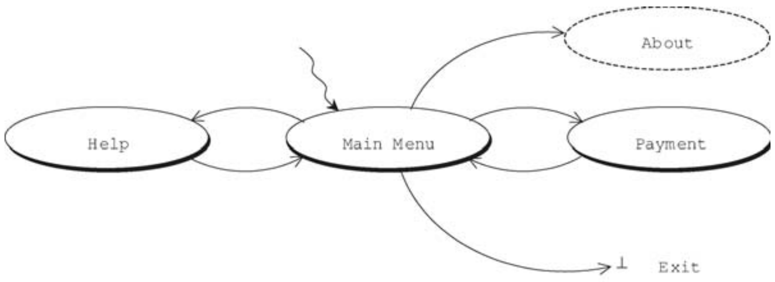


Figure 10.6 The Park applet's GUI structure

Figure 10.7 shows the main menu for the applet on the left, and the 'About' box on the right.

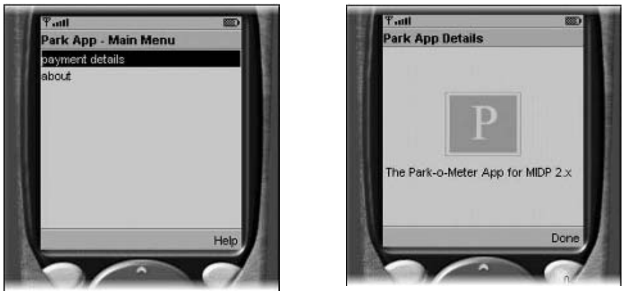


Figure 10.7 The Park applet's main menu

The most interesting screen in this applet is the Payment Details screen, which persists the user data from session to session. In the demo applet it is implemented very naïvely, as shown in Figure 10.7.

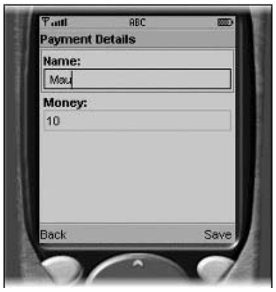


Figure 10.8 The payment details form

However, we are interested more in implementation solutions than the applet's realism.

The code

The MIDlet subclass (the `ParkMain` class) manages the overall UI, the user commands, and the transitions between screens. In the `ParkMain` class the `commandAction()` method handles all the applet's command management. Most of the code implements the transition diagram shown in Figure 10.6 – for example, when the user issues the `HELP` command from the main menu screen, the help screen is shown.

The `ParkMain` class implements the Explicit Navigation design strategy discussed in Chapter 6.

We diverged from the simplest implementation to deal with main menu commands more efficiently. More object-oriented mechanisms are possible, such as using specialized `CommandItem` events, but minimizing implementation complexity is a key objective when writing J2ME applications. The techniques shown here try to minimize the number of employed classes and objects – that is, static classes and their runtime instances – while maintaining a simple software design by minimizing the number of classes.

Following this approach, all the application's screens are gathered in the `Screens` class. This class is invoked by the `ParkMain` instance whenever a screen is needed. The `Screens` class lazily creates the required displays. Consider the help screen, for example. Such a screen is only needed a few times, and experienced users might never invoke it. Keeping it `null` until it is needed saves runtime space and initialization time. Lazy instantiation, used here for screens, which are created only when needed, is a key technique in the implementation of MIDP UIs.

There are cases in which keeping a reference to a screen that has already been created is counterproductive. This may happen for example when the screen needs to be created anew each time, or when it is accessed only once per session. This latter case is exemplified by the `About` screen, displayed by the `showAbout()` method. In this case it would be a waste of space to keep a reference to the screen throughout the whole life of the applet (assuming of course that the 'About' screen contains only standard product information, rather than data that needs to be frequently referenced). Some J2ME applications can run for weeks, so careful memory management is essential.

Separating screens from control code may be beneficial for non-trivial applets, in that it separates presentation from control and keeps the implementation organized coherently, even if it might favor closer coupling among classes. In this example, the MIDlet and the `Screens` instances are tightly coupled.

Finally, the AppData class contains all the business data required by the application. In the trivial implementation used here, AppData uses has only two attributes:

- The user's amount of parking credit – the money attribute.
- The user's name.

This class is also responsible for retrieving and saving data persistently, by means of the J2ME MIDP RecordStore mechanism, which is properly initialized in its constructor.

10.8 Summary

This chapter has discussed factors relevant to graphical user interfaces in Java 2 Micro Edition briefly, demonstrating the built-in support for GUIs in the J2ME MID profile. It also discussed some simple demonstration examples of the use of such libraries, together with some high-level strategies for organizing the user interface of a MIDP applet GUI.

11 Java Tools and Technologies

This chapter discusses an aspect that is critical for Java application development, and one that is often overlooked – the right mix of ingredient tools and technologies for a project. It focuses mostly on GUI development and is slightly biased towards open source software (OSS) over commercial products. It deals with Java GUI development tools and technologies only.

After introducing the practice of tool selection for Java technology and covering some aspect of OSS, we will focus on perhaps the most crucial, and often irreversible, choice in Java GUI tool selection: whether to opt for Swing or SWT. After discussing the various issues related to these two toolkits in detail, other tools and technologies available to Java GUI developers are outlined.

This chapter is structured as follows:

11.1, Introduction to tool selection discusses the general issues involved in selecting ingredient technologies and tools for building a Java GUI.

11.2, Evaluating open source software illustrates various aspects of OSS technology evaluation in more detail, introducing the OSS maturity model.

11.3, SWT or Swing? is dedicated to the differences in the two foundational technologies for Java desktop GUIs.

11.4, Other GUI technologies discusses some alternative technologies to SWT and Swing for Java GUIs.

11.5, Utility libraries lists various (mostly OSS) GUI utility libraries, including development, security and deployment tools, sets of specialized components, and utility libraries such as JGoodies, Glazed Lists and others.

11.6, Test tools discusses some GUI testing tools for Java.

11.7, Profiling tools illustrates some profiler tools for Java GUIs.

11.8, GUI builders discusses some visual editors for content assembly that are available for Java developers.

11.9, Presentation layer technologies demonstrates some Swing look and feels and presentation technologies for SWT.

11.10, Declarative GUIs with Java discusses the various alternatives for specifying GUIs declaratively for Java-based applications.

11.1 Introduction to tool selection

Literally hundreds of Java libraries, plug-ins, and tools for Java development are available on the market, with either OSS or commercial licenses. Not taking advantage of such a bounty would be a pity, given the maturity of many of these products, which have been built over years, and the added value they provide, often as OSS. Perhaps the strongest aspect of Java technology as a whole lies in its community of developers and its orientation towards open source and free collaboration: the OSS offering is just the final by-product of this active, open, collaborative climate.

Such an abundance of products poses problems over the best mix of libraries and tools. Many situations are possible, ranging from development teams tactically choosing libraries and tools for a single project, to a company selecting tools and libraries for a long-term strategic investment in personnel training and large-scale adoption for multiple projects.

Assuming that you have a development environment already set up, which hopefully provides modern comforts such as code editors, unit testing, refactoring tools, concurrent versioning, and continuous integration, to develop a decent GUI the following ingredients are usually required:

- Support libraries – specific layout managers, look and feel, favorite logging facility, XML parsing, and so on.
- A GUI testing tool of your choice. Such a tool would also be used for acceptance testing.
- Optionally, a visual GUI builder.
- A set of development tools specific to client application development, such as deployment support (either using JNLP or creating an installer package), code obfuscators, license management, and others.
- Domain-specific libraries, where required, such as a library for representation of currency and monetary values.

11.2 Evaluating open source software

Before reviewing the best OSS product currently available for Java GUI development, how should you evaluate the usefulness of an OSS technology, and how can you make an informed plan about the ingredient technologies that will be used in a project?

When evaluating the adoption of a tool or a library, some practical considerations apply:

- The type of product license, and whether it is compatible with other OSS you plan to use and with your overall business goal. For example, suppose you plan to build an application for playing music using the Eclipse RCP. You might find an OSS Java library that plays all sorts of popular music formats, released under the GPL license[1]. In such a scenario, it is not legally possible to combine such a library with the Eclipse RCP and obtain a commercial product.

- Satisfying your requirements. The most important point is how effective the OSS is for solving your problems. This is key. It doesn't matter how well documented, mature, and powerful a tool or a library is if it doesn't meet your needs.
- The community involved with the project – whether any active on-line forums or other means for useful exchange exist with people that already use and are knowledgeable with the technology or tool.

  One consequence of a vibrant user community can be a reduction in the cost of professional services.
- The availability of useful documentation. This can be easily checked with a Web search. The quality and coverage of the documentation required depends on the importance of the OSS in your application scenario. If you are looking for something useful but not critical to your development, such as, say, a GUI test tool for a small internal application, you may not be concerned if there is no documentation for advanced customization features. In contrast, if you are looking for a critical component of your GUI, you should be careful in assessing the availability of effective documentation for advanced users.
- How the tool or technology you are investigating integrates with your existing basket of technologies. For example, if you use the JBuilder IDE and you find an OSS layout manager library, you need to know how well it can be integrated with JBuilder's visual editor, and whether it is available as an IDE extension than can be added to your development environment.

  The only sure way to assess the compatibility of a new tool or library with an existing environment and your class path is by testing. Overlooking compatibility issues can lead to degradation of an implementation. For example, including a library that use an XML file for configuration, while your application is already using a preferences file, results in an application with configuration data scattered over two separate files.
- The current maturity level of the product and its evolution strategy. Some OSS libraries start off small and pretty but grow to be huge and ugly by trying to solve everybody's problems in the most comprehensive way. In such cases,

1. For more on OSS licenses, see the discussion about RCP licensing in Chapter 13.

you are either forced to fork[2] the code base, and thus take responsibility for the code, or put up with bloated installation JAR files and amend your code for newer features you might not actually need.

Open Source Maturity Model

This section discusses a specific, formal model for assessing the maturity of OSS. Despite being a general technique that can be applied to comprehensive scenarios, including hardware, infrastructure software and large applications, the main ideas can also be used for selecting the best tools for smaller projects and in daily work.

The Open-Source Maturity Model (OSMM) described in (Bernard 2004) proposes a model for assessing open source products for their readiness for use in an industrial production environment. It can be useful to companies that are evaluating how a given OSS technology can fit within a given software development organization. In many cases though, when limiting to consider only OSS Java development technologies and for small projects, there is little need to resort to a fully-fledged model.

Factors such as functionality, support, documentation, training, product integration, and professional services are considered in the OSMM. The model considers two types of users: *early adopters*, who are more keen to adopt new but unfinished OSS technology, and their counterparts, the *pragmatists*, as well as three levels of implementation of the OSS in a project: experimentation, pilot, production.

The OSMM assesses an OSS product's maturity in three steps:

1. Assessing product elements. The output of this phase is a set of scores for each of the key product elements. Sub-steps of this phase are:
   – Define requirements. Determine the required functionality for the current scenario.
   – Locate resources. Determine whether essential resources are available to assist your organization in implementing the open source software. Examples include specialized consultants, or an approved partner company.
   – Assess element maturity. Maturity levels range from *non-existent product* to *production-ready.*
   – Assign a final element score in the range 1–10.
2. Assign weights summing to 10 to each element's maturity score to reflect its importance. For example, in evaluating a GUI testing tool, good documentation could be more important to the product's overall maturity assessment than the availability of professional services. Default values are shown in Table 11.1.

2. Some licenses don't allow modification of OSS source code that is to be used commercially.

3. Calculate overall product OSMM score.

   OSMM ranking = (Element Score × Element Weightings)

   The output of an OSMM assessment is a numeric score between 0 and 100 that is then compared with recommended values. Table 11.2 shows the minimum values suggested by default by the model.

Table 11.1 Table 1 OSMM default weights

Software	4
Support	2
Documentation	1
Training	1
Integration	1
Professional Services	1

Table 11.2 Table 2 OSMM recommended minimum scores

	Type of user	
Purpose of Use	**Early adopters**	**Pragmatist**
Experimentation	25	40
Pilot	40	60
Production	60	70

11.3 SWT or Swing?

After a general introduction to OSS evaluation comes perhaps the most critical choice in technology selection for a Java GUI project, one that will shape the development and dictate support and testing tools: the base GUI toolkit. Deciding which GUI toolkit to use for a GUI is extremely important, as this choice is hard to reverse. This section discusses and compares both toolkits thoroughly.

The toolkits

We assume readers are more experienced with Swing than SWT, so a quick introduction for readers not familiar with SWT is also provided.

Although it is a valid choice in various situations, for brevity AWT has only been considered briefly here.

The Swing toolkit and typical problems using it

Swing has been around since 1997, and a large number of resources such as documentation, code example, discussion forums, and so on, are available on line and in books. This also implies the existence of a large number of experienced developers proficient in Swing.

While Swing can be seen as a more conservative and less risky choice over SWT, it nevertheless suffers from a number of well-known issues that developers need to deal with:

- Swing applications need to be finely tuned, both as regards the final appearance of GUIs, by choosing or customizing an existing look and feel, adjusting pixels for baseline text alignment[3] and other fine details, and for the final implementation, which needs to be profiled and optimized for almost every non-trivial application.

  Worse, operating system vendors constantly update their platforms both for the appearance and richness of GUI components, involving Swing look and feel implementations in a never-ending chase in which native GUIs are constantly leading innovation and Swing is lagging behind[4].
- Swing is currently too basic a toolkit to support any but the most basic GUIs. In fast-paced production environments it therefore has to be complemented by other support libraries to provide cost-effective implementation and high-quality GUI detail. Producing good GUIs using the Swing toolkit alone is still too labor-intensive and needlessly hard.
- Swing's history lacks a complex project for effective testing, as Eclipse was for SWT, and is characterized by premature release and the commitment of Sun to diehard compatibility with legacy applications written as long ago as 1997. Because of this, it feels cumbersome and convoluted in some aspects. It is easy to criticize some of its architectural choices and implementation details, but nevertheless some parts of it are not of excellent quality.
- In the past Sun's support for Swing has been inadequate for its large developer base and its diffusion to the variety of applications built on top of Swing. Today there still are many bugs that have been open since the late

3. See Chapter 9.
4. This does not even consider the case of users personalizing their desktop environment: Swing Version 1.5 does not yet fully support native OS themes and some other customizations.

1990s[5], and the toolkit itself has been merely maintained in recent years. Sun's Swing development team has coped with these issues heroically, but perhaps only the advent of SWT and the jolt of fresh competition it brought to the scene has revived Sun's efforts with Swing.

Standard Windowing Toolkit

Readers familiar with Standard Windowing Toolkit (SWT) library can safely skip this section: the next section discusses the differences between SWT and the Swing toolkit.

Despite common folklore, SWT is not tied into Windows[6]. SWT runs on Apple Macintosh, Linux (using GTK or Motif), and a number of J2ME platforms. The design strategy of SWT focuses on building a simple, essential GUI toolkit that produces GUIs that are closely linked to the native environment, but abstract enoughs to be portable across supported platforms. SWT delegates common components such as labels, lists, tables and so on to native widgets, as AWT does, while emulating more sophisticated componesnts such as toolbars on Motif in Java, similar to Swing's strategy.

SWT has been designed to be as inexpensive as possible. One result of this is that it is native-oriented to the current platform: SWT provides different Java implementations for each platform, and each of these implementations makes native calls to the underlying platform implementation through the Java Native Interface, JNI. AWT is different, in that all platform-dependent details are hidden in native C code and the Java implementation is the same for all the platforms. This is illustrated in Figure 11.1.

Despite similarity in the features they provide, SWT and AWT have different design objectives:

- SWT explicitly aims at using native-driven widgets and being in control of the underlying OS GUI toolkit, while AWT attempts a simple form of cross-platform GUI support.
- AWT's overall philosophy is to provide a least-common- denominator across all platforms, while SWT also supports widgets by emulating them on platforms on which they are not supported natively.
- AWT hides the native layer from the Java programmer, while SWT make it available.

5. For example '*ButtonGroup-cannot reset the model to the initial unselected state*' or '*JMenuBar.setHelpMenu() not yet implemented*' are still open from 1997, as can be seen at http://bugs.sun.com/bugdatabase.
6. Even though its API has been designed in a very Windows-centric fashion.

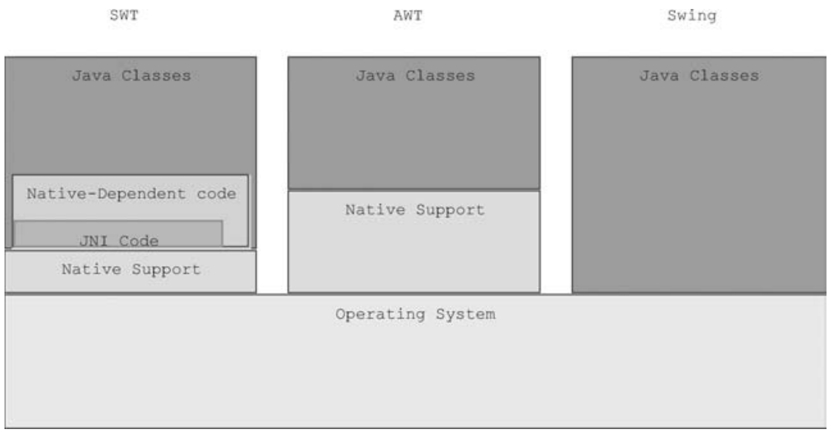


Figure 11.1 Java GUI toolkit architectures

- AWT employs different peers on different platforms, and exposes a common, cross-platform widget to the Java developer, while SWT provides less insulation from the native widgets.

The following table sketches the different widgets available for the main Java GUI toolkits.

Table 11.3 Comparison of visual components in Standard toolkits

Component	SWT	Swing	AWT
Advanced button	✓	✓	
Advanced text area	✓	✓	
Button	✓	✓	✓
Internal windows		✓	
Label	✓	✓	✓
List	✓	✓	✓
Menu	✓	✓	
Progress bar	✓	✓	
Sash	✓	✓	

Table 11.3 Comparison of visual components in Standard toolkits (Continued)

Component	SWT	Swing	AWT
Scale	✓	✓	
Slider	✓	✓	
Spinner	✓	✓	
TabFolder	✓	✓	
Table	✓	✓	✓
Text area	✓	✓	✓
Toolbar	✓	✓	✓
Tree	✓	✓	

The following figure shows the main classes of the class hierarchy of the SWT toolkit.

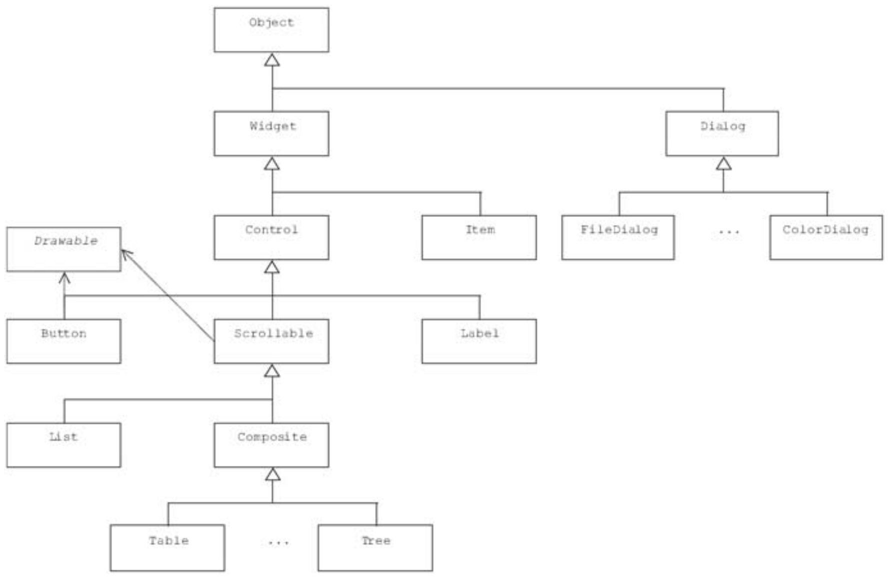


Figure 11.2 SWT widgets essential class hierarchy

Swing developers learning SWT might experience some difficulty in getting into its API style, which is more practical and simplified than Swing's. The main things that usually perplex Swing developers learning SWT are:

- The use of *styles* – bit masks represented as integers that customize the various aspects of a widget.
- The fact that when creating every widget, it is mandatory to specify the parent container as a parameter in the constructor.

For example, check box and button widgets are obtained using the same SWT widget (`org.eclipse.swt.Button`) with two different styles (respectively `SWT.PUSH` and `SWT.CHECK`).

After initial puzzlement, developers usually start to appreciate the coherence and predictability of the API and its good balance between the amount of control of low-level details and overall ease of use.

Native resource management with SWT

An important difference between SWT and Swing is in SWT's handling of native resources – platform resources that are allocated natively through the SWT API. Such resources must be explicitly released by the programmer through the `dispose()` method. The JVM's garbage collector finalizes unreferenced SWT objects, just as for any other object, but it does not dispose of the native resources used by them.

Native resources are represented in SWT by the following objects or their subclasses: `Color`, `Cursor`, `Display`, `Font`, `GC`, `Image`, `Printer`, `Region`, and `Widget`. Apart from the last, in the case of `Container` widgets, when disposing a parent container automatically disposes of all its contents, all other instances should be carefully disposed when no longer needed. The rule is that the object that created them is also responsible for disposing of them.

Typical problems when using SWT

SWT also has its own shortcomings, the main ones being:

- Developers cannot expect wide diffusion of SWT to less popular platforms. Porting the SWT toolkit to new platforms and maintaining existing ones is complex work that require a deep knowledge of the various GUI platforms and of SWT's inner workings, so it is hard for the open source initiative to successfully port SWT to minor platforms.
- SWT is a new API and as such requires costly learning. On the other hand, the greater spread of Swing means that it is widely taught in universities, and many organizations already have developers skilled in Swing who can mitigate the learning effort for novice programmers. The balance is changing,

however, as SWT gains in popularity over Swing, at least for some application typologies.

- There is not yet a real market of SWT widgets and third-party libraries. This is not a serious hurdle in itself, as SWT will continue to thrive under the shadow of the Eclipse project, which today provides all reusable classes and utilities for SWT-powered applications. In some niche application domains, however, this could be a problem, such as the rich ad-hoc components or chart widgets that are available for the Swing toolkit.
- Some developers and managers feel that SWT and its related technologies (JFace, high-level utility classes, and the Eclipse RCP) have yet to prove their maturity and viability as a fully-fledged base toolkit, not just the GUI framework that powers Eclipse. Investing in learning and building a code base on such a stack of technologies is still seen as controversial by some. This perception may change with time and other factors, such as the evolution of competing technologies like AJAX[7].

JFace

A good design choice made by SWT's architects was to separate the low-level features (basic widgets, basic content handling, and events) clearly from the utility support built on top of widgets (data handling, commands, application windows, wizards, handy support for native resource disposal, and so on). The latter layer is provided by the JFace library. Developers normally use JFace support on top of SWT, and manipulate raw SWT widgets only when specifying content details or handling low-level events. All data handling and high-level control (commands) is processed by means of JFace.

One of the advantages of separating basic widget support from higher-level features is that SWT remains compact and self-contained. This in turn makes SWT easier to learn for a novice, and more straightforward to use also for experienced developers.

Choosing a toolkit

Since SWT was released there has been a lot of discussion in the Java GUI developer community over which toolkit is the best – of an unexpectedly exasperated tone. A sort of religious war has been raging among developers over such an apparently mundane topic.

Putting away any religious bias, the solution to this puzzle is clear: there is no single 'best' toolkit. Both SWT and Swing have their own strong points and weaknesses and are individually best suited for specific problems. This is actually great

7. A discussion of new Web-oriented GUI technologies (such as Ajax) is provided in *Bandwidth and interaction* in Chapter 9 – see page 363.

news for Java developers, because it widens the possibilities. Increased power comes at a price though – Java GUI developers need to stay up-to-date on more than a single library.

The hidden cost of learning

Learning the basics of SWT is not too difficult for a Swing developer, given the fact that both toolkits share the same architectural concepts (single event dispatch thread, event model, overall toolkit widgets). SWT and Swing result in two different programming experiences, however, Swing being higher-level and Smalltalk-like, while SWT feels more like C/C++. Indeed at times it feels pretty much like programming Windows MFC. These tactical differences in the API style can confuse inexperienced programmers and double the workload of learning and mastering both toolkits. This is the major drawback, and is the hidden cost of the coexistence of two independent toolkits for Java.

Apart from basic concepts, however, such as events, layout managers, and simple widget handling, the two toolkits are rather different, both in philosophy and practical features. Failing to acknowledge this and trying to use them without considering their specificity – for example, trying to customize the appearance of SWT widgets to the pixel, or avoiding fine-tuning the details of Swing GUIs – is another example of the 'going against the flow' complexity booster discussed in Chapter 6.

The speed myth

It is not normally possible to assess whether SWT is faster than Swing, because there are so many parameters to consider in a fair comparison, such as raw speed depending on a given port of SWT, the type of application, or the time spent profiling and optimizing the particular application. Such factors make a thorough assessment of the two technologies possible in only few cases.

The Swing team has worked hard to improve performance as much as possible in recent releases, so that the 'raw speed' issue – SWT uses native OS-specific resources, and thus faster than Swing – seems less important than in the past. In these circumstances runtime performance of all but trivial GUIs depends mainly on the overall design and amount of care spent in its optimization, rather than the GUI toolkit alone. Having said that, one would expect an SWT profiled[8] application to take less memory and run faster than an equally profiled Swing application, but this is more a personal expectation than a mathematical law.

8. An application whose runtime execution has been carefully examined by means of a profiler tool and optimized accordingly.

Heavily-loaded widgets

Another common assumption is that SWT supports situations in which components need to sustain large volumes of data or other non-trivial situations, such as very large trees, much better than Swing. While such differences still remain, they are smaller than one would expect.

Consider a directory with 10,000 files on the local file system. If you don't mind cluttering your own file system, you could create such a directory by executing the following raw lines of code:

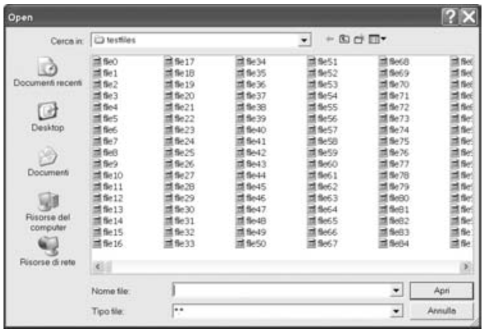
```
public static void create10000Files(){
  File f = null;
  for (int i = 0; i < 10000; i++) {
       f = new File("C:/temp/testfiles/file"+i);
       try {
            f.createNewFile();
       } catch (IOException e) {
            e.printStackTrace();
       }
  }
}
```

Opening a Swing file chooser dialog on such a directory and waiting for it to populate takes roughly 2.1 seconds[9], shown on the left in Figure 11.3. The equivalent dialog in SWT takes 1.3 seconds to completely start up. When using 50,000 files instead, the Swing dialog takes roughly 4.9 seconds to fully start up, and its SWT counterpart needs 7.6 seconds.

Even though this is a simple example, it nevertheless shows a couple of things:

- The power of competition. Swing performance used to be *much worse* than SWT in the past. In one project in 2000 we spent almost a month optimizing a Swing file chooser that was hanging when visualizing directories with thousands of files in a core part of the application. This was a substantial hindrance, and such accidents gave Swing a bad reputation among developers.
- Using the underlying native widgets via SWT is more reassuring for developers than using Swing, especially if they know up front that an application risks a potential performance bottleneck.
- Figure 11.3 shows how closely the Swing appearance in J2SE 1.5 using Windows look and feel matches the native operating system. Such a

9. These measurements are indicative only and provide data useful for a first evaluation. They were performed on a 1.6 GHz Pentium M 725 with 512 MB running Windows XP Professional, with 90MB of available physical memory. The application was executed first without measurement, then the results of the next three executions were averaged. Data gathered in this way is by no means reliable or indicative of real performance.



file0	file17	file34	file51	file68
file1	file18	file35	file52	file69
file2	file19	file36	file53	file70
file3	file20	file37	file54	file71
file4	file21	file38	file55	file72
file5	file22	file39	file56	file73
file6	file23	file40	file57	file74
file7	file24	file41	file58	file75
file8	file25	file42	file59	file76
file9	file26	file43	file60	file77
file10	file27	file44	file61	file78
file11	file28	file45	file62	file79
file12	file29	file46	file63	file80
file13	file30	file47	file64	file81
file14	file31	file48	file65	file82
file15	file32	file49	file66	file83
file16	file33	file50	file67	file84

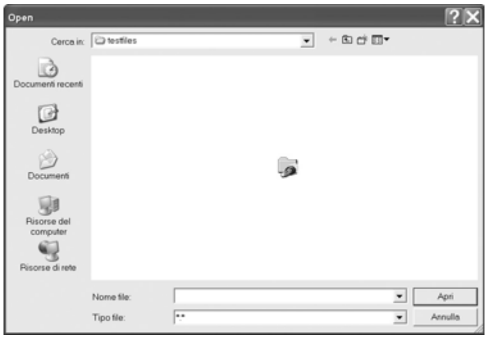
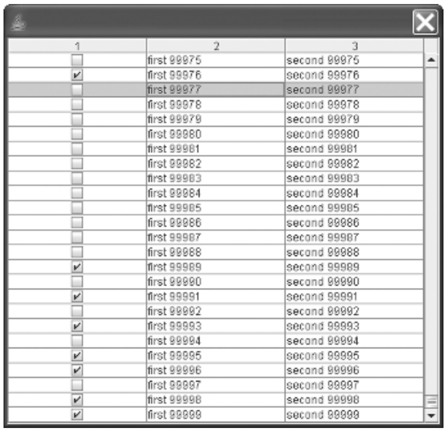


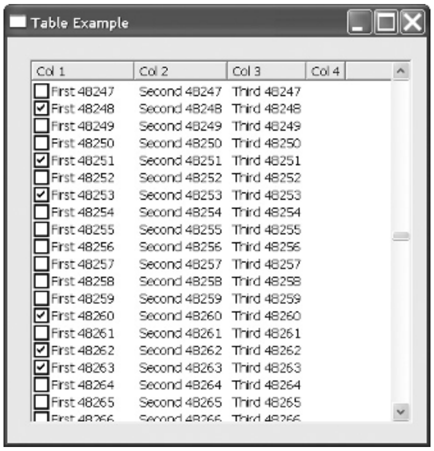
Figure 11.3 Swing (left) and SWT (right) file choosers

similarity is limited mostly to the presentation layer rather than effective interactions details, which still differ – for example, the native platform's contextual menus for files are not supported in the Swing dialog.

A similar comparison can be done for data tables with 10,0000 elements and a specific render, in this case a check box for Boolean values. No conspicuous differences emerge, although the SWT version seems slightly more responsive than the Swing case. Figure 11.4 shows this with Swing on the left and SWT on the right.



1	2	3
☐	first 99975	second 99975
✓	first 99976	second 99976
☐	first 99977	second 99977
☐	first 99978	second 99978
☐	first 99979	second 99979
☐	first 99980	second 99980
☐	first 99981	second 99981
☐	first 99982	second 99982
☐	first 99983	second 99983
☐	first 99984	second 99984
☐	first 99985	second 99985
☐	first 99986	second 99986
☐	first 99987	second 99987
☐	first 99988	second 99988
✓	first 99989	second 99989
☐	first 99990	second 99990
✓	first 99991	second 99991
☐	first 99992	second 99992
✓	first 99993	second 99993
☐	first 99994	second 99994
✓	first 99995	second 99995
✓	first 99996	second 99996
☐	first 99997	second 99997
✓	first 99998	second 99998
✓	first 99999	second 99999



Col 1	Col 2	Col 3	Col 4
☐ First 48247	Second 48247	Third 48247	
✓ First 48248	Second 48248	Third 48248	
☐ First 48249	Second 48249	Third 48249	
☐ First 48250	Second 48250	Third 48250	
✓ First 48251	Second 48251	Third 48251	
☐ First 48252	Second 48252	Third 48252	
✓ First 48253	Second 48253	Third 48253	
☐ First 48254	Second 48254	Third 48254	
☐ First 48255	Second 48255	Third 48255	
☐ First 48256	Second 48256	Third 48256	
☐ First 48257	Second 48257	Third 48257	
☐ First 48258	Second 48258	Third 48258	
☐ First 48259	Second 48259	Third 48259	
✓ First 48260	Second 48260	Third 48260	
☐ First 48261	Second 48261	Third 48261	
✓ First 48262	Second 48262	Third 48262	
✓ First 48263	Second 48263	Third 48263	
☐ First 48264	Second 48264	Third 48264	
☐ First 48265	Second 48265	Third 48265	

Figure 11.4 Swing (left) and SWT (right) large tables

In conclusion, data loads foreseen for an application can influence the choice of toolkit, but performance differences with JSE 1.5 are less dramatic than one might expect.

Scenarios to which SWT+JFace is better suited than Swing

Behind the IBM marketing talk and the implacable facade of the Eclipse Foundation there is real value in SWT (and JFace) that can support developers better than existing alternatives in many practical cases. The following list summarizes the main scenarios in which SWT is a good choice over Swing and other Java GUI technologies:

- Where the main target platforms are Windows, MacOS, Linux, and some J2ME profiles, SWT provides the many benefits discussed previously without limiting development – although admittedly at various levels of soundness.
- When the advantages of native widget support are important, SWT should be considered over Swing. These advantages are:
  - Different (that is, native) appearance and behavior on different platforms – SWT makes it is possible to completely mimic a native application.
  - A simpler API – all low-level details are left to the native GUI infrastructure.
- When the use of an RCP framework is advisable, such as for large projects, or applications with planned long-term maintenance, and it is viable to use SWT, using the Eclipse RCP[10] should be considered over similar technologies. The Eclipse RCP is in fact probably one of strongest points in favor of the adoption of SWT.
- Developers should be willing to embrace the technology, because effective use of it ultimately depends on them. This means that developers should be keen to learn the new API and deal with the typical problems SWT brings.

Scenarios in which Swing is a better choice than SWT

There are several cases in which Swing carries advantages over SWT:

- When maximum platform independence is needed Swing should be preferred. Platform independence should always be considered from a cost-driven perspective: vague long-term options should be examined very critically, as full platform independence is a costly and laborious feature to achieve.
- When providing the exact appearance and behavior of the GUI across a wide range of platforms is a requirement.
- For GUIs with particular graphics requirements (fancy, or extremely customized GUIs). Such developments should focus on Swing because of its greater rendering flexibility.

10. See Chapter 13.

- If developers already experienced in Swing and the wider diffusion of Swing skills are available. This means developers with a *deep* knowledge of Swing. Given the complexity of the toolkit, having just – say – delivered simple forms is not enough to be considered truly experienced with a complex framework such as Swing.
- When ad-hoc components are required. Using Swing to create a component such as the one discussed in the example application in Chapter 16 is straightforward, while with SWT the Draw2D library, part of the GEF library, must be used. There is no equivalent to Swing's `Graphics` and `Graphics2D` classes for accessing low-level raster rendering within standard widgets in SWT.

  Building ad-hoc components with SWT also renders one of its strong points useless – its support for native widgets. If ad-hoc components form the main parts of the GUI, some of the benefits of SWT, such as its native widgets, are lost, while you pay the price of its shortcomings, such as lack of availability on all Java-powered platforms.

From Swing to SWT

Some of the main differences between Swing and SWT are listed in the following table.

Table 11.4 Some of the differences between Swing and SWT

Concept	Swing	SWT+JFace
Data provider	MVC Models (that is, `TreeModel`, `TableModel`, and so on)	Implementations of IContentProvider
Data presentation	Cell renderers (that is, `CellRenderer`)	Implementations of ILabelProvider
Customization of data presentation in data-bound widgets (trees, tables, and so on)	Complete (every widget can be used)	Limited (for example, for tables, only images and check boxes can be embedded in table cells)
Main choosers (file, color, and so on)	Available as panels	Mostly available only as dialogs
Providing scroll behavior	Explicitly add `JScrollPane` instance	On main widgets use `SWT.*_SCROLL` style.

For more details, convenience wrapper classes, and a general discussion about Swing versus SWT, a thorough tutorial for migrating Swing code to SWT available on line[11].

Mix and match

It is of course possible to avoid the choice of one toolkit over the other by mixing them in the same application. Swing and SWT are increasingly being combined in a variety of development scenarios. There is often a need to employ legacy code in a newly-written SWT GUI, or to use a specialized third-party Swing widget in a SWT GUI, such as for example packaging an existing Swing GUI into an Eclipse plug-in, or to use some of the features of SWT from within a Swing GUI.

While mixing SWT and Swing widgets in the same application is technically feasible, there are a few design points that must be observed:

- Swing and SWT are different toolkits in appearance and behavior. Mixing them in the same GUI will confuse users. To avoid this, always try to follow the GUI guidelines and style of the embedding toolkit. For example, when developing an application using SWT and a third-party Swing widget, install the native look and feel for Swing. More generally, when embedding Swing widgets within SWT GUIs, always use the look and feel of the current platform.
- Following from the previous point, it is good plan never to mix the same type of widgets in the same application. For example, despite the fact that a Swing `JTree` and an SWT `Tree` can be made to look very similar, they have different behaviors and subtle interaction differences that will confuse users.
- From a visual viewpoint, carefully circumscribe the use of widgets from one toolkit in another to minimize user confusion. For example, limit to a single Swing ad-hoc component within an SWT application, or embed a whole Swing-powered panel into an Eclipse plug-in.

Here are the various possible technical combinations that allow a mix of the two GUI toolkit technologies.

- *Swing with SWT.* The most interesting situation is using Swing widgets from within an SWT application, often from within Eclipse. This is made straightforward by using the SWT class `org.eclipse.swt.awt.SWT_AWT` and adding the Swing or AWT widgets to it. This is the most frequent solution in SWT applications that need to include a Swing GUI, chiefly because this

11. The *Migrate your Swing application to SWT* tutorial is available from IBM's developerWorks at: http://www-128.ibm.com/developerworks/edu/j-dw-java-swing2swt-i.html

approach works well in practice and it is well supported by the SWT toolkit on a variety of platforms.

- *Swing on top of SWT*. SwingWT[12] is an open source project that aims to build the Swing API on top of SWT instead of AWT. The project has already been adopted for some applications despite being currently still in beta.
- *SWT on top of Swing*. Some OSS projects focus on making SWT available on top of Swing. This is particularly useful for executing an SWT GUI on a platform that is not yet supported by the relative platform-dependent runtime. Some tools that adopt such an approach, all in the early stages of development, are SWTSwing and SWT on Swing[13]. At the time of writing, SWTSwing appears to have been discontinued.

SWT and Swing together: the Java GUI dream team?

It is possible that the apparent schism within Java GUI toolkit technology between SWT and Swing will be partially resolved with time. It is still unclear how, as various options are viable. Practically, it is possible to incorporate some of the SWT ideas or architecture into AWT, even though the two libraries are in reality totally independent. SWT technology was in fact built as a total replacement for Swing – there are no references to AWT and Swing from within SWT. Even simple classes such as `Point` and `Rectangle` were duplicated in SWT.

The inclusion of portions of SWT within the standard Java API would however pose some technical challenges, as SWT is not completely platform independent. It would also further Balkanize and clutter the Java GUI API, which has grown by accretion from AWT to Swing, and through the various API revisions and enhancements of Swing. Perhaps the biggest obstacles are political and organizational – SWT is maintained outside Sun's control and is constantly evolving alongside Eclipse. SWT also requires maintenance for its platform-dependent lower layer on a variety of machines.

The GUI technology landscape has never been so rich and promising for Java. By choosing between SWT and Swing, or a combination of both, developers can build sophisticated user interfaces that leverage the power of Java technology.

11.4 Other GUI technologies

By themselves, or as the foundation of an RCP application, Swing and SWT are not the only available base technologies for GUI development. This section reviews other GUI technologies that are related to Java, both open source and commercial. The list is partial and by no means complete.

12. http://swingwt.sourceforge.net/
13. Available at: http://www.3plus4.de/swt/

Table 11.5 GUI technologies related to Java

Name	Notes	URL
Remote SWT	Exports the graphic display of a Java SWT application running on one host on a remote host, also transmitting GUI events.	`http://rswt.sourceforge.net/`
Canoo ULC	Execute Java Swing application with domain logic running on the server. Requires a small-footprint client installation.	`http://www.canoo.com/`
Asperon	Combination of Java and XML running in a browser using JVM 1.1.	`http://www.asperon.com/`
Thinlets	Execute in browser using JVM 1.1.	`http://thinlet.sourceforge.net/home.html`
XUL	Content in XML, Control and Business Domain in Javascript, for Firefox/Mozilla browsers.	`http://www.mozilla.org/projects/xul`
OpenLazlo	XML and Javascript rendered in Flash, standalone client or with server support (J2EE)[a].	`http://www.laszlosystems.com/`

a. Despite not using the Java language on the client (as of Version 3.x) OpenLazlo's popularity is growing among Java developers.

Almost all the products listed in the table are bound to the Web browser, testifying to the enormous interest for empowering Web technology with the power of fully-fledged Java GUIs. The technologies in Table 11.5 that use declarative languages, such as XUL and Thinlets, are discussed later in this chapter.

11.5 Utility libraries

This section lists a number of well-known and useful libraries and frameworks available for creating Java GUIs. Most of them target the Swing toolkit, essentially for historical reasons that reflect its longer availability than SWT. Similar libraries for SWT are expected to appear with time.

The list provided here is not meant to be exhaustive or fully descriptive of the features and details of each library. Interested readers are encouraged to visit the related Web sites for more information.

Security tools

Before diving into widgets and components, let's present some useful tools and libraries for GUI development.

Note that source code obfuscation is always needed to secure a Java application. License keys and the inner workings of the application can be tracked and hacked easily when full decompilation of executables is possible. Effective code obfuscation also impacts class design. At design time it is important to individuate carefully the set of core classes and methods that will be made inaccessible by means of obfuscation. Discussing the important topic of code obfuscation in depth is beyond the scope of this section.

The following list illustrates some of the available tools and libraries useful for securing the investment and hard work needed to build a professional Java application. In addition, downloading a decompiler such as JAD, DJ or Cavaj will help in testing the robustness of the security strategy chosen.

- yGuard is an OSS Java obfuscator packaged as an Ant task, and thus can easily be integrated into many development environments.

  `http://www.yworks.com/en/products_yguard_about.htm`
- Zelix KlassMaster is a Java obfuscator that claims to provide an unmatched obfuscation technology while also attempting various optimizations to the final obfuscated code. It is licensed commercially.

  `http://www.zelix.com/klassmaster`
- JLicense is a library for managing, creating, and validating license keys, and also includes a simple GUI tool. Its use in binary format is free, but source code needs to be purchased separately.

  `http://www.websina.com/products/jlicense.html`
- TrueLicense is an OSS library for handling the creation and validation of licenses. It uses the Java Cryptography Extension library (JCE). A Swing-based wizard is provided for installing new licenses by users .

  `https://truelicense.dev.java.net`

Paradoxically, an OSS license tool can be more secure than a home-made or even a commercial one, because its architecture and implementation has been publicly exposed and all sorts of attacks and weak points have been studied as a result, allowing countermeasures to be added to the public code. If you feel current OSS license tools are not secure enough for your application, you can opt for a commercial solution or a custom one, possibly by modifying an existing OSS license library.

Deployment tools

An important part of concluding the development of a client application is being able to deploy it effectively on the target machine. A number of options are available:

- For Java Web Start technology, a number of OSS and commercial tools ease the generation of JNLP files and certificates, such as CSR Generator for creating certificate signing requests, or the Java Web Start tools that form part of JDK 5.0 and are available in various IDEs.

  `http://www.apgrid.org/csrgenerator`
- Advanced Installer for Java provides native support for the installation of Java code on Windows platforms. It allows also the creation of MSI (Microsoft Installer) files.

  `http://www.advancedinstaller.com/java.html`
- Install4J is a commercial multi-platform installer that provides native integration with the underlying platform.

  `http://www.ej-technologies.com/products/install4j/overview.html`
- IzPack is an OSS Java installer tested on Windows, MacOS X, Linux, and BSD platforms.

  `http://www.izforge.com/izpack`
- GCJ is an OSS compiler of Java to native code. Compilation can be done both directly on Java source code and also on `.class` files. The result is a native executable that is better performing and more secure, even if this approach loses platform independency. This approach also suffers from a major drawback: compilation of Swing and AWT applications is not yet fully supported, although that for SWT is working.

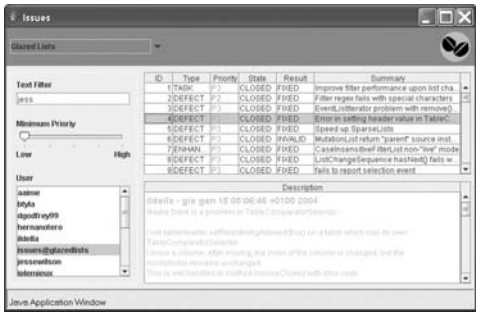
  `http://gcc.gnu.org/java/status.html`

Deployment also impacts users, and its design should therefore consider them. In some cases, such as shrink-wrapped products distributed on line as shareware, a technically simpler solution such as Java Web Start might prove unappealing for some users, as they might feel that accepting the certificate required to launch the application represented a security risk.

The remainder of this section discusses widgets and components available to Swing and SWT developers.

Glazed Lists

Glazed Lists is a library for handling GUI-savvy list collections. It provides a number of useful features, such as easy creation of table and list models, support for filtering and sorting (and their combinations), and a thread-safe architecture. The library supports both Swing and SWT toolkits. Performance on large lists has been considered, with optimized sorting and filtering algorithms. A number of non-GUI utility manipulations on lists are also provided. Figure 11.5 shows a screenshot of Glazed List at work.



ID	Type	Priority	State	Result	Summary
1	TASK	P3	CLOSED	FIXED	Improve filter performance upon list cha…
2	DEFECT	P2	CLOSED	FIXED	Filter regex fails with special characters
3	DEFECT	P3	CLOSED	FIXED	EventListIterator problem with remove()
4	DEFECT	P3	CLOSED	FIXED	Error in setting header value in TableC…
5	DEFECT	P3	CLOSED	FIXED	Speed up SparseLists
6	DEFECT	P3	CLOSED	INVALID	MutationList return "parent" source inst…
7	ENHAN	P3	CLOSED	FIXED	CaseInsensitiveFilterList non-"live" mode
8	DEFECT	P3	CLOSED	FIXED	ListChangeSequence hasNext() fails w…
9	DEFECT	P3	CLOSED	FIXED	fails to report selection event

Figure 11.5 Screenshot of the Glazed List demo

Glazed Lists is provided with the LGPL OSS license. It is available at `http://publicobject.com/glazedlists/`.

JGoodies Swing Suite

JGoodies Swing Suite is a comprehensive, professional suite for easing the burden of writing Swing applications. It ranges from basic support (look and feel, data validation, factories, utility code for form-based GUIs, data binding) to reusable components such as wizards, splash windows, log-in, license, 'About' dialogs, and so on, and includes various utility classes (enhanced help, lazy loading support). While most packages are distributed under a commercial license, some of them are freely available.

The following useful parts of the JGoodies Swing suite are released as OSS:

- *Animation*. A compact library for creating real-time animations with Swing, using concepts and notions from the W3C specification for the Synchronized Multimedia Integration Language (SMIL).
- *Data binding*. A useful library for binding Swing widgets to data sources (data models) in various ways.
- *Forms*. An easy to use and effective layout manager and builder specialized for form-based GUIs.
- *Looks*. A set of professional look and feels. Looks is discussed in more detail on page 436.
- *Validation*. A library for performing data validation and notification.

A screenshot of the Looks demo is shown in Figure 11.6.

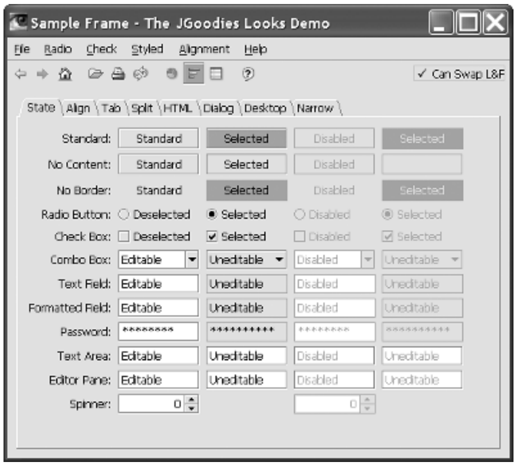

Sample Frame - The JGoodies Looks Demo
File Radio Check Styled Alignment Help
Can Swap L&F
State Align Tab Split HTML Dialog Desktop Narrow
Standard: Standard Selected Disabled Selected
No Content: Standard Selected Disabled
No Border: Standard Selected Disabled Selected
Radio Button: Deselected Selected Disabled Selected
Check Box: Deselected Selected Disabled Selected
Combo Box: Editable Uneditable Disabled Uneditable
Text Field: Editable Uneditable Disabled Uneditable
Formatted Field: Editable Uneditable Disabled Uneditable
Password:
Text Area: Editable Uneditable Disabled Uneditable
Editor Pane: Editable Uneditable Disabled Uneditable
Spinner: 0 0

Figure 11.6 Demonstration screenshot of JGoodies Looks

Among the many things I personally appreciate most in the JGoodies libraries is the careful attention to detail they provide. For example, see the text baseline alignment in the form shown in Figure 11.6, where the text in the label is aligned with the text in the corresponding text field or combo box.

JGoodies is available at `http://www.jgoodies.com`.

L2FProd Common Components

L2FProd Common Components is a set of Swing components available as OSS. The list of Swing widgets includes 'tip of the day,' property sheets, expandable/collapsible lists, and others. Figure 11.7 shows the L2FProd `JFontChooser` component, both as a panel contained in a tabbed pane, and as a pop-up dialog.

L2FProd Common Components is available at `http://www.l2fprod.com`.

Other OSS component libraries

Other OSS component libraries relevant to Java GUIs include:

- Buoy is a set of widgets on top of the Swing library that aims to provide a simpler development environment at a higher level of abstraction than plain Swing, with a minimal footprint of less than 200KB. Buoy is released as OSS.

  `http://buoy.sourceforge.net/`

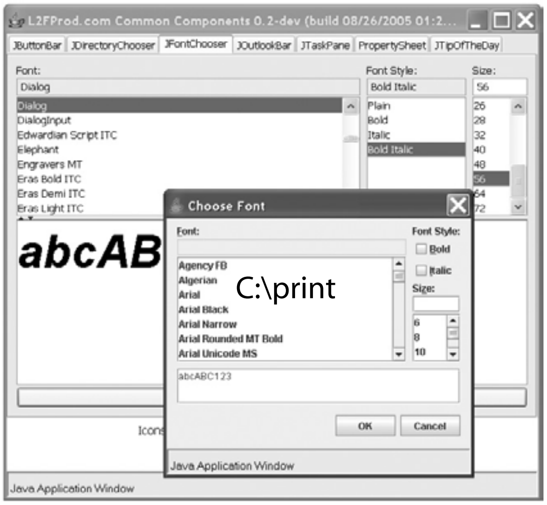


Figure 11.7 Demonstration screenshot of L2FProd Components

- The Java GUI Programming Extensions, Java-GPE, is an OSS library for developing Swing GUIs. Java-GPE includes some look and feels, some classes for preference dialogs, and other utility classes. See Figure 11.8 for a screenshot from the demo application.

  `http://www.markus-hillenbrand.de/javagpe`

- UICompiler is an OSS project that also provides a set of Swing widgets, ranging from a look and feel to various specialized choosers and general purpose components. A screenshot of the file chooser is shown in Figure 11.9 on page 418.

  `http://uic.sourceforge.net`

- Blazze provides a set of Swing components and utility classes specialized for business application GUIs. It is licensed as OSS.

  `http://blazze.sourceforge.net`

- JBalloon is a compact set of classes that provides balloon tooltips for Swing GUIs under the LGPL license.

  `http://www.allworldsoft.com/software/17-476-jballoontooltip.htm`

- Geosoft provides some utility classes under LGPL. Among them, the 2D graphics library provides a useful 2D graphics library and a rendering engine featuring layered hierarchical graphical objects, 3D world extents, style support, smart annotation, and image support.

  `http://geosoft.no`

- Batik is a Swing library for managing and rendering SVG[14] files licensed under the Apache OSS license. This library is used by many other products for managing and exporting data formats to SVG.

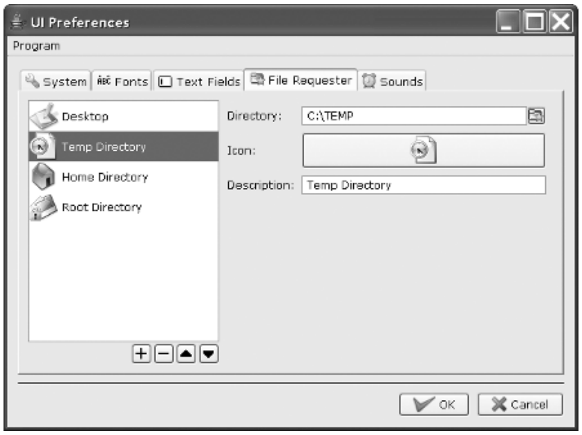
  `http://xml.apache.org/batik`
- JFreeChart is a library for generating charts licensed under LGPL. It can generate 2D and 3D pie charts, bar, line and area charts, scatter plots, Pareto, Gantt, and many other types of chart. It provides zooming, printing, and exporting to PDF, SVG, and bitmap formats.

  `http://www.jfree.org/jfreechart`
- Various OSS architectural frameworks for separating data from views are also available, such as TikeSwing or MVCMediator. For a critical discussion of the usefulness of such libraries in your application, see Chapter 6.

  TikeSwing: `http://sourceforge.net/projects/tikeswing`

  MVCMediator: `http://www.danmich.com/mvcmediator/1.0`
- CUF is a utility library and an application framework for building GUI applications in Swing that is also available for .NET. It provides callback handling close to .NET's delegates (Java CUF), a JTable extension, declarative state management of widgets, data binding, and more. Little documentation is currently available.

  `http://cuf.sourceforge.net`



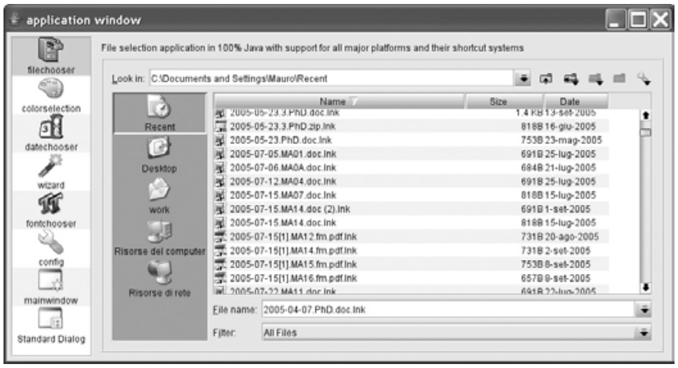
Directory: C:\TEMP

Icon:

Description: Temp Directory

Figure 11.8 Demonstration of Java GPE preferences

14. SVG (Scalable Vector Graphics) is a language for describing two-dimensional graphics in XML.



File selection application in 100% Java with support for all major platforms and their shortcut systems

Look in: C:\Documents and Settings\Mauro\Recent

Name	Size	Date
2005-05-23.3.PhD.doc.lnk	1.4 KB	13-set-2005
2005-05-23.3.PhD.zip.lnk	818B	16-giu-2005
2005-05-23.PhD.doc.lnk	753B	23-mag-2005
2005-07-05.MA01.doc.lnk	691B	25-lug-2005
2005-07-06.MA0A.doc.lnk	684B	21-lug-2005
2005-07-12.MA04.doc.lnk	691B	25-lug-2005
2005-07-15.MA07.doc.lnk	818B	15-lug-2005
2005-07-15.MA14.doc (2).lnk	691B	1-set-2005
2005-07-15.MA14.doc.lnk	818B	15-lug-2005
2005-07-15[1].MA12.fm.pdf.lnk	731B	20-ago-2005
2005-07-15[1].MA14.fm.pdf.lnk	731B	2-set-2005
2005-07-15[1].MA15.fm.pdf.lnk	753B	8-set-2005
2005-07-15[1].MA16.fm.pdf.lnk	657B	8-set-2005
2005-07-22.MA11.doc.lnk	691B	22-lug-2005

File name: 2005-04-07.PhD.doc.lnk

Filter: All Files

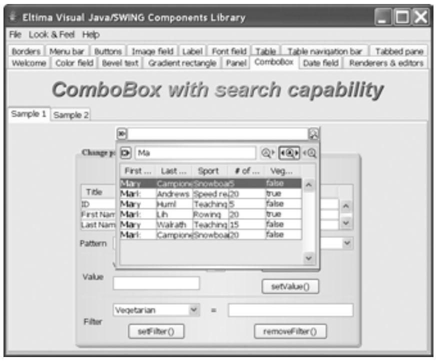
Figure 11.9 Demonstration of UI Compiler components

Some commercially-available Swing components

A great number of Swing-based widgets are available commercially from various vendors, some tens of libraries. Some of them are summarized here. The objective is not to promote one vendor over another, nor to provide an exhaustive and detailed list of the available features for each product, but just to highlight their existence. The list is not exhaustive and vendors were chosen without any prejudice towards their products.

- Eltima components provide a set of feature-rich widgets, although some of them might need some tweaking to fit into a standard Swing application coherently. Figure 11.10 shows a demo screenshot downloaded from the Eltima Web site.

  `http://www.eltima.com`


Eltima Visual Java/SWING Components Library
File Look & Feel Help
Borders Menu bar Buttons Image field Label Font field Table Table navigation bar Tabbed pane
Welcome Color field Bevel text Gradient rectangle Panel ComboBox Date field Renderers & editors
ComboBox with search capability
Sample 1 Sample 2
Change p
Title
ID
First Nam
Last Nam
Pattern
Value
setValue()
Filter
setFilter()
removeFilter()

Ma

First ...	Last ...	Sport	# of ...	Veg...
Mary	Campione	Snowboa	5	false
Mark	Andrews	Speed re	20	true
Mary	Huml	Teaching	5	false
Mark	Lih	Rowing	20	true
Mary	Walrath	Teaching	15	false
Mark	Campione	Snowboa	20	false

Filter: Vegetarian =

Figure 11.10 Screenshot of Eltima demonstration

- JAPI Libraries sports a set of specialized components (XML editor, and various browsers) and utility APIs.

  `http://www.japisoft.com`

- JSuite from Infragistics is a comprehensive although pricey set of components and utility classes for Swing: some AWT components are also supported. It includes Gantt charts, scheduling and calendar panels, advanced tables, navigation support, and more.

  `http://www.infragistics.com`

- ICESoft provides various browsers implemented in Swing, capable of rendering PDF, XML, XSL/XSLT, and many other formats and technologies.

  `http://www.icesoft.com`

- Javio provides a Web browser that supports HTML 4.0, CSS, and Javascript, plus other features, a graphic modeler component, and several widgets and tools for editing and viewing JavaHelp files, all implemented in Java.

  `http://www.javio.com`

- JGraph provides a range of specialized Swing ad-hoc graphic components that implement direct manipulation, in-place editing, zooming, pluggable routing algorithms, and more. Figure 11.11 shows the demo application of the Layout Pro component. JGraph also provides some general-purpose Swing components.

  `http://www.jgraph.com`

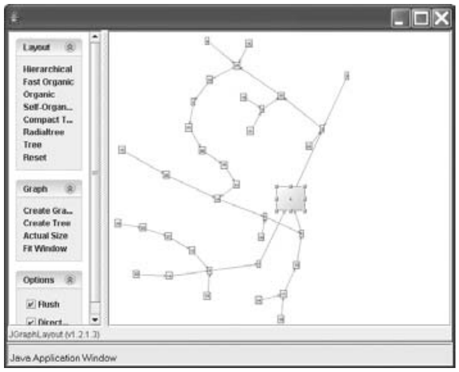


Figure 11.11 Screenshot of JGraph Layout Pro demonstration

- InfoNode provides a number of widgets, a dockable windows library for creating GUIs similar to the Eclipse IDE workbench, and a look and feel family. InfoNode products are available under a dual license: as a commercial product, or under the GPL license.

`http://www.infonode.net`

- ILOG's JViews library is a set of ad-hoc and custom components ranging from diagrams, maps, process control, Gantt diagrams, 2D and 3D charts, and others.

  `http://www.ilog.com`
- Quest's JClass Desktopview library delivers a number of specialized widgets, including 2D and 3D chart components.

  `http://www.quest.com`
- JProductivity Components! is a suite of calendars and date-oriented widgets that also includes a calculator and other components.

  `http://www.jproductivity.com`
- JIDE provides a large set of components that include a dockable window framework, action support, extended tables, and other widgets. A number of look and feels are available as well that mimic Visual Studio and Eclipse appearances.

  `http://www.jidesoft.com`
- ConfigureJ from JASE software is a Swing library for creating configurable menus, toolbars, and pop-ups that can be customized by the user.

  `http://www.configurej.com`
- Webcharts3D from Greenpoint Inc. is a set of Swing components for rendering various types of chart.

  `http://www.gpoint.com`
- I-net Crystal-Clear is not a component library, but is a report generator that can generate various types of report in many formats.

  `http://www.inetsoftware.de`
- yWorks' yFiles is a library of ad-hoc components for visualizing, analyzing, and automatic layout of graphs and diagrams.

  `http://www.yworks.com`

11.6 Test tools

This section lists only the most popular GUI testing tools specifically for Java desktop GUIs. Being written specifically for Swing/Java GUIs, these tools allow for a more robust widget location and other useful features tailored for Java GUIs. Unit test tools and general GUI test tools that can also test Java GUIs, such as

Eggplant, Rational Functional Tester, Xeus, and many others, are beyond the scope of this discussion.

- Abbot is a GUI testing framework that was initially available only for Swing, but has now also been ported to SWT – although its SWT support still needs improvement. Due to its API-centric nature, Abbot combines well with a unit test tool such as Junit. Costello is the companion tool for Abbot that enables recording and playback of Abbot GUI tests. Both Abbot and Costello are available as OSS.

  `http://abbot.sourceforge.net`
- Jacareto is a GUI testing tool released under the GPL. It performs recording and playback of GUI scripts, and can also be used for packaging animated demonstrations of existing Swing applications.

  `http://jacareto.sourceforge.net`
- Jemmy is an (OSS) module of the Netbeans IDE that can also be used without Netbeans. Like the other tools listed here, it can record and play test scripts and automatic demos of existing applications.

  `http://jemmy.netbeans.org`
- JFCunit is an extension to Junit for testing GUIs and is distributed as OSS. An example of use of JFCUnit is provided in Chapter 8.

  `http://jfcunit.sourceforge.net`
- Marathon is a GUI testing framework built with Swing. It supplies a test script recorder, a script player, and an editor to edit test scripts manually. Scripts are implemented in Python. Among other features, it provides useful support for writing acceptance tests.

  `http://marathonman.sourceforge.net`
- qftestJUI is a test tool for Swing GUIs that provides a number of interesting features such as record and playback of test scripts, including an integrated test and debugger, and others. It is distributed under a commercial license.

  `http://www.qfs.de`

11.7 Profiling tools

Profiling a Java application is an essential activity that is needed for refining the implementation of any complex GUI. This section lists only the major products, given the many OSS simple but effective Java profilers that are available.

- JProfiler is a commercial product featuring thread, memory, CPU profiling, and specialized views.

  `http://www.ej-technologies.com`

- Netbeans Profiler is the profiler part of Netbeans. Its functions include CPU, memory, and thread profiling, as well as basic JVM monitoring by means of dynamic bytecode instrumentation[15]. Note that, depending on the current license scheme of JFluid, a proprietary Sun technology used in the Netbeans profiler, the whole profiler is not licensed as OSS.

  `http://profiler.netbeans.org`
- Eclipse Profiler is currently still in beta, but is nevertheless interesting for its implementation built with SWT.

  `http://sourceforge.net/projects/eclipsecolorer`
- JProbe is a commercial profiler that provides various features such as thread, memory, CPU and heap profiling, and a number of task-oriented GUIs for solving common profiling issues such as memory leaks and performance bottlenecks.

  `http://www.quest.com/jprobe`
- OptimizeIt from Borland is a powerful profiling tool that integrates with various IDEs.

  `http://www.borland.com/us/products/optimizeit`

11.8 GUI builders

Visual builders are tools for visually composing GUIs that then generate the final Java code. They help to define widget layout and various components details visually by means of direct manipulation. They leverage the fact that both SWT and Swing-AWT have JavaBeans-compliant APIs, and as such can be processed by automatic tools.

The utility of such tools can be explained in terms of cognitive burden. Humans find it much easier to understand the map of a place as an image rather than as a textual description, no matter how clever the description may be.

The following partial list summarizes some visual editors for Java GUIs. It is not exhaustive – several tens of visual editors for Java GUI are currently in existence.

- JBuilder provides one of the first really complete GUI builders that remains a competitive and unintrusive tool (see item 1 of the list on page 423).

  `http://www.borland.com/jbuilder`
- Netbeans, with its much-trumpeted Matisse visual editor, was a leap beyond its old GUI builder, even if some issues remain unresolved.

  `http://www.netbeans.org/`

15. See *JRE runtime management* in Chapter 5, page 214.

- Eclipse, with its VE (Visual Editor) plug-in, has had to solve a couple of technical problems, that of integrating SWT and Swing, and building a powerful foundation visual environment that can be extended easily by third-party developers. Given these constraints design choices were made that currently hinder its use as of Version 1.1, such as using a whole JVM per screen. These issues will hopefully be optimized in forthcoming versions.
- Intellij Idea also provides its own GUI builder environment. This provides basic two-way support and a number of layout managers, and, most importantly, it integrates elegantly with the rest of the IDE platform.

Good visual builders speed up development time, despite what diehard coders such as the author might think about them. After all, there is little to be proud of in the ability to put together a complex form with a grid bag layout without touching a visual editor or the documentation, unless of course you don't have anything better to do with your time and mental energies. More importantly, the real world contains various types of developers, each with different skills and roles. Employing a good visual builder in your project can help less-skilled developers take charge of more mundane tasks.

Substandard GUI builders only decrease development time at first. Going beyond basic use and integrating the generated code into a larger code base, or tweaking it, often loses any time saved by consuming the precious time of (usually) skilled developers.

A GUI builder is itself a GUI, and as such its main purpose should be to simplify the work of its users. Assume that the objective of a GUI builder should be to lower the cognitive burden of developing real-world GUIs instead of using a text editor. A GUI builder should therefore affect only the visualization and manipulation of a GUI, not the way the generated code is structured, nor impose other hindrances like support files – such as the ghastly Netbeans' `.form` files – or any other by-product of the GUI builder tool.

Here is a checklist of desirable features for a GUI builder:

1. Integration with the application's existing code. The generated code should be exportable or importable into the GUI builder without any restrictions. This avoids vendor lock-in, and generally ensures higher code quality and productivity.
2. Modification and customization of every aspect of the generated code. Once imported, the code should be made available for modifications as required.
3. Architectural flexibility. This is the higher-level version of the external code integration property. Many GUI builders completely ignore the fact that Java is an OOP language, and that developers work using design patterns as well as their own frameworks and conventions. OOP-conscious developers are often faced with the dilemma of using GUI builders and then having to

tweak the resulting code heavily. With some weaker GUI builders, this means losing the possibility of importing the GUI into the visual editor, or initially writing everything by hand.

4. Layout manager support. Unfortunately, this is the weakest aspect of any GUI builder, in that they need some form of built-in support for a given layout manager to be able to lay out widgets in the visual environment. The newest or less-known layout managers are often unsupported in GUI builders.
5. Quality of the generated code. GUI builders often generate unreadable, lengthy, and structured code that is hard to understand. This is important for applications with many screens and where code maintenance is an important issue.
6. Real two-way support. 'Two-way support' is the term used to indicate the automatic alignment of the source file with the visual representation provided by the GUI builder. Ideally, whatever modification is done in either of these views should immediately be reflected in the other, and vice versa. If something is easier to do in the text editor, rather than opening up a couple of dialogs and clicking around in the visual builder view, then the tool should support editing from the source code to the visual environment in every respect.
7. Optimization. Like any good user interface, GUI builders should be optimized for common tasks. Developers often need to put together form-based GUIs with labels, fields, and buttons aligned on a grid basis managed in some form of dynamic layout.

The next section covers look and feels and visual customizations for Swing and SWT GUIs.

11.9 Presentation layer technologies

Customizing the visual appearance of a Java GUI can provide a strong branding for a product, or give it the final touch that will make users enthusiastic. This is the 'magical power' of professional GUI design, but as well as magic, graphical appearance can also spoil an otherwise well-designed product when badly employed. Fine-tuning the appearance of an application should be done very carefully and not as an afterthought. Developers, following their mental model, tend to think about a visual theme or a look and feel as very easily replaceable, and thus secondary to the rest of the GUI. This is wrong.

Assuming that the visual appearance of a GUI is not an important issue is very dangerous, a fact to which anybody who has ever chosen a look and feel in haste can testify. Customizing the visual appearance of a GUI can also have powerful political connotations. The author has personally witnessed the use of a new look and feel as a 'political' means of quickly and easily demonstrating a result – any result – in a troubled project.

This section deals with the practical choice of a look and feel or OS theme. Issues like how to properly customize the visual appearance, whether to allow users to change the installed look and feel, and similar GUI design concerns were discussed in Chapter 3. For brevity the term 'look and feel' is also used to refer to SWT presentation layer customizations.

Assessing a look and feel

Choosing the right look and feel (L&F) for an application requires many parameters to be considered and shouldn't be an oversight – the L&F is often selected as an afterthought on very ephemeral and subjective considerations, without considering users.

A Swing L&F is a Java library, and as such the general discussion for assessing OSS in Section 11.2 applies. Specifically, the main criteria for assessing the suitability of a L&F for a given application are:

- Fitness for purpose
- Usability testing
- Technical considerations
- Esthetic considerations

Fitness for purpose refers to the intended use of the presentation layer in an application. A GUI's visual appearance does not have to please developers or managers, but it should have a purpose that is coherent with the whole product experience. Simplifying things, a good criteria is to focus on the principal categories of users for an application. This is not an exact rule, but it will avoid gross mistakes.

What is going to be the main group of users? When a GUI should please occasional users, as in a kiosk GUI, for example, then an 'eye-candy' visually pleasant L&F is a perfect choice. When the GUI is geared towards repetitive users, too many bells and whistles get in the way of daily work, and usability-driven GUIs should be preferred over visually compelling ones. Of course, both a visually appealing and a usable L&F is possible at the same time for a given application. The best way to assess the right balance of simplicity and visual appearance in any scenario is to perform a usability test with users, covering the tasks that are more frequent in real use and more dependent on visual appearance.

Technically-dependent performance is the easiest parameter to assess with profiling, without the need to be exact or statistically meaningful. While some statistics are more significant for Swing than for SWT GUIs, measurements never hurt. The next subsection considers the following data for various L&Fs:

- *Memory consumption*. How much memory does the L&F alone consume to support the required visual tricks?
- *Time consumption*. How fast is the L&F to load and operate? As an extreme case, an 'eye-candy' and amateurish L&F could prove too slow to render critical components, making the whole user experience unacceptable.

Esthetic considerations are also important when choosing the best L&F for a given application. GUIs are mundane artifacts, and they are subject to all the same fashions, personal tastes and tiresomeness of this kind of thing. Think of the sort of GUIs software had back in 1997. Would you like your hard-built application look like one of those? How do you think your customers would react to it?

Swing look and feels

There are many L&F implementations around, so many that a comprehensive list would be too large and of little practical use[16]. To provide a fair evaluation, the same application was used to measure some performance data, a tweaked version of the SwingSet2 demo application that forms part of Sun's distribution of JDK 1.5.

Developers often stop bothering about L&F and visual details as soon as a good candidate is found that works well with their application. An extra step that most professional L&Fs support is customization of the L&F to optimize visual details and enrich the appeal of the GUI. More important than optimizing appearance details, most L&Fs support different color themes. Color themes, with their varying contrasts, are especially useful for visually-impaired or color-blind users[17].

Ocean 1.5

Ocean is the replacement for the glorious Steel L&F[18], the default cross-platform L&F for the Swing library. A cross-platform L&F should define behavior and appearance that are consistent between all the platforms on which Java can run. Despite Swing architecture making this technically simple[19], a number of other details are also required, such as pixel-level tuning of images, or smoothing the various differences between Java's and the native L&F. For example, MacOS Aqua users are used to certain conventions, while Windows users to others. Ocean uses

16. The L&Fs listed here were selected from http://javootoo.l2fprod.com/.
17. A starting point for color themes associated with color blindness can be found at http://www.visibone.com/colorblind/.
18. The Steel L&F is more commonly known as Metal. The name *Metal* is used to refer to the cross-platform Java L&F at large, of which Steel and Ocean are particular implementations.
19. The same Swing code runs on all the J2SE platforms.

graphics gradients to enhance the look of an application, and a caching mechanism for speeding up rendering.

In cases in which visual compatibility with existing applications running on JSE 1.5 or higher is needed, the old Steel L&F can still be obtained by setting the system property `swing.metalTheme=steel`. For compatibility with the older L&F, Ocean uses bold fonts for labels and text. This can be turned off with the system property: `swing.boldMetal=false`.

Table 11.6 Ocean 1.5 details

L&F name	Ocean
Time to load L&F at startup (seconds)	0
Time to launch application (seconds)	1.71
Free memory (KB) Initial - after startup	67.29
License	Sun's Binary Code License Agreement
Author / Company	Sun Microsystems Inc.
URL	`http://java.sun.com/products/jfc/`

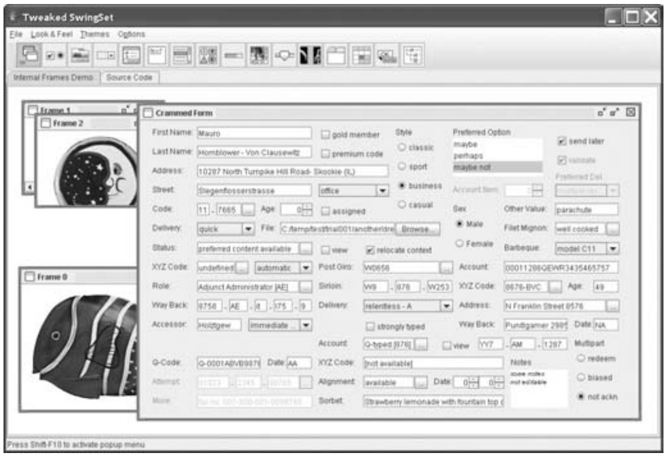
The above data was measured over three runs, discarding first-time executions to avoid JIT compilation overhead and other initializations, and also spurious values, for example when a garbage collection occurred. These measurements depend heavily on the machine on which they are performed and are indicative only.

The time to load the L&F is the time measured between the moment the L&F loading is started and when it is completed. It is not rigorously determined: a L&F that forks a thread to complete its installation will show a shorter loading time. Clearly for Ocean, because it is the default option, the time to load is zero.

The time to launch figure is an indication of the time needed to launch the application after all initialization is done and the GUI is ready for user interaction. For the purpose of these benchmarks, a particularly crammed form was added to the standard demos to see the effect of the L&F for form-based GUIs.

Finally, free memory shows the total amount of free memory consumed by the application, that is, initial free memory minus the free memory remaining after start-up. This figure gives some indication of the L&F's memory footprint.

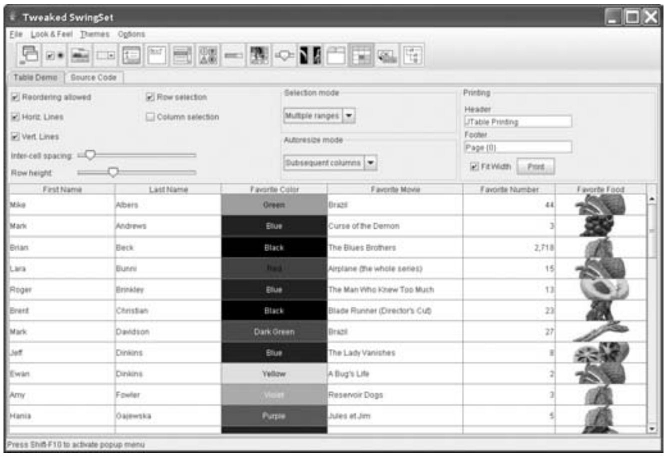
The larger these numbers – time to load, time to launch, and the memory occupied by the application – the more expensive it is to use the given L&F. Of course they only give an initial intuitive and non-rigorous evaluation of the L&F's performance. They are by no means substitutes for profiling and other measurements performed within the real context of a given application use domain. Figure 11.12 shows two screenshots of this L&F.



First Name: Mauro
Last Name: Hornblower - Von Clausewitz
Address: 10287 North Turnpike Hill Road- Skookie (IL)
Street: Siegenfiosserstrasse | office
Code: 11 . 7665 Age: 0
Delivery: quick File: C:/temp/test/trial001/anotherdre
Status: preferred content available
XYZ Code: undefined | automatic
Role: Adjunct Administrator [AE]
Way Back: 8758 . AE . it . 175 . 9
Accessor: Holzgew | immediate ...
Q-Code: Q-0001ABVB9876 Date: AA
Attempt: 11573 . 1345 . 09765
More: Tel. Nr. 001-800-001-0098765

☐ gold member
☐ premium code
☐ assigned
☐ view
☑ relocate context
Post Giro: W0656
Sirloin: W9 . 876 . W253
Delivery: relentless - A
☐ strongly typed
Account: Q-typed [876]
XYZ Code: [not available]
Alignment: available Date: 0 0
Sorbet: Strawberry lemonade with fountain top c

Style: ○ classic ○ sport ● business ○ casual
Preferred Option: maybe / perhaps / maybe not
Account Item:
Sex: ● Male ○ Female
Other Value: parachute
Filet Mignon: well cooked
Barbeque: model C11
Account: 00011286QEWR3435465757
XYZ Code: 8676-BVC Age: 49
Address: N Franklin Street 8576
Way Back: Pundtgamer 2985 Date: NA
☐ view YY7 . AM . 1287
Notes: some notes not editable
☑ send later
☑ reinstate
Preferred Del:
Multipart: ○ redeem ○ biased ● not ackn



☑ Reordering allowed
☑ Horiz. Lines
☑ Vert. Lines
Inter-cell spacing:
Row height:
☑ Row selection
☐ Column selection
Selection mode: Multiple ranges
Autoresize mode: Subsequent columns
Printing
Header: JTable Printing
Footer: Page {0}
☑ Fit Width

First Name	Last Name	Favorite Color	Favorite Movie	Favorite Number	Favorite Food
Mike	Albers	Green	Brazil	44	
Mark	Andrews	Blue	Curse of the Demon	3	
Brian	Beck	Black	The Blues Brothers	2,718	
Lara	Bunni	Red	Airplane (the whole series)	15	
Roger	Brinkley	Blue	The Man Who Knew Too Much	13	
Brent	Christian	Black	Blade Runner (Director's Cut)	23	
Mark	Davidson	Dark Green	Brazil	27	
Jeff	Dinkins	Blue	The Lady Vanishes	8	
Ewan	Dinkins	Yellow	A Bug's Life	2	
Amy	Fowler	Violet	Reservoir Dogs	3	
Hania	Gajewska	Purple	Jules et Jim	5	

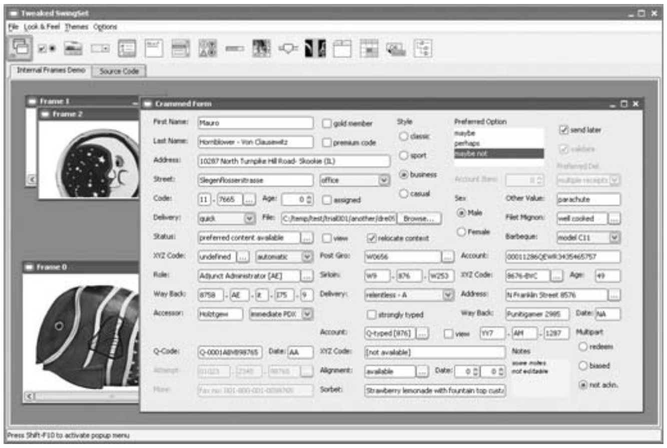
Figure 11.12 Ocean 1.5

Synthetica

Synthetica was one of the first professional L&Fs available on the market. Despite being based on Synth, which loads its data from external XML files, its performance isn't too bad, possibly because the example uses the default XML file contained in the JAR file along with the classes and loaded by the JRE at class loading time. Figure 11.13 shows two screenshots of this L&F.

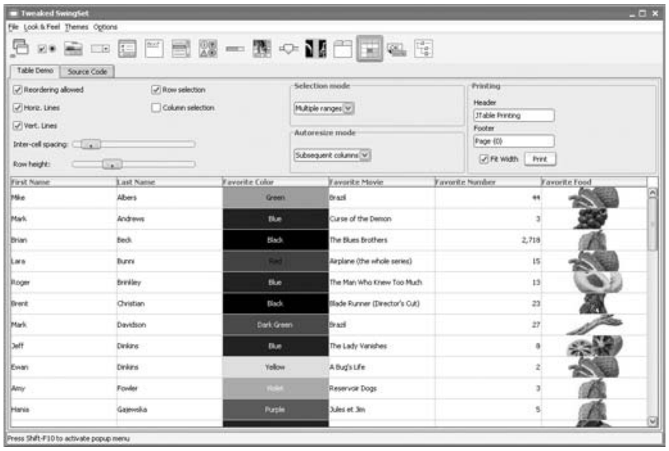
Table 11.7 Synthetica details

L&F name	Synthetica 1.0.0
Time to load L&F at startup (seconds)	0.61
Time to launch application (seconds)	0.98
Free memory (KB) Initial - after startup	1527.18
License	Dual license: LGPL and commercial
Author / Company	Javasoft
URL	`www.javasoft.de/jsf/public/products/synthetica`



First Name:	Mauro	gold member	Style: classic / sport / (●) business / casual	Preferred Option: maybe / perhaps / maybe not	[✓] send later
Last Name:	Hornblower - Von Clausewitz	premium code			
Address:	10287 North Turnpike Hill Road- Skokie (IL)				
Street:	Siegenflosserstrasse	office			
Code:	11 - 7665	Age: 0	assigned	Sex: (●) Male / Female	Other Value: parachute
Delivery:	quick	File: C:/temp/test/trial001/another/dre05			Filet Mignon: well cooked
Status:	preferred content available	view	[✓] relocate context		Barbeque: model C11
XYZ Code:	undefined / automatic	Post Giro:	W0656	Account:	00011286QEWR3435465757
Role:	Adjunct Administrator [AE]	Sirloin:	W9 - 876 - W253	XYZ Code:	8676-BVC Age: 49
Way Back:	8758 - AE - it - 175 - 9	Delivery:	relentless - A	Address:	N Franklin Street 8576
Accessor:	Holztgew / immediate PDX		strongly typed	Way Back:	Puntigamer 2985 Date: NA
		Account:	Q-typed [876] view YY7 - AH - 1287	Multipart:	redeem / biased / (●) not adn.
Q-Code:	Q-0001ABV898765 Date: AA	XYZ Code:	[not available]	Notes:	some notes not editable
Attempt:	01023 - 2345 - 98765	Alignment:	available Date: 0 0		
More:	Fax no: 001-800-001-0098765	Sorbet:	Strawberry lemonade with fountain top cust		

Figure 11.13 Synthetica 1.0.0



☑ Reordering allowed
☑ Horiz. Lines
☑ Vert. Lines
Inter-cell spacing:
Row height:
☑ Row selection
☐ Column selection

Selection mode
Multiple ranges
Autoresize mode
Subsequent columns

Printing
Header
JTable Printing
Footer
Page {0}
☑ Fit Width

First Name	Last Name	Favorite Color	Favorite Movie	Favorite Number	Favorite Food
Mike	Albers	Green	Brazil	44	
Mark	Andrews	Blue	Curse of the Demon	3	
Brian	Beck	Black	The Blues Brothers	2,718	
Lara	Bunni	Red	Airplane (the whole series)	15	
Roger	Brinkley	Blue	The Man Who Knew Too Much	13	
Brent	Christian	Black	Blade Runner (Director's Cut)	23	
Mark	Davidson	Dark Green	Brazil	27	
Jeff	Dinkins	Blue	The Lady Vanishes	8	
Ewan	Dinkins	Yellow	A Bug's Life	2	
Amy	Fowler	Violet	Reservoir Dogs	3	
Hania	Gajewska	Purple	Jules et Jim	5	

Press Shift-F10 to activate popup menu

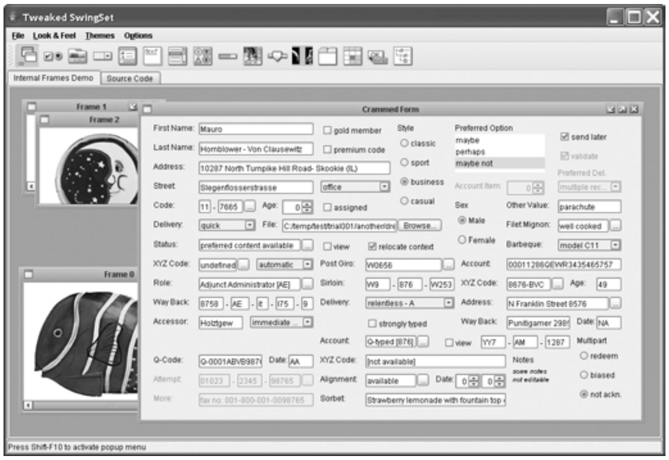
Figure 11.13 Synthetica 1.0.0 (Continued)

Alloy

The Alloy L&F is a commercial L&F that extends the standard cross-platform Java L&F. It has been around for many years, although it still lacks support for some of the latest features, such as scrollable tabs in tabbed panes. Figure 11.14 shows two screenshots of this L&F.

Table 11.8 Alloy details

L&F name	Alloy 1.4.4
Time to load L&F at startup (seconds)	0.06
Time to launch application (seconds)	1.15
Free memory (KB) Initial - after startup	650.93
License	Commercial
Author / Company	INCORS GmbH
URL	`http://www.compiere.org/looks/`

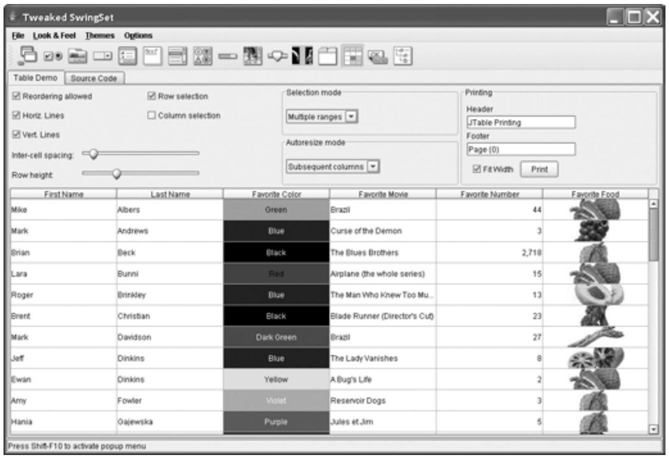


First Name: Mauro
Last Name: Hornblower - Von Clausewitz
Address: 10287 North Turnpike Hill Road- Skookie (IL)
Street: Siegenflosserstrasse — office
Code: 11 - 7665 Age: 0
Delivery: quick File: C:/temp/test/trial001/anotherdir
Status: preferred content available
XYZ Code: undefined — automatic
Role: Adjunct Administrator [AE]
Way Back: 8758 - AE - it - l75 - 9
Accessor: Holztgew — immediate ...
Q-Code: Q-0001ABVB987 Date: AA
Attempt: 01023 - 2345 - 98765
More: fax no: 001-800-001-0098765

☐ gold member
☐ premium code
☐ assigned
☐ view
Style: ○ classic ○ sport ● business ○ casual
☑ relocate context
Post Giro: W0656
Sirloin: W9 - 876 - W253
Delivery: relentless - A
☐ strongly typed
Account: Q-typed [876]
XYZ Code: [not available]
Alignment: available Date: 0 0
Sorbet: Strawberry lemonade with fountain top

Preferred Option: maybe / perhaps / maybe not
Account Item: 0
Sex: ● Male ○ Female
Account: 00011286QEWR3435465757
XYZ Code: 8676-BVC Age: 49
Address: N Franklin Street 8576
Way Back: Puntigamer 2985 Date: NA
☐ view YY7 - AM - 1287
Notes: some notes not editable

☑ send later
☑ validate
Preferred Del.: multiple rec...
Other Value: parachute
Filet Mignon: well cooked
Barbeque: model C11
Multipart: ○ redeem ○ biased ● not ackn.


Tweaked SwingSet
File Look & Feel Themes Options
Table Demo
Source Code
Print
Press Shift-F10 to activate popup menu

☑ Reordering allowed ☑ Row selection
☑ Horiz. Lines ☐ Column selection
☑ Vert. Lines
Inter-cell spacing:
Row height:
Selection mode: Multiple ranges
Autoresize mode: Subsequent columns
Printing — Header: JTable Printing Footer: Page {0} ☑ Fit Width

First Name	Last Name	Favorite Color	Favorite Movie	Favorite Number	Favorite Food
Mike	Albers	Green	Brazil	44	
Mark	Andrews	Blue	Curse of the Demon	3	
Brian	Beck	Black	The Blues Brothers	2,718	
Lara	Bunni	Red	Airplane (the whole series)	15	
Roger	Brinkley	Blue	The Man Who Knew Too Mu...	13	
Brent	Christian	Black	Blade Runner (Director's Cut)	23	
Mark	Davidson	Dark Green	Brazil	27	
Jeff	Dinkins	Blue	The Lady Vanishes	8	
Ewan	Dinkins	Yellow	A Bug's Life	2	
Amy	Fowler	Violet	Reservoir Dogs	3	
Hania	Gajewska	Purple	Jules et Jim	5	

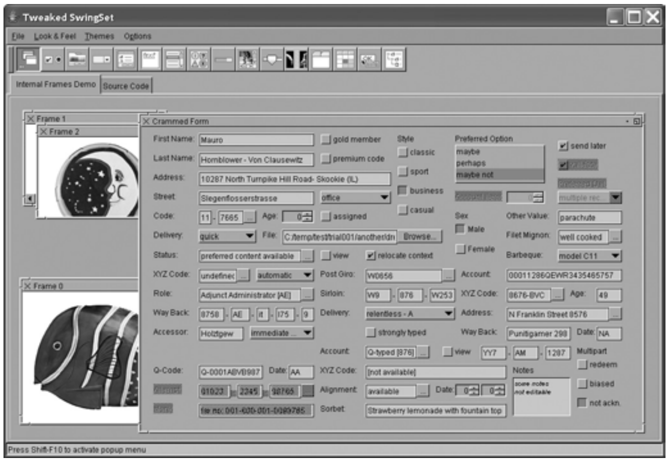
Figure 11.14 Alloy 1.4.4

Metal 3D

Despite being relatively lightweight on memory, this L&F has a rather outdated look that appears cluttered when used in crammed forms (see Figure 11.15).

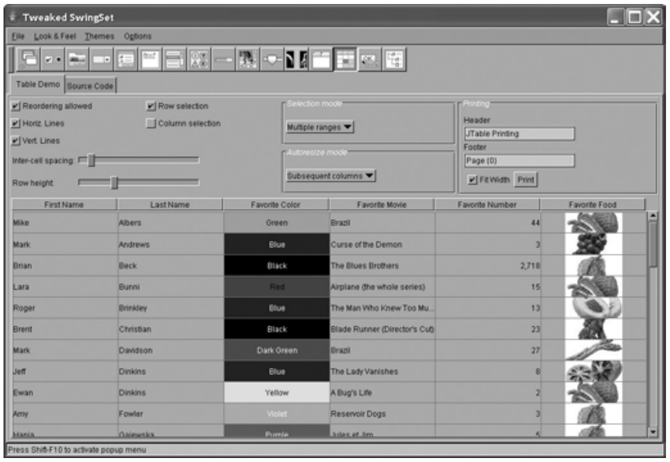
Table 11.9 Metal 3D details

L&F name	Metal 3D
Time to load L&F at startup (seconds)	0.3
Time to launch application (seconds)	1.30
Free memory (KB) Initial - after startup	274.19
License	LGPL (OSS)
Author / Company	Marcus Hillenbrand
URL	`http://www.markus-hillenbrand.de/3dlf/index.shtml`



First Name: Mauro
Last Name: Hornblower - Von Clausewitz
Address: 10287 North Turnpike Hill Road- Skookie (IL)
Street: Siegenflosserstrasse | office
Code: 11 - 7665 | Age: 0
Delivery: quick | File: C:/temp/test/trial001/another/dn
Status: preferred content available
XYZ Code: undefinec | automatic
Role: Adjunct Administrator [AE]
Way Back: 8758 - AE - it - 175 - 9
Accessor: Holztgew | immediate ...
Q-Code: Q-0001ABVB987 | Date: AA
01023 - 2345 - 98765
file no: 001-600-001-0089785
[ ] gold member
[ ] premium code
[ ] assigned
[ ] view
[✓ relocate context]
Style: [ ] classic [ ] sport [✓ business] [ ] casual
Preferred Option: maybe / perhaps / [✓ maybe not]
Post Giro: W0656
Sirloin: W9 - 876 - W253
Delivery: relentless - A
[ ] strongly typed
Account: Q-typed [876]
[ ] view YY7 - AM - 1287
XYZ Code: [not available]
Alignment: available | Date: 0 0
Sorbet: Strawberry lemonade with fountain top
Sex: [✓ Male] [ ] Female
Other Value: parachute
Filet Mignon: well cooked
Barbeque: model C11
Account: 00011286QEWR3435465757
XYZ Code: 8676-BVC | Age: 49
Address: N Franklin Street 8576
Way Back: Puntigamer 298 | Date: NA
Notes: some notes not editable
[✓ send later]
multiple rec...
Multipart: [ ] redeem [ ] biased [✓ not ackn.]

Figure 11.15 Metal 3D



[✓] Reordering allowed
[✓] Horiz. Lines
[✓] Vert. Lines
[✓] Row selection
[ ] Column selection
Inter-cell spacing
Row height
Header: JTable Printing
Footer: Page {0}
[✓] Fit Width

First Name	Last Name	Favorite Color	Favorite Movie	Favorite Number	Favorite Food
Mike	Albers	Green	Brazil	44	
Mark	Andrews	Blue	Curse of the Demon	3	
Brian	Beck	Black	The Blues Brothers	2,718	
Lara	Bunni	Red	Airplane (the whole series)	15	
Roger	Brinkley	Blue	The Man Who Knew Too Mu...	13	
Brent	Christian	Black	Blade Runner (Director's Cut)	23	
Mark	Davidson	Dark Green	Brazil	27	
Jeff	Dinkins	Blue	The Lady Vanishes	8	
Ewan	Dinkins	Yellow	A Bug's Life	2	
Amy	Fowler	Violet	Reservoir Dogs	3	
Hania	Galewska	Purple	Jules et Jim	5	

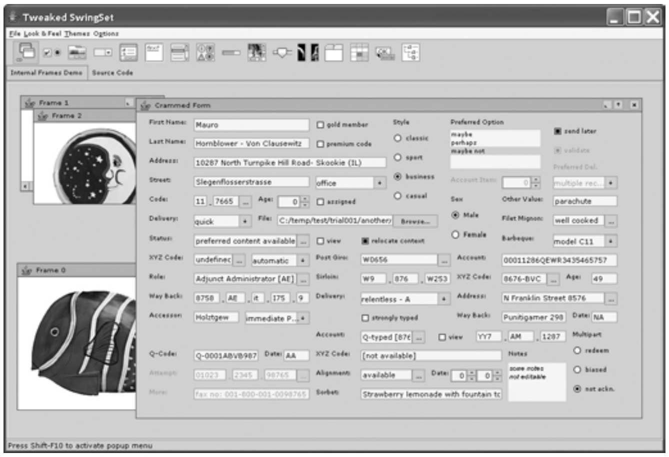
Figure 11.15 Metal 3D (Continued)

Hippo

The purpose of Hippo is to provide a simple, clean, and essential look and feel. It provides a clean result for complex form GUIs, although it is still not fully complete (see Figure 11.16).

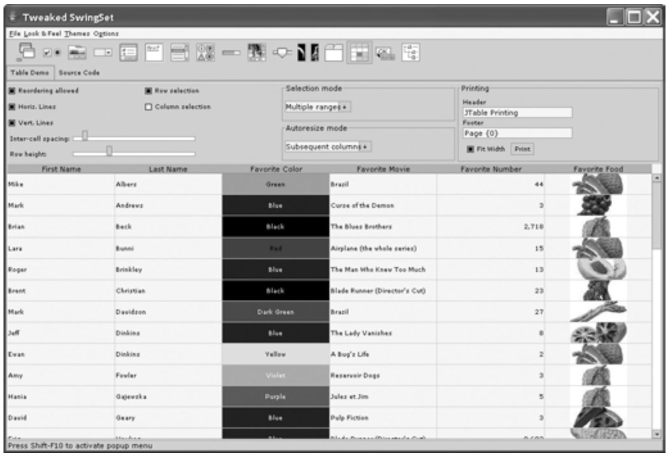
Table 11.10 Hippo details

L&F name	Hippo 0.7.1
Time to load L&F at startup (seconds)	0.03
Time to launch application (seconds)	1.34
Free memory (KB) Initial - after startup	904.29
License	BSD (OSS)
Author / Company	Robert Blixt
URL	`http://www.diod.se/`



First Name: Mauro ☐ gold member
Last Name: Hornblower - Von Clausewitz ☐ premium code
Address: 10287 North Turnpike Hill Road- Skookie (IL)
Street: Siegenflosserstrasse office
Code: 11 . 7665 Age: 0 ☐ assigned
Delivery: quick File: C:/temp/test/trial001/another
Status: preferred content available ☐ view ▣ relocate content
XYZ Code: undefinec automatic Post Giro: WD656
Role: Adjunct Administrator [AE] Sirloin: W9 . 876 . W253
Way Back: 8758 . AE . it . 175 . 9 Delivery: relentless - A
Accessor: Holztgew immediate P... ☐ strongly typed
Account: Q-typed [876
Q-Code: Q-0001ABVB987 Date: AA XYZ Code: [not available]
Attempt: 01023 . 2345 . 98765 Alignment: available Date: 0 0
More: fax no: 001-800-001-0098765 Sorbet: Strawberry lemonade with fountain tc

Style: ○ classic ○ sport ● business ○ casual
Preferred Option: maybe / perhaps / maybe not
▣ send later
▣ validate
Preferred Del.: multiple rec...
Account Item: 0
Sex: ● Male ○ Female
Other Value: parachute
Filet Mignon: well cooked
Barbeque: model C11
Account: 00011286QEWR3435465757
XYZ Code: 8676-BVC Age: 49
Address: N Franklin Street 8576
Way Back: Punitigamer 298 Date: NA
☐ view YY7 . AM . 1287
Notes: some notes not editable
Multipart: ○ redeem ○ biased ● not ackn.


Tweaked SwingSet
File Look & Feel Themes Options
Table Demo
Source Code
Print
Press Shift-F10 to activate popup menu

▣ Reordering allowed ▣ Row selection
▣ Horiz. Lines ☐ Column selection
▣ Vert. Lines
Inter-cell spacing:
Row height:
Selection mode: Multiple ranges
Autoresize mode: Subsequent columns
Printing — Header: JTable Printing; Footer: Page {0}; ▣ Fit Width

First Name	Last Name	Favorite Color	Favorite Movie	Favorite Number	Favorite Food
Mike	Albers	Green	Brazil	44	
Mark	Andrews	Blue	Curse of the Demon	3	
Brian	Beck	Black	The Blues Brothers	2,718	
Lara	Bunni	Red	Airplane (the whole series)	15	
Roger	Brinkley	Blue	The Man Who Knew Too Much	13	
Brent	Christian	Black	Blade Runner (Director's Cut)	23	
Mark	Davidson	Dark Green	Brazil	27	
Jeff	Dinkins	Blue	The Lady Vanishes	8	
Ewan	Dinkins	Yellow	A Bug's Life	2	
Amy	Fowler	Violet	Reservoir Dogs	3	
Hania	Gajewska	Purple	Jules et Jim	5	
David	Geary	Blue	Pulp Fiction	3	

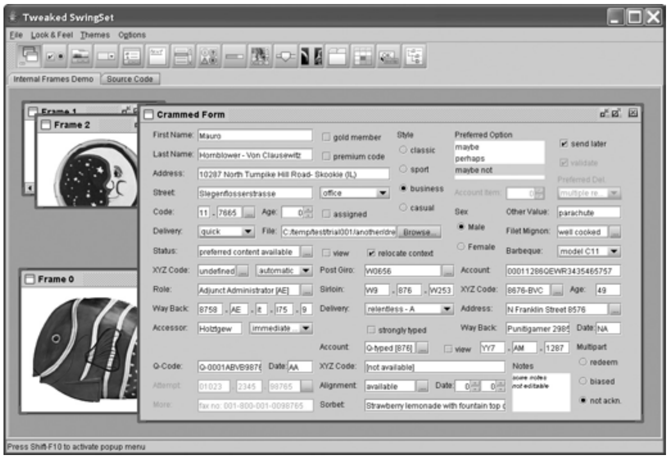
Figure 11.16 Hippo 0.7.1

Compiere

Compiere L&F is part of an OSS framework for building ERP applications. The Compiere L&F is geared towards form-based GUIs, is relatively fast to load, and can be customized through a dedicated GUI (see Figure 11.17).

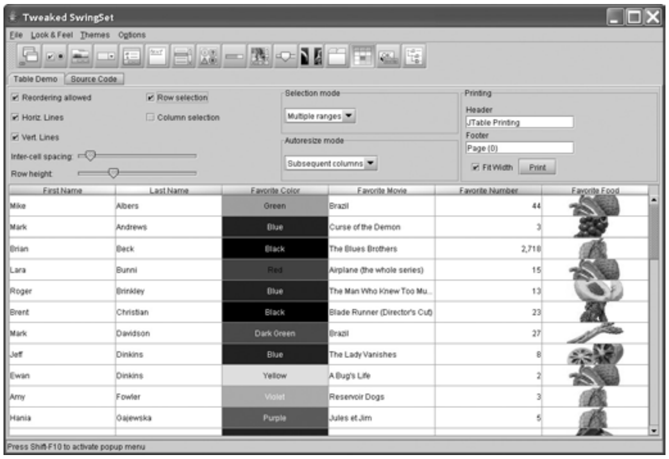
Table 11.11 Compiere details

L&F name	Compiere Looks 1.2.0
Time to load L&F at startup (seconds)	0.01
Time to launch application (seconds)	1.40
Free memory (KB) Initial - after startup	1460.33
License	Variant of Mozilla Public License (OSS)
Author / Company	Compiere Inc.
URL	`http://www.compiere.org/looks/`



First Name: Mauro
Last Name: Hornblower - Von Clausewitz
Address: 10287 North Turnpike Hill Road- Skookie (IL)
Street: Siegenflosserstrasse office
Code: 11 . 7665 Age: 0
Delivery: quick File: C:/temp/test/trial001/another/dre
Status: preferred content available
XYZ Code: undefined automatic
Role: Adjunct Administrator [AE]
Way Back: 8758 . AE . it . I75 . 9
Accessor: Holztgew immediate ...
Q-Code: Q-0001ABVB9876 Date: AA
Attempt: 01023 . 2345 . 98765
More: fax no: 001-800-001-0098765
gold member
premium code
assigned
view
relocate context
Post Giro: W0656
Sirloin: W9 . 876 . W253
Delivery: relentless - A
strongly typed
Account: Q-typed [876]
XYZ Code: [not available]
Alignment: available Date: 0 0
Sorbet: Strawberry lemonade with fountain top c
Style: classic sport [✓ business] casual
Preferred Option: maybe perhaps maybe not
Account Item: 0
Sex: [✓ Male] Female
Account: 00011286QEWR3435465757
XYZ Code: 8676-BVC Age: 49
Address: N Franklin Street 8576
Way Back: Punitgamer 2985 Date: NA
view YY7 . AM . 1287
Notes: score notes not editable
send later ✓
validate ✓
Preferred Del: multiple re...
Other Value: parachute
Filet Mignon: well cooked
Barbeque: model C11
Multipart: redeem biased [✓ not ackn.]

Figure 11.17 Compiere Looks 1.2.0



[✓] Reordering allowed
[✓] Row selection
[✓] Horiz. Lines
[ ] Column selection
[✓] Vert. Lines
Inter-cell spacing:
Row height:

Selection mode: Multiple ranges
Autoresize mode: Subsequent columns

Printing
Header: JTable Printing
Footer: Page {0}
[✓] Fit Width

First Name	Last Name	Favorite Color	Favorite Movie	Favorite Number	Favorite Food
Mike	Albers	Green	Brazil	44	
Mark	Andrews	Blue	Curse of the Demon	3	
Brian	Beck	Black	The Blues Brothers	2,718	
Lara	Bunni	Red	Airplane (the whole series)	15	
Roger	Brinkley	Blue	The Man Who Knew Too Mu...	13	
Brent	Christian	Black	Blade Runner (Director's Cut)	23	
Mark	Davidson	Dark Green	Brazil	27	
Jeff	Dinkins	Blue	The Lady Vanishes	8	
Ewan	Dinkins	Yellow	A Bug's Life	2	
Amy	Fowler	Violet	Reservoir Dogs	3	
Hania	Gajewska	Purple	Jules et Jim	5	

Press Shift-F10 to activate popup menu

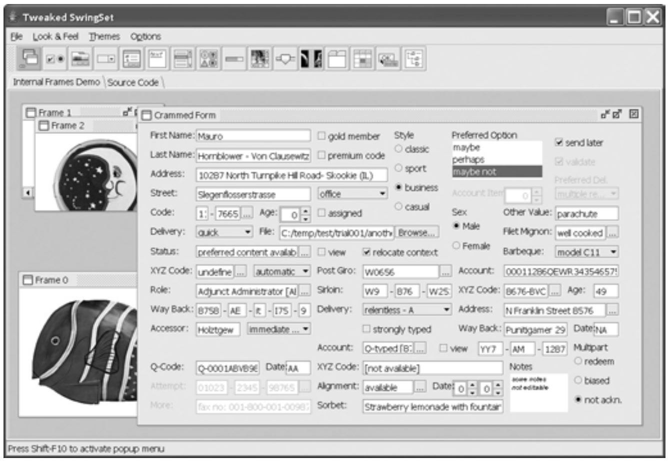
Figure 11.17 Compiere Looks 1.2.0 (Continued)

JGoodies Looks

JGoodies Looks is a family of look and feels that provides quality design to the pixel and multi-platform coherence. The Plastic L&F, for example, has been designed especially for Windows users (see Figure 11.18).

Table 11.12 JGoodies Looks details

L&F name	JGoodies Looks 1.3.1
Time to load L&F at startup (seconds)	0.05
Time to launch application (seconds)	1.37
Free memory (KB) Initial - after startup	1355.59
License	BSD (OSS)
Author / Company	Karsten Lentzsch
URL	`http://www.jgoodies.com/`



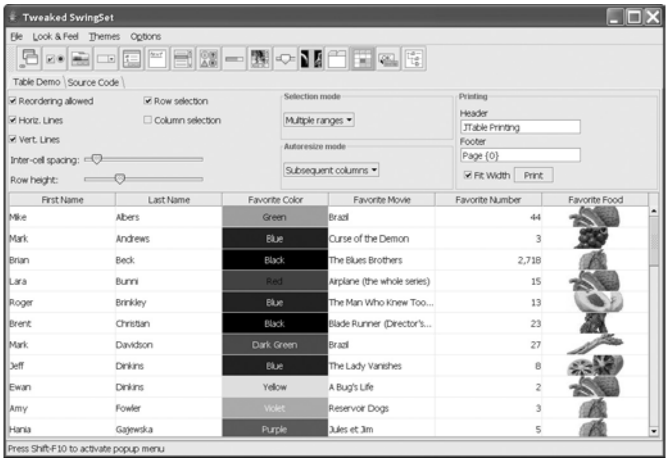
First Name: Mauro ☐ gold member
Last Name: Hornblower - Von Clausewitz ☐ premium code
Address: 10287 North Turnpike Hill Road- Skookie (IL)
Street: Siegenflosserstrasse office
Code: 1: - 7665 ... Age: 0 ☐ assigned
Delivery: quick File: C:/temp/test/trial001/anoth
Status: preferred content availab ... ☐ view ☑ relocate context
XYZ Code: undefine ... automatic Post Giro: W0656 ...
Role: Adjunct Administrator [Al ... Sirloin: W9 - 876 - W25
Way Back: 8758 - AE - it - 175 - 9 Delivery: relentless - A
Accessor: Holztgew immediate ... ☐ strongly typed
Account: Q-typed [B: ... ☐ view YY7 - AM - 1287
Q-Code: Q-0001ABVB9E Date: AA XYZ Code: [not available]
Attempt: 01023 - 2345 - 98765 ... Alignment: available ... Date: 0 0
More: fax no: 001-800-001-0098 Sorbet: Strawberry lemonade with fountain

Style: ○ classic ○ sport ● business ○ casual

Preferred Option: maybe / perhaps / maybe not

☑ send later
☑ validate
Preferred Del.: multiple re...
Account Item: 0
Sex: ● Male ○ Female
Other Value: parachute
Filet Mignon: well cooked ...
Barbeque: model C11
Account: 00011286QEWR34354657!
XYZ Code: 8676-BVC ... Age: 49
Address: N Franklin Street 8576 ...
Way Back: Punitgamer 29 Date: NA
Multipart: ○ redeem ○ biased ● not ackn.
Notes: some notes not editable

Press Shift-F10 to activate popup menu



☑ Reordering allowed ☑ Row selection
☑ Horiz. Lines ☐ Column selection
☑ Vert. Lines
Inter-cell spacing:
Row height:

Selection mode: Multiple ranges
Autoresize mode: Subsequent columns

Printing
Header: JTable Printing
Footer: Page {0}
☑ Fit Width

First Name	Last Name	Favorite Color	Favorite Movie	Favorite Number	Favorite Food
Mike	Albers	Green	Brazil	44	
Mark	Andrews	Blue	Curse of the Demon	3	
Brian	Beck	Black	The Blues Brothers	2,718	
Lara	Bunni	Red	Airplane (the whole series)	15	
Roger	Brinkley	Blue	The Man Who Knew Too...	13	
Brent	Christian	Black	Blade Runner (Director's...	23	
Mark	Davidson	Dark Green	Brazil	27	
Jeff	Dinkins	Blue	The Lady Vanishes	8	
Ewan	Dinkins	Yellow	A Bug's Life	2	
Amy	Fowler	Violet	Reservoir Dogs	3	
Hania	Gajewska	Purple	Jules et Jim	5	

Press Shift-F10 to activate popup menu

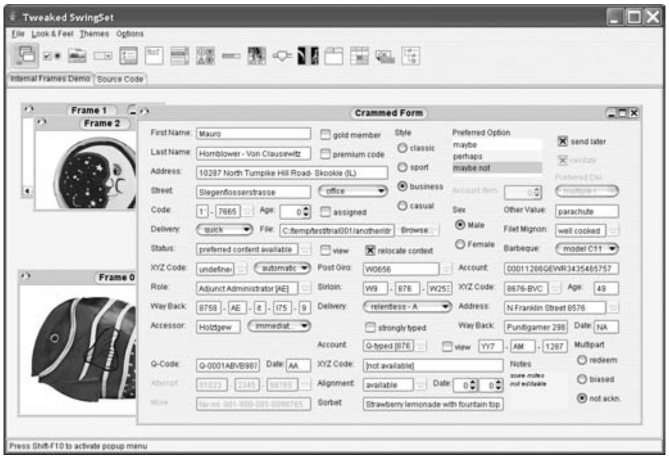
Figure 11.18 JGoodies Looks, Plastic L&F

Liquid

Liquid provides a Swing look and feel based on the Mosfet Liquid KDE 3.x theme (see Figure 11.19).

Table 11.13 Liquid details

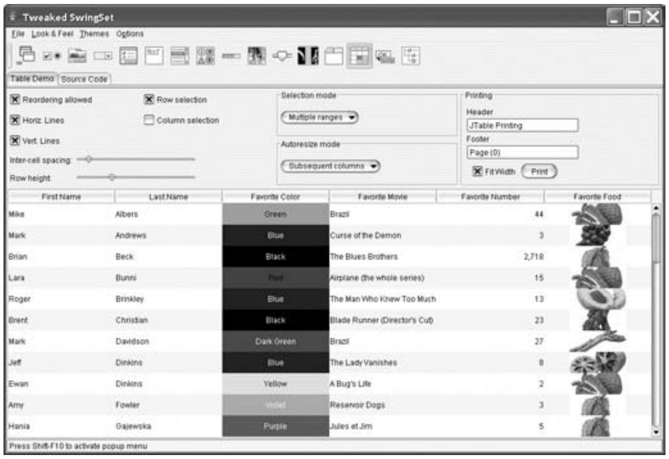
L&F name	Liquid
Time to load L&F at startup (seconds)	0.13
Time to launch application (seconds)	1.11
Free memory (KB) Initial - after startup	617.76
License	(OSS)
Author / Company	M. Lazarevic and E. Vickroy
URL	`https://liquidlnf.dev.java.net/`



First Name: Mauro [ ] gold member
Last Name: Hornblower - Von Clausewitz [ ] premium code
Address: 10287 North Turnpike Hill Road- Skookie (IL)
Street: Siegenflosserstrasse office
Code: 1 - 7665 Age: 0 [ ] assigned
Delivery: quick File: C:/temp/testtrial001/anotherdir
Status: preferred content available [ ] view [✓ relocate context]
XYZ Code: undefine automatic Post Giro: W0656
Role: Adjunct Administrator [AE] Sirloin: W9 - 876 - W25
Way Back: 8758 - AE - it - I75 - 9 Delivery: relentless - A
Accessor: Holztgew immediat [ ] strongly typed
Account: Q-typed [876 [ ] view YY7 - AM - 1287
Q-Code: Q-0001ABVB987 Date: AA XYZ Code: [not available]
Alignment: available Date: 0 0
Sorbet: Strawberry lemonade with fountain top

Style: classic sport [✓ business] casual
Preferred Option: maybe perhaps maybe not
[✓ send later]
Sex: [✓ Male] Female
Other Value: parachute
Filet Mignon: well cooked
Barbeque: model C11
Account: 00011286QEWR3435465757
XYZ Code: 8676-BVC Age: 49
Address: N Franklin Street 8576
Way Back: Punitgamer 298 Date: NA
Notes: some notes not editable
Multipart: redeem biased [✓ not ackn.]

Figure 11.19 Liquid



[✓ Reordering allowed] [✓ Row selection]
[✓ Horiz. Lines] [ ] Column selection
[✓ Vert. Lines]
Inter-cell spacing
Row height

Selection mode: Multiple ranges
Autoresize mode: Subsequent columns

Printing
Header: JTable Printing
Footer: Page {0}
[✓ Fit Width]

First Name	Last Name	Favorite Color	Favorite Movie	Favorite Number	Favorite Food
Mike	Albers	Green	Brazil	44	
Mark	Andrews	Blue	Curse of the Demon	3	
Brian	Beck	Black	The Blues Brothers	2,718	
Lara	Bunni	Red	Airplane (the whole series)	15	
Roger	Brinkley	Blue	The Man Who Knew Too Much	13	
Brent	Christian	Black	Blade Runner (Director's Cut)	23	
Mark	Davidson	Dark Green	Brazil	27	
Jeff	Dinkins	Blue	The Lady Vanishes	8	
Ewan	Dinkins	Yellow	A Bug's Life	2	
Amy	Fowler	Violet	Reservoir Dogs	3	
Hania	Gajewska	Purple	Jules et Jim	5	

Press Shift-F10 to activate popup menu

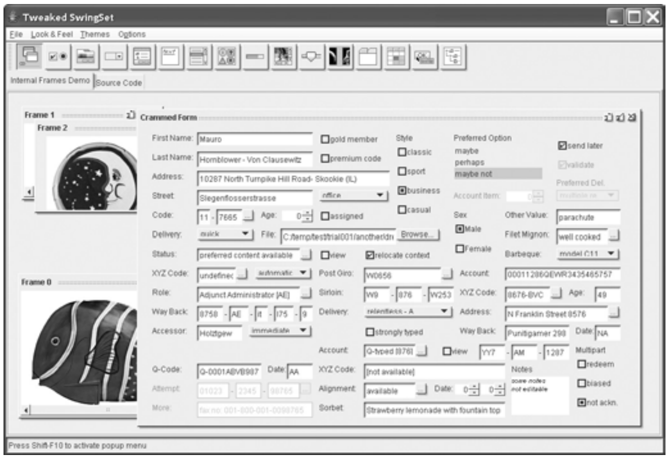
Figure 11.19 Liquid (Continued)

Oyoaha

Despite Oyoaha's coherent design, some visual details, particularly the 3D effect of buttons and text fields, result in a cluttered ensemble when the L&F is employed in non-trivial forms. For an example, see Figure 11.20.

Table 11.14 Oyoaha details

L&F name	Oyoaha 3.0
Time to load L&F at startup (seconds)	0.08
Time to launch application (seconds)	0.92
Free memory (KB) Initial - after startup	1017.26
License	(OSS)
Author / Company	Philippe Blanc
URL	`http://www.oyoaha.com/lookandfeel/`



First Name: Mauro
Last Name: Hornblower - Von Clausewitz
Address: 10287 North Turnpike Hill Road- Skookie (IL)
Street: Siegenflosserstrasse office
Code: 11 - 7665 Age: 0
Delivery: quick File: C:/temp/test/trial001/anotherdir
Status: preferred content available
XYZ Code: undefined automatic
Role: Adjunct Administrator [AE]
Way Back: 8758 - AE - it - 175 - 9
Accessor: Holztgew immediate
Q-Code: Q-0001ABVB987 Date: AA
Attempt: 01023 - 2345 - 98765
More: fax no: 001-800-001-0098765

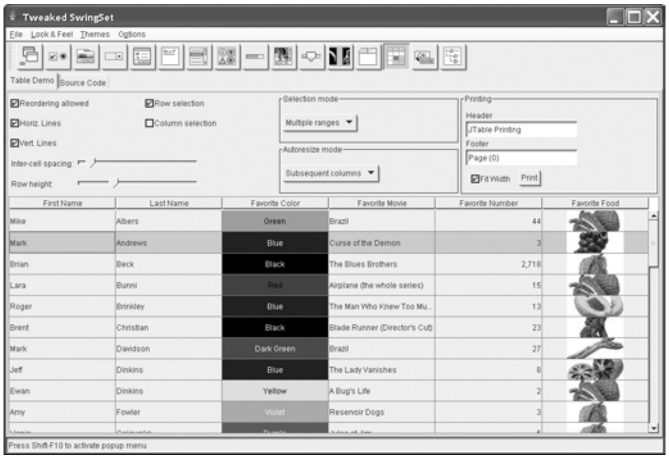
☐ gold member ☐ premium code ☐ assigned ☐ view ☑ relocate context ☐ strongly typed

Style: ☐ classic ☐ sport ☒ business ☐ casual

Post Giro: W0656
Sirloin: W9 - 876 - W253
Delivery: relentless - A
Account: Q-typed [876] ☐ view YY7 - AM - 1287
XYZ Code: [not available]
Alignment: available Date: 0 0
Sorbet: Strawberry lemonade with fountain top

Preferred Option: maybe, perhaps, maybe not
Account Item: 0
Sex: ☒ Male ☐ Female
Other Value: parachute
Filet Mignon: well cooked
Barbeque: model C11
Account: 00011286QEWR3435465757
XYZ Code: 8676-BVC Age: 49
Address: N Franklin Street 8576
Way Back: Punitgamer 298 Date: NA
Notes: some notes not editable

☑ send later ☑ validate
Preferred Del.
Multipart: ☐ redeem ☐ biased ☒ not ackn.


Tweaked SwingSet
File Look & Feel Themes Options
Table Demo
Source Code
Print
Press Shift-F10 to activate popup menu

☑ Reordering allowed ☑ Row selection
☑ Horiz. Lines ☐ Column selection
☑ Vert. Lines
Inter-cell spacing:
Row height:
Selection mode: Multiple ranges
Autoresize mode: Subsequent columns
Printing — Header: JTable Printing; Footer: Page {0}; ☑ Fit Width

First Name	Last Name	Favorite Color	Favorite Movie	Favorite Number	Favorite Food
Mike	Albers	Green	Brazil	44	
Mark	Andrews	Blue	Curse of the Demon	3	
Brian	Beck	Black	The Blues Brothers	2,718	
Lara	Bunni	Red	Airplane (the whole series)	15	
Roger	Brinkley	Blue	The Man Who Knew Too Mu...	13	
Brent	Christian	Black	Blade Runner (Director's Cut)	23	
Mark	Davidson	Dark Green	Brazil	27	
Jeff	Dinkins	Blue	The Lady Vanishes	8	
Ewan	Dinkins	Yellow	A Bug's Life	2	
Amy	Fowler	Violet	Reservoir Dogs	3	

Figure 11.20 Oyoaha 3.0

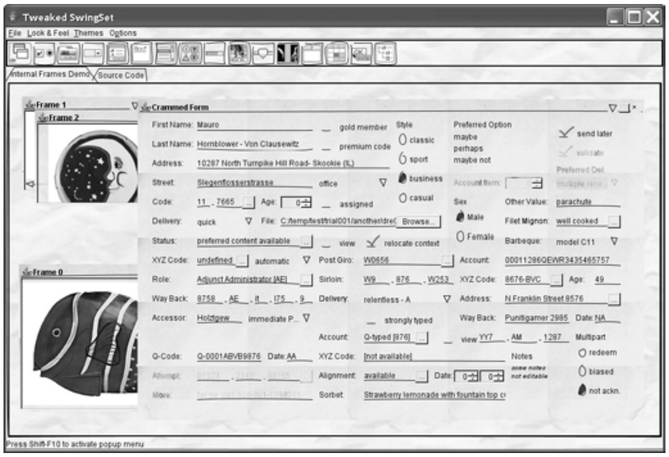
Napkin

Napkin is a simple look and feel that provides an informal and provisional appearance to Swing GUIs. It is not intended to be used with a final product, but only in development and during demonstrations to users. This enables developers to avoid committing to a given L&F until needed by highlighting the fact that the GUI is not ready.

AS it is a L&F that is not used in production, Napkin's performances lags behind other L&Fs, with a relatively large memory occupancy due to the many bitmap it uses, as well as long time to load and launch values. Given the intended use of this L&F, these are not problems, especially if the application is executed on powerful development machines. Napkin is a good example of the many possible uses of Swing L&F technology. See Figure 11.21.

Table 11.15 Napkin details

L&F name	Napkin Beta 0.07
Time to load L&F at startup (seconds)	0.95
Time to launch application (seconds)	1.85
Free memory (KB) Initial - after startup	2394.29
License	BSD (OSS)
Author / Company	Ken Arnold
URL	`http://napkinlaf.sourceforge.net/`



First Name: Mauro
_ gold member
Last Name: Hornblower - Von Clausewitz
_ premium code
Address: 10287 North Turnpike Hill Road- Skookie (IL)
Street: Siegenflosserstrasse
office
Code: 11 . 7665
Age: 0
_ assigned
Delivery: quick
File: C:/temp/test/trial001/another/dre
Status: preferred content available
_ view
✓ relocate context
XYZ Code: undefined
automatic
Post Giro: W0656
Role: Adjunct Administrator [AE]
Sirloin: W9 . 876 . W253
Way Back: 8758 . AE . it . I75 . 9
Delivery: relentless - A
Accessor: Holztgew
immediate P...
_ strongly typed
Account: Q-typed [876]
_ view YY7 . AM . 1287
Q-Code: Q-0001ABVB9876
Date: AA
XYZ Code: [not available]
Attempt: 01023 . 2345 . 98765
Alignment: available
Date: 0 0
More: fax no: 001-800-801-0098765
Sorbet: Strawberry lemonade with fountain top c

Style
○ classic
○ sport
● business
○ casual

Preferred Option
maybe
perhaps
maybe not

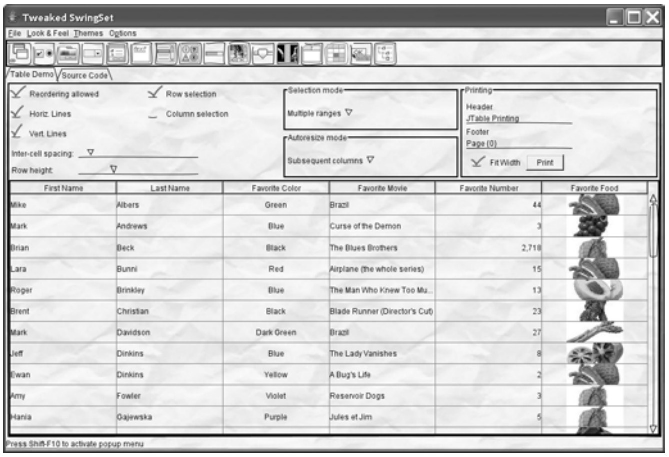
Account Item: 0

Sex
● Male
○ Female

Notes
some notes
not editable

✓ send later
✓ validate
Preferred Del.
multiple rece...
Other Value: parachute
Filet Mignon: well cooked
Barbeque: model C11
Account: 00011286QEWR3435465757
XYZ Code: 8676-BVC
Age: 49
Address: N Franklin Street 8576
Way Back: Punitigamer 2985
Date: NA
Multipart
○ redeem
○ biased
● not ackn.

Figure 11.21 Napkin L&F



✓ Reordering allowed
✓ Horiz. Lines
✓ Vert. Lines
Inter-cell spacing:
Row height:
✓ Row selection
Column selection
Selection mode: Multiple ranges
Autoresize mode: Subsequent columns
Printing
Header
JTable Printing
Footer
Page (0)
✓ Fit Width
Print

First Name	Last Name	Favorite Color	Favorite Movie	Favorite Number	Favorite Food
Mike	Albers	Green	Brazil	44	
Mark	Andrews	Blue	Curse of the Demon	3	
Brian	Beck	Black	The Blues Brothers	2,718	
Lara	Bunni	Red	Airplane (the whole series)	15	
Roger	Brinkley	Blue	The Man Who Knew Too Mu...	13	
Brent	Christian	Black	Blade Runner (Director's Cut)	23	
Mark	Davidson	Dark Green	Brazil	27	
Jeff	Dinkins	Blue	The Lady Vanishes	8	
Ewan	Dinkins	Yellow	A Bug's Life	2	
Amy	Fowler	Violet	Reservoir Dogs	3	
Hania	Gajewska	Purple	Jules et Jim	5	

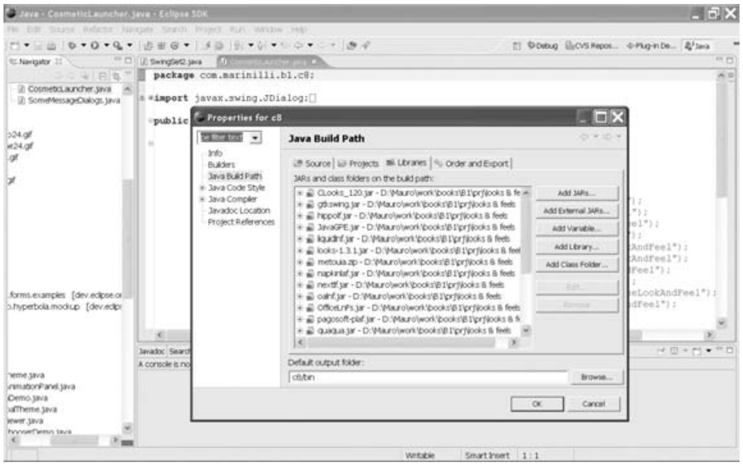
Press Shift-F10 to activate popup menu

Figure 11.21 Napkin L&F (Continued)

SWT Presentation

Although less flexible than Swing's counterpart, SWT also allows for the customization of the toolkit's visual appearance via native OS themes. This option is still poorly supported as of Eclipse 3.0. To enable Windows XP themes in SWT, a special manifest file is included in the same directory that contains the JRE that launches the application.

Figure 11.22 shows Eclipse on Windows with two different themes: XP (above) and Windows classic (below).



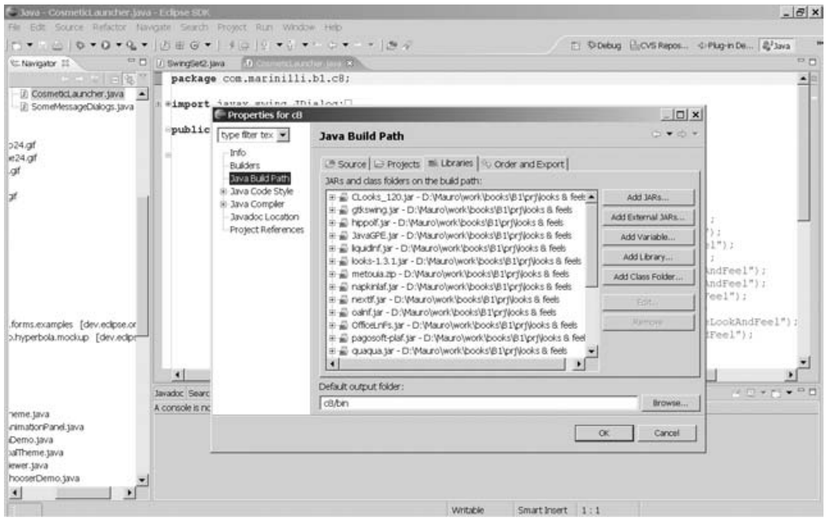
```
package com.marinilli.bl.c8;

import javax.swing.JDialog;

public
```

JARs and class folders on the build path:

```
CLooks_120.jar - D:\Mauro\work\books\B1\prj\looks & fe
gtkswing.jar - D:\Mauro\work\books\B1\prj\looks & feels
hippolf.jar - D:\Mauro\work\books\B1\prj\looks & feels
JavaGPE.jar - D:\Mauro\work\books\B1\prj\looks & feels
liquidlnf.jar - D:\Mauro\work\books\B1\prj\looks & feels
looks-1.3.1.jar - D:\Mauro\work\books\B1\prj\looks & feel
metouia.zip - D:\Mauro\work\books\B1\prj\looks & feels
napkinlaf.jar - D:\Mauro\work\books\B1\prj\looks & feels
nextlf.jar - D:\Mauro\work\books\B1\prj\looks & feels
oalnf.jar - D:\Mauro\work\books\B1\prj\looks & feels
OfficeLnFs.jar - D:\Mauro\work\books\B1\prj\looks & feels
pagosoft-plaf.jar - D:\Mauro\work\books\B1\prj\looks & fe
quaqua.jar - D:\Mauro\work\books\B1\prj\looks & feels
```

Default output folder: c8/bin

Figure 11.22 Eclipse 3.0 with different themes



```
package com.marinilli.b1.c8;
```

Java Build Path

JARs and class folders on the build path:

```
CLooks_120.jar - D:\Mauro\work\books\B1\prj\looks & feels
gtkswing.jar - D:\Mauro\work\books\B1\prj\looks & feels
hippolf.jar - D:\Mauro\work\books\B1\prj\looks & feels
JavaGPE.jar - D:\Mauro\work\books\B1\prj\looks & feels
liquidnf.jar - D:\Mauro\work\books\B1\prj\looks & feels
looks-1.3.1.jar - D:\Mauro\work\books\B1\prj\looks & feels
metouia.zip - D:\Mauro\work\books\B1\prj\looks & feels
napkinlaf.jar - D:\Mauro\work\books\B1\prj\looks & feels
nexttf.jar - D:\Mauro\work\books\B1\prj\looks & feels
oalnf.jar - D:\Mauro\work\books\B1\prj\looks & feels
OfficeLnFs.jar - D:\Mauro\work\books\B1\prj\looks & feels
pagosoft-plaf.jar - D:\Mauro\work\books\B1\prj\looks & feel
quaqua.jar - D:\Mauro\work\books\B1\prj\looks & feels
```

Default output folder: c8/bin

Figure 11.22 Eclipse 3.0 with different themes

11.10 Declarative GUIs with Java

Many projects aim to provide declarative capabilities to Java GUIs, mostly to express content, even if some projects also strive to provide a minimum amount of interaction and control behavior.

XML-based formats

Unsurprisingly, the largest family of declarative formats is based on XML. There are more than a dozen such XML schemas[20], with projects like JDNC[21], Mozilla XUL, Luxor, SwiXml, XUI, Beryl XML GUI, Purnama XUI, SwingML, Thinlet, jXUL, KoalaGML, WidgetServer, Gui4j, and XAMJ.

The Thinlet project is an example of this family of formats, a LGPL-licensed, tiny-footprint (39 KB) interpreter of Thinlet XML files. Thinlets can run in a Java 1 JVM, the default shipped with Microsoft Internet Explorer, and other J2ME profiles, and don't require Swing. Figure 11.23 shows a sample GUI demo using Thinlet.

20. For a quick comparison, can see: http://xul.sourceforge.net/counter.html.
21. JDNC is discussed briefly as an alternative implementation for the example application in Chapter 14.

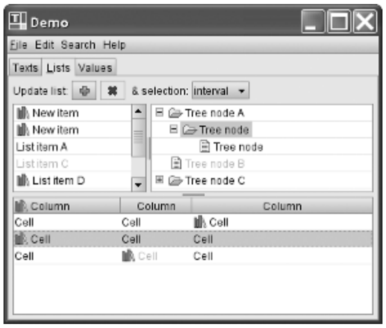


Figure 11.23 Thinlet demonstration

An extract of the source file that generates the GUI in Figure 11.23 is shown in Listing 11.1 below.

Listing 11.1 The `demo.xml` file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<panel columns="1" gap="4">
  <menubar weightx="1">
  <menu text="File" mnemonic="0">
   <menuitem text="New" icon="/icon/new.gif" mnemonic="0" />
   <menuitem text="Open..." icon="/icon/open.gif" />
   <menuitem text="Save" icon="/icon/save.gif" />
   <menuitem text="Save As..." icon="/icon/saveas.gif" />
   <separator />
   <menuitem text="Page Setup" icon="/icon/pagesetup.gif" />
   <menuitem text="Print" icon="/icon/print.gif" />
   <separator />
  …
  </menubar>
  <tabbedpane selected="1" weightx="1" weighty="1">
    <tab text="Texts">
      <panel columns="5" top="4" left="4" bottom="4" right="4" gap="4">
        <label text="Find in the text:" mnemonic="10" />
  …
    <tab text="Lists" mnemonic="0">
      <panel columns="1" top="4" left="4" bottom="4" right="4" gap="4">
        <panel gap="4">
          <label text="Update list:" />
  …
```

Another approach that avoids the use of Java on the client altogether is to take advantage of other presentation technologies, such as Macromedia Flash, that are installed on clients as Web browser plug-ins. SWF bytecode can be generated dynamically on the server side by tools such as OpenLaszlo, using servlet technology, or Flex. Both these approaches make use of XML-based user interface languages in which the XML is generated and prepared on the server side, compiled, and send to the client's Flash player.

11.11 Summary

This chapter discussed some of the most popular technologies and products for developing Java GUIs. We introduced the issue of evaluating an OSS in general, and specifically for the purpose of creating Java GUIs. Major Java GUI technologies and tools currently available were discussed and compared, including development aids, third-party components, utility libraries, and presentation technologies.

12 Advanced Issues

This chapter deals with various issues that are encountered less often by developers. Rather that being classified as 'advanced,' these topics can be seen as solutions to specialized problems that seldom occur in average GUIs and would not be of interest for the average reader, but that are still useful to consider, as they apply to a wide range of real cases.

The chapter is organized as follows:

12.1, Building on top of existing libraries discusses some of the issues related to creating APIs and frameworks, also taking advantage of usability.

12.2, Memory management for complex GUIs illustrates problems and possible solutions with practical examples.

12.3, Restructuring existing GUI code discusses various issues related to renewing and restructuring existing Java GUI code.

12.4, Exploiting technology proposes alternative uses of some Java GUI technologies.

12.5, Domain-specific and Little languages discusses the use of this technique for Java GUIs.

12.6, The future of Java GUIs attempts to forecast the future of Java GUI technologies.

12.1 Building on top of existing libraries

A frequent habit of designers is to create reusable classes in order to save development time in future projects. Even if full reusability is often an unfulfilled dream, there are certain common patterns, as we have seen throughout this book. We have discussed some possible strategies, focusing mostly on more reusable patterns. Here we will explore another approach to code reuse for non-trivial GUIs: to formalize the support for higher-level attribute implementation into reusable classes.

We mentioned this issue when discussing OOUIs. Imagine that we have to develop a business application that needs many data structures that are in turn composed of simpler attribute data such as strings, integers, files, and so on. We could assign a great deal of common behavior to these attribute classes, sparing developers from writing many lines of service code, for example to implement persistence, or for sophisticated content assembly, and so on. Providing a high-level library for such utility functions will also help to speed up the software design process. This is not a new idea. Standard libraries and much corporate effort have devised this strategy to simplify GUI development, especially for predictable applications such as business management, database-oriented domains, and so on. We will explore this idea in the next section.

Attributes

A common solution for easing the implementation of non-trivial applications is to think of a given business class as a compound of high-level attributes. Such attributes can themselves be composed of other attributes, and so on, implementing a Composite pattern (Gamma et al. 1994). Basic attributes (which we will call *fields*) will wrap basic data types, such as strings, dates, and numerical intervals. Apart from wrapping business data at a higher abstraction level, attributes provide several useful services.

Usually attributes are responsible for handling the following services:

- Providing GUI interfaces for accessing the business data.
- Providing automatic mechanisms for default values, message bundles, preference storage, serialization, and so on
- Negotiating appearance and layout with their composite parents (similarly to Swing's and AWT's components).
- Providing a business logic layer for administering their data.

Many proprietary or publicly-available GUI frameworks exist, such as JFace for the Eclipse platform, that – with different degrees and perspectives – take advantage of the attribute concept.

By way of demonstration, will discuss a simple, lightweight example implementation of a possible attribute framework. The class diagram in Figure 12.1 shows the design of such a library.

Adopting the Composite pattern, we could have simple 'leaf' attributes, our fields, and compound attributes, represented by the abstract class `CompositeAttribute`. Fields are the basic building blocks we use to build complex data structures. Each field has a name and other common properties that are used for initialization, such as a default value, for example, or for GUI purposes, such as tooltip and a mnemonic key for example. Taking advantage of the `Viewable` interface[1], each attribute can provide different views of its data suitable for aggregation into a larger view in its parent's composite attribute.

1. See Chapter 15, *The Viewable interface* on page 538.

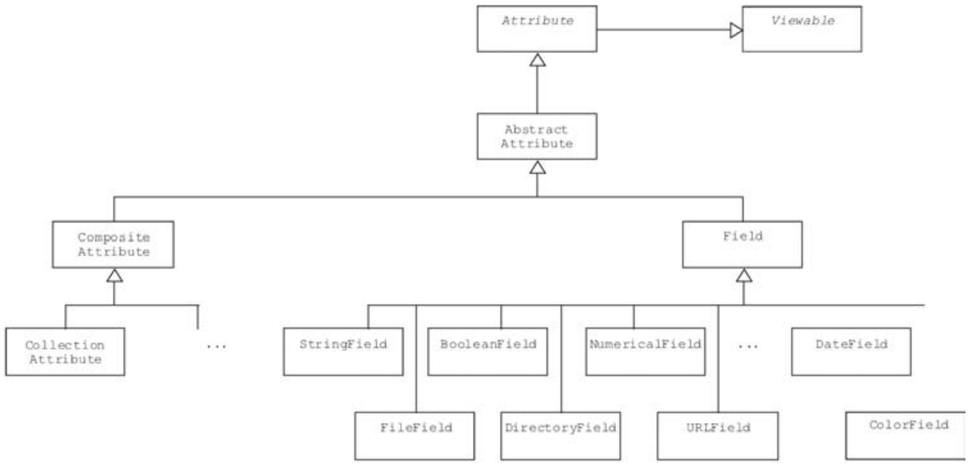


Figure 12.1 A complete framework for attribute management

Among developers, attribute frameworks are often seen as suspect, given the additional complexity they bring to a design and the subsequent application development. Usually such frameworks justify themselves in terms of future reusability (but we know how vague this can be) and faster development. But like any other class library, they need to be properly mastered.

This solution is needlessly powerful for simple GUIs in which there is no need for multiple views of the same attributes. In such cases the attribute framework can be greatly simplified. Avoiding the use of multiple views and a fully-fledged MVC architecture leads to the design shown in Figure 12.2 below.

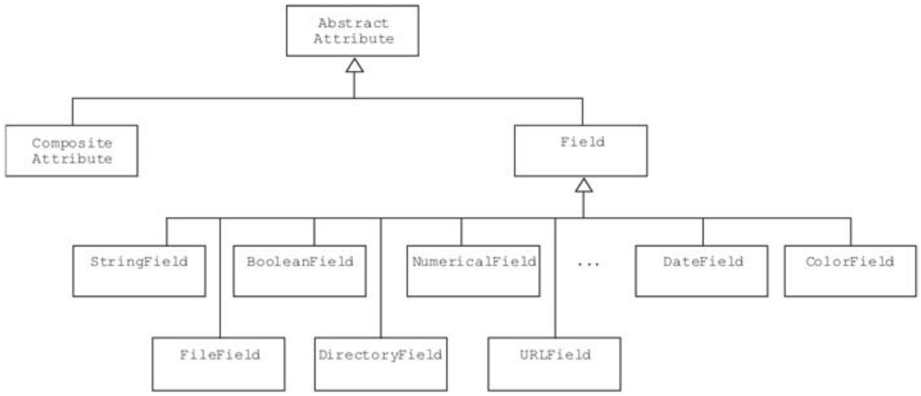


Figure 12.2 A slimmed-down class diagram for attribute management

Let's look at this second solution in detail, sketching out a skeleton of a basic attribute framework. Our attributes will provide the following services:

- Preferences persistence.
- Default values.
- Naming and other labeling facilities, such as tooltip, field name, and mnemonic key, obtained from the message bundle properties file.
- Simple graphical views, with basic commit/rollback behavior similar to the `Viewable` interface.

An implementation of the `AbstractAttribute` class is provided with the code bundle for the book.

Attributes can carry domain-dependent information that is needed in the content layer, such as validation constraints, or whether or not an attribute is mandatory, for example. This enables a clear and systematic separation between the business domain and the other functional layers to be obtained.

The `createLabelFor` method is a convenient method for obtaining a label component from the attribute. The `createComponent` method is implemented by `AbstractAttribute`'s subclasses.

The persistence mechanism, which is used mostly for user-customized data, has been implemented using the preferences mechanism present in J2SE from Version 1.4 onwards. The text messages have been managed using properties files directly, to ease localization, relying on the `ServiceManager` class.

Concrete subclasses of `AbstractField` (`BooleanField` and `StringField`) are provided with the code bundle for the book. The `Employee` class in the code bundle is an example of a composite attribute that represents data for an employee. In our simple implementation, an employee is composed of three elementary fields:

- `name` – the employee's first name, implemented with a `StringField` instance.
- `surname` – the employee's family name, implemented with a `StringField` instance.
- `senior` – whether or not the employee is a senior worker, implemented with a `BooleanField` instance.

The `Employee` class is itself an attribute, so it can be combined with other attributes to create bigger attributes, such as a `PayRoll` class, or standard compound attributes such as a `CollectionAttibute` for modeling sets of employees.

In our implementation we choose to adopt static encapsulation – that is, the attributes of a composite class are instances of variables of that class – instead of dynamic encapsulation, where a collection variable holds all the attributes. This means that a lower degree of automation is possible. Methods such as `doCommit()` or `doRollBack()` therefore do not need to invoke any sub-attribute. The `doLayout()` method is different, because a semantic notion of each sub-attribute is needed to achieve an effective layout.

Static attribute encapsulation has its own advantages. The main ones are readability – inspecting the class source is enough to understand its attributes – and simplicity, as the attribute instances look like normal class members.

Finally, the `Example` class creates an `Employee` and requests it to show its contents on the screen in a test frame. The final result is shown in Figure 12.3.

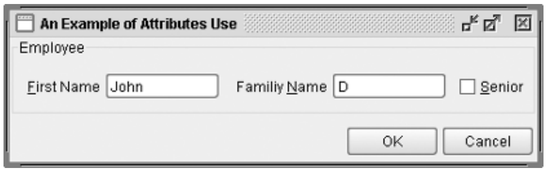


Figure 12.3 The Example class shows an employee (PGS)

The example implementation is very simplistic and lacks many useful features, for example the ability to open the attribute's visual component in read-only mode, better persistence support, localization, and so on.

Roll your own framework

Sometimes there is a need to extend an open source library that falls short in the features needed for the current project. At other times we might want to collect utilities and code we keep on writing over and over again for every GUI into a coherent API, or we are called on to provide some specialized framework for an organization, and so on. In all these situations, developers need to wear the API provider's hat rather than that of the client.

Java GUI developers are often faced with this kind of task for two main reasons. Reusable GUI libraries aren't so reusable in practice, given the sheer number of requirement our 'reusable' code should fulfill for real-world GUIs. For example, imagine that you have found the 'perfect' calendar widget for your applications, but that it unfortunately doesn't fit within your existing GUI design because it cannot be inserted in a lightweight pop-up window like all your other choosers.

Open source libraries *encourage* developers to tweak someone else's source code and enhance it, providing of course acknowledgements are given and license compliance is met.

Building GUI code for other developers instead of final users is a gratifying experience that need extra technical care in its details. Two very useful documents on this perspective are *Evolving Java-based APIs* (Des Rivières 2000) and, specifically on the client side, *How to Use the Eclipse API* (Des Rivières 2001). There is much more material available on this topic, but this is out of our scope.

Designing usable APIs and frameworks

This subsection discusses the idea of applying usability principles to general API and OOP framework design. This is of course not a new idea, but the original approach taken here is to leverage sound usability and HCI advice taken from the first part of the book to guide us through our revisit of API design as a product.

Unsurprisingly, GUI design and API design have many points in common. Both have users of many types and levels of experience, API users being mainly developers. Both design processes leverage devised metaphors and abstract concepts embodied in the design, be it a GUI design or an API, to solve user's needs effectively. On the other hand the user experience of the product – the new API to be designed – is shaped by many factors: class and method names, the design patterns employed, concepts and abstractions, documentation, and the general 'feeling' of its use, as perceived by the developer.

As examples of existing APIs we refer to the APIs and frameworks for GUIs such as Swing, Netbeans API, and Eclipse API, as the reader is likely to be more familiar with these. Nevertheless, the ideas discussed here apply to any kind of API and OOP framework, not only graphical ones.

Here is a list of the main GUI design concepts, which briefly discusses how they translate to API design. The term *user* and *developer* are used here as synonyms.

- Focus on your users:
  - Design for different types of users. Usually you'll have novice users, expert full-time developers, and possibly also knowledgeable, part-time developers as well. Each class of users has its own needs and priorities.
  - Provide a user-centered, task-oriented, and context-aware design. Performing a classic task analysis, adding context and user data, seems an almost trivial suggestion, but it will shape the final API design tremendously. Think about what the main tasks your API will solve are, with user goals, how developers carry out these task now without your API, their context of work, task breakdown decompositions, ethnographic study of developers in their work environment, and so on.

For example, a task could be to create and customize a data-bound table for expert SWT users, for adding to an existing panel. But beware – such an approach can bring a subtle but conceptually devastating consequence: API designs, just as GUI designs, depend heavily on their context of use and on their intended users. So there is no such a thing as perfect API for all seasons.

- Ensure consistency and predictability. This can be also understood in terms of lowering the use of long-term memory (LTM) and using short-term memory (STM) as much as possible. To ease management of STM, use no more than 7±2 items (that is, parameters in methods, methods in interfaces, and so on). Consistency involves the use of design guidelines, analogous to GUI design guidelines, that prescribe how errors should be handled, patterns to use and those to avoid, naming conventions, and so on.
- Design effective metaphors and concepts to:
  - Solve user problems. This is implicit, but is important to point out. Unless your API won't solve problems, it won't be worth developer's time using it.
  - Behave as expected by its intended users. For example, if you are designing a domain-specific language (a 'little language') don't use an exotic, fancy syntax with which Java developers are not familiar.
  - Keep close to the application domain. Don't use first-order logic for a dynamic layout API, even if it looks cool. Concepts too distant from the domain and users should be avoided. For example, even if quantum physics provides a wonderful and elegant metaphor for (say) a GUI toolkit, hide the internal details of the implementation, so that users don't need to study physics before putting together a form using your API.
  - Communicate the API effectively. Documentation, training, and other forms of learning are very important, but ultimately what makes an API (or a GUI design) usable are the concepts themselves.
- Test your API with user representatives not previously exposed to the design process for usability – classic usability testing – and effectiveness – productivity: how easy is to achieve the required goal with the API? You might want to test also for flexibility – can expert users tweak any aspect of the framework – and future modifications – what are the unforeseen needs of our users? Developers are like GUI users: putting them in control of the GUI will make them feel better. How can your API be modified to accommodate them without degrading its architecture? Releasing a badly-tested API implies many modifications on a tight schedule that will deteriorate the initial design.

- Make the API pleasant to use and easy to learn.
  - There is no need to torture developers, who are stressed enough already. Designing your API to provide early gratification is a good technique for keeping them interested and stimulated.
  - Hide unnecessary complexity, for example by separating interface from implementation[2] or by providing a carefully thought-out class hierarchy.
  - Provide default behavior, to employ it usefully with minimal effort, and ready-to-use, predefined behavior for common cases.
  - Provide informative feedback. Learning an API by trial and error is a common habit among developers. Providing useful feedback, such as the Amazon Web Service API feedback in the example application in Chapter 13, eases API comprehension and developer productivity.
- Design in error prevention and management. Many techniques can be used here:
  - Use immutable objects when possible, providing their mutable counterpart when performances or usability are a concern.
  - Decide what to do with null. Despite appearing a trivial suggestion, many 'home-grown' APIs fail to define such a basic issue consistently and effectively, causing a number of minor weaknesses, and opening the door to some annoying errors. By generalization, also consider using Special Case design (Fowler et al. 2003).
  - Design the API as you might design a GUI, to try and make errors impossible. For those errors that cannot be eliminated by clever design, provide both a 'baby-proof' path – safe objects that only allow a protected and safe subset of all possible data and behavior – and a 'pro' path, allowing for maximum freedom and customization power, but also allowing inconsistent and dangerous behavior.
  - Devise a comprehensive failure strategy, as discussed in Chapter 5.

12.2 Memory management for complex GUIs

A common problem with large Java applications is managing the memory needed during execution. The situation can be complicated when such applications need to run for a long time, requiring more sophisticated memory policies.

The simplest solution for such a class of Java applications is to provide a means for the end user to control the JRE's garbage collector directly. Clearly this is feasible only when the typical user population can be assumed to be knowledgeable enough to manipulate such a low-level feature like the garbage collector. This

2. See Chapter 7.

is the case in all major development environments implemented in Java, such as JBuilder Eclipse or Idea.

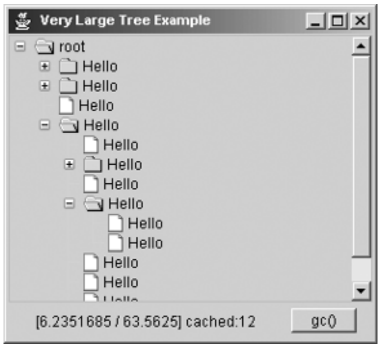
When providing access to the garbage collector is not feasible, for application-dependent reasons, for end user characteristics, or whatever other reason, you have to employ some ad-hoc strategy. A common solution is to provide a low-priority thread that takes care of memory management, invoking the garbage collector when needed.

There is another reason for adopting such an ad-hoc approach. Usually the garbage collector takes some time to perform its operations, and this appears to the end user as if the application is freezing for a moment. This kind of pause, which can appear random, can be unacceptable. In such cases it's advisable to 'pilot' garbage collection at a specific time, such as just before a heavily interactive session. In general, there could be too many different situations in practice to discuss them all here, but we can suggest a simple test situation that can be adapted to manage a wide range of practical cases.

Some applications are required to handle vast amount of data, such as the large datasets shown in table components of database clients GUIs. The amount of data to be viewed is largely decided by the user. This poses some problems, because memory management should be adapted to the current user interaction.

A practical case

Imagine a tree component that is bound to a large data source that can grow almost infinitely. This could be the case for example with a client application that shows data from a remote source that can supply a very large amount of information. Clearly, some solution is needed to make this tree component manageable without compromising GUI interaction. A screenshot is shown in Figure 12.4.



[6.2351685 / 63.5625] cached:12

Figure 12.4   A very large and expensive tree (Tiny)

Below the tree widget the demo application, available in the source code bundle, shows the currently-available memory, overall memory, and the number of current cached nodes.

The `Node` class represents a single tree node that gobbles up a large quantity of memory (0.5 MB of dummy data), so a few new nodes can consume almost all available client memory.

The `VeryLargeTree` class is the cornerstone of the example. This class invokes the `setRowHeight()` method with a fixed size, and `setLargeModel()` with `true` on a Swing `JTree` instance, to prompt the object to use an alternative code path optimized for models with large data sets. One important consequence of this setting is that the model will be queried more often – clearly in order to reduce the tree's cache size.

An interesting method in the `VeryLargeTree` class is `treeWillExpand`, part of the `TreeListener` interface. This method is invoked whenever a folder node of the tree is going to be expanded. In our fictitious example, we simply fill the folder node with data obtained from the dummy server. Just before the tree expansion takes place, this method is invoked, and a new request is issued to the server by means of a queue of worker `Runnable` instances.

The example uses a lazy instantiation mechanism for the tree nodes. Whenever the user expands a node, the branch is populated with fresh data from the server. This avoids useless memory allocation for those folders that the user will never explore, but we pay for it with a less interactive GUI.

Another section of code in the example is the `MemoryManager` class, which implements a simple caching mechanism, the `Cache` inner class. The method `removeEldestEntry()` removes the eldest entry in the cache. The `findAllChildren()` method recursively finds all the child nodes of a given element that are candidates for removal whenever the node is deleted by the tree. The `MemoryMonitor` class controls the `MemoryManager` instance, and shows the memory state through a status label. This class provides a way to activate the JRE's garbage collector explicitly.

Finally, the `DummyServer` class simulates a remote server that returns data nodes with an unpredictable latency, simulated by a pseudorandom delay via the `simulateIOLatency()` method.

A simpler and 'lighter' implementation, in that it takes advantage of special reference types, part of the J2SE standard API, is also possible. The `SimpleLargeTree` uses the `WeakReference` type, contained in `java.lang.ref`. This makes it possible to implement 'soft' reference types that are automatically cleared out when the JRE runs out of available memory. This means that programmers don't have to bother too much about cache maintenance, because reclaiming memory held by weak references will be done automatically by the JRE. This simplifies coding, but shields programmers from tight control over memory management.

Which of these strategies is best will depend on the situation. Other solutions are possible, using other special types of references provided in the `java.lang.ref` package that allow for more control over garbage collection.

12.3 Restructuring existing GUI code

We have discussed how to apply our basic reference architecture and other techniques to build high-quality GUI code from scratch. Sometimes, however, the opposite problem arises – existing GUI code must be restructured in a scenario that differs from development, for example code that was written elsewhere, that was written more than a few years ago, and so on. Sometimes it's cheaper to throw legacy GUI code away, using it only to capture requirements for a new implementation, while at other times this choice is quite hard to make in the general case – it could be wiser to keep it, even in the form of an unmaintainable patchwork of code.

Complex or large GUIs built over the years have usually absorbed so much change in their source code under the influence of tight deadlines, different developers, and so on, that they appear very hard to maintain, especially when the initial developers are no longer available. Despite that, building such GUIs from scratch can prove to be too dangerous an enterprise, and step-by-step refactoring could be the wiser approach to enhancing the quality of the code base while keeping the product 'alive' without incurring release delays.

Several reasons for modifying such sources can exist:

- It must be ported to newer technologies, such as porting an old AWT applet to Swing, or renewing an old open source library.
- The architecture and the overall software quality need to be enhanced, for example for performance optimization.
- Routine software maintenance is required, where the goal is to intervene in a specific portion of existing code.
- New features must be added to the application. This implies a deep understanding of at least some parts of the code, to modify it without disrupting the rest of the application.

Restructuring may range from applying general-purpose techniques, Java-specific manipulations, or other processes, such as applying coding guidelines. Nevertheless, some general principles apply – it's important to:

- Understand the legacy code, or at least enough of it to carry out what is required. Getting a 'grip' on old code may be hard, even for circumscribed and specific pieces of software like Java GUIs. No matter how deep your understanding of Java GUIs and your toolkit knowledge, understanding and modifying convoluted old code successfully is always time-consuming.

- Work with a clear objective in mind, such as enhancing performance, or porting the code to a newer technology or a different layering architecture, and so on.
- Develop a clear plan of what is going to be done, and deliver it one step at a time.

As any programmer knows, modifying existing code is a complex task, and an exhaustive discussion is out of scope here. One basic point, though, is the relationship between the overall restructuring cost and the fraction of modified code. Figure 12.5 shows data collected from NASA software projects (Selby and Porter 1998). Even if this data focuses on code reuse[3] rather than code restructuring, it still shows some interesting facts.

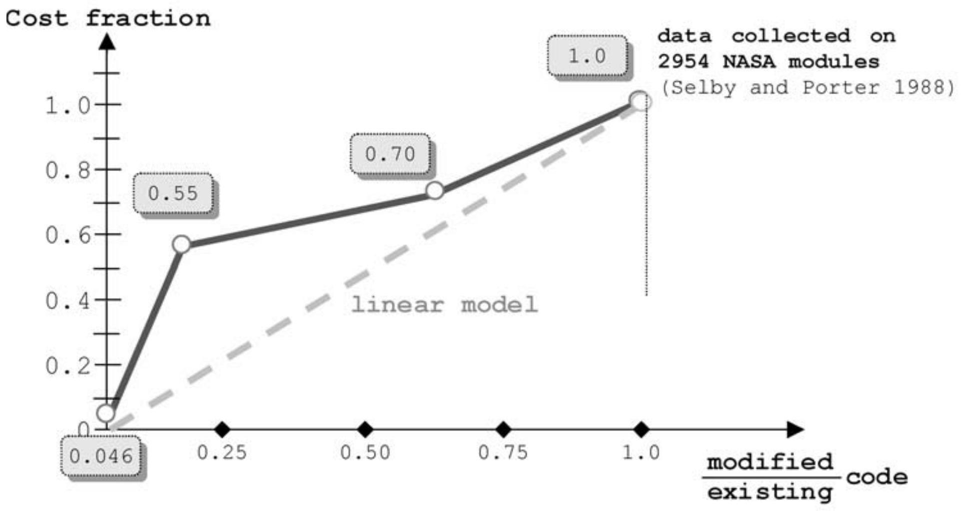


Figure 12.5 The cost of modifying existing code

3. Even if OOP was meant to cut down such high reuse costs, it cannot spare today's programmers from code restructuring. Writing poor code, or the need to renew code from time to time, is still an open problem in the software industry.

First, whenever we embark on a code restructuring project, we always pay a toll, even if we don't touch a line of code – see the first point on the left, which shows that preliminary costs amounted to 4.6% of the cost of initially building the code. Such costs comprise analysis, code comprehension, retrieving meaningful documentation, and so on.

Another interesting observation is that code modification has a nonlinear cost associated with it as the proportion of modified code grows. This is intuitive – if programmers don't know the code very well, even small modifications in the early phase may cause the whole application to behave unexpectedly. This is why the first segment of the graph is much steeper than the others, while the last part is less expensive (less steep) because by this stage developers are familiar with the code, allowing additional changes to be made more cheaply.

> This discussion of code modification costs doesn't take Agile coding approaches into account. Having a solid suite of tests for code that is to be modified, and proceeding in small iterative steps, as discussed in Chapter 5, ensures a cheaper and less risky restructuring process. Alas, such practices were not the norm even as recently as a few years ago.

Essentially, GUI code restructuring involves the following activities:

- Code analysis and comprehension, the necessary prerequisite for all other manipulations.
- Code refactoring, as we introduced in Chapter 5.
- Code porting to newer features, such as to new libraries, new deployment technologies, and so on. In general this kind of porting is not painless. and is hard to estimate for *a priori*.
- GUI-specific higher-level manipulations, such as introducing particular abstractions and architectural strategies.
- GUI-specific lower-level activities, such as providing internationalization support, or increasing the performance of some GUI code.

It is therefore important to tackle GUI code renewal and porting incrementally, especially complex GUI code. Once the process has been started, one should bear the high initial costs of modifying existing code effectively. A sound approach is to take advantage of the vast literature on the subject, and consolidate best practices about code enhancement.

Porting an old applet – a case study

A real case can help to illustrate some general issues about maintaining existing GUI code. The Rubik Cube applet shown in Figure 12.6 on page 461 was developed by two students of mine[4]. At one point the program was completed: it allowed users to play the Rubik's Cube puzzle by means of direct manipulation, rotating and manipulating the cube by mouse dragging in a virtual 3D space. It was implemented by interacting with low-level canvas and mouse events, pure AWT code, desirable for an applet that, as a result, doesn't need 7 MB additional downloads for the JRE.

A year or so later another group of students came with the idea of building a simple general framework for solving Rubik's Cube. Basically the idea was to provide heuristics, or even a small AI planner[5], to drive resolution, or at least some sort of tip for a player stuck at a difficult point. Clearly the old AWT applet was the perfect fit for the GUI of the new program. I handed the students all the documentation for the old applet and they started happily working on the project.

It turned out that the simplest and most effective GUI for representing moves in this domain was a tree widget. This posed some problems, though, because there are no native tree widgets for AWT. Instead of resorting to one of the various AWT trees available on the Internet, the idea of porting the old AWT applet to newer technologies, such as Swing, was becoming more compelling, which in turn would have fostered a whole host of new technologies, such as Java Web Start, standard help support, and others.

From an implementation viewpoint, referring to the classification introduced in Chapter 3, the applet is made up of a custom component tree for the moves (on the left-hand side of the applet's display area) and an ad-hoc component for the cube representation on the right-hand side.

A screenshot of the port of the old code to a Swing applet is shown in Figure 12.6. The tree on the left-hand side represents the moves computed by a resolution algorithm that takes advantage of heuristics. By double-clicking on a move node in the tree, the move is directly performed on the 3D representation of the cube.

4. Various people were involved, at various levels of commitment – I am sorry if I can't cite their names.
5. An artificial intelligence planner is a tool that plans domain-dependent steps automatically in order to solve a problem using AI techniques.

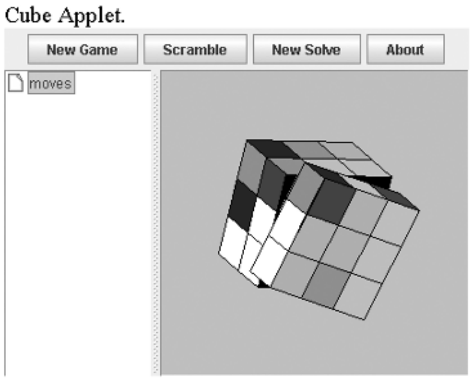


Figure 12.6 A Rubik's Cube applet (Ocean1.5)

The option of throwing everything away and rebuilding from scratch wasn't feasible because of the high cost of building the ad-hoc Cube component. In other situations, though, this could be the best choice (and not just because I was called in to consult on it). When old code is relatively easy to rebuild with a newer technology, the simplest way is just to use the running application as a black-box prototype embodying a given set of requirements for the new program, while throwing away its implementation.

We considered the available possibilities. Let's recap them, organized by deployment means:

- Java applet:
  - AWT applet, running in any browser. This is the simplest solution for deployment, and it is almost straightforward in this case, but has some drawbacks: AWT lacks a tree component, which would need to be created in-house or bought from a third-party vendor. In the long term, when expanding the applet further, we might be forced to switch to more powerful technologies, and forced to migrate an expanded code base.
  - Swing applet, running in any browser before Java plug-in installation. This solution provides more benefits in the long run. All major libraries and utilities are built for Swing rather than AWT (help support, layout manages, and so on).
- Java application:
  - Standalone AWT application. Similar to the case of the AWT applet configuration, even though it wastes the most important strong point of AWT: ease of deployment. For this reason it's less attractive than the AWT applet solution.

- – Standalone Swing application. While a number of deployment means can be devised, we discuss JNLP here, mostly.
- – Standalone SWT application. This is the same as the AWT case, although in this case SWT provides a standard tree component that works well with large tree models. On the other hand, developers not familiar with SWT need further study of the technology, and this can be an important practical hurdle. This is even truer for ad-hoc components such as the cube panel, which require deep knowledge of the underlying graphics libraries. Luckily, SWT's low-level graphics rendering model is close to AWT's.
- – Native application created from pure Java code, for example using GCJ with SWT libraries. This could be an interesting possibility, but we should sacrifice pure Java deployment for a number of (limited) native executables. Given the current context – non-for-profit educational software – we abandon this choice, even if it would have been very interesting in its runtime performance for large puzzle solution plans.

As you can see, the array of possible configurations is complex even for such a simple case. One possible pitfall lies in the programmers' experience. Developers not familiar with the target technology can become unexpected and dangerous obstacles in a porting process.

We opted for the Swing applet solution, which appeared to provide the best cost/risks/results ratio, although with a subsequent porting to a JNLP application in the longer term.

We started from the first step, porting the old AWT applet to Swing. The idea was to solve the most basic problems first, lowering risks and fostering many other porting steps[6]. While porting graphics from AWT to Swing, essentially migrating from the old `Graphics` class to `Graphics2D`, was easy, some unexpected problems arose along the way, all related to incorrect assumptions made by the earlier developers, such as too low-level code. User interaction such as mouse dragging wasn't working as expected in the Swing porting, and in-depth corrections were needed to make things work. We will get back to this point in a later section that discusses tips for ensuring greater longevity for Java GUIs.

Applets are still among us. Even if they didn't fulfill the triumphant vision of Java's early days, there are still a lot of them around, ranging from sophisticated, commercial software to educational, scientific simulators, video games. and so on. In many companies it is not unusual to find bloated applets – all the rage at the end of '90s – still on duty in intranet business applications.

6. See the ranking of such steps in Chapter 5.

When choosing the right client configuration, you should consider several factors:

- The intended user population and operating scenario. Will an Internet connection be needed? Is one available, and to what extent can it be relied on? Do the conditions and scenario prevalent when the code was first implemented still apply today? A new port of an old Java GUI is a perfect opportunity to consider such extra-functional issues.
- The cost of filling the gap between the old and the new technologies, including the porting of higher-level runtime models and all the work involved.
- Deployment means. This involves deciding between applet, JNLP or a 'plain' application, or even native executables (and their related family of installers), or other possible arrangements.
- The developer's skills. Obviously developers are more productive and effective when working with known technologies.
- The business model and other organizational constraints. In our previous example we were developing a non-for-profit applet with no demanding timeline, but in industrial scenarios the situation could be much more complex, and such constraints could impact heavily on the chosen porting strategy.

In conclusion, Java client technology is still evolving after more than ten years, and the many possible choices available on the market demand a clear view and a careful decision-making process by lead developers and architects.

Long-life GUIs

Despite the heroic commitment of Sun to supporting compatibility in the past, it is still possible to experience glitches and unforeseen changes in behavior when porting applications written for a given JRE to a newer version.

The problem is that compatibility can be ensured only at the API level. A performance enhancement in Swing's internals, such as resource loading, or small enhancements in the way some details of the low-level event pump are handled, *should* be transparent to client applications. This might not be case with software built without attention to long-term maintenance, however. We are not talking about good design and architectural details here, but about a clean separation from low-level behavior. The main source of incompatibility with a newer Java release lies in incorrect assumptions hard-wired into the code, making it dependent on obscure and undocumented implementation details in the technologies used in the application (for example the GUI toolkit). This is the case with the order of execution of some low-level operations, for example. These details are meant to be internal to the library used by an application, and relying on them will jeopardize the stability of code in the long term.

Even harmless code such as that for detecting specific mouse configurations, to support right-clicking on a tree for example, can work well with old JREs but start to behave bizarrely with newer versions. Other problems could arise as well, such as unfortunate variable naming – for example `enum` as a variable name plus a switch to JRE 1.5 or newer, or the like – but these are usually easily solved compared with the kind of low-level incompatibility often found in graphics-oriented code such as is used in video games or direct-manipulation GUIs.

Incorrect assumptions about specific thread timing combinations, and other incidental situations that make the application work but are not formally documented anywhere, can cripple code when executing it with a newer JRE. Unfortunately these sorts of issues are hard to detect at coding time, and developers are always uneasy about getting rid of hard-won code that works – perhaps a little murkily and with a couple of low-level hacks – but that works nevertheless.

As a rule of thumb, it is always better to resort to code that follows formally-documented features, to make a GUI independent of low-level, empirically-proven details.

Providing new deployment support

We discuss the issue of deployment separately because it is a common theme that can be handled easily by following some simple steps.

The JNLP protocol is suited to the deployment of Java applications via Web browsers. Rich Java clients usually take advantage of the Java Network Launching Protocol (JNLP) for launching and deploying Java application over the Web. This protocol works by means of special XML files (`.jnlp` files) that instruct the JRE how to deploy the application. This is done through a special launcher application on the client that is bundled with every JRE.

Once your Java application is ready for deployment, you publish its JAR files, together with the special JNLP file, on your Web server. Your customers only have to click on the link to the JNLP file to launch it automatically without any extra intervention. Actually JNLP does much more than this – interested readers can find more details on Sun's Java Web Start site or in (Marinilli 2001).

The specific case of porting to a newer deployment technology is interesting because it is a common situation that luckily is easy to manage in practice. The main point about porting applications to JNLP concerns external resource loading. The JNLP protocol works thanks to the J2SE class loader mechanism – application resources such as icons, property files and the like would not be accessible if loaded by any other mechanism. This can be easily demonstrated by substituting all external access with the following idiom:

```
URL res = this.getClassLoader().getResource(name);
    if (res!=null)
      // use the resource URL as needed
```

Remember that you should also fix your development environment, as it might now fail to load external resources if not properly set up. Here again a wise software architecture, one that gathers all external accesses into a minimum number of places, such as the Service layer implementation proposed in Chapter 7, can greatly ease the porting effort.

There are basically three different ways to deploy and subsequently manage your software in J2SE: by using applets, taking advantage of the Java Plug-In facility, by using JNLP-deployed applications, as discussed above, or by simply providing your own deployment solution, for example supplying your customers with installation CD-ROMs.

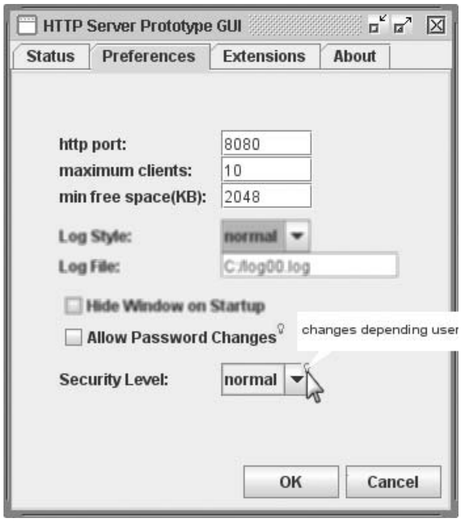
It is worth making a final point about applet deployment porting, useful for the many corporate applet-deployed applications still around. Applets are container-managed programs, developed to run in an applet container. This means that they have an underlying lifecycle model that is too simple for any but limited application scenarios, covering only `init()`, `start()`, `paint()`, `stop()`, and `destroy()` methods.

Apart from most business applications, today's applets are usually strongly graphics- and interactivity-oriented, such as video games or simulators, and this in turn makes them lower-level oriented, employing low-level GUI events and making assumptions about them. When porting an applet to an application, usually a JNLP-powered one, one should always consider the hidden cost of porting the program model as well, that is, the cost of making the applet code work outside the applet container.

12.4 Exploiting technology

Java GUI technology is powerful, although its uses are still limited to the production of GUI code. The technology can be used in other ways, however, such as applying it to development phases other than production and execution. For example, in Chapter 11 we discussed a look and feel, Napkin, that is specifically targeted at prototypes and early GUI designs.

Figure 12.7 shows an example of an original use of Swing's flexibility in rendering presentation details and applied to analysis and early development iterations. The GUI in Figure 12.7 is a prototype in which various design choices are still to be validated and finalized with customers, and on which additional analysis is required. Blurred widgets have therefore been used to represent items that need further work. Comments can also be embedded in the GUI itself that can help to understand the final intended behavior of the application, even from this early, limited version.


HTTP Server Prototype GUI

Status | Preferences | Extensions | About

http port: 8080
maximum clients: 10
min free space(KB): 2048
Log Style: normal
Log File: C:/log00.log
Hide Window on Startup
Allow Password Changes
changes depending user
Security Level: normal

OK | Cancel

Figure 12.7 An example of the use of presentation technology

Instead of communicating this information in a document, possibly bloated, costly to maintain, and ultimately unnatural when compared with the application in Figure 12.7, we embed it directly in the real thing, removing it as the application proceeds iteratively to its final refinement. After all, the GUI in Figure 12.7 *is* the application as we know it today. Such details could also be made visible or invisible under the control of a runtime flag.

12.5 Domain-specific and Little languages

Java is a general-purpose object-oriented programming language. It can represent and manipulate information in any domain of interest as long as it is represented using the particular flavor of the OO paradigm realized by the Java language. We can model business logic rules and weather forecast data, creating new specialized classes and intertwined relationships between them. The generality of Java has a drawback, however – its abstractness. For example, developers have experienced the complexity of specialized OOP frameworks in expressing domain-specific concepts, for example how cumbersome it is to use layout managers in non-trivial situations. This is because dynamic layout classes express what are essentially visual concepts using a text-based syntax optimized for general-purpose problems.

'Little languages,' or 'Mini-languages' (Hunt and Thomas 2000), are specialized, small languages that are created to fit a specific purpose and which are then 'embedded' into a more general-purpose language such as Java. This allows them to be more effective and simple to use than a fully-fledged specialized OO class framework. Little languages can grow out of the development environment and become full languages, although simple and extremely specialized, such as the Hibernate or Ant XML file formats, for example, which can be thought of as two little languages specialized respectively for Object–Relational Mapping and expressing tasks for driving the building process of Java programs.

On a more restricted level, the string format used by layout managers such as JGoodies's FormLayout, in which cell constraints are expressed compactly with strings like `right:pref:grow, left:max(50dlu;pref), l:m:g` instead of dozens of lines of code, can be thought of as a little language specialized in the effective high-level definition of layouts. Even the rather crude properties file protocol shown for developing throw-it-away prototypes in Chapter 5 could be polished and to form a little language for customizing and populating widgets quickly.

The syntax of choice for little languages is usually a simple one-line text format or XML schema. In many cases, however, some form of more powerful language is needed to combine the required expressivity with ease of use. Scripting languages such as Groovy, Jython, Beanshell, and many others, provide a powerful environment that accommodates even the most complex problems. Escalating to such a powerful solution, though, can be costly.

One example of the use of such language support could be the representation of business domain logic, encoding business rules in a scripting language that can also be used by non-developers, and which can be treated explicitly by the application itself, easing deployment and perhaps providing features like a simple COTS[7] business rule editor.

7. COTS, 'Component Off the Shelf' are software components built by a third party organization and ready to be employed in software.

Deployment of business rules can also be achieved easily by means of JNLP technology. By using conventional OOP, instead of adding another level of complexity to a client application by introducing a script interpreter engine, it is possible to provide all the required features using the same technology that is used for initial deployment. Code updates can then be used to patch application JAR files with new business rules.

12.6 The future of Java GUIs

While authors should always avoid making risky forecasts in a book, some trends can be recognized in the medium term, at least as regards the technologies available for Java GUIs.

Infrastructure and utility platforms are expected to flourish, growing more powerful and providing sophisticated concepts and tools, such as high-level support, sophisticated composable unit mechanisms, aspect-oriented support, and the like. While it is unclear whether a fully-fledged market for 'macro components' will ever gain momentum – paralleling the enterprise world and the history of J2EE – more powerful application platforms and specialized frameworks are expected to ease the lives of developers, at least in common scenarios such as rich client development and form-based GUIs.

The evolution of declarative languages for GUIs discussed in Chapter 11, and other non-Java based languages such as XHTML 2.0 for form-based GUIs, and other competing languages, is also interesting. Declarative languages are already used in conventional Java GUIs, in the form of little languages of greater or lesser sophistication. The Adaptation pattern discussed in Chapter 6 lends itself naturally to the declarative definition of aspects of GUIs independently of the rest of the application. Declaring a GUI has several advantages over representing it procedurally, especially for medium to large applications: it is easier to separate the various issues and keep them more maintainable, in many cases the resulting representation is more natural and easier to understand, and reuse is made easier because of the clearer decomposition.

Perhaps the real issue of non-trivial GUIs built with Java technology lies in the *language* issue. OOP scales to very complex scenarios and applications, but at the price of complexity and manageability. Domain-specific languages and tools can relieve this burden for such a well-known and circumscribed class of software applications, but professional GUIs will always remain complex beasts that need dedicated, multidisciplinary developers when tackling complex application domains and providing usable, cost-effective software for their customers.

As a concrete example, imagine running a poll among developers: would they prefer to have an `XPanel` component that provides full XHTML 2.0 form support à la Eclipse Flat Look – that is, defining complex forms with validation and binding by means of a standard, XML-based declarative language – or would they prefer a comprehensive OOP framework that delivers the same high-level powerful functionality by means of subclassing, API support and the like? My personal choice would be to give both a try, but in the process the XML-based language would perhaps be easier to understand and to tweak than a fully-fledged OOP framework. But this is a personal choice – much lies in the quality of the implementation and how well it exploits the specific details of the domain.

12.7 Summary

In this chapter we discussed specific design problems and some possible solutions. We have seen the notion of specialized, high-level attributes for managing complex data. We discussed some of the issues and possible solutions to the problem of handling runtime memory in large Java applications. We presented a practical example of porting a Java program to a new Java configuration, and we also discussed the use of little languages in our applications. The chapter concluded with a brief discussion of upcoming innovations in Java GUI technology.

13 Rich Client Platforms

This chapter discusses the practical use of Java rich client platforms (RCPs) for developing desktop application GUIs. The trade-offs between adopting an RCP infrastructure instead of using a more lightweight infrastructure such as a plain GUI toolkit plus possibly some support libraries are also covered. An example RCP application for the Eclipse RCP is included, focusing on its architecture.

The chapter contains the following sections:

13.1, Introduction to Java rich client platforms discusses the main issues involved in employing an RCP framework to build an application GUI.

13.2, The NetBeans RCP briefly introduces this RCP.

13.3, The Spring RCP provides an overview of the Spring RCP.

13.4, The Eclipse RCP discusses the Eclipse RCP in detail, including its architecture, windowing infrastructure, and GUI design guidelines.

13.5, Choosing the best RCP for your needs helps in picking the RCP best suited for your application. A section discusses the general issue of when to adopt an RCP for a project.

13.6, Legal issues introduces the main license issues related to the adoption of an RCP.

*13.7, An example Eclipse RCP application*shows a practical implementation of an application based upon the Eclipse RCP.

13.1 Introduction to Java rich client platforms

An RCP is an infrastructure framework for building medium-to-large desktop applications. Basing an application on an RCP simplifies development in many ways, providing structured GUI window support – including high-level widgets such as dockable windows, utility dialogs, high-level integration for internationalization and accessibility – coherent GUI guidelines, a common application environment, and an abundance of utilities, including user preferences, configuration support, and data-driven editors. An RCP can be intuitively thought as of the analog of an application server for the client side, even if this analogy works only at a high level, the server side and GUI development being rather different worlds. Both aim at relieving developers of the responsibility of domain-independent development, leaving them free to focus on domain-specific development.

The slow evolution of RCPs in the Java world compared to application servers and middleware is due mainly to the unexpected explosive success of the Web, which has made RCPs appear as 'dinosaurs from the desktop era' (as they were dubbed a couple of years ago). This is no longer true: today's Java RCPs are modern, Web-aware platforms that perfectly fit the need for developing cost-effective, Internet-aware, medium to large desktop application GUIs in Java[1].

The case for RCP applications

RCPs are not a panacea, of course. They tend to be costly to learn and even more time-consuming to master in detail. Employing a RCP is not always a cost-effective solution. In small projects there is no need to resort to an RCP application, especially if the developers are not familiar with the technology and deadlines are tight. Learning an RCP is learning a complex OOP framework, with all the related issues. Learning an RCP can be seen as an investment for the future, meaningful in a large project, or over several small ones, that could take advantage of a RCP. RCPs do provide a number of benefits, such as proven solutions to common problems, and a higher-level framework than low-level GUI toolkits such as SWT+JFace or Swing, in which developers can focus on domain-specific code rather than reinventing the wheel. They also benefit users, because the risk of producing bizarre or clumsy GUI designs is much lower when using a RCP framework.

RCPs fit nicely into an iterative development scenario: given their modular structure, developers are forced to conceive their code in terms of loosely-coupled components that can be added or updated with newer releases,. This makes it easy to build an extensible application on top of an RCP, with all the benefits that this brings.

RCPs are in many ways still focused on their origins: the integrated development environment (IDE). General-purpose and useful components – referred to as 'modules' or 'plug-ins,' depending on the RCP, are not yet available, such as a general-purpose security model, or an administrative console and services for system administrators. However, there are plenty of well-executed CVS clients, Ant support tools, and sophisticated text-based source editors that are fine if something like a development environment is to be built, but of little use in the majority of applications. This is unlikely to be a long-term problem, as a market for RCP extensions is growing rapidly, along the same lines as the well-established 'ecosystem' for IDE extensions.

1. Especially Eclipse RCP, with its native support for integrated browser and its Web-like form widgets.

What's in an RCP

RCP support ranges from high-level GUIs to data models, GUI design and presentation details or, put another way, everything but the domain. Figure 13.1 illustrates the services provided by RCPs using the same functional decomposition adopted in Chapter 1, in which grayed areas represent the degree of coverage of the features provided by RCPs. These areas represent what is already provided for solving most application problems in the specific functional area.

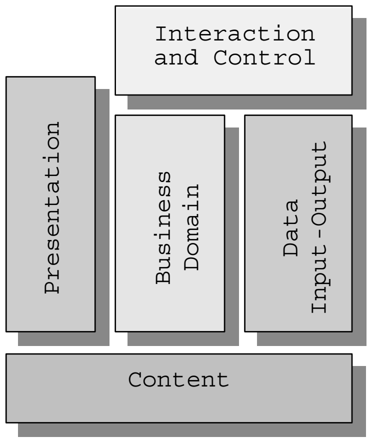


Figure 13.1 RCP support organized functionally

While RCPs provide many features 'out-of-the-box,' developers have a certain degree of freedom in customizing them – for example, presentation graphics are easier to modify in NetBeans RCP-powered applications than in a GUI built using the Eclipse RCP. For some domains, though, even the wide array of support libraries provided by RCPs is not enough, and developers have to resort to specific solutions, such as domains that require ad-hoc components.

From the diagram in Figure 13.1 it is clear that RCPs can be used as powerful tools for building client applications using the Domain-Driven Design approach[2], in which complex domains can be represented effectively by assigning all non-domain concerns to infrastructure components.

2. See (Evans 2004).

GUI design guidelines and RCPs

Large IDEs built collaboratively by large teams, such as NetBeans and Eclipse, have precise guidelines for rationalizing their GUI design and smoothing user interaction. Strict GUI design guidelines become necessary for extensible applications that need to accommodate many third-party components seamlessly. While NetBeans fully adopted the Java Look and Feel design guidelines[3] (Java L&F Design Guidelines 2001), Eclipse created its own GUI design guidelines for effective use of the SWT toolkit.

Clearly, there are no strict rules regarding the adoption of the original platform's GUI guidelines in an application, even though all the GUI details and machinery used by the application will conform to these guidelines, and departing widely from them could make the overall GUI of an application look awkward.

Conversely, the GUI details of the various widgets and windowing support employed in large IDEs might not necessarily fit your own project: when in doubt, the wisest choice is always to avoid using such free solutions in a GUI and to limit an RCP-powered GUI for simplicity and usability.

SWT and the Eclipse RCP deserve a separate discussion, which is done in Section 13.4., while an example application based on the Eclipse RCP is discussed in Section 13.7.

13.2 The NetBeans RCP

The NetBeans RCP (NRCP) is a general-purpose platform for building desktop application GUIs with Java. It is the result of a refactoring of the code originated in NetBeans, the open source IDE provided by Sun.

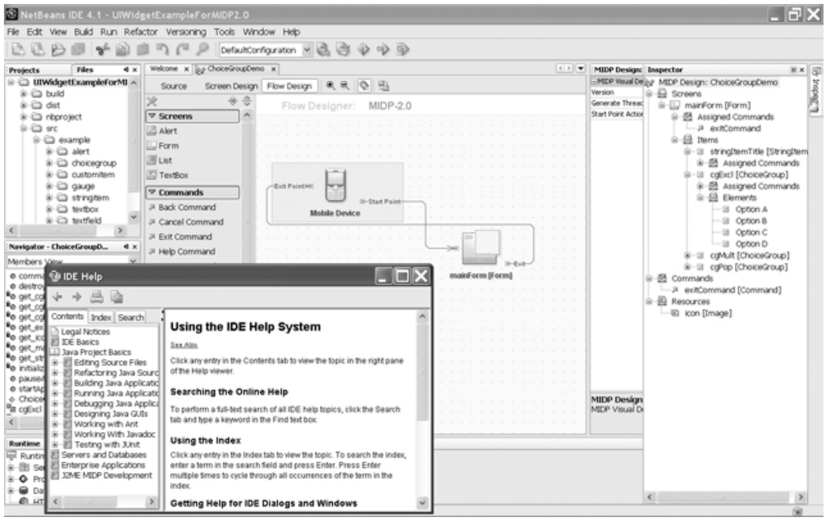
NRCP provides roughly the same features as the Eclipse RCP, which is introduced in Section 13.4. Both RCPs provide a modular architecture for adding new components, and provide agile installation bundles, stripped of any unnecessary code[4]. The main difference between the two RCPs lies in the GUI toolkit employed: NRCP uses Swing and related technologies, such as JavaHelp for help support, while Eclipse RCP uses SWT and JFace, which were built expressly to support the IDE.

The learning curves differ too: there is a wealth of documentation (both in literature and on line) for Eclipse-related technologies, but little material regarding Netbeans RCP. Eclipse also provides many generated template applications that speed learning of the platform considerably. The NetBeans IDE does provide a wizard for creating an application based on NCRP.

3. These guidelines were discussed in Chapter 3.
4. The NRCP 4.1 download is 4.41 MB.

The most surprising thing about NRCP is its lack of popularity, despite the fact that it was around long before the Eclipse RCP. This is perhaps partly due to the current momentum of SWT technology over Swing. It is a pity, because the set of features NRCP offers to developers is quite impressive, ranging from window management, generic data access, scripting support, auto-update, and user settings management.

Figure 13.2 shows a screenshot of Version 4.1 of the NetBeans IDE, showing some of the windowing components that are available for NRCP applications. Note the sliding panel containing a tree on the right-hand side of the screen.



Using the IDE Help System

See Also

Click any entry in the Contents tab to view the topic in the right pane of the Help viewer.

Searching the Online Help

To perform a full-text search of all IDE help topics, click the Search tab and type a keyword in the Find text box.

Using the Index

Click any entry in the Index tab to view the topic. To search the index, enter a term in the search field and press Enter. Press Enter multiple times to cycle through all occurrences of the term in the index.

Getting Help for IDE Dialogs and Windows

Figure 13.2 NetBeans 4.1 window support

NRCP architecture

NetBeans' modules, equivalent to Eclipse plug-ins, are components that provide various functionalities to the NetBeans platform. They are implemented following the Java standard for extensions that is built into the JAR file definition, as defined in the JAR `MANIFEST.MF` file. The modular architecture of NetBeans takes advantage of various Java standards, such the JavaBeans Activation Framework (JAF) for determining the type of arbitrary data and others.

Rather than a fully modular architecture, current NetBeans organization more closely resembles a sort of large API-centered ecosystem, in which several complex APIs interact at a fine level of detail. In the future the architecture of NetBeans will probably evolve towards a more componentized approach, following Eclipse's architecture.

The API for windowing support is based on the idea of supplying components to be displayed, which are positioned and rendered as decided by the platform's window management system. All components are subclasses of the `org.openide.windows.TopComponent` class, which provide methods for basic GUI management, such as activating the component view, opening it, and so on, in a high-level fashion.

13.3 The Spring RCP

The Spring RCP is a framework built on top of Swing to support the construction of medium to large client applications. The Spring RCP is still in its early stages of development[5] compared with both NetBeans and Eclipse RCPs. In contrast to the latter two RCPs, the Spring RCP is not the result of refactoring the implementation of an existing IDE, so its features don't include those typical of development-oriented tools, such as version control and advanced text editor support.

Perhaps the strongest point of the Spring RCP over the other RCPs lies in its coupling with server-side Spring applications, both in its coding approach and runtime object communication. The Spring RCP project focuses on providing 'an elegant way to build highly-configurable, GUI-standards-following Swing applications faster by leveraging the Spring Framework and a rich library of UI factories and support classes[6].' This means that developers familiar with the traditional Spring server application framework's 'elegant' implementation style (a popular mix of inversion of control[7] and aspect-oriented programming[8]) can apply the same approach to client applications.

It is too soon to say whether the independent efforts of both NetBeans and Eclipse RCPs will be able to provide the same level of quality over the same array of features that these two projects already provide. Nevertheless, the Spring RCP takes an interesting approach that has already proved popular among Java developers building server applications.

5. At the time of writing (late 2005) the framework is still in alpha, so there is little practical utility in discussing such an early product in detail.
6. Taken from the Spring RCP Web site, http://www.springframework.org/spring-rcp
7. Inversion of control, or its Spring-specific variant known as 'dependency injection' are techniques for reversing the traditional flow of control in which client code invokes server code directly, by making server code invoke callback methods in client code. The dependency inversion principle is described in Chapter 7 – see page 358.
8. Aspect-oriented programming (AOP) is an approach to programming (which happens to complement OOP nicely) to express different behaviors in a program in a more effective and modular way.

13.4 The Eclipse RCP

The Eclipse RCP (ERCP) grew out of the standard Eclipse project when it became clear that much of the work spent in building the various parts of Eclipse could also be used for building general applications, following the example of NetBeans. In fact, Eclipse itself can now be thought of as a specific application for software development built using the ERCP.

This apparently simple refactoring was due mainly to the high quality of Eclipse's design and plug-in architecture, which allows for a clear decomposition of code into separate units. Something described as 'applies to Eclipse' in this section means that is applicable to the ERCP as well.

Eclipse plug-in architecture

In Eclipse plug-ins are loaded in their own class loader. Given the rules of visibility for Java class loaders, plug-ins cannot access other classes or resources loaded from other plug-ins. The basic interoperability mechanism among plug-ins is provided by *extension points* that define how a given plug-in can be extended by other plug-ins and, symmetrically, by providing extensions to other plug-ins by extending their extension points. The details are declared in the `plugin.xml` manifest file for that plug-in.

Extension points can be used to override the default behavior of a plug-in, or for example to group related elements in the GUI, such as grouping commands in a common point in the GUI. Every component in an ERCP 'ecosystem' is defined in this way, apart from the importing of Java packages from another plug-in, achieved by means of the specific `requires` attribute in the `plugin.xml` file. At start-up the platform scans the `plugin.xml` declarations, creating an in-memory registry of all available plug-ins, although they are loaded only when needed by another plug-in.

The following (simplified) `plugin.xml` file defines an application launched through the class `com.marinilli.Application`, containing only one command:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
   <extension
         id="application"
         point="org.eclipse.core.runtime.applications">
      <application>
         <run
               class="com.marinilli.Application">
         </run>
      </application>
   </extension>
   <extension
```

```
         point="org.eclipse.ui.commands">
      <category
            name="Save"
            id="app.category">
      </category>
   </extension>
</plugin>
```

From Eclipse 3.0 plug-ins can be added and removed (with certain restrictions) at runtime as well as at deployment time, the former being referred to as *dynamic* plug-ins. From Eclipse 3.1, some kernel support is factored out of the `plugin.xml` file into OSGI's `MANIFEST.MF`.

Eclipse RCP plug-ins

The Eclipse RCP can be defined as the minimum set of Eclipse plug-ins that can be used to build an application using the Eclipse 'bare-bones' infrastructure. Only two items are necessary:

- The Eclipse runtime plug-ins, which is comprised of three plug-ins, including the OSGI[9] kernel implementation.
- The GUI support, composed of three distinct layers: the SWT library, with some native auxiliary code at the lower level, the JFace library for data-driven support to SWT, and the workbench support for detachable windows, dialogs, and so on.

A downloaded RCP installation from the Eclipse Web also contains other auxiliary plug-ins that are needed for file-related chores such as XML and various other infrastructure support. Auxiliary plug-ins can be omitted to further decrease the installation bundle size.

This set of plug-ins is the minimum set. Clearly SWT can be used without any other plug-ins, so that GUIs can be built using this toolkit without using ERCP at all, as discussed in Chapter 11. The ERCP 3.1 bundle for Windows is 5.80 MB to downloaded and 8.38 MB unzipped, excluding JRE.

Figure 13.3 shows the main plug-ins organized in a layered fashion. Plug-ins shown with dashed borders are optional.

9. The Open Services Gateway Initiative (OSGI) is an industry group responsible for defining an open standard for component interoperability: see http://www.osgi.org for more details.

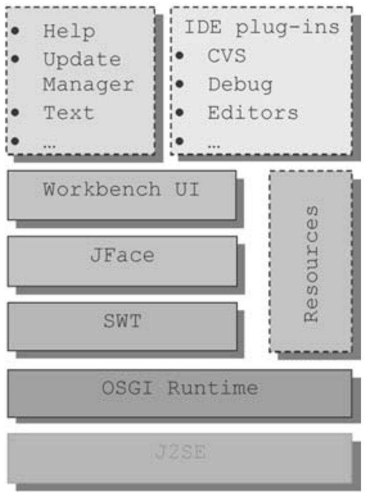


Figure 13.3 Eclipse layers

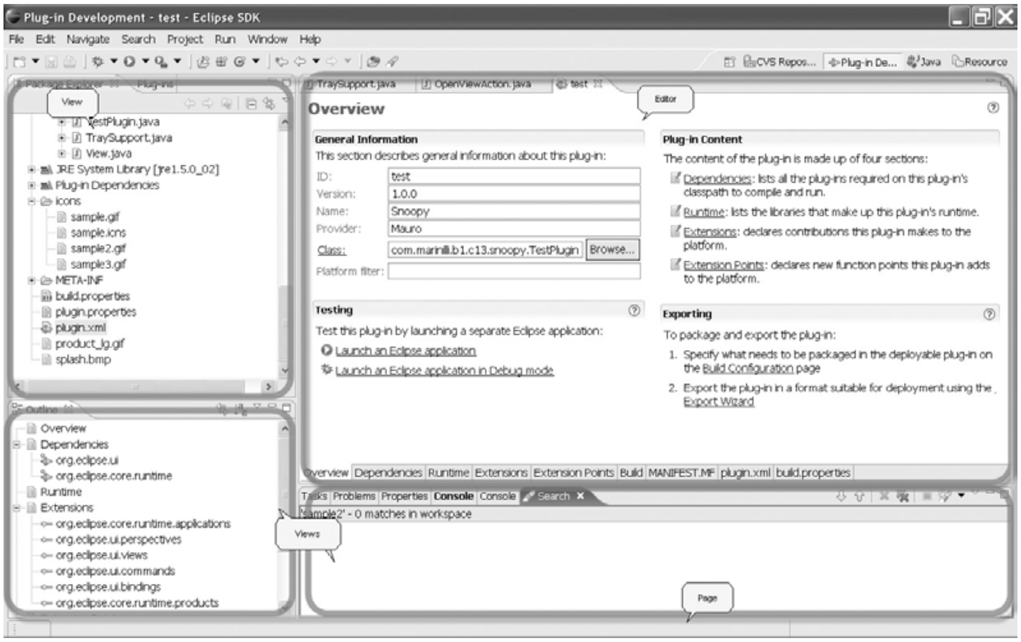
As the figure shows, the main plug-ins in ERCP are the OSGI Runtime and the user-interface support items – SWT, JFace, and the Workbench plug-in. All other plug-ins can be optionally used in building an application.

The plug-in component model in Eclipse 3.1 is powerful and effective, allowing the creation of very rich component 'ecosystems,' a thousand or more plug-ins in some installations. It does still lack the features mentioned previously, however: a sound security model that provides various levels of security, managing trusted and non-trusted plug-ins, sandboxing[10] non-signed plug-ins, and so on.

The workbench – the building blocks of ERCP GUIs

The Eclipse workbench, consisting of the Eclipse main window and its structure, can be reused in ECRP applications. Eclipse developers are familiar with its organization, shown in Figure 13.4.

10. Sandboxing is a technique for containing code execution within predefined rules. It is used for example in the code managed in the applet container, and also for Java Web Start applications. For example, an unsigned (untrusted) Java applet cannot access the local file system.



Overview

General Information

This section describes general information about this plug-in:

ID: test
Version: 1.0.0
Name: Snoopy
Provider: Mauro
Class: com.marinilli.b1.c13.snoopy.TestPlugin
Platform filter:

Plug-in Content

The content of the plug-in is made up of four sections:

Dependencies: lists all the plug-ins required on this plug-in's classpath to compile and run.

Runtime: lists the libraries that make up this plug-in's runtime.

Extensions: declares contributions this plug-in makes to the platform.

Extension Points: declares new function points this plug-in adds to the platform.

Testing

Test this plug-in by launching a separate Eclipse application:

Launch an Eclipse application

Launch an Eclipse application in Debug mode

Exporting

To package and export the plug-in:

1. Specify what needs to be packaged in the deployable plug-in on the Build Configuration page
2. Export the plug-in in a format suitable for deployment using the Export Wizard

'sample2' - 0 matches in workspace

Figure 13.4 The Eclipse workbench

The term 'workbench' usually indicates the set of classes and visual components that implement the Eclipse GUI structure, and that are also used for ERCP applications.

The main parts of the workbench are:

- *Windows*. It is possible to open several windows on the same workbench.
- *Pages* are containers for the current *perspective*, and are used mainly for implementation purposes.
- *Perspectives* are organizations of views, editors, and actions within a window. By customizing views, editors, and actions using a perspective, GUI designers can optimize the same components and commands for different tasks, thus creating more productive and task-centered GUIs.
- *Editors* are areas of a window devoted to information manipulation. In complex GUIs, such as Eclipse itself, editors are the main focus of the user interaction after the data to work on has been selected using a suitable exploration view, so they occupy a central position within the window layout.

- *Views* are the window areas other than editors. Views are used for exploring and selecting data, for showing properties, or other auxiliary information. Depending on the kind of application you are building you will use one or more views to organize results, select data, and so on.

These entities are related as shown in the following UML class diagram, which represents relationships among concepts, rather than the real Java implementation in ERCP. See Figure 13.5.

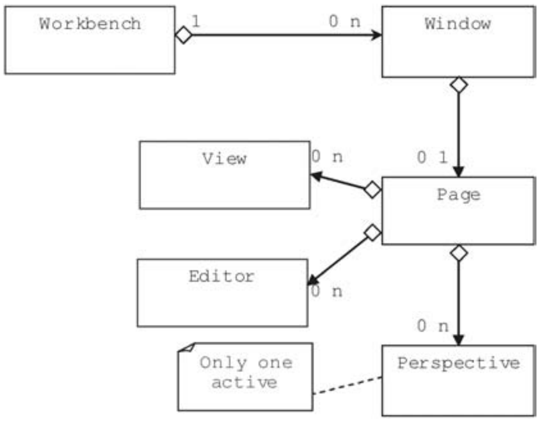
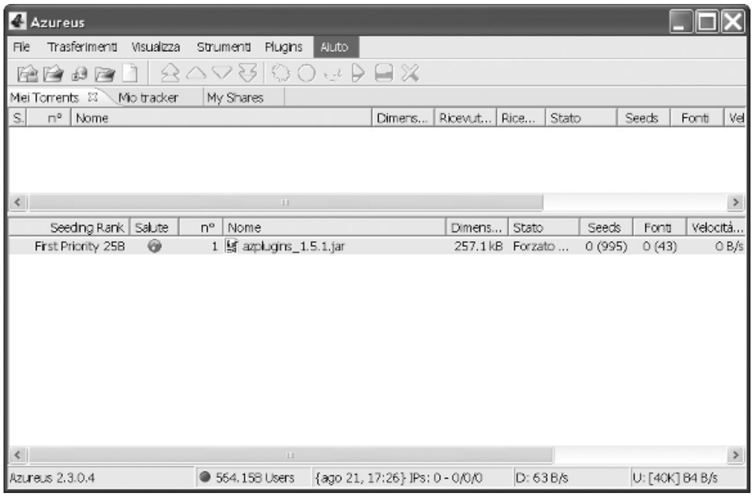


Figure 13.5 Eclipse workbench structure

The workbench's windowing organization, despite initially being intended for IDE applications only, is highly flexible and can be adapted to a wide range of different scenarios. Figure 13.6, for example, shows a workbench instance for the Azureus application, a client for the BitTorrent file sharing protocol, which was built with ERCP. The main window is implemented using only ERCP workbench views, organized in a fixed layout.

The ERCP plug-in architecture and the workbench support together provide a powerful mechanism for implementing composable units[11]. Despite the visual composition of loosely-coupled GUI-based components being supported only at the rather coarse-grained level of composing views and editors within a perspective, it is also possible to define plug-ins that participate in the composition of a single view or editor.

11. See Chapter 6.



Seeding Rank	Salute	n°	Nome	Dimens...	Stato	Seeds	Fonti	Velocità...
First Priority 258	●	1	azplugins_1.5.1.jar	257.1 kB	Forzato ...	0 (995)	0 (43)	0 B/s

```
Azureus 2.3.0.4   ● 564.158 Users   {ago 21, 17:26} IPs: 0 - 0/0/0   D: 63 B/s   U: [40K] 84 B/s
```

Figure 13.6 *An ERCP-based workbench example*

GUI design guidelines for ERCP applications

This section gives GUI design guidelines for the Eclipse IDE[12] annotated with the related rule numbers. Basically, these guidelines can be summarized as 'Follow what the Eclipse IDE does.'

Views, editors, and perspectives in Eclipse should conform to the following rules:

- Views are used to navigate a hierarchy of information, open an editor, or display the properties of an object (7.1). Commit in views must be done in immediate mode (7.2) while in editors commit is achieved in deferred mode – that is to say, data in views in saved immediately, while in editors it is saved only by means of an user action such as 'Save.'
- Editors are used to edit or browse a file, document, or other primary content. Only one instance of an editor may exist for each editor input within a perspective, but different input can be opened in separate editor instances.
- Perspectives should be created for long-lived tasks that involve the performance of smaller, non-modal tasks (8.1). When only one or two views need to be shown, it is suggested to extend an existing perspective type, rather than create a new one (8.2).

12. Available at http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html.

The following is a brief list of other details from the GUI design guidelines for Eclipse, with the relevant rule numbers in parenthesis:

- Use capitalization (1.5, 1.6) in text with headline style for menus, tooltips, titles of windows, dialogs, tabs, column headings, and push buttons. Capitalize the first and last words, and all nouns, pronouns, adjectives, verbs, and adverbs. Do not include ending punctuation. Use capitalization of the first word (and any proper names) only for all labels in dialogs or windows, including those for check boxes, radio buttons, group labels, and simple text fields.
- Error handling (1.8). When an error occurs that requires either an explicit user input or immediate attention from users, communicate the occurrence with a modal dialog.
- Icons. There are eight different types of icons used in Eclipse, and new icons should use the standard color palette provided by IBM (with a separate palette for wizard icons). The guidelines also prescribe file naming and directory structure for placing icons.
- Object properties should be placed in a view (the Properties view) when they can be calculated quickly, otherwise they should be placed in a dialog.
- Preferences. Global preferences are handled in a common Preferences dialog (15.1). Local preferences, such as those related to single views or editors, should be handled locally. Use the root page in the Preferences dialog for frequently-used preferences, or those preferences that have widespread effect. The root preference page should not be blank (15.4).
- Use Flat Look design for user scenarios that involve extensive property and configuration editing (16.1).

The native twist

As long as an application is close to the Eclipse IDE in concept (or ideally is one of its plug-ins) it is safe to follow the guidelines introduced above. Things start to become blurred when the GUI design of an application departs from the original IDE concept.

Because of SWT's 'native' nature, it is possible instead to use OS-specific guidelines, such as the Windows GUI design guidelines. While this limits cross-platform portability, it may make sense when there is only a need to target a single platform. In such cases working directly with the native platform GUI design style is the best solution.

13.5 Choosing the best RCP for your needs

So far we have described the three main open source RCPs available to Java developers for building medium- to large-scale applications. These provide roughly the same features – Netbeans and the Eclipse RCP, in particular, have very similar functionalities. What differs is the level of documentation and the community base supporting them, and perhaps more importantly, the underlying GUI toolkit used.

With the constant refinement of open source RCPs, GUI development for all but trivial Java GUIs has changed dramatically. Apart from specific application domains, RCPs can be used in a wide range of scenarios and business domains. Gone are the days when the knowledge of the GUI toolkit alone was enough to build GUIs – expensively, and sometimes with poor results. In the author's opinion, RCPs should be taught in university courses along with GUI toolkits, both as a practical means of building GUIs, and as an example of current industrial practice in constructing large applications[13].

A potential problem in adopting an RCP instead of a more lightweight solution such as the SWT/Swing toolkit and a GUI support library lies in the comprehensive nature of RCP frameworks. RCP frameworks tend to have an 'all-or-nothing' effect on development. Before attempting a solution to a given problem, developers using an RCP should first focus on the way the RCP handles that situation, and follow that design, instead of providing their own solution that might not work. This requires time-consuming learning that goes beyond a mere study of the documentation, otherwise problems could manifest themselves later in development.

Ultimately every RCP fits a specific purpose. Spring RCP is aimed at developers familiar with Swing and the Spring framework. NetBeans RCP addresses the needs of the Swing community at large, while applications based on the Eclipse RCP are shaped by the SWT toolkit[14].

When deciding which RCP to adopt in a new project, if any, perhaps the most important point lies in the choice of the GUI toolkit – that is, in the choice between Swing and SWT. Other details such as documentation and support, the availability of a strong developer community, and the other factors discussed for tool selection in Chapter 11, all shape a final decision. For general documentation and practical support for the generation of plug-ins, ERCP is currently preferable to the NRCP, but this may possibly change in future.

When to employ an RCP

When should one consider using an RCP for a small project? How small does a project have to be not to warrant an RCP?

13. The Eclipse RCP seems the best suited to this from an architectural viewpoint.
14. Although it is still possible to use Swing, as discussed in Chapter 11.

Answering these questions involves knowledge of many details, such as the developers' previous knowledge of the platform, the schedule for the planned releases, the future maintenance and extension plan for the application, and so on. As a rule of thumb, building more than eight screens, or the need to actively maintain an application for more than six months, justifies the adoption of an RCP, given that the time to learn it and the other constraints, such as the type of application and the domain fit with an RCP, are satisfied. These are just rough rules of thumb that should be customized to each case – each project has its own peculiarities, and a universal set of rules is not practical.

In conclusion, learning an RCP or adapting it in a medium to large project can be seen as a form of long-term investment. Even if both NetBeans or Eclipse disappear in the near future, their source code is public, so that they still remain a viable choice even for long-term projects. As regards cost and their competitiveness with other similar frameworks, much remains to be done in enhancing ease of use, documentation, code generation facilities, and other means of lowering the initial adoption cost. In this respect, NetBeans needs to catch up with Eclipse.

13.6 Legal issues

You might think that all this discussion of RCPs sounds interesting, and perhaps also useful, but what about the small print on the license page? What are the legal constraints over use of an RCP as part of a commercial product, or in some other form? Here is a brief overview, but for definitive information on such delicate issues, you should refer to the licenses themselves.

Eclipse

The Eclipse Public License (EPL) evolved from the Common Public License (CPL) created by IBM 'to encourage a model in which commercial products could be based on open-source efforts[15].' The EPL differs from the CPL in a few details, such as a specific, less restrictive treatment of software patents[16]. Here is a brief summary of the key points in the EPL: a *contributor* is a person or organization that creates the initial code under EPL, or who originates changes or additions, or who distributes the code under the EPL:

- Only mere distributors can be anonymous, all other contributors cannot.
- Contributors creating 'modules' that use or modify existing EPL code can distribute the final result under their own terms, as long as the portion of the

15. From *A history of IBM's open-source involvement and strategy*, IBM Systems Journal, July 2005. Available on line.
16. For more details, see http://www.eclipse.org/legal/index.php.

derivative work under the EPL license is still acknowledged under an EPL-compatible license. This clause causes incompatibility of EPL/CPL with a popular OSS license, GPL[17]. Contrary to the EPL, extending/modifying code licensed with the GPL forces the derivative product to be licensed under GPL.

- Contributors can compile code licensed under the EPL and distribute it commercially under EPL terms.
- Contributors that modify EPL code but don't distribute it, perhaps for internal use, are not obliged to make their modifications available to others.

Some problems might arise if for example you plan to use GPL-licensed code with code licensed under the EPL, because these licenses are incompatible. You can refer to the Free Software Foundation license page for more details[18].

Netbeans

NetBeans is covered by the Sun Public License[19] (SPL), which is a variant of the Mozilla Public License. Both of them are 'free software' licenses, in that they envisage software as a free artifact to which contributors are free to make modifications and use them privately, or distribute changes, without specific permissions apart from those prescribed in the license itself. This doesn't mean that versions of so-called 'free software' cannot be distributed commercially – license terms for 'free software' can still be enforced under copyright law.

Given its distribution philosophy, the SPL is not compatible with the GPL. For more details, the reader is urged to consult the SPL text.

In conclusion, ERCP, NRCP, and Spring RCP allow commercial products to be built on top of their platforms provided that legal information is provided when the application is downloaded, and that the legal requirements dictated by the various licenses are satisfied.

13.7 An example Eclipse RCP application

In this section we discuss a practical example of an application built on top of ERCP. Given the number of examples freely available on line or discussed in other books, we will focus on the architecture of the application and on its high-level GUI design aspects.You can download the code and try the application yourself.

17. The GNU General Public License (GPL). For details, see: http://www.gnu.org/licenses/gpl.html.
18. Available at: http://www.fsf.org/licensing/licenses/index_html.
19. Details available at: http://www.netbeans.org/about/legal/license.html.

To install the code of the example application, you need to:

i. Download the freely-available RCP distribution from the Eclipse Web site.
ii. Download the bundle code for this chapter.
iii. Copy the contents of the `features` and `plugins` directories into the same directories as in the RCP directory (this procedure is also valid also for installing Eclipse plug-ins).
iv. Copy the `.ini` file of your platform (Windows, Mac, GTK), renamed as `config.ini` into the ERCP's `configuration` directory, to replace the existing file of that name.

The application

Snooper is a demo application for gathering data about people. You may have seen this kind of application at work in many movies or TV shows, where detectives or secret agents type in the name of some bad character and everything about him is magically discovered, from his driving license number to his favorite brand of shoe.

Despite the fact that such an application would be on the edge of legality on grounds of personal data management, privacy, national security, and so on, at least in some countries, our management has decided that this is the ultimate high-margin market, so we are asked to provide an initial release in three months time.

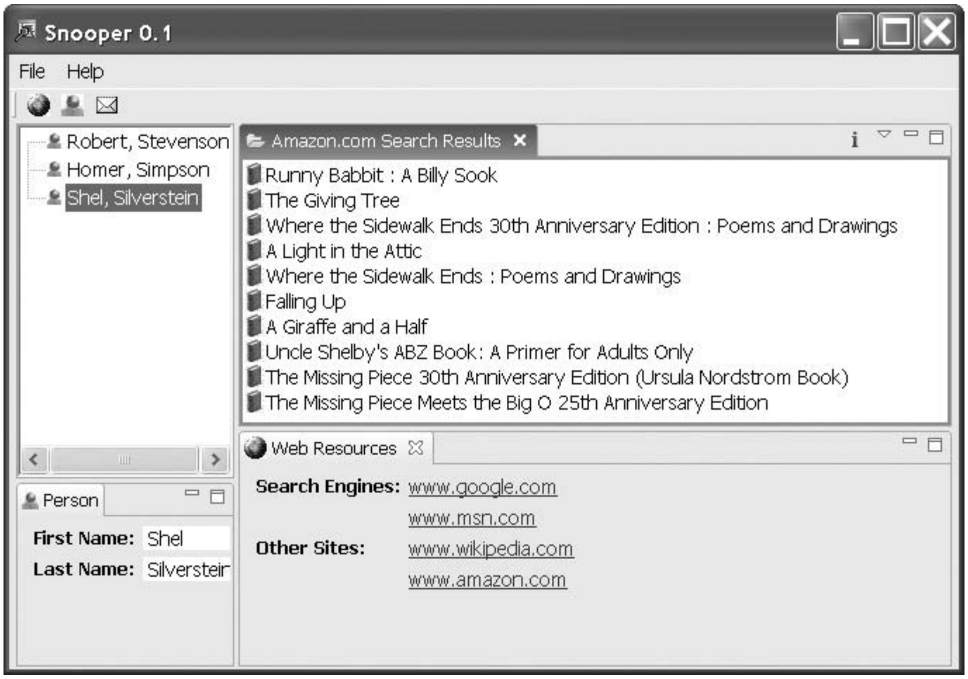
We decide to use ERCP because it provides us with a robust and powerful platform that can host future extensions, such as specialized searches, access to specific databases, facial image processing, and all that fancy spy story stuff, and it only needs to be available on the major OS platforms.

We would like to make the application available in a modular fashion, so that premium customers can buy extra modules for more sophisticated research.

You will be disappointed to hear that the implementation provided with this chapter doesn't connect to a Pentagon database – for security reasons – but, more humbly, to free resources available on the Web. Figure 13.7 shows the application with the Amazon search service loaded.

The main interaction we design for the application in Figure 13.7 is as follows:

1. The user selects an individual from a persistent list, or creates a new person as needed.
2. Selecting a person populates the other views with data about the selected individual. For search views, or other calculated views, if the search button is not used, nothing is displayed. When the button is clicked, the search process starts for all the registered search providers (that is, all plug-ins loaded at deployment time). For simplicity, no 'stop search' action has been provided.
3. Previously-launched searches, if not refreshed, show old data.



Runny Babbit : A Billy Sook
The Giving Tree
Where the Sidewalk Ends 30th Anniversary Edition : Poems and Drawings
A Light in the Attic
Where the Sidewalk Ends : Poems and Drawings
Falling Up
A Giraffe and a Half
Uncle Shelby's ABZ Book: A Primer for Adults Only
The Missing Piece 30th Anniversary Edition (Ursula Nordstrom Book)
The Missing Piece Meets the Big O 25th Anniversary Edition

First Name: Shel
Last Name: Silversteir

Search Engines: www.google.com
www.msn.com
Other Sites: www.wikipedia.com
www.amazon.com

Figure 13.7 The Snooper application

The first release of the application is a little vapid, in that you only get material such as published books and data from standard Web sites, but it nevertheless provides an interesting example of a simple ERCP application.

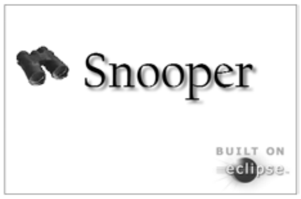
The GUI design of the application is organized as follows:

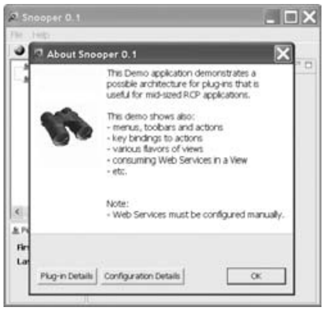
- We use ERCP views to represent all data managed within the application. We don't need editors because of the nature of the application: all data input is performed in immediate mode – see *GUI design guidelines for ERCP applications* on page 482.
- Every module provides one or more views for user interaction.
- View capabilities are related to domain-specific issues, as will be shown later.

Building the application using modules provides many advantages, such as an easier iterative development environment – we can add more features later without modifying the core components – flexibility and extensibility, the possibility of more powerful billing and licensing schemes, and clearer organization of the development teams.

The code provided shows many useful tricks, such as splash screen, a localized 'About' dialog (Figure 13.8), OS-dependent system tray support (Figure 13.9), and others. See the code bundle for this chapter for more details.

The remainder of this section focuses on the architecture of ERCP applications, and how to employ its plug-in architecture usefully.





This Demo application demonstrates a possible architecture for plug-ins that is useful for mid-sized RCP applications.

This demo shows also:
- menus, toolbars and actions
- key bindings to actions
- various flavors of views
- consuming Web Services in a View
- etc.

Note:
- Web Services must be configured manually.

Figure 13.8 Bells and whistles in Snooper

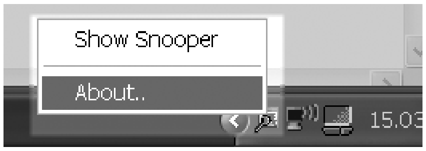


Figure 13.9 The system tray support for Windows

Introducing client-side modular architectures

This section introduces general issues about the modular architecture of RCP applications that are based upon a plug-in architecture, whether the Eclipse RCP, NetBeans RCP, or other plug-in support frameworks. These general points are then illustrated in the context of the Snooper application in the next section.

Plug-ins and pomegranate seeds

Most people find pomegranates tedious to eat. One needs a lot of patience to deal with the inner intricacies of a pomegranate, so it comes as no surprise that such fruits are not as popular as, say, peaches. Working with ERCP is much like eating pomegranates. You have no choice but to deal with its 'seeds' one by one, because much like a pomegranate, the bulk of an ERCP application is all in the seeds themselves.

Learning to deal with plug-ins is tedious and time-consuming, despite what the IBM marketing department may claim. Mastering the subtleties of the OSGI manifest or the flow of control during the platform start-up[20] takes substantial time that detracts from that available for, say, a thoughtful componentization of the various plug-ins with which an application is to be implemented.

No wonder that few developers find the time, or are willing to see the ERCP plug-in architecture (or NetBeans') as an opportunity for fine, useful design, rather than as a necessary hindrance on the route to the next release of their application.

Packing everything into one or two big plug-ins and using the plug-in architecture only as a necessary means of integration of code with the rest of the ERCP framework is a valid strategy for delivering small applications quickly. Unfortunately, even for small- to medium-scale applications, the application's structure tends to degrade quickly as maintenance and code additions are performed during the application's lifecycle. A sound modular structure becomes a serious concern in medium to large RCP applications and where long-term maintenance is an issue.

The next section illustrates a simple approach to using the plug-in architecture in ERCP and in other frameworks for building modular RCP applications, but first, a brief introduction to software components.

Pomegranate seeds and software components

The use of the term 'software component' when talking about loosely-coupled modules that execute on a client-side application is a little misleading. The definition of a software component is still controversial: many different approaches at various levels of granularity, and focusing on different application scenarios and business domains, all define themselves as 'components,' such as J2EE EJB, .NET components, JavaBeans, CORBA, and COM components. All these approaches share two characteristics:

- They focus on writing software units that comply with a given specification.
- Such units may be reusable in other contexts[21].

Clearly we are interested in a smaller subset of the characteristics of enterprise and distributed software components. For example, GUI aspects are extremely important for us, while remoting capabilities are not, as our 'components' all reside in the same RCP instance. For these reasons, and also because we used the term 'component' throughout the book when referring to aggregates of basic

20. Sooner or later you need to face such details unless you can maintain your plug-in as a carbon copy of those automatically generated by Eclipse PDE (Plug-in Development Environment) wizards.

21. For more details. see for example: http://en.wikipedia.org/wiki/Software_component

widgets (that is, visual components), we will give our components the more specific term of *module*.

To the Java programmer, components tend to resemble Java packages, as they can be used to group elements into logical structures. Clearly components go beyond mere structuring of multiple classes, providing a semantically-rich grouping mechanism that discreetly takes advantage of the theoretical underpinnings of object orientation – encapsulation, self-containment, loose coupling, and so on.

Figure 13.10 introduces UML2 component diagrams, which we will use here even though we are not discussing fully-fledged software components. The diagram uses the UML2 feature of visual stereotypes to represent components.

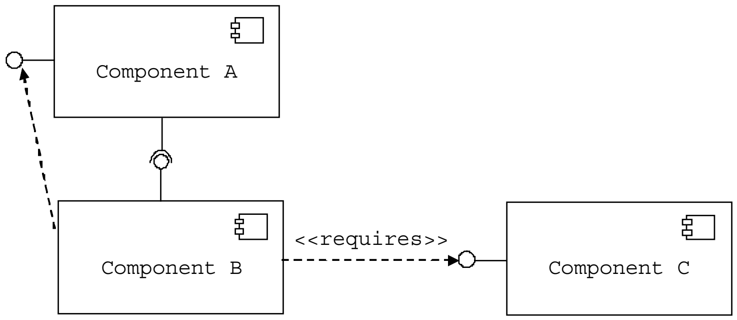


Figure 13.10 An example component diagram

In the diagram Component B exposes one interface to the outside, while using the interfaces provided by Component A and Component C. Component A requires an interface (a set of services) that is provided by B, so Component B provides the services Component A requires. The 'lollipop' icon expresses an interface a component is providing to other components, while the half circle represents a required interface.

A standard specification for a component model on the client side of Java applications would give a great boost to medium to large GUIs in Java, especially for enterprise-level GUIs such as business-critical applications, ERP, or financial software. Unfortunately, the availability of two major GUI toolkits, and specifications such as JSF, make things more difficult for the creation of an effective standard for client-side components. But a closer look suggests that such apparent difficulties in fact represent strengths that such a specification could leverage.

Designing modular RCP applications

Plug-ins therefore interact mainly by exposing interfaces to other plug-ins through extension points and by providing extensions to existing interfaces (that is, extension points from other plug-ins). In a modular mindset like that briefly introduced above, some guidelines can be defined:

- A module in a GUI-focused RCP should be fully self-contained: it has to include its own GUI, typically as a number of views or editors, as well as actions, help support, business domain classes, and so on, and it should also contain everything needed for its own build: sources, resource files, test cases, and the like.
- A module should be designed and made to interact with other modules only by means of the underlying RCP architecture. When using Eclipse this is accomplished by means of its plug-in architecture and, as a general design guideline, by providing one module per plug-in. Despite the fact that this 1:1 mapping could seem limiting in some situations (such as large modules) it provides a general, simple, and practical rule of thumb.
- Modules represent either business domain concepts or services. Two commonly-used service modules are the `Frame` module, a base infrastructure module providing framework support to all other modules, and optionally the `Main` module, which glues together all other modules and launches the application[22].

These guidelines build on the existing RCP infrastructure, and also apply to RCPs other than ERCP.

Here we take a more humble approach than that of designing general software components. For example, reuse is a secondary concern in our approach, which focuses on enhancing software design, development, and maintenance of medium and large RCP applications. Modules are a means of rationalizing and raising the abstraction level in the development of an RCP application.

The Snooper application architecture

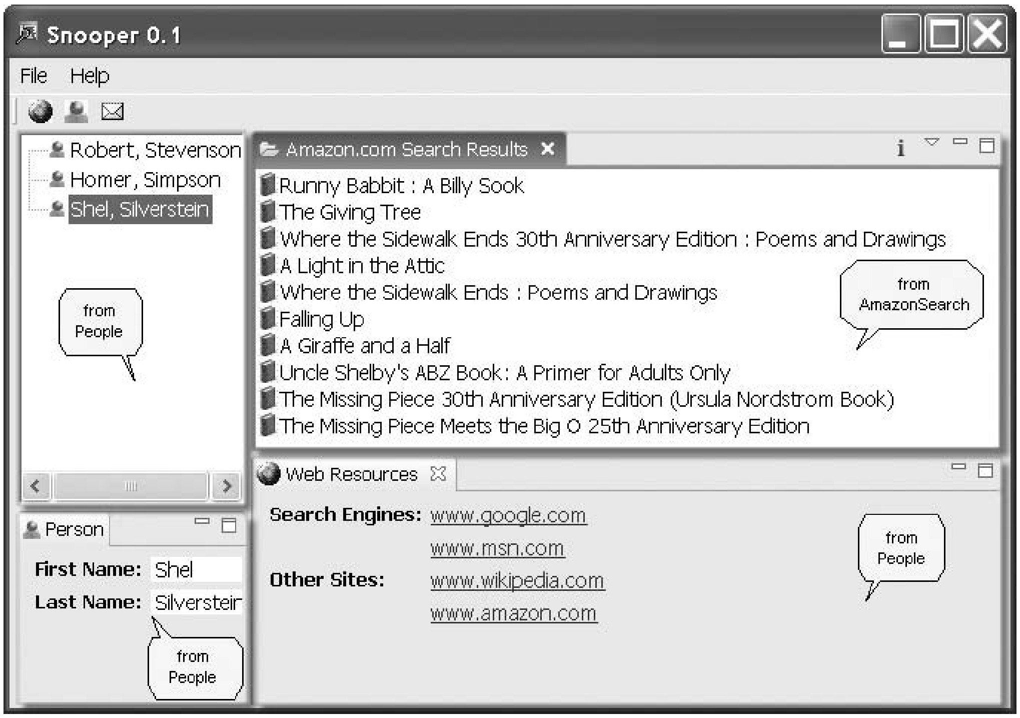
The Snooper RCP application is designed around four modules:

- The `People` module handles:
  - The management of a collection of individuals, shown on the left-hand side of Figure 13.11.

22. In simple applications the one module that has the best fit semantically with the application's purpose can work as the main module, glue together all other modules, and launch the whole application.

  - The currently-selected person, shown in the **Person** view in Figure 13.11.
  - The list of specialized searches for the selected person on a set of search Web sites, the **Web Resources** view at the bottom right-hand side of Figure 13.11.
- The `AmazonSearch` module, which provides search management specialized for the Amazon site. This is shown in the **Amazon.com Search Results** view in Figure 13.11.
- The `Main` module, which glues together the other plug-ins in order to run the application.
- The `Frame` module, which provides common support services to all the plug-ins.

Figure 13.11 shows the ERCP views provided by the various plug-ins in the architecture. References to the underlying platform (`org.eclipse.ui` and `org.eclipse.core.runtime` plug-ins) are excluded for brevity.



Robert, Stevenson
Homer, Simpson
Shel, Silverstein

Runny Babbit : A Billy Sook
The Giving Tree
Where the Sidewalk Ends 30th Anniversary Edition : Poems and Drawings
A Light in the Attic
Where the Sidewalk Ends : Poems and Drawings
Falling Up
A Giraffe and a Half
Uncle Shelby's ABZ Book: A Primer for Adults Only
The Missing Piece 30th Anniversary Edition (Ursula Nordstrom Book)
The Missing Piece Meets the Big O 25th Anniversary Edition

First Name:	Shel
Last Name:	Silversteir

Search Engines:	www.google.com
	www.msn.com
Other Sites:	www.wikipedia.com
	www.amazon.com

Figure 13.11 Component diagram of the Snooper application

The two business domain modules both extend the window support provided by ERCP and provide their own specific interfaces. More specifically:

- The `People` module provides three Eclipse views (**Person**, **People List**, and **Web Resources**) and an extension point useful for attaching customized searches. The `People` module depends only on the `Frame` module.
- The `AmazonSearch` module provides one Eclipse view, **Search Results**, and extends the `People` module's extension point for implementing searches on the selected person provided by the `People` plug-in.

For usability reasons the views are designed so that:

- The collection of people cannot be moved or closed, and its location is fixed on the left-hand side of the main window. This view works as the exploration area where the subject for searches is selected.
- The **Person** view cannot be closed, but it can be moved and docked as required within the main window.
- The other views can all be closed or docked as required in the main window.

This arrangement is just one possible solution: the same application could be partitioned into a different set of plug-ins. More importantly, the design is far from perfect. It should be seen as the first version in an iterative development, in which imperfections are tolerated for the sake of simplicity.

In this example, creating half a dozen almost-empty modules to anticipate their use in future releases could be a mistake, because it entails the costs of creating the modules without any functional added value. This design is also imperfect in that it centralizes too many responsibilities in the `People` module, which could well be split into two or three separate modules in a subsequent iteration. For example, the **Web Resources** view should be a separate search provider, while `People` will remain as a collection of `Person` instances, with `Person` possibly a separate module with its own view, and so on.

Figure 13.12 shows an UML2 component diagram that illustrates the organization of the plug-ins used in Snooper.

The modules in the figure are organized in a layered scheme to aid comprehension. The lowest layer, shown at the bottom of the diagram, is composed of the `Frame` module. Business modules are represented in the middle, while the topmost module depends implicitly or explicitly on all the modules below it. Note that `People` provides the extension point for the various search providers currently installed, currently extended only by the `AmazonSearch` module.

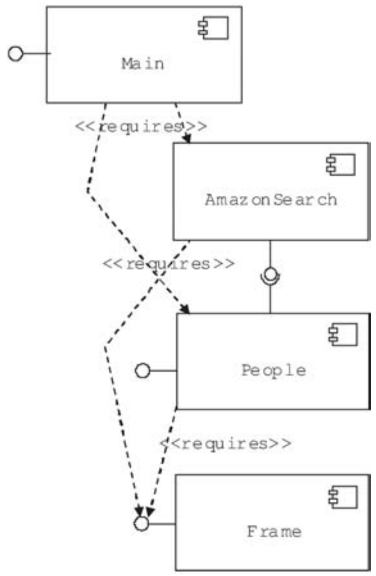


Figure 13.12 Component diagram of the Snooper application

The `People` plug-in exposes the extension point for registering external search providers, as shown in its `plugin.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
   <extension-point id="peopleSearch" name="peopleSearch"
schema="schema/people.exsd"/>
…
```

The `people.exsd` schema file describes the details of the extension point provided. The `AmazonSearch` plug-in extends such an extension point, declaring it in its `plugin.xml` file and implementing the corresponding interface shown in the following section of code:

```
package com.marinilli.b1.c13.snooper.search;
import com.marinilli.b1.c13.snooper.model.Person;
public interface ISearchProvider {
  public void launchSearch(Person p);
}
```

Whenever the **Search** button is clicked in the GUI, all the registered search plug-ins are requested to start a new search using the currently-selected person. This mechanism is implemented by the `People` plug-in, invoking the `launchSearch()` method on all registered plug-ins.

13.8 Summary

This chapter introduced the main RCPs currently available to Java developers, briefly discussing their characteristics. It then focused on the Eclipse RCP, providing an example application that illustrates the many options available to developers. The example is also useful to focus discussions of general design strategies for medium- to large-scale desktop application GUIs built on top of a Java RCP, using the Eclipse RCP as a practical example.

14 The Personal Portfolio Application

In this chapter we will look at a complete application that covers all the important issues in professional GUI design and subsequent development – or, better, two distinct designs and implementations that solve the same needs. The application has been developed from scratch for this chapter. Although the scenario for which it is designed is totally fictional, it was chosen for its resemblance to real-world situations, especially in the common problems it addresses. The demo applications are available for download together with the source code for the book. The scenario is introduced with the first implementation: an alternative approach is discussed later in the chapter.

14.1 The scenario

A publisher of scientific and technical material maintains a large document repository that is constantly growing with the addition of new documents from disparate authors – books, scientific papers, conference proceedings, technical articles, and so on. The repository is accessed by subscription via a Web interface, and is known as the Personal Portfolio application.

One category of users find the GUI rather poor. Librarians, who use the repository intensively, as well as editors and other frequent users, are unhappy with the current browser-based interface, in that it offers limited functionality, and performing advanced and repetitive searches on the repository is extremely time-consuming. Furthermore, the publishers are considering launching new advanced services such as specialized news streaming, personalized e-learning content, P2P-based information dissemination, and more.

An engineering task force has been set up to address the issue. The objective is to come up with an initial working prototype of a new GUI within three months. The whole project is focused on addressing power users and their needs, while creating a platform to which to add future advanced services.

After informal interviews with representative users and within the publisher's engineering branch, the landscape became clearer:

- The old Web-based GUI was adequate for average and non-repetitive users, while repetitive users – for example, those who access the system more than once in a day – need a 'geared-up' search facility.

- Despite the fact that the repository is accessed by thousands of subscribers world-wide, power users who demanded an advanced GUI were limited to roughly a hundred people scattered around the world.
- Repetitive, power users were generally enthusiastic about the project and willing to collaborate in its development. A group of such user representatives was formed to investigate the new GUI, together with people from the publisher's organization. Another, larger, group of expert users was formed for early testing of the prototype.
- The new GUI needs to be made expandable to handle sophisticated services, which will be tested within the power user community. This will give the publisher a privileged channel for testing new services, and in the medium to long term, an edge over their competitors.
- The whole project will not affect the existing GUI and the well-established Web repository site, but will instead be organized separately.
- After an initial start-up phase, during which a preliminary prototype with some limited functionality will be tested within the power user community, the project will eventually evolve into a new, stable service for advanced users. Such a service will provide early access to new features.

That is the setting. Unfortunately, nobody in the engineering team discovered this book, and this chapter in particular…

This development scenario is representative of a type of project that includes the following characteristics:

- A complex, professional GUI needs to be developed within a relatively tight timescale.
- Regardless of the inherently distributed nature of the problem, traditional HTML-base Web-based interfaces cannot satisfactorily be employed.
- The focus is on quality, even if constrained by time-to-market and development cost-effectiveness, just like any other real-world software project.
- Despite a small set of users being located geographically close enough for some preliminary meetings, the test group and the remaining user population is physically out of the reach of the development team. This poses some interesting organizational and technical problems.

A note on lifecycle models

This chapter follows the Rational Unified Process software lifecycle model and terminology. We introduced this model in Chapter 1: for convenience here we highlight the phases of the process only. For more details on UML and the Rational Unified Process, see for example (Fowler 2003) or (Rosenberg and Scott 1999).

The main phases of the software lifecycle according to the RUP model are illustrated in Figure 14.1.

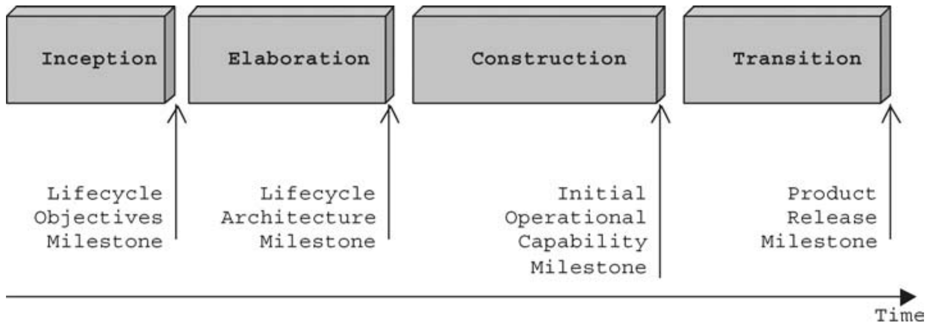

Inception
Elaboration
Construction
Transition
Lifecycle Objectives Milestone
Lifecycle Architecture Milestone
Initial Operational Capability Milestone
Product Release Milestone
Time

Figure 14.1 The phases and milestones of an RUP project

For brevity we will only briefly touch on all the details of each phase – see Figure 14.2: we discuss the software architecture in some detail, as well as the final source code. The deployment, an important part of the whole engineering scenario, is also briefly discussed.

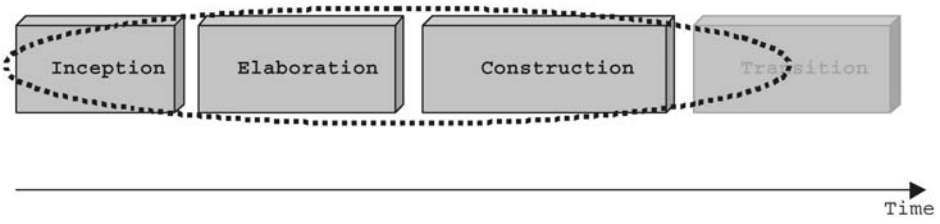


Figure 14.2 RUP phases covered in this example

Management and other issues, such as the business case, evaluation and change control are glossed over, the focus being only on technical details.

14.2 Analysis

The project team started the analysis activity, following the RUP approach.

Early analysis

The core design group of users was chosen from those geographically close to the development team, to speed up the initial design, while the test user group was set up to cover people from different countries to test the prototypes extensively.

These groups were formed taking care to select a representative set of the user population. Parameters like geography, culture, role in the end user population, background computing skills, platform used, and so on were all taken into account.

Figure 14.3 shows an initial use case diagram representing the use cases obtained from an initial analysis.

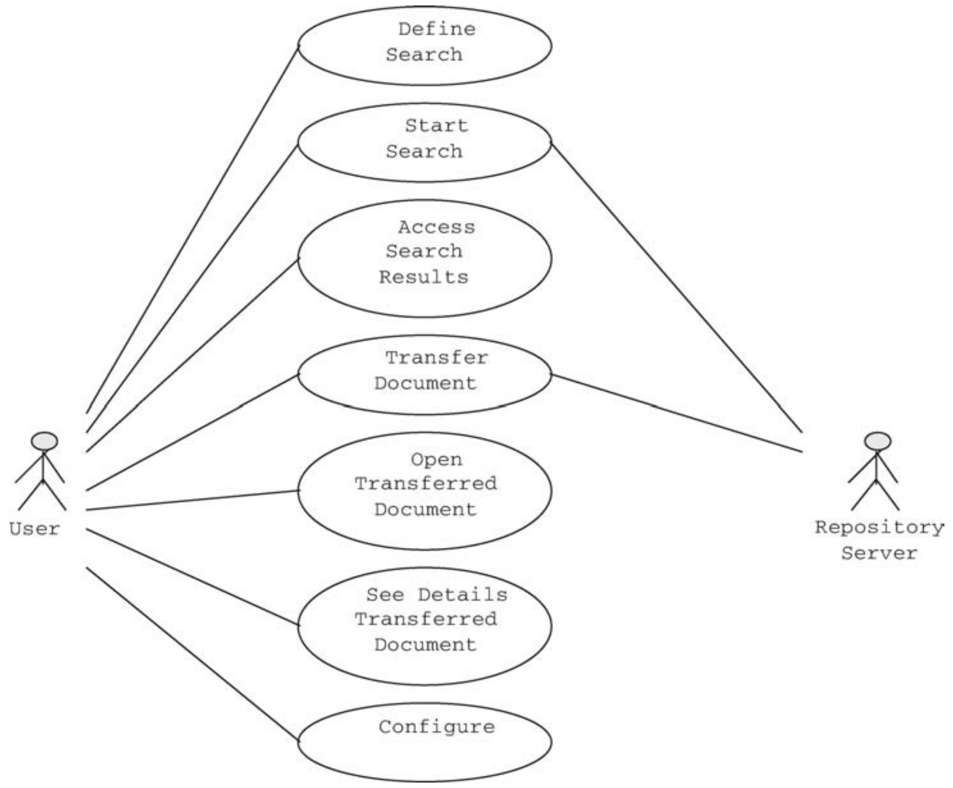


Figure 14.3 An initial use case diagram

Two actors were considered: the end user (a human) and the repository server (a server machine). From this early analysis seven use cases were elicited, as shown in Figure 14.3. Prior to refining the use case diagram, the intent needs to be better defined, by means of a vision document.

The vision

Before getting into the analysis phase in details, it's important to focus on the philosophy behind the product and its clear-cut definition. This focus is a typical RUP concept – that of a clear 'vision' of the product being developed that meets stakeholders' real needs.

In the publisher's situation, the vision is focused around the notion of a personal document portfolio that manages document searches and found documents on behalf of the user. More generally, however, the vision document should fulfill the following requirements:

- State the problem(s) that the application will solve.
- Define the stakeholders, including the end users of the product, and their needs.
- Identifying the product's required features.
- Characterize the functional requirements, which can later evolve into the use cases.
- Define non-functional requirements.
- Identify design constraints.
- Define a glossary of the key terms.

To fulfill these requirements requires a formal document, not just a mere list. This chapter briefly covers the vision for the following aspects:

- Stating the problems that the application will solve:
  - Managing personalized document searches for expert users.
  - Relieving the strain and unnecessary complication of repetitive and advanced searches performed with the existing Web-based GUI.
  - Enabling the creation of an access point for future services for expert users of the digital library.
- Identifying the stakeholders, among them the end users of the product, and their needs:
  - End users: expert, repetitive users of the old digital document repository system. Often such users perform their searches on behalf of others, as is the case with librarians.
  - Engineers and other developers within or external to the publisher.
  - The rest of the user population, even if they won't be affected by the new system.

- Defining the product features:
  - Providing an advanced user interface for the digital repository's repetitive users.
- Characterizing the functional requirements. The system will provide at least the following basic functionalities:
  - Authenticate the given user.
  - Create a new search.
  - Store and reuse/modify a search.
  - Submit the search to the server.
  - Managing billing, subscriptions and other related services.
- Identifying the non-functional requirements:
  - Usability requirements. The application's usability should be at least equal as that provided by the existing Web-based interface.
- Defining the design constraints. The main ones will be:
  - A user's searches should be located on the server to support future features such as collaborative working, and to improve reliability.
  - The software interface with the server should be the same as the old Web-based interface, to minimize risk.
  - In general, the new application should have a minimum impact on the existing server software.
- Eliciting a glossary of key terms. The main terms will be:
  - *Search*. A set of values that embody a query to the remote digital repository.
  - *Document*. A single item obtained by searching the remote digital repository. Documents are consumed by users.
  - *Search preferences*. The information that composes a search. A common set of properties, such as how often the search should be performed, keywords, and the like.
  - *Digital document repository system*. The existing, Web-based interface to the digital library service offered by the publisher.
  - *Personal portfolio*. This concept has been elicited from the early design meetings for capturing the application's basic approach, and subsumes the overall 'vision.' The application will provide a personal portfolio of documents taken from the digital repository on behalf of the user.

The main risks to the project are found to be bounded mainly by usability. Since the application will integrate with, rather than replace, the older digital document repository, it should be superior to the existing application in order to be successfully accepted by expert users.

A further risk is the economic impact of the project.

Some scenarios

The following scenarios emerge from a few interviews with users in the design group.

User search

Since the initial interviews, the focus has been on identifying the most typical interaction. This is illustrated by the following scenario, represented in natural language.

1. User accesses the application.
2. User logs in.
3. User creates a new search and enters the search details, then the search is launched on the server automatically.
4. Server gives a response to the query.
5. The application presents search results.
6. User chooses some documents from the search results and transfers them to their local machine.
7. The application handles requested documents, checking for permissions, security, and performs billing.
8. User opens the transferred documents in the relevant OS-specific tool, for example a PDF viewer.
9. The application automatically saves the user's search for future use if user doesn't explicitly delete it.

Some alternative paths could be:

- Authentication failed.
- 2.1: the user is informed of the problem and a 'Retry Y/N' message is displayed. If the user chooses 'Y,' step 2 is repeated, otherwise the application exits.
- Server or connection is currently unavailable.
- 1.1: the application signals the condition with some unobtrusive feedback. From now on certain functions (like creating a new search) are not available.

User manages search results

Another scenario could involve the following, a session in which the user's objective is to access retrieved documents.

1. User accesses the application.
2. User logs in.
3. User accesses a previous search by means of the GUI.

4. User changes search preferences (search name, search parameters, and so on) to modify the set of retrieved documents.
5. User commits the changes. The search is submitted to the server.

From this point on this scenario follows the preceding one. In a real-world use case document, more refined scenarios would follow.

A refined use case diagram

After further interviews with end users and more analysis activity, the following use case diagram is defined.

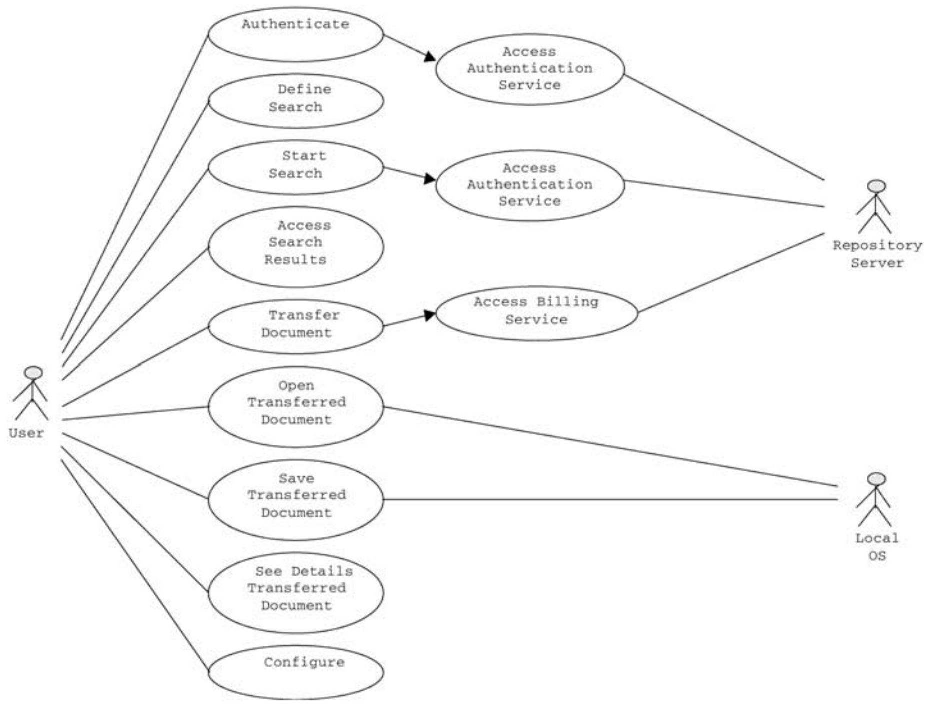

Authenticate
Access Authentication Service
Define Search
Start Search
Access Authentication Service
Access Search Results
Repository Server
Transfer Document
Access Billing Service
User
Open Transferred Document
Save Transferred Document
Local OS
See Details Transferred Document
Configure

Figure 14.4 A refined use case diagram

Note that this refined diagram includes a new actor, the local operating system (OS), which is needed to store and view downloaded documents, together with a more structured use case organization. This diagram forms the input to the first GUI paper mock-up prototype.

The next step in following an RUP methodology is to elicit the actors-system boundary classes.

Individuating boundary classes

Boundary classes are those classes that lie at the boundary between the system and the external actors. Using the concept of a personal portfolio of documents, and studying the previous scenarios from the viewpoint of the end user actor, we could think of the main boundary class as a collection of search objects: search objects gather retrieved documents from the remote repository, and users will mainly deal with a collection of searches.

After initial interviews a partial conceptual model of the application domain is nailed down, as shown in Figure 14.5.

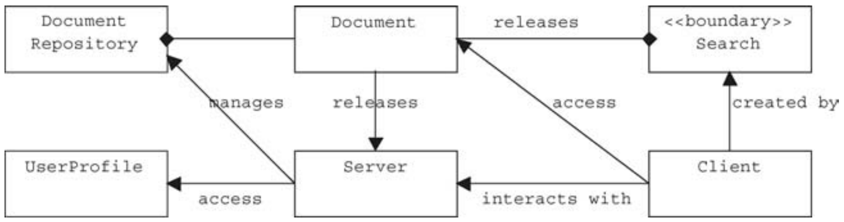


Figure 14.5 An initial conceptual class diagram

The class diagram in Figure 14.5 is related to the conceptual domain only, and shouldn't be confused with the implementation class diagram that we will consider in a later section.

Before proceeding to the design, we need to make some basic choices about the underlying technology. Clearly, our design will be radically affected by this strategic choice.

14.3 Choosing a technology

The choice of a suitable implementation technology is basically restricted to Web-based technology, essentially JSP, php, or other Web page–based technologies, and some form of client technology.

The choice of Java is driven by several cultural and practical issues:

- The repository server is implemented with J2EE technology.
- Developers feel more comfortable with Java rather than with other technologies.
- It takes advantage of existing development tools to minimize risk and additional cost, such as expensive licenses, the evaluation of and training for new software, and so on.

- The end user population relies on recent machines with a variety of OSs, and Java would be a cost-effective choice at least for the first generation of the application – the riskier one.

This scenario dictates that, among the various Java client options, the most useful deployment technology is the JNLP protocol, as it is available for all Java-enabled platforms. Deployment aspects are considered later in this chapter.

The choice of JNLP choice ensures the following benefits:

- A team of developers is available that is familiar with the implementation platform.
- Synergy between client and server technology is guaranteed.
- Ease of deployment and debugging facilities over the Web is supported. This is particularly useful for the frequent deployment of updated prototypes.

Now that a stable set of functional requirements for the application exists and the technology has been chosen, the GUI design phase can begin.

14.4 An initial GUI design

The main tasks that need to be addressed by the application are:

- Authenticate the user
- Create and manage repository searches
- Access documents retrieved by these searches
- Manipulate and modify the searches
- Store the searches

The preliminary design phase produces a number of documents that describe the basic functional requirements, the product vision, a number of use case diagrams and scenarios, plus some models that sketch the application domain, and other informal observations. All this information is ready for convergence into a tentative GUI prototype. The team follows a participatory design approach, in which users actively contribute to the GUI design. This method is chosen because of the nature of the current case – the intended user population, the focus on usability, and so on.

An initial GUI paper mock-up

From an informal meeting among designers, the first design proposal is sketched out.

The main window is composed of a toolbar, a list of all the active searches, and a status bar, as shown in Figure 14.6. Each item in the list represents a search, with a title and other useful data, such as the current status of the search, an is available for quick selection. The commands in the toolbar affect the currently-selected search item in the list.

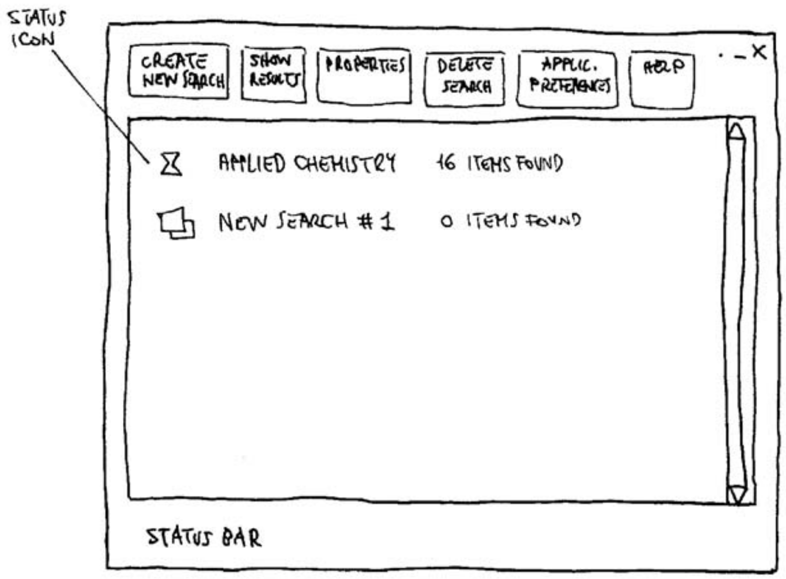


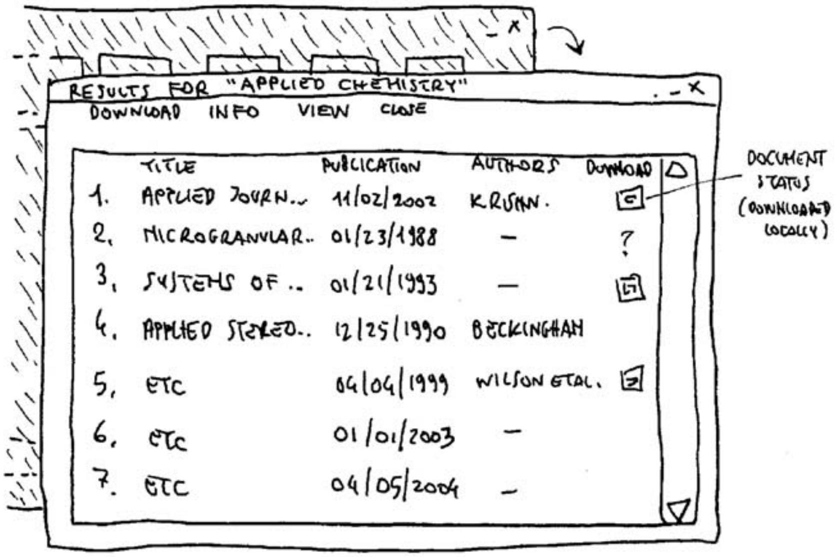
Figure 14.6 The first paper mock-up

By double–clicking on a search item, another window pops up that lists the documents retrieved by the search. This second window is sketched in the paper mock-up in Figure 14.7.

Double–clicking on a single document in the list starts the download process, following which the relevant viewer for the specific document type is opened. The buttons in the toolbar manipulate the currently-selected document in the list. The **Info** command pops up a dialog with the document's data, such as author, publication date, and so on, without downloading the document. The **View** command works like a double click: first the document is downloaded, then it is opened.

From a conceptual viewpoint, the prototype relies on the idea of a list of searches created and maintained by the user. Any of these searches can be opened to show all the retrieved documents and, in turn, the documents can be manipulated by the user.

This mock-up is used as a starting point for a subsequent discussion with representative users. The objective of this second meeting is to produce an early GUI prototype that is representative of user's needs, and which in turn will be validated with a larger user population.


RESULTS FOR "APPLIED CHEMISTRY"
DOWNLOAD INFO VIEW CLOSE
DOCUMENT STATUS (DOWNLOADED LOCALLY)

	TITLE	PUBLICATION	AUTHORS	DOWNLOAD
1.	APPLIED JOURN..	11/02/2002	KRISHN.	▣
2.	MICROGRANULAR..	01/23/1988	—	?
3.	SYSTEMS OF ..	01/21/1993	—	▣
4.	APPLIED STEREO..	12/25/1990	BECKINGHAM	
5.	ETC	04/04/1999	WILSON ET AL.	▣
6.	ETC	01/01/2003	—	
7.	ETC	04/05/2004	—	

Figure 14.7 Another paper mock-up

A second GUI paper mock-up

The design team holds a second meeting, in which a selected group of users discussed the GUI design sketched in Figure 14.6. Their objective is to nail down an initial stable prototype that is validated by representative users.

This meeting results in a major redesign of the GUI: the users explain the most frequent tasks for which they intend the GUI to be suited, using the first prototype as a common discussion ground. A crucial aspect that emerges from this meeting is the desire of users to be able to fully customize their workspace. They see this as one of the major limitations of the existing Web interface.

Many different interaction strategies emerge from this second meeting. End users turn out to have radically different (and unexpected) approaches to solving the same tasks: for example, one uses several slightly different searches to explore the documents on a given topic, carefully recording the best searches on paper for future reference, while another is accustomed to launching broad queries and then scanning the large list that results.

Other important issues expressed by users were:

- To be able to use more natural interaction styles, such as 'drag and drop' (there is a significant Apple Macintosh community among end users).

- The importance of having a lot of information in one screen, rather than continuously switching between different windows.
- Gearing up the GUI for repetitive users, providing powerful 'horizontal' features rather than complex specialized 'vertical' ones – continuously popping up windows to inspect search outcomes seemed too awkward.
- To focus the whole GUI on common tasks, or at least making them as easy as possible.
- The difficulty of making comparisons between two different searches.
- A bookmark concept, very useful in practice, is lacking.
- A properties panel beside the search lists, including some basic information about retrieved documents, would help the details of each search to be inspected more easily.

The designers gather all these suggestions, and after further interaction, worked out the prototype shown in Figure 14.8.

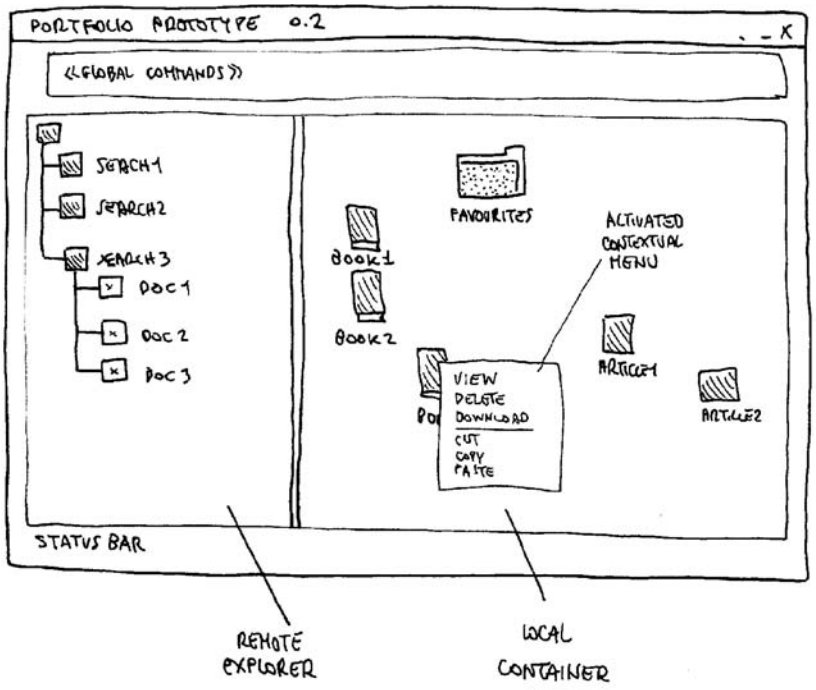


Figure 14.8 The revised paper mock-up

This GUI is substantially different than that shown in Figure 14.6 – 'drag and drop' and a more flexible interaction allow for a richer GUI experience and more intuitive interaction.

Lurking behind the GUI design sketched in Figure 14.8 are some interesting conceptual considerations. As often happens, limitations in a GUI design often derive from a poor conceptual model. As well as the concept of an *explorer* object that represents the searches performed by users, as previously implemented by the list in the prototype in Figure 14.6, the prototype in Figure 14.8 introduces a new and useful abstraction, the *local container*. This is an object that contains all the documents of interest to the current user, and follows the Desktop metaphor used by all modern OS GUIs. This container maintains an image of selected documents that can be manipulated by the user, and documents transferred in the container can later be downloaded to the user's local file system.

A typical interaction for creating a search object now leads to the creation of a new node in the tree on the left hand-side of the GUI, termed the *remote explorer* area. Retrieved documents are shown as subnodes of the search node, mimicking a file system hierarchy. Clicking twice on a document node, or dragging a node to the right-hand side of the screen, transfers the relevant document into the local container. Documents in the container can be managed just like documents in a desktop environment. In particular, by double-clicking on a container document, it is possible to view the document's contents, perform background user authentication, billing, and other operations, and finally download the document content to a local cache.

The 'drag and drop' metaphor is intended to be coherent outside the container as well – dragging a document out of the application window and onto the OS desktop area should cause the document to be downloaded to the desktop or to a target folder.

Locating the exploration task on the left-hand side of the screen and the manipulation area on the right follows a general and widely-accepted pattern in modern GUIs, as we saw in Chapter 4. Taking advantage of this kind of convention is usually a 'win-win' approach: on one hand designers get useful guidelines for limiting the initial design space to the promising avenues, while on the other users feel comfortable using a GUI that resembles software with which they are familiar.

Nailing down the logical model

This is an important and often overlooked aspect of GUI design. Designers may need to change their assumptions later in the development process, but a sound conceptual analysis is still indispensable at an early stage to achieve a professional GUI design. Designers should carefully refine the conceptual model behind the GUI, searching for inconsistencies and conceptual fallacies.

The two abstract concepts the new document repository client's GUI relies on so far are:

- *Remote explorer*. A collection of search objects defined by the user and the retrieved document metadata. The document metadata only contains sparse data, to avoid downloading useless information. The search objects reside on the server, allowing for collaborative features, and are updated when the application is started.

- *Local container*. This represents a collection of information about the documents transferred by the user as a result of document searches. Document transferred into this container are not yet downloaded. The final step in fully accessing a document – after initiating a search, selecting one or more retrieved documents, and transferring them into the local container – is by downloading the document and viewing it via a suitable OS-dependent viewer application. Document metadata stored in the local container is kept in a cache on the user's local file system, together with document content files.

Clearly, this is only the first refinement of the GUI's conceptual model, but it is important to define it a soon as possible, even if it may be changed in the future.

A throw-away GUI prototype

Having tested the mock-up with users, the designers are ready to implement it in order to build a more vivid representation of the final GUI that can be validated by a larger number of users. Technology now enters directly into the design process. The aim is not to produce a working GUI prototype, but rather to capture basic interactions (inexpensively), and hence requirements, in further interactions with users.

They produce the prototype shown in Figure 14.9.

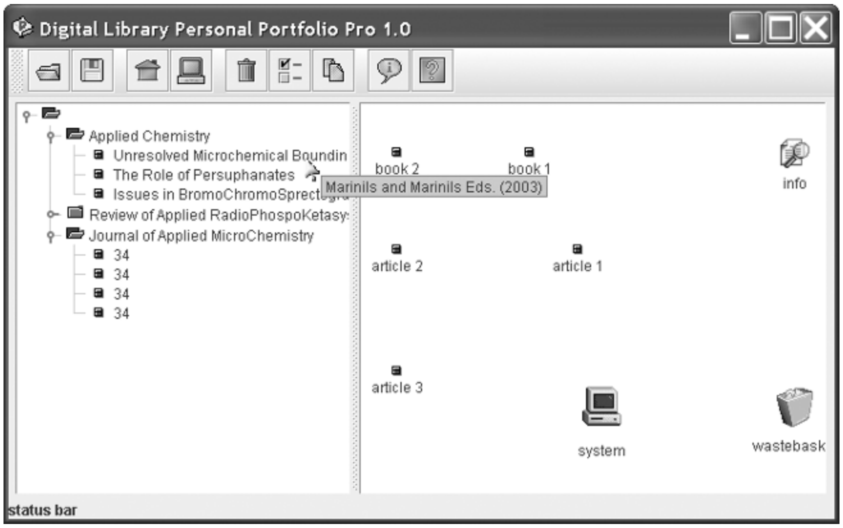

Digital Library Personal Portfolio Pro 1.0
Applied Chemistry
Unresolved Microchemical Boundin
The Role of Persuphanates
Issues in BromoChromoSpect
Review of Applied RadioPhospoKetasy
Journal of Applied MicroChemistry
34
34
34
34
book 2
book 1
Marinils and Marinils Eds. (2003)
info
article 2
article 1
article 3
system
wastebask
status bar

Figure 14.9 The throw-away prototype (Ocean1.5)

The GUI closely resembles the revised paper mock-up discussed in the previous section (Figure 14.8). Users can drag items from the tree on the left-hand side into the container on the right of the window, mimicking the transfer process.

Three system icons are placed in the virtual desktop container on the right-hand side, as shown in Figure 14.10. These icons are used for 'drag and drop' manipulation of documents . For example, dragging a document onto the wastebasket icon removes the document from the local container.

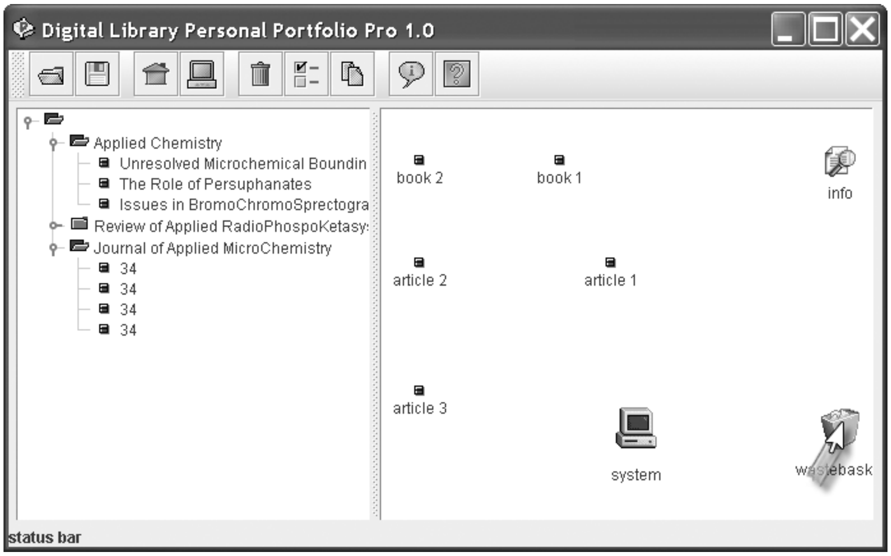

Digital Library Personal Portfolio Pro 1.0
Applied Chemistry
Unresolved Microchemical Boundin
The Role of Persuphanates
Issues in BromoChromoSprectogra
Review of Applied RadioPhospoKetasy:
Journal of Applied MicroChemistry
34
34
34
34
book 2
book 1
info
article 2
article 1
article 3
system
wastebask
status bar

Figure 14.10 The throw-away prototype at work (Ocean1.5)

We will go into the implementation details of this prototype later.

Validating the throw-away prototype

Usability tests done on a larger user population reveal that some of the GUI assumptions were wrong. In particular, the system folders on the right-hand side in Figure 14.9 and Figure 14.10 are misunderstood by the vast majority of users – only few of them can correctly work out their use. Clearly, it seems that the design team, including the users who participated in the design, were biased in their preliminary assumptions.

In such cases – when the interaction needed to activate some functionality is not clear – the best solution is to rely on the underlying platform guidelines. Here the designers adopted the Java Look and Feel design guidelines[1], so in this case a typical interaction would follow a contextual menu style. By right-clicking on the chosen item, users could access all the available functionalities for the selected item.

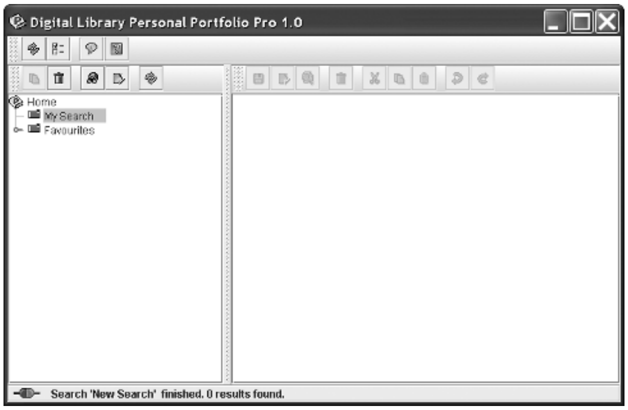
A second version of the prototype is produced in which the system icons were removed. This second version, using contextual menus, was successfully validated with users.

Finally the design team came out with a reliable and detailed design, ready to be used as a specification for the first release of the Portfolio project.

14.5 The final GUI

Before getting into the details of the implementation, let's review the final GUI from an end user perspective. This will help to better clarify the interaction details while keeping the discussion at a intuitive and concrete level.

Figure 14.11 shows how the final application looks. Suppose we create a new search **My Search** using the toolbar button, or by right-clicking on a remote explorer folder via the contextual menu. The new search folder will appear in the remote explorer, as shown in Figure 14.11.



Search 'New Search' finished. 0 results found.

Figure 14.11 Creating a new search (Ocean1.5)

1. See Chapter 2.

After a while the first results appear from the server. We can manipulate them via the contextual menu, as shown in Figure 14.12.

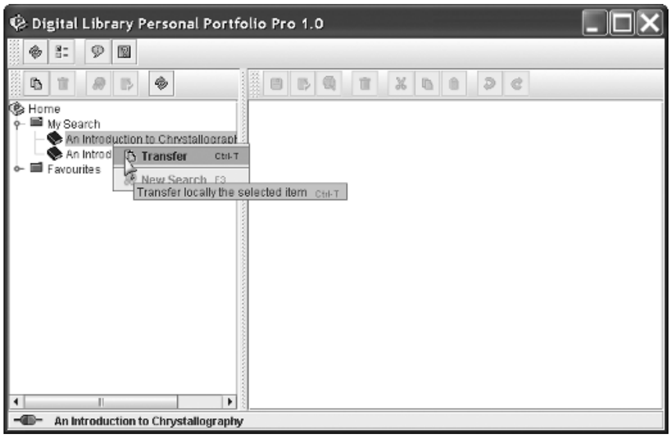

Digital Library Personal Portfolio Pro 1.0
Home
My Search
An Introduction to Chrystallograpi
An Introd
Favourites
Transfer Ctrl-T
New Search F3
Transfer locally the selected item Ctrl-T
An Introduction to Chrystallography

Figure 14.12 Manipulating search results (Ocean1.5)

A document can be transferred into the local container in several ways: by invoking the 'transfer' command, by dragging it into the local container, or simply by double-clicking on its tree icon. When the transfer process begins, the corresponding node in the remote explorer becomes disabled – see Figure 14.13.

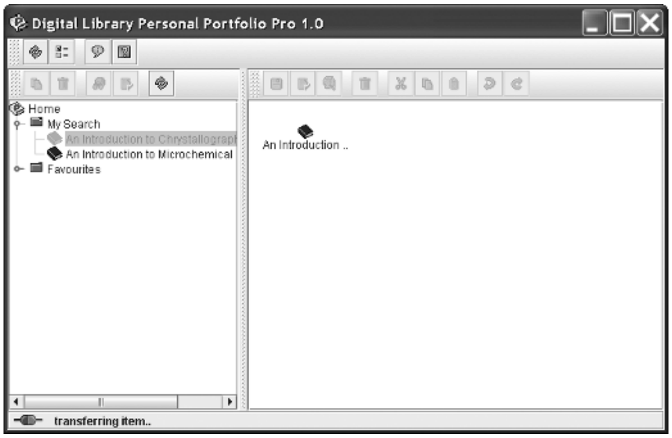

Digital Library Personal Portfolio Pro 1.0
Home
My Search
An Introduction to Chrystallograpi
An Introduction to Microchemical
Favourites
An Introduction ..
transferring item..

Figure 14.13 The GUI at work (Ocean1.5)

When the document is fully transferred into the local container, it can be manipulated with a richer set of commands, as shown in Figure 14.14. Double-clicking on documents in the local container opens them for viewing – the corresponding icon will change to signal this when implemented in the final version.

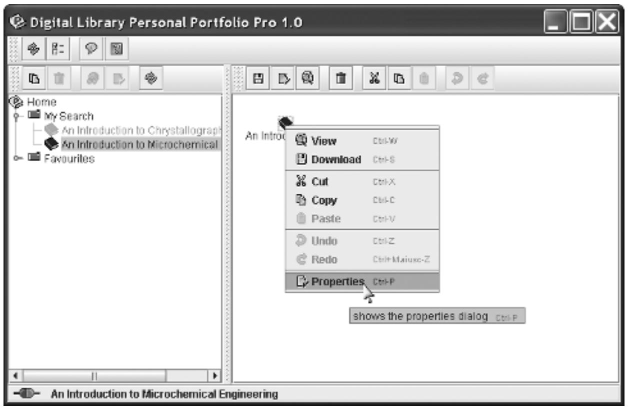

Digital Library Personal Portfolio Pro 1.0
Home
My Search
An Introduction to Chrystallograpł
An Introduction to Microchemical
Favourites
An Introd
View Ctrl-W
Download Ctrl-S
Cut Ctrl-X
Copy Ctrl-C
Paste Ctrl-V
Undo Ctrl-Z
Redo Ctrl+Maiusc-Z
Properties Ctrl-P
shows the properties dialog Ctrl-P
An Introduction to Microchemical Engineering

Figure 14.14 Manipulating a document transferred locally (Ocean1.5)

Apart from the usual operations, the GUI provide a configuration command, **Preferences**, that follows the standard Java Look and Feel design guidelines: icon size, the text used on buttons and other configuration details can be set from the preference dialog[2].

The best way to understand the various parts of the GUI is by launching the demo application and interacting directly with it. Let's now see how it was implemented.

14.6 Implementation

The project team is now ready to get into the implementation of their application. They proceed in a typical top-down manner, beginning from the software architecture and finishing with its final implementation. The description here focuses on architectural and reusable techniques rather than code-level aspects.

2. Preference dialog design is discussed in Chapter 4.

Software requirements

An important step before proceeding with an implementation is eliciting its required properties. These properties can be seen as fine-grained design constraints, as describe in the vision document on page 501:

- Separation into different composable units. Separating the code into coherent parts is a highly desirable property, making the code easier to manage, and facilitating project team structure and possibly code reuse.
- Traceability of detailed software requirements to functional requirements, another desirable property for an implementation, and often mandatory in real-world projects.
- Maximizing software reuse. This could be a rather tough requirement to meet – fully reusable code tends to be more expensive to build and only pays back the investment in its creation in the future.

The team's software solution will be designed to satisfy these high-level requirements, but we don't have the space to discuss their list of detailed software requirements here.

The software architecture

The team begin from the boundary – conceptual – classes established during the analysis process.

Boundary classes

Their earlier analysis had the purpose of better identifying the boundary classes, at least as regards the end user. RUP methodology focuses on beginning the implementation phase from the boundary classes, which in turn are refined iteratively to obtain the final implementation class architecture. The class diagram in Figure 14.15 shows the two major classes with which the user interacts.

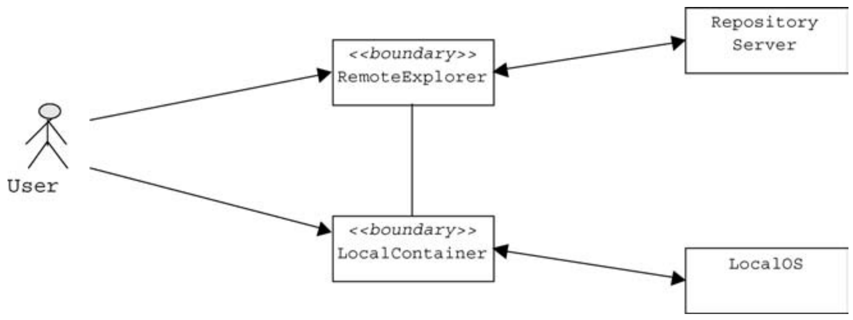


Figure 14.15 Class diagram for main boundary classes

These two classes correspond to the two specialized macro-level components:

- *The remote explorer*, a collection of searches defined by the user and their corresponding retrieved document images. The remote explorer is represented by a dynamic tree view loosely synchronized with the remote document repository.
- *The local container*, a collection of document information manually transferred by the user from the document searches. The local container is represented by a desktop-like container. An example of an implementation of such as container is given in Chapter 16.

In this version of the application the team adopt the OOUI (*Object-Oriented User Interface*) approach outlined in Chapter 2. Chapter 15 contains a practical implementation of such an interface.

The organization of code into packages can be naturally derived from the software requirements, and in particular, the identification of the two specialized components. Such an organization is detailed below.

Package organization

The team know from their analysis that their application will be essentially composed of three parts:

- The remote explorer component.
- The local container component.
- A global framework comprising all global-level functionalities and encapsulating the other two components.

They will then add a further set of logical classes to the latter package for gathering business objects. The package decomposition of the code is shown in the UML diagram Figure 14.16.

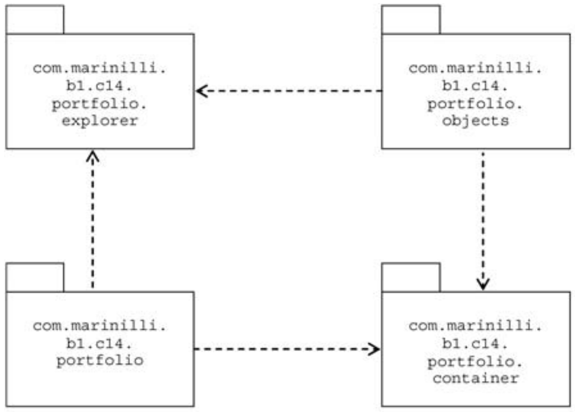


Figure 14.16 Class diagram showing packages dependencies

Note from the figure that the two visual components, which correspond to classes in the packages `explorer` and `container`, don't interact directly, as this would disrupt reusability and code separation, some of the concerns for the construction phase.

The other two packages are designed to allow for the integration of the two components as one coherent and reliable macro-component – that is, the whole application.

The team adopt a typical top-down approach that fits nicely both with the chosen implementation approach (OOUI) and with the RUP design philosophy. The next steps will be iteratively to refine the design to the final classes, then turning them into code. The only digression from this pure top-down approach will be to take the JFC classes that will constitute the basic building blocks into account.

Business objects

Eliciting the business classes involved is a key step in implementation analysis. For the first release of the Portfolio application, only three types of documents will be available through the system:

- *Articles*, both academic papers and technical articles published by some branch of the publishing group.
- *Books*. Book properties include titles, pages, authors, and so on. For simplicity the team assume that searches are done only on keywords, as for all other publication types.
- *Subscriptions*. These are special internal publications from the publishing house.

A `Publication` is the top-level type in the simple hierarchy used for handling published documents. A `Publication` object will follow the OOUI approach: it will be `Viewable` (in the sense of being able to provide graphical views of its content) and `Configurable` (that is, capable of providing special views for configuring itself)[3]. This results in the static class diagram of Figure 14.17.

Publications represent the document data types stored on the server, and are logically gathered in the `objects` package. As we will see, these documents can be alternatively seen as nodes in the remote explorer tree or icons in the local container.

The next step is to refine these two major components.

3. See Chapter 15 for more details about these interfaces.

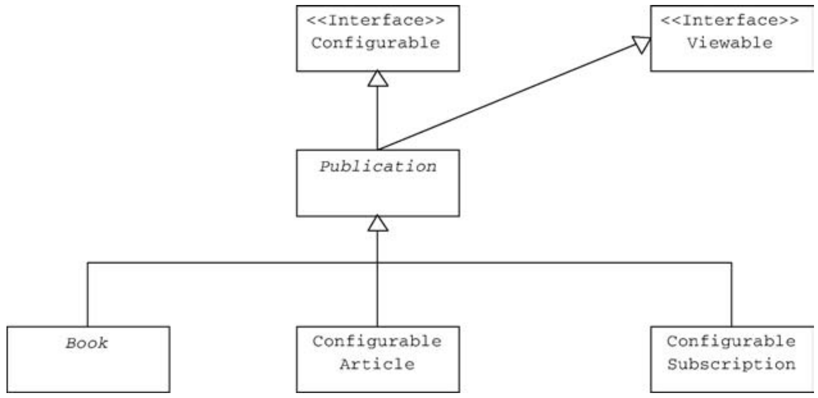
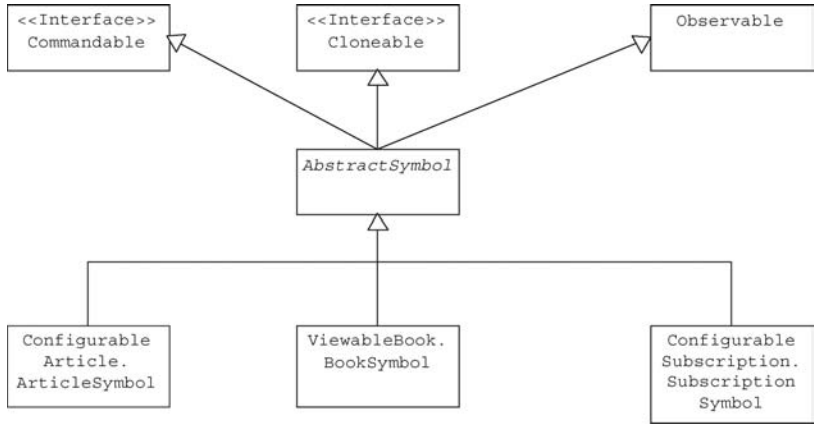


Figure 14.17 Class diagram for publications

The local container

This component is logically a folder of documents selected by the user. We implemented it as a sandbox instance – that is, following the 'desktop' metaphor. Icons in the sandbox all belong to the `AbstractSymbol` class type. Given the type of publications the system will handle, three different concrete subclasses are required for representing articles, books, and subscriptions. Note that in this first release folders are not supported.

The static class diagram is shown in Figure 14.18.



<<Interface>> Commandable
<<Interface>> Cloneable
Observable
AbstractSymbol
Configurable Article. ArticleSymbol
ViewableBook. BookSymbol
Configurable Subscription. Subscription Symbol

Figure 14.18 Class diagram for the local container's symbols

`AbstractSymbols` can issue their own commands within the sandbox container. This is modeled with the `Commandable` interface, and is discussed in Chapter 15.

There is an interesting point to notice at the code level. `AbstractSymbol` subclasses, one for each kind of document, are implemented as inner classes of their corresponding business objects. This implementation approach tends to minimize inter-class references and makes the code more readable, but at the price of a stronger binding among different code packages (here `objects` and `container`).

The remote explorer

The remote explorer component is a collection of remote searches. Each search in turn contains a set of publication nodes, each of which can be of the three different types discussed before, plus search nodes and special 'system' nodes. This is summarized in the static class diagram of Figure 14.19.

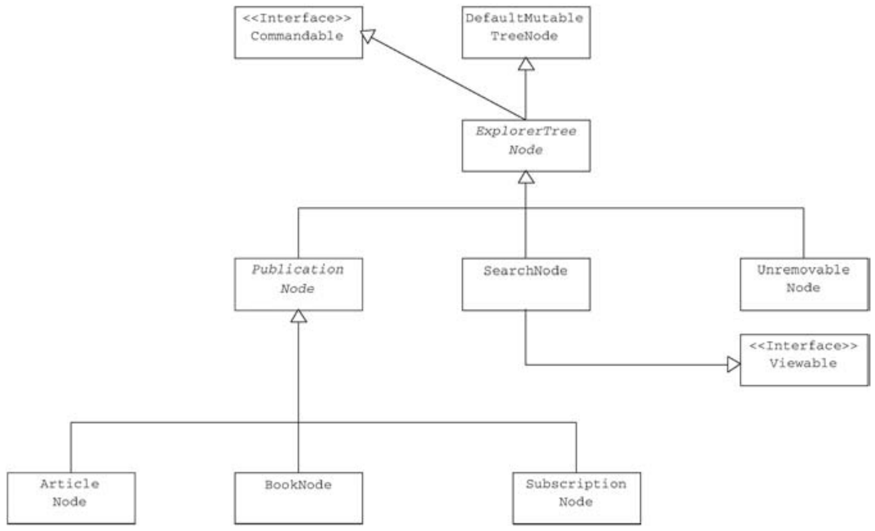


Figure 14.19 Class diagram for the remote explorer's nodes

The publication nodes – elements in the remote explorer tree – are conceptually different than publication instances[4], which in turn are different than publication symbols – that is, items contained in the local container.

4. Document instances are application domain entities.

The retrieval process follows the following mechanism:

- When a search operation is performed, the server returns a set of suitable document metadata bundles to the client. The first release doesn't use any expiration or automatic refresh mechanism, users just have to refresh the search manually.
- This document metadata needs to be as compact as possible to speed up transfer time and ease the burden on the server. It therefore contains only a brief description of each document and the id needed to eventually access it. These lightweight document representations are rendered with nodes in the remote explorer tree.
- When the user selects one or more such document metadata bundles and drags them into the local container, the application queries the remote document repository with the corresponding document ids, and the related publication instances are downloaded. These in turn contain further details of the publication, but still no content data.
- Only when the user explicitly requests download or view of the publication's content, by manipulating the local container representation, is the document content downloaded.

Control

Control here refers to the functional layer introduced in Chapter 1. The GUI must always preserve its coherence when responding to external events, such as the Internet connection suddenly disappearing, or user–initiated events, such as clicking a toolbar button. Interactivity is all about maintaining this consistency, and you can judge professional GUIs by the way in which they ensure the correct behavior under all conditions.

One of the key problems with control code is that it generally needs to span many heterogeneous classes. This gives rise to a natural tendency for control code to be scattered among many different classes, resulting in a spaghetti-like web of class references. This Balkanization of control logic has many drawbacks. First of all, it lacks a clear and systematic software engineering approach – the arbitrary definition of control responsibilities tends to generate subjective code organization that is hard to understand, and even worse to maintain. Furthermore, the web of references among classes may disrupt code reusability and architecture modularity.

We have discussed the Mediator pattern and its variants previously in this book. Now we will see a hierarchical application of this pattern to a concrete, non-trivial case. In our implementation we will call instances of the mediator class *directors*.

We basically have two components in our software architecture. These two components (the remote explorer and the local container) should not interact directly, to promote their future reuse in different contexts. Each has its own specialized director. The simplest integration approach is to provide a third director that will take care of coordinating the other two, while also providing control for global-level commands in the GUI. The class diagram in the following figure shows this architecture.

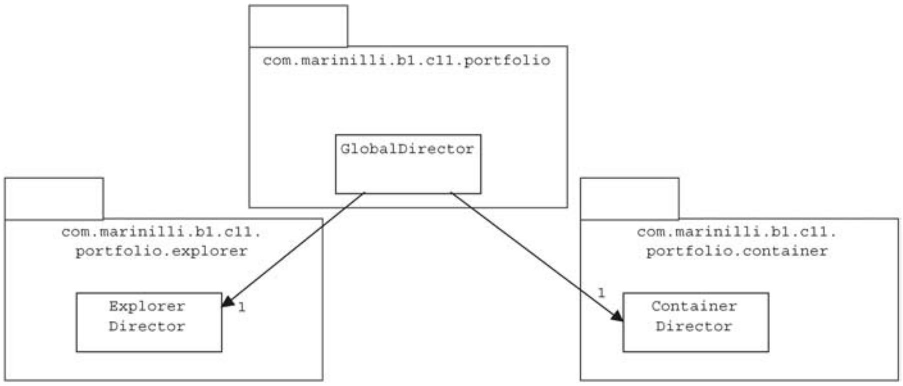


Figure 14.20 A hierarchical organization of directors

This makes both code modularity and a neat class architecture possible. The `GlobalDirector` class is also useful for wrapping components' complexity by providing a single interface to the rest of the world (that is, global-level classes) for many different functionalities.

Looking at the details of the control code for the director classes results in the class diagram in Figure 14.21.

Each director manages its own action classes, whether 'shallow' or 'deep[5].' Hence, for example, the remote explorer director class manages the following actions:

- Delete a search.
- Refresh the current search.
- Issue a new search.
- Inspect a search's properties.
- Transfer a document metadata bundle (the outcome of a search) into the local container.

Note that 'undo' features are only provided in the local container director. This is an accidental consequence of our GUI design.

5. See Chapter 6.

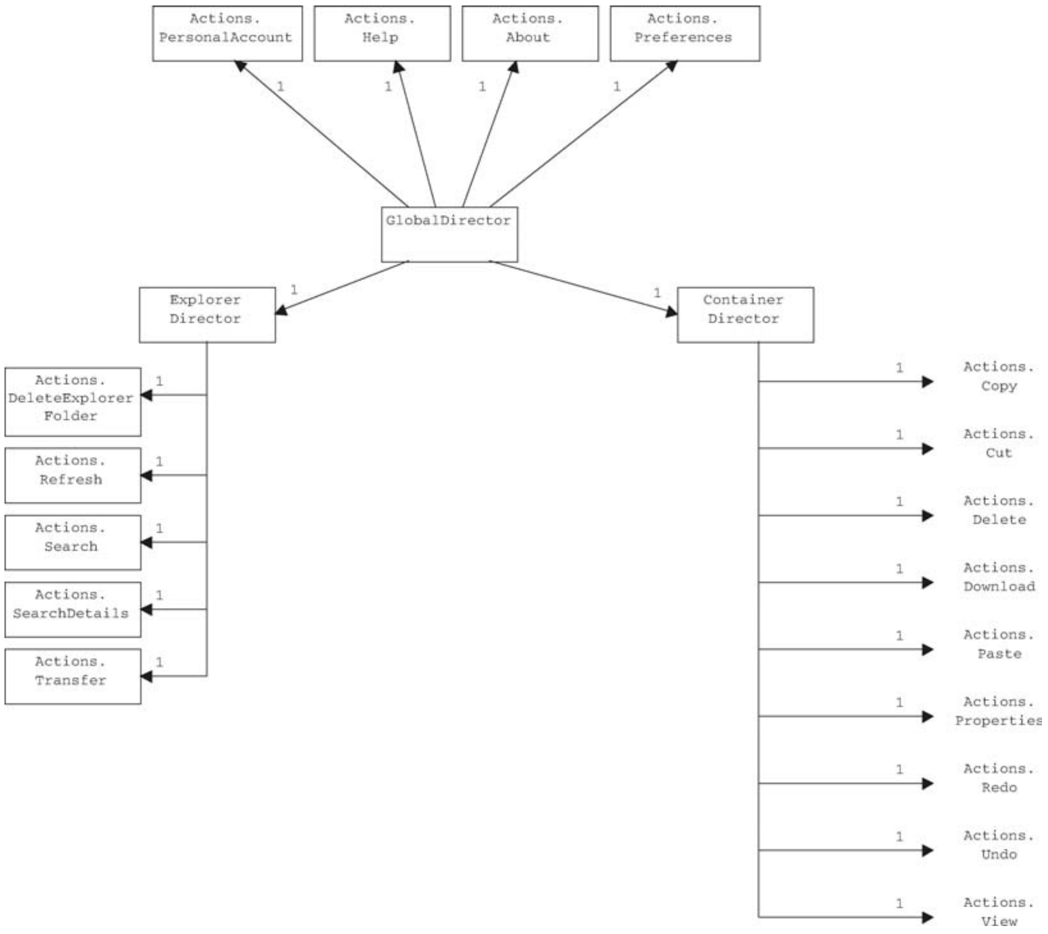


Figure 14.21 Class diagram for directors and their actions

This sketches enough of the static structure of the implementation – now for some runtime aspects.

Start-up

The start-up of a complex Java GUI is always a delicate phase that should be engineered carefully, as the initial impact of an application on its users is greatly influenced by the way it launches. Start-up time should always be minimized as much as possible. Fortunately, this kind of application can be optimized for this aspect.

We saw from Chapter 4 that the main technique for cutting start-up time relies on extensive use of lazy initialization technique and local caching. At start-up an application needs to restore its state, which was stored in an instance of the `Application` class. For simplicity in this implementation we only persistently store and retrieve a small amount of configuration data, but this doesn't hinder start-up efficiency in general for the complete implementation of the Portfolio application. The start-up phase for class creation and arrangement is shown in the sequence diagram in Figure 14.22.

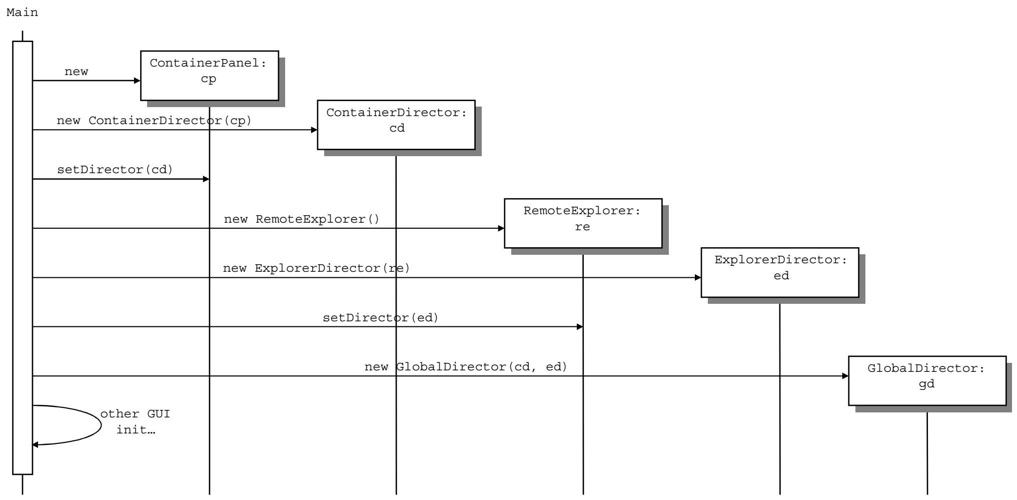


Figure 14.22 *Startup sequence diagram*

This sequence diagram refers to the activation procedure performed by the `Main` class, the main application class. After creating and properly setting all the three directors and their related visual counterparts, some GUI initialization is performed and the application is ready to take off.

14.7 Resources

Resource loading is an important part of any application start-up process. Professional GUIs tend to make extensive use of resources such as message bundles, help data, images, and so on.

Before getting into implementation code details, a few points about resources management are relevant. This is an important aspect for advanced GUIs, but if you are not interested in it, just skip this section.

The code for the Personal Portfolio application makes extensive use of a service layer library. This library provides code with service-layer features such as advanced resource retrieval to support issues such as localization, image management, and other resource-intensive aspects.

Localization bundles

Message bundles are organized on a per-package basis to support localization, as you can see in the sample code for this chapter. Any string and any image can be changed by modifying these text files, which can even be done by non-programmers. But this is not enough – professional resource management also requires the externalization of tooltips, accelerators, mnemonics, and any other locale-sensitive or important data.

Images

The team organize images on a two-level basis:

- When not specified, they are fetched directly by their identifying string as usual. The service library supports a distinction between 'small' and 'large' images.
- When explicitly stated, an image is treated by the service layer as 'large' or normal.

This distinction allows all button icons to be switched from small to large, for example. This kind of functionality is essential, for example for users with visual deficiencies. Apart from this, there are a large number of images that the GUI manages. The main types are:

- Icons, used for buttons and menu items.
- Deployment images, used for short-cuts, during download, and so on.
- Labeling images, used in the 'About' box, in the help data, and so on.

Note that a splash window and its related images has not been provided, to further speed up application start-up.

14.8 The code

This section goes into details of the code behind the Portfolio application. We don't have space to discuss the whole code of the application, so we will focus on a few classes that shed light on the underlying implementation, chosen for complexity and for the reusable high-level solutions they embody.

The remote explorer director

The logical organization of command and control management within the application has already been discussed, so it's time look at the details of a director. The remote explorer director is interesting because it is quite sophisticated, for example by comparison to the `GlobalDirector`, and helps to clarify its basic interaction within the remote explorer.

The basic structure of a director is dictated by its superclass, `AbstractDirector`. Among other things, this affects where actions are located, for example in a hash table for external access, and as instance variables for convenience of internal manipulation, and where their initialization takes place[6].

Actions managed by the remote explorer director are all of the 'shallow' type. That is, they delegate command execution – the code executed whenever the user activates them – to the director. The container director, in contrast, handles a number of 'deep' actions – those that fully implement the Command design pattern. No undoable actions need to be performed within the remote explorer, so deep actions aren't really needed.

A further duty of a director class in this architecture is to package the toolbar and other similar structures so that the external container can place them where needed. This is done by the `getActionToolBar()` method. The director class is also responsible for coordinating the GUI, especially for action enabling. This is performed by the `checkAction()` method that is invoked whenever the director's actions state needs to be updated.

The remote explorer component takes charge of listening for `DropTargetListener` events using the standard methods of this interface. This means that when the user drops something, the standard `drop` method is invoked. This will in turn invoke the `transfer()` method – which can also be invoked by activating the transfer action, or simply by double-clicking on a document metadata bundle in the remote explorer tree. Note that the transfer method essentially fires a `RemoteExplorerEvent` for initiating the transfer process.

The `getActionToolBar()` method is invoked on correct initialization of the remote explorer's content. In the current implementation search objects are not saved persistently and refreshed at start-up, but instead a random search only is added, for demonstration purpose.

Explorer events

The remote explorer director also acts as a source of `RemoteExplorerEvents`. As we know from Chapter 6, events offer one of the most effective techniques for decoupling groups of classes. This allows new code to take advantage of existing

6. See the `setupActions()` method.

classes without modifying them. This technique is used in the `RemoteExplorer` component for interacting with other classes. The events thrown by this class are received by the `GlobalDirector`, which couples the remote explorer with the local container, as shown in the class diagram in Figure 14.23 below.

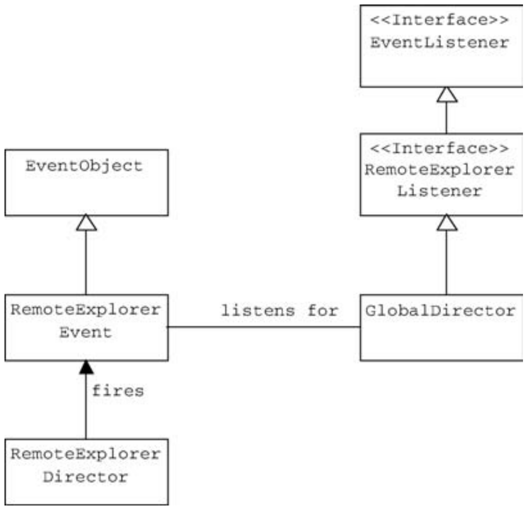


Figure 14.23 Class diagram for remote explorer events

The `RemoteExplorerEvent` class is shown in Figure 14.24.

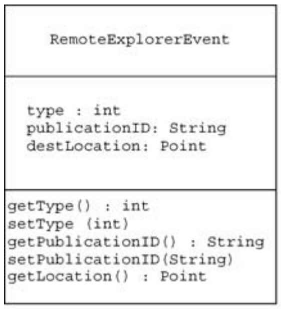


Figure 14.24 Remote explorer event class

Listeners for `RemoteExplorerEvent` events react depending upon the type of event received. The current version supports only `ITEM_TRANSFERRED` events, listened for by the global director, which takes care of transferring the document into the local container, requesting the remote document repository for it using its id.

Representing application data

A common problem for non-trivial GUIs is storage of application preferences and other properties. The Portfolio application uses the Memento design pattern to encapsulate all meaningful data in a single class, the `Application` class, that can be made persistent through sessions.

The simplest way to view the role of the `Application` class is to see it as a simple Java Bean that stores useful application properties and is able to fire `PropertyChangeEvent` events[7] whenever some sensitive property is modified. But this Bean is also able to show its contents graphically for configuration purposes. In fact, thanks to the `Configurable` interface, the `Application` class can create `ConfigurableViews` of itself – that is, of the application's general preferences data –just like any other entity in the architecture[8]. The properties handled by the current implementation of this class are:

- `largeIcons`, a Boolean value representing whether the icons should be large or small.
- `textOnButtons`, which may have one of four possible values:
  - No text to be shown with button icons
  - Text shown on top of the command icon
  - Text shown to the left of the command icon
  - Text shown to the right of the command icon

Note that while the `textOnButtons` property is updated immediately at runtime, the icon size property needs the application to be restarted for changes to take effect.

Implementing searches

The `SearchNode` class helps to illustrate a solution to the common problem of interacting with remote hosts over unreliable and unpredictable connections. This class implements a remote explorer tree node type and contains two inner classes:

- The `ContentView` class supports inspection and modification of search properties such as keywords, text caption, and so on.
- The `ServerSearch` class implements a particular server request.

This simple framework implements a work queue that queues `Runnable` instances to be served by invoking their `run()` methods sequentially. The solution provided here shows an alternative, older design that provides the same functionality that is now provided by the standard `SwingWorker` class, which should be preferred in general.

7. See for example the `setTextOnButtons()` method.
8. See the `ConfigurableView` inner class.

Client–server communication is a common problem for thick client applications that need to connect to a remote computer network – connections delays, and the state of the connection itself, cannot be predicted. When accessing the network, therefore, the best solution is to fork a thread so that the user can perform other operations while the application is waiting for the server 's response.

Depending on the design approach, this will be signaled to the user either by a status bar message, as in the Portfolio application, by a progress dialog that allows users to abort the process, or in other ways, for example by modifying the mouse cursor shape. The demonstration code employs a minimal notice strategy, using only the status bar and visually disabling the transferred/transferring node, because its GUI is intended for experienced, repetitive users.

The prototype

Throw-it-away prototypes are often neglected pieces of software, mistreated by programmers because they serve a limited function, restricted in time and in overall interactivity. A support library can ease the development of such software enormously. The source code for the `Prototype1` class can be found in the source bundle of this chapter – we are now left with the deployment aspects of the Portfolio application.

14.9 Deployment issues

Deployment is an often overlooked part of the software lifecycle. Professional products are characterized by the way they ship and how they can be managed remotely. Deployment services are essential to high-quality software (Marinilli 2001).

Server support

A fully-fledged implementation of Portfolio requires server-side code that responds to clients requests via HTTP. This would have required readers to install a servlet container in order to see the application working. To simplify the installation, the server side has been omitted and has been surrogated by the `ServerProxy` class. This class replaces a real server in many aspects. Although this class is not properly a part of the application, useful only for simulating a real remote server to understand how the application reacts, it is nevertheless necessary to look briefly at how the remote server is simulated.

Essentially, publications on the 'server' side are created randomly. The document repository is simulated by a simple hash table where publications are stored for future retrieval by the client. The process of creating publications is performed in the `getPublicationNode()` method. A pseudorandom number between 0 and

100 is created in this method. If the number is greater than 40, a new node for the current search is retrieved, otherwise the search is finished. In this way searches are filled with randomly-created documents. The retrieval time is also randomized to simulate connection delays, using the `simulateIOLatency()` method.

The server behavior affects the client's performance. When a new search is issued, the number of retrieved documents, and the duration of the retrieval process for each of them that is experienced whenever the user tries to transfer a document into the local container, are determined randomly. This behavior has been simulated to better imitate 'real' server connections.

14.10 An alternative, cost-driven implementation

The GUI design the team proposed, along with its implementation, were both nice and interesting, but expensive to build: the more GUI design involved, the more usability testing is required, and analogously, the more code that is written, the more tests must be created to validate it.

The first design proposed is expensive both because of the GUI interaction devised and because of the relatively large code base needed to implement it. The approach of domain-related composable units might also not be reusable other than in very similar projects, as well as being vulnerable to changes in business details. Focusing on more practical, industrial considerations, a more cost-driven design and implementation would be advisable. Following this approach doesn't sacrifice usability completely, just counters it with engineering constraints.

Here is a description of an alternative design that will solve the requirements outlined the *Analysis* section, but in a more cost-driven fashion.

Choosing a higher-level starting point

Optimal reuse of existing technology and design is key in a cost-driven approach. In the first design the team built a small composable unit framework from scratch and architected the Portfolio application around it. Here we focus on minimizing the amount of code and GUI design we provide, with the idea that 'less is better.' The adoption of effective practices and technologies is instrumental in this approach.

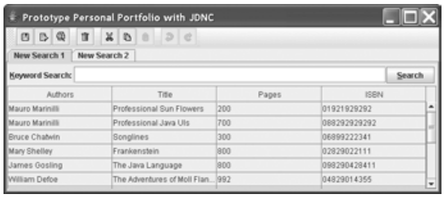
We choose JDNC (Java Desktop Network Components) as the basis of our cost-driven implementation, because it provides a proven, higher-level set of components that will minimize the code base. JDNC is a perfect example of what a high-level, specialized library can do for developers in terms of savings in development time. Many other third-party high-level toolkits exist, but we focus on JDNC because it represents the 'natural evolution' of the Swing library provided by Sun.

Choosing a more sophisticated technology is only one prerequisite for a cost-driven design – the other is its effective use. If I opt for an automatic excavator instead of a shovel to dig a hole, the real benefit will still depend upon my ability to use the tool!

To provide a truly cost-driven design, we need to focus on the GUI design first. By examining the set of requirements in the *Analysis* section, we note that 'drag and drop,' although nice to have, is not essential to a usable GUI, and could well be left for a future release. Further, the nature of the data provided by the server is inherently tabular, not hierarchical. A table widget would represent it in a more effective and inexpensive way. By using a high-level widget, features like search, ordering and filtering will be provided by the toolkit, enhancing the overall usability of the GUI even though using the (supposedly) less intuitive representation of a table, versus trees or other more domain-oriented designs.

A cost-driven prototype using JDNC

The prototype shown in Figure 14.25 was built by implementing only few classes, mostly for data support, with only one class for implementing content, but it nevertheless provides a wide array of GUI features not covered by the previous application, such as ordering, results filtering, and so on.



Authors	Title	Pages	ISBN
Mauro Marinilli	Professional Sun Flowers	200	01921929292
Mauro Marinilli	Professional Java UIs	700	088292929292
Bruce Chatwin	Songlines	300	06899222341
Mary Shelley	Frankenstein	800	02829022111
James Gosling	The Java Language	800	098290428411
William Defoe	The Adventures of Moll Flan...	992	04829014355

Figure 14.25 A cost-driven prototype built with JDNC

Clearly, JDNC components don't allow for the wide array of customizations and design freedom, both in implementation and in the GUI, that are provided by raw Swing widgets, as seen in the previous application. Despite that, they provide a cost-effective solution to most frequent implementation scenarios.

A brief introduction to JDNC

JDesktop Network Components are a family of GUI technologies based on J2SE (and Swing) that aim to reduce the complexity of GUI building in common scenarios, such as data-driven network-rich clients. They are organized into layers so that developers can use those parts that best fit their development needs.

The most basic JDNC layer is a set of Swing classes that extended basic Swing widgets to provide features like table sorting, better validation, and the like.

(These extensions are increasingly being absorbed into the standard Swing library.) A further layer built on top of the Swing extension classes is represented by classes that implement high-level, rich visual components that can easily be connected to data sources and which offer a simplified API for developers not familiar with Swing – although a deeper Swing knowledge is clearly needed for special customizations.

On top of this layer, a further set of classes implement a declarative markup language that can accommodate developers' needs very easily in a restricted, although quite large, number of practical cases.

JDNC and its various layers are a promising and much-awaited development of Swing which, with its basic palette of widgets, is still too labor-intensive to use in professional GUIs. It is yet to be seen whether the higher levels of the JDNC layering scheme, such as the markup language, will prove successful among developers. What JDNC does provide, though, is a very important refinement of basic Swing widgets for common practical cases.

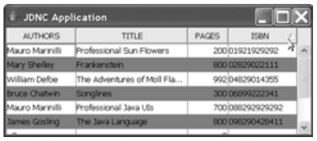
An example of JDNC declarative language

The JDNC markup language allows developers to define most important properties for abstract components, which are then interpreted by the `org.jdesktop.jdnc.runner.Application` class. As a very basic example of this approach, Listing 14.1 shows the definition of a table widget with simple customizations of data source at line 7 and row colors at lines 15–18.

Listing 14.1 Defining a table with JDNC markup

```
00: <?xml version='1.0'?>
01:   <om:resource xmlns:om="http://www.openmarkup.net/2004/05/om"
02:     xmlns="http://www.jdesktop.org/2004/05/jdnc"
03:     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04:     xsi:schemaLocation=
05:       "http://www.jdesktop.org/2004/05/jdnc schema/ jdnc-1_0.xsd">
06:     <table>
07:      <tabularData source="http://...">
08:         <metaData>
09:           <columnMetaData name="AUTHORS"/>
10:           <columnMetaData name="TITLE"/>
11:           <columnMetaData name="PAGES" type="integer"/>
12:           <columnMetaData name="ISBN" />
13:         </metaData>
14:       </tabularData>
15:      <highlighters>
16:         <alternateRowHighlighter oddRowBackground="white"
17:           evenRowBackground="light grey"/>
18:       </highlighters>
19:     </table>
20:   </om:resource>
```

The resulting table is shown in Figure 14.26. All these features are available programmatically using the `JNTable` class. Note the availability of automatic sorting on column headers and field type validation on columns. For example, non-integer values are not allowed in the Pages column, as defined by line 11 of Listing 14.1.



AUTHORS	TITLE	PAGES	ISBN
Mauro Marinilli	Professional Sun Flowers	200	01921929292
Mary Shelley	Frankenstein	800	02829022111
William Defoe	The Adventures of Moll Fla...	992	04829014355
Bruce Chatwin	Songlines	300	06099222341
Mauro Marinilli	Professional Java UIs	700	088292929292
James Gosling	The Java Language	800	098290428411

Figure 14.26 A simple table defined with JDNC content markup

14.11 Summary

In this chapter we have seen a complete yet simplified real-world example application. We discussed all its lifecycle phases, following the RUP terminology – inception, elaboration, construction, and transition – from a practical viewpoint, trying to highlight the interesting points while emphasizing more reusable ideas and solutions.

We discussed the Personal Portfolio application in two proposed incarnations. We have shown with a practical case study how object-oriented technology can play a critical role in developing quality GUIs. Leveraging existing technical skills and the set of simple approaches highlighted in previous discussions can produce top-quality software in a cost-effective way.

15 An Example OO User Interface

In this chapter we will explore some software design techniques for building professional user interfaces, demonstrating a way to implement GUIs with the Java programming language by taking advantage of the OOUI conceptual approach introduced in Chapter 1, within the reference architecture introduced in the previous chapter. As well as providing a set of Java classes that implement this approach, we will also see it at work in a complex example that uses several of the design patterns mentioned in Chapters 6 and 14, as well as a number of practical code tactics. All the ideas proposed here are illustrative and can be used separately in a wide range of contexts.

The chapter is structured as follows:

15.1, Introduction briefly discusses some general characteristics of the implementation solutions proposed in this chapter.

15.2, Implementing object-oriented user interfaces introduces a simple framework for implementing object-oriented user interfaces (OOUI).

15.3, Some utility classes extends the simple framework introduced previously with some useful classes.

15.4, Configuration views discusses the specifics of configuration views.

15.5, Interacting with the user discusses some general-purpose implementation strategies for representing user interactions within the proposed OOUI framework effectively.

15.6, Managing user commands clarifies how user commands are represented in the proposed framework.

15.7, An example application describes the implementation of the Library application using the proposed OOUI framework.

15.8, An alternative implementation using Naked Objects shows a different GUI design and implementation of the same problem using an existing OOUI framework, Naked Objects. It illustrates the great simplification that a specialized framework provides to development, although at the price of a much constrained GUI design.

The chapter concludes with a summary.

15.1 Introduction

As technology evolves, developers and designers can rely on more and more powerful computers, enabling them to afford more sophisticated designs, a trend that doesn’t only apply to the user interface – the software architecture behind the UI has also been evolving. Nowadays, the creativity of software developers can rely on a wealth of computational resources – not a bad thing in itself, but one that inevitably adds a great deal of complexity that must be explicitly addressed.

As in the rest of the book, we use the term ‘GUI’ as a synonym for any generic graphical user interface, OOUIs included.

A matter of style

This chapter illustrates a variety of design choices, so that the reader can grasp the benefits and weaknesses of each. Depending on your needs, you may prefer one solution over another. Personally, I still haven’t found the UI software architecture ‘silver bullet,’ and I don’t expect to find it in the near future.

As an example, let’s take a classic UI software design approach. The top-level container (usually a `JFrame` instance) contains all the required widgets as instance variables, eventually using other GUI-related classes as needed, for example a subclass of `JTree`. Visibility and object communication is provided by the fact that ‘everybody sees each other’ thanks to the instance membership, so that, say, an `actionPerformed` method of such a class can manipulate all the widgets as it needs to.

This is not a bad approach per se – it works wonderfully for small GUIs, it’s simple to understand and to master, it produces GUIs in a breeze – but unfortunately it has its drawbacks. Among these, it doesn’t scale well – have you ever had two dozens or more buttons to cope with? It also tends not to produce readable code for large classes. This approach to GUI development is used by automatic GUI builders found in the most popular IDEs, such as JBuilder, Netbeans Matisse, and the like, and is also one followed by many programmers, especially novices. We will take advantage of it as well whenever it is suitable.

The purpose of this chapter is to explore several design approaches to the software architecture behind a GUI made with Java. The solutions proposed here are partial and not intended to be definitive, as the task of choosing the correct software architecture for a given GUI design is a complex one and involves many variables, such as the architect’s preferences and habits, the GUI’s inherent complexity, the project size, and so on.

The solutions proposed have several properties in common:

- They tend to be initially costly, both conceptually in understanding and practice, and as regards practical coding, as they usually involve a more elaborate code organization, but they will pay back in the long run.
- They tend to scale well – that is, to be more useful for large or complex GUIs designs. They can however be usefully employed for mid-sized or even simple projects.
- They were designed explicitly with priority given to the GUI design. Sometimes programmers tend to favor the software side of the development process, resulting in GUIs that are simpler to build, but which may ultimately be poorer.
- They are illustrative, rather than polished, commercial frameworks. The code proposed here is not intended as a final product ready to be employed in a production environment. However, all its flaws are highlighted.
- They are the result of many years of programming experience.

The code provided for this chapter, apart for the `book` package, which is related to the example application, is also intended to be reusable in other projects.

15.2 Implementing object-oriented user interfaces

Object-oriented programming (OOP) is a good match with Object-oriented User Interfaces (OOUI): although the two concepts are not strictly related, it is not by chance that GUI widgets model nicely as objects.

On the other hand, from an OOP purist's viewpoint, many of the GUI libraries provided with Java, such as Swing or SWT, are not perfectly object-oriented. This is because one of their goals is to enable the composition of a GUI by means of a rapid development tool (RAD), dictating the choice of the Java Beans mechanism. Such *accessory methods* – setter and getter methods, such as `setTitle` – are used liberally in GUI libraries, even if they break the pure OOP paradigm. Accessory methods are a necessary evil, and although they violate one of the main principles of object-oriented development, data hiding, they help developers in many ways if carefully used. Writing your code using accessory methods dovetails nicely with pre-existing standard libraries such as SWT or Swing[1].

Several existing OOUI implementations exist for Java, for example the Favabeans project. Compared with such frameworks, we adopt a more lightweight approach

1. For an alternative approach to these issues, see (Holub 1999).

when designing an OOUI implementation, favoring simplicity and clarity over advanced features and exhaustiveness.

In the following we indicate OOUI objects by capitalizing their names – the Book object – to distinguish them from their Java implementation, the `Book` class.

The Viewable interface

From a GUI design perspective, views are visual proxies of the data manipulated by users. How does one implement such a design paradigm? An object, among its many attributes, may be given the ability to provide a graphical representation of itself. This is not a violation of the data-hiding principle – it is in fact the proper way to build a GUI with an object-oriented language, especially when the GUI design is an OOUI.

Every suitable object should be able to provide a graphical representation of its own state, at least as far as it concerns the current user. We refer to this as a *view* of the object. Providing only one type of view is generally not enough, because we could require the same object to supply several different views depending on external situations. We can also imagine objects capable of providing their own help data, or perhaps providing several levels of detail for the same view, and so on.

As software designers we could provide a method `getView(parameterType)` that would return the proper view given the right type. For illustration, and to make this discussion easier to understand, we instead provide a static list of all possible views in our generic `Viewable` interface. Wherever a new view is required, this generic interface can be extended as required. This approach tends to produce more understandable code, even if its classes won't fully implement all the interface's methods. A specialization of the generic viewable interface that deals with object configuration is considered later.

We will use four different types of views:

- *Brief views* are those that should fit in a small display area, like a short text label defining the identity an object.
- *Content views* are the canonical views – those that show the object's full state, for editing or for inspection.
- *Help views* – it's handy to accommodate the help facility within this mechanism, as it makes every viewable object responsible for showing its own help data.
- *Partial views* are the 'wildcard' views. In some special situations objects need to offer different views, depending on some external parameter such as the current user's role. A partial view is provided by an Object to expose visually some particular aspect of its state. Usually some extra parameter is needed to

specify domain-specific details and a convention shared with the `Viewable` class to use it.

Listing 15.1 shows the `Viewable` interface. Every object that implements this interface provides one or more views of itself to the outside world.

Listing 15.1 The `Viewable` interface

```
package com.marinilli.b1.c15.util;

/**
 * The Presentation Layer
 *
 * @author Mauro Marinilli
 * @version 1.0
 */

public interface Viewable {

  public View getBriefView(boolean editable);

  public View getContentView(boolean editable);

  public View getHelpView(boolean editable);

  public View getPartialView(boolean editable, Object
argument);

}
```

Although the method naming seems correct, it hides a pitfall. In most common cases objects are intended to create a new view whenever one of the methods in the `Viewable` interface is invoked. So, for example, `getBriefView` should be renamed `createNewBriefView`. Some objects may have the semantic constraint of always returning the same view instance, so that the view is *static* to the class and not dependent on the particular instance. For simplicity, to maintain the same signature for every `Viewable` class, we keep this potentially misleading naming convention.

Some simplifying assumptions

A general note is needed at this point. We are planning to build, bit by bit, a complete framework to address the most common issues of GUI software design. We need to make some choices and to decide the level of complexity and resulting sophistication of this framework. For example, I used to have an `EmptyView` Singleton object, a subclass of `View`, to neatly handle the case of a view type requested from an object that doesn't support it. This solution turns out to be especially useful during development – I even subclass it for more specialized behavior. To keep the discussion focused on general GUI software implementation patterns and to avoid getting bogged down in detail, we'll skip this kind of refinement and provide just the basic framework, stopping in the middle of the route towards a comprehensive, sophisticated GUI class framework. The set of classes proposed here, and the concept behind them, can nevertheless be adapted to manage complex GUIs.

A key factor in building professional GUIs is reliance on a set of utility classes that consistently and exhaustively adopt and enforce the use of a coherent set of GUI design guidelines – in our case, the standard Java Look and Feel design guidelines. It's also important to be able to build high-quality GUIs quickly and cheaply. These advantages can be obtained both with general-purpose reusable classes and with more abstract design patterns, focusing on GUI design as well as software design, as we saw in Chapter 4 and we will see in the remainder of the book.

The OOUI approach can be employed with SWT and other GUI toolkits as well.

Implementing views

Viewable objects return views of themselves, but is a `View` class really necessary? Why not provide a `Component` – that is, a generically displayable object, both in Swing and AWT frameworks – directly, instead of yet another layer of indirection?

Having an explicit, general `View` type is handy in many situations. For completeness we want to implement both standard deferred and immediate mode interactions[2], so we need some form of control over the view object. Consider the common situation of modifying the widgets' state in a dialog and then saving the changes with the **OK** button. The consequence of this action should be to commit the changes to the view, which is in charge of keeping view's screen data state[3] aligned with the related domain data.

When working in deferred mode there should be a way to signal to the view the undo of any changes that have occurred so far, so a `doRollBack` method should be provided. It's possible to imagine several mechanisms to describe this behavior. We will choose the simplest: just one interface, `View`, that models both deferred mode and immediate mode behaviors. Usually only one of these two methods will be used. The `View` interface therefore has three methods: `doCommit()`, `doRollBack()`, and `getComponent()`.

2. See Chapter 4, *Waiting strategies* on page 141.
3. See Chapter 8, *Runtime data model* on page 329.

Conceptually, a view is responsible for coupling the widgets and the domain data. Widgets' data, such as the string manipulated by `setText/getText` in a `JTextField` component, is thought of as a kind of temporary buffer that exists as long as the corresponding GUI item exists. The domain data is the source and the possible destination of the widgets' data, following the MVC pattern.

The approach of using `View/Viewable` and other auxiliary classes can be extended to handle every GUI transaction, so that even the main application window could show itself via this mechanism, for example. We are not interested here in proof-of-concept of a pure OOUI implementation with Java – we are more interested in exploring some useful design solutions in order to apply them in common programming practice. We therefore limit the use of the `View/Viewable` mechanism to domain objects only, instead of applying it to every GUI object.

Let's recap on what usually happens. Figure 15.1 shows a typical `View/Viewable` interaction, in which a visual container, say a `JPanel` or a `JFrame`, asks `viewableObject` for a given visual representation of itself. The graphical container extracts a visual component, say a `JLabel` instance, from the returned view and composes a larger view with it as required. The MVC pattern fits well with this approach: models are the responsibility of the `Viewable` object, while related Swing widgets are provided by `View` instances.

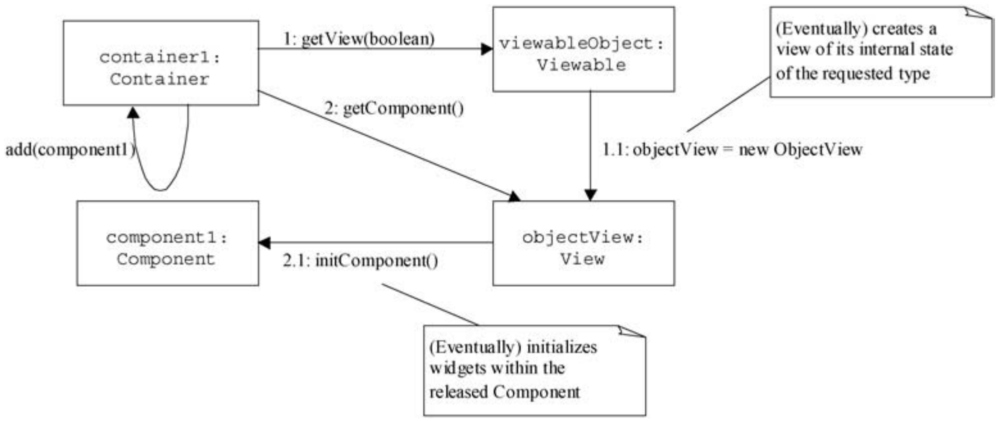

container1: Container
1: getView(boolean)
viewableObject: Viewable
(Eventually) creates a view of its internal state of the requested type
2: getComponent()
add(component1)
1.1: objectView = new ObjectView
component1: Component
2.1: initComponent()
objectView: View
(Eventually) initializes widgets within the released Component

Figure 15.1 A typical interaction

The `View/Viewable` mechanism described here has several benefits for software developers:

- It tends to produce more reusable code. GUI code is encapsulated in the related class, so that whenever the class is reused in a new context, the chances are that its views will work in the new GUI as well.

- The mechanism works well with high-quality GUI projects. The extra care and discipline spent in the software design turns out to be paid back in terms of reliability and overall quality.
- It can be used in its broadest form as a common ground for discussions among development team members, either engineers or GUI designers.
- It creates a common ground that can be exploited to produce a number of reusable utility classes that turn out to be very useful in practice. This is essential for minimizing the cost of software development, especially in terms of time. We will not introduce such utility classes here because they would be of little conceptual interest, and because you are encouraged to develop your own. The more OOUI-aware reusable components are employed, the more this approach pays back.

The approach has also some potential drawbacks:

- It tends to produce more intricate code in which the same results are obtained in a more indirect way compared with more straightforward, simple software design strategies[4].
- It needs time to get used to, especially initially, making it hard for developers to get involved in the project, especially at later stages. Once this strategy is mastered, designers usually tend to develop frameworks on top of it, which in turn makes it hard for newcomers to pick up the details of a project.

15.3 Some utility classes

There are a number of refinements that could be added to the ideas exposed above. We will see just a few, to give an idea of the possibilities and of the most common issues they raise. Any class library should provide convenience classes to address the most frequent cases.

Brief views

Let's look at some default implementations. Given the ability to handle generic views, we could use default implementations to ease their creation.

A brief view can be implemented in a standard way, by means of a single `JLabel` paired with a 'More' button. If the view is editable, the button is enabled, allowing the user to modify the object's state – the implementation of the `More-Button` invokes the content view for editing. We can see an instance of this class at work in Figure 15.6 on page 548, where it is used to describe the book template currently used.

4. The advantages of such an apparently convoluted approach depend on the nature of the project. Another example of this approach is shown in Chapter 14.

What we refer to as a 'More' button is a button with the conventional label '**...**' (ellipsis), indicating the generic behavior of opening another window for further interaction. An example is shown in Figure 15.2 on page 545 in the Book's brief view. For more details, see the use of this convention in (Java L&F Design Guidelines 2001).

The implementation of the `DefaultBriefView` class provided in the code bundle for this chapter illustrates some interesting points:

- It consists of a brief view, obtained by a viewable instance passed through its constructor, with a button placed beside it. Whenever the user clicks on the button, a content view of the same viewable object pops up automatically in a deferred mode standard dialog.
- The model behind `DefaultBriefViews` is a subclass of `PlainDocument`, the simplest text model provided by Swing. By taking advantage of Swing's data models, we are implicitly using the MVC pattern in our class framework and leveraging its advantages.
- We handle widgets within the class itself. This turns out to be by far the simplest and most effective approach in situations with specialized, reasonably short classes.
- The `DefaultBriefView` removes its visual components from the underlying model whenever they are removed from the container that held them. This greatly improves garbage collection, minimizing the risk of dangling pointers.
- For simplicity, we implemented the `View` interface in the widget class itself, a tactic that you will see often in the demonstration code. This minimizes the possible problems caused by the additional indirection layer introduced by the Viewable-View mechanism.
- Implementing the `View` interface on the visual component itself can give rise to unexpected problems, the commonest being name-space pollution. Widget classes usually come from deep hierarchies and have large signatures (many methods) that can conflict with domain-related names for methods or fields in the business classes, which belong in the application layer in our terminology.

Other reusable classes could be provided, for example for handling general collections of items.

Making collections viewable

The other general-purpose class presented in this chapter, `DefaultViewableList`, deserves comment. Being reusable, it can be used in many different contexts. It subclass the Swing default model class `DefaultListModel`, and provides content views in the form of `JList` instances, opportunely bounded to the origin's model. At runtime such views can be manipulated by the user with a contextual menu to provide the available commands.

We adopt a fully-automated approach to list manipulation. Those operations that are allowed on the origin's collection instance (create a new element, remove a selected element, modify a selected element) are available to the user. This aspect crops up again in the discussion of command composition later in this chapter.

15.4 Configuration views

Interactions with a GUI can be thought of as being divided in two broad groups: operational and configurational. When you drive a car, you turn the steering wheel and operate the pedals to control the vehicle. This is a normal, operational interaction. If there is too little room, you can move the seat back and modify its geometry accordingly to your preferences. Here you are configuring the car, that is, modifying some parameters that are changed less often and that don't impact on normal operations.

Configuring GUI items is a common operation, and specialized facilities, often under the form of a configuration dialog, are found in many GUIs.The Java Look and Feel design guidelines terms such operations *preferences*.

A `Configurable` type that specializes `Viewable` for objects that can be configured is available with the source code bundle for this chapter. This interface is made up of two methods:

- The `getConfigurationView` is the straightforward specialization of the `View/Viewable` mechanism for configuration views.
- The other method, `getCategory`, is an auxiliary method used for tagging the configuration views, and is used in a specialized container.

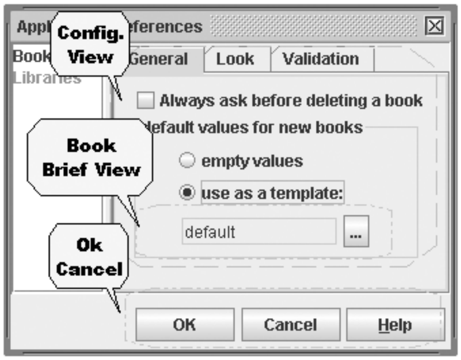
We will see this interface at work in the following section.

A utility class

Designing OOUI objects requires a decision about whether to make them configurable – that is, to make them implement the `Configurable` interface. Although in theory a utility class can inspect the current namespace to discover all `Configurable` instances automatically, it is far simpler to embed in the code the list of configurable objects gathered in a specialized dialog. This list of configurable items changes only at design time. In the implementation proposed, we have designed a `ConfigurationDialog` class that gets an array of `Configurable` objects as a parameter for its constructor.

To see some real action, we have to anticipate this chapter's example application. Such an application offers two OOUI objects for user manipulation, books and libraries. That is enough to understand the next couple of figures, while we will see the details of the Library example application itself later in the chapter.

Figure 15.2 shows the structure of the Configuration dialog used by the example application. Within a deferred mode dialog, a `JSplitPane` divides the list of all available items on the left from the configurable data presented on the right. When the user selects **Books**, the book's `ConfigurationView` is shown on the right. This design avoids a confusing set of nested tabbed panes, while preserving the same data presentation[5].



Always ask before deleting a book
Default values for new books
empty values
● use as a template:
default

Figure 15.2 The configuration dialog

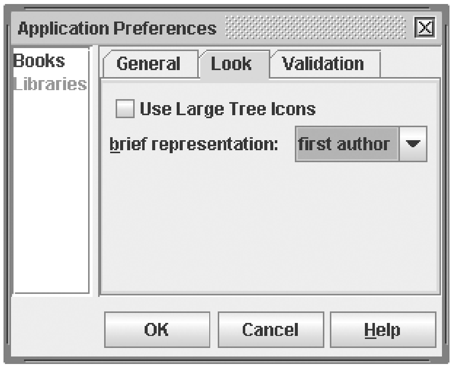
Note the following:

- The aspect of the item presented on the left – the categories into which the system configuration is organized – is not bound to a given class, but is obtained by querying the `getcategory` method in the `Configurable` interface.
- The `Configuration` view of the `Book` class uses a brief view of itself to describe the template book used for creating new book instances. More precisely, they are two different instances of the same class.
- This utility class can be used for *any* configurable set of classes. For enhanced usability, each of these classes should oblige their configuration views to use

5. Although with some limitations.

a deferred mode style of interaction, otherwise users could become confused. This constraint is not enforced in our class framework.

Figure 15.3 shows how the book OOUI object's appearance can be configured.



Use Large Tree Icons
brief representation: first author

Figure 15.3 The book configuration panel -- book appearance

15.5 Interacting with the user

In the last section we discussed an approach to the task of building a configuration facility for GUIs. This section looks in some detail at the more general problem of setting out the software architecture for managing effectively GUI-user coordination.

The Commandable interface

We plan to make OOUI objects capable of releasing different visual representations of their internal state. What we need is a way to model the set of possible actions that each of them can support. Users are already familiar with contextual menus and with right-clicking on a GUI object to see what commands the object supports. The only method in the `Commandable` interface therefore returns an array of menu items suitable for attachment to a pop-up or drop-down menu. We use an array of `JMenuItems` instead of a generic collection make the code easier to understand.

Composing commands

A common approach is to compose commands from multiple OOUI objects at a centralized point in the GUI, to facilitate user access to specific actions, such as an object that returns menu items ready to be incorporated into a toolbar or a menu bar. This behavior also occurs in classes that are responsible for contained objects, in a conceptual hierarchy that is similar to the Facade design pattern.

Such behavior can be created using the `DefaultViewableList` component. Despite its long name, this is just a visual container of generic list objects. The container itself issues several collection-related commands, such as add and remove items. When the user selects a given object, the list of all available commands for the object is displayed in a pop-up menu, as shown in Figure 15.4.

Behind the scenes, the container negotiates with the selected `Commandable` object to acquire the list of available commands to be incorporated into the menu. In Figure 15.4 Library objects can be created and removed, commands that are the responsibility of the visual container, and offer two additional commands:

- **Properties**, to inspect an object's internal state.
- **Inventory**, which is a Library business-related command.

Note the separator in Figure 15.4 that divides the two command groups.

This mechanisms allows for maximum flexibility – every object knows which commands it supports and whether they are enabled or not at any specific moment, while maintaining clearly-defined responsibilities between different OOUI objects.

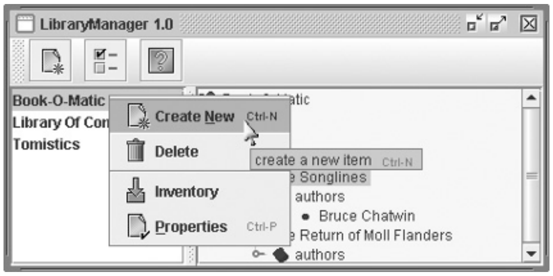


Figure 15.4 The contextual menu for libraries

Similar behavior is employed in the tree view for Libraries, where no object-dependent action is allowed, as shown in Figure 15.5.

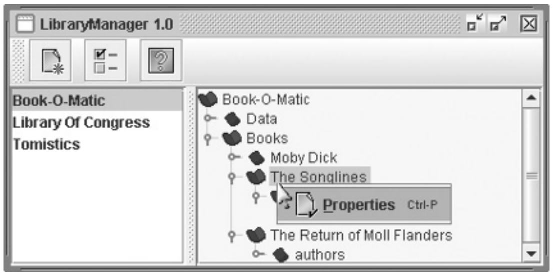


Figure 15.5 The contextual menu for books

The DefaultViewableList class goes a step further, adapting its menus dynamically to the kind of list in use. For a collection that cannot be expanded or shrunk, for example, only the **Properties** command is available, as shown in Figure 15.6.

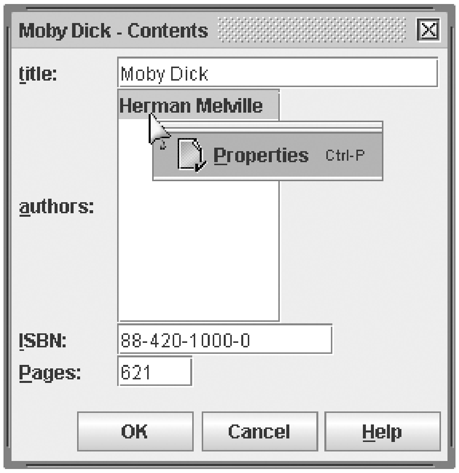


Figure 15.6 The contextual menu for a fixed-size list

The object interactions typical of this mechanism are illustrated by the sequence diagram in Figure 15.7.

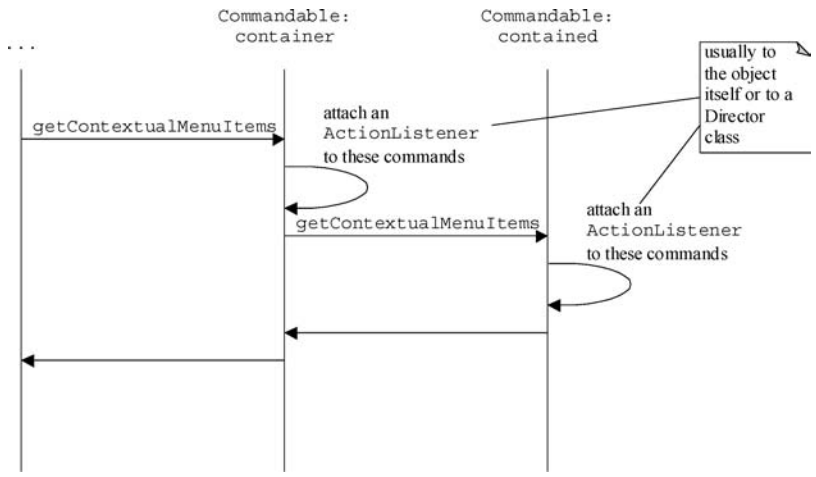

...
Commandable: container
Commandable: contained
getContextualMenuItems
attach an ActionListener to these commands
getContextualMenuItems
attach an ActionListener to these commands
usually to the object itself or to a Director class

Figure 15.7 Typical interactions for composing commands

15.6 Managing user commands

So far we have looked at the `Commandable` interface and two different uses of the `Action` class. It's a good idea to take additional care when designing user commands. Apart from standard commands like **OK**, **Cancel**, **Help**, commands should be designed both from a GUI viewpoint and from an implementation viewpoint, because they will be the main interaction points with users. Whether you use shallow or deep actions in your programs, it's useful to keep them centralized in repository classes in your code.

The example implementation uses two classes:

- `Commands`, a factory class that creates all the actions used in the GUI, indexed by unique string identifier.
- `ActionRepository`, an interface that contains a version of `ShallowAction`, an implementation of the `AbstractAction` Swing class seen in Chapter 6.

The factory methods in the `Commands` class could have been implemented by cloning prototypes stored in a hash table and indexed by the command strings. Instead, the more hard-wired approach of a long chain of `if` statements is used here. As a rule of thumb, it's a good idea to consider the dynamic option (the hash table) when the number of actions is greater than about a dozen.

Indexing actions with a unique key string has several advantages:

- It keeps the action's creation centralized, for better code readability and maintenance.
- It allows for faster string comparison (the '==' operator can be used).
- It avoids any spurious operation with actions, such as a string mistyping, which could be hard to track down in large programs.

In the solution proposed here, such strings are treated as class tokens, in that they uniquely identify a class rather than a single action instance.

The 'token' command strings in the `Commands` class are only used internally, and not for presentation, so they must not be localized.

To keep the example simple, actions are not localized. Chapter 8 contains examples of localized (locale-dependent) actions.

We are now ready to put all these pieces together in a concrete example application.

15.7 An example application

The task is to build a GUI that manages various libraries. Libraries have several attributes (such as name, address, and so on), and a collection of books that are publicly available.

We will build this application using the OOUI approach to drive the application implementation. The example was constructed with the idea of using an architecture of sufficient complexity to illustrate some of the main issues that can arise in a real-world software development environment following the OOUI approach. However, too complex an example would be confusing and distracting.

The example shows a GUI that serves merely as a showcase for the ideas presented earlier. Try it for yourself: compile the code provided and run the `com.marinilli.b1.c15.book.MainFrame` class, or more simply, just launch the `c15.jar` file provided for this chapter.

The main window in Figure 15.8 shows a list of libraries on the left and the currently-selected library on the right.

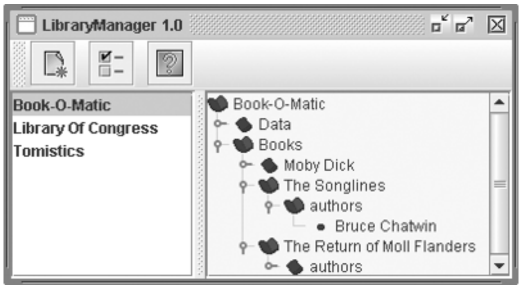


Figure 15.8 The library application GUI

Library objects can be manipulated by right-clicking on the list of the available libraries on the left-hand side. In particular, domain-dependent operations can be invoked, such as making an inventory of the selected library, or performing collection operations such as creating or deleting libraries. The implementations of these actions are all provided by the standard behavior of the `DefautViewableList` class.

A few sample commands have been implemented – the toolbar shows three. As well as library creation, there is a **Preferences** button, which displays the system configuration dialog, and a **Help** button.

The nice thing about the OOUI approach is that it spans naturally from the GUI design phase to the implementation details. Hence we focus our development effort at designing and building the OOUI objects employed in the GUI that will later be implemented as Java classes.

OOUI objects

Chapter 1 showed that OOUI objects are domain-related, conceptual objects that are offered to users for manipulation. When it comes to implementing such objects in practice, a single OOUI object is usually realized by several Java classes.

First we need to define carefully the OOUI objects needed in the GUI. We have only two OOUI objects in this simple example: Books and Libraries. Libraries are collections of Books, with some additional attributes. This situation is represented in the UML class diagram in Figure 15.9.

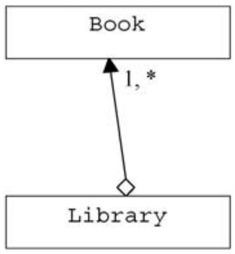


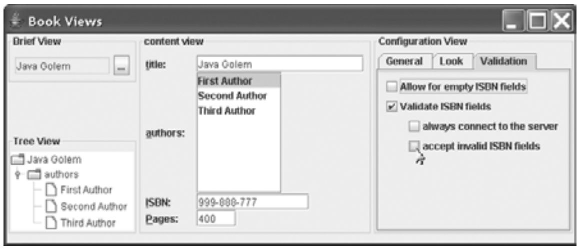
Figure 15.9 OOUI object relationships

We indicate OOUI objects with capitalized names, such as 'Book' or 'Library.' Don't confuse these with the Java classes that are needed to implement them. In the end, the user's experience should be as close as possible to manipulating a unique OOUI object, no matter if trees or tables (Java classes) are used to represent its state and the intended interaction. Thus the class diagram in Figure 15.9 refers to 'abstract' OOUI objects, not to Java classes.

Book objects and their implementation

For brevity we concentrate only on the implementation of the Book object, leaving the classes that implement the Library object for interested readers.

We want the Book object to provide different views, depending on the part of the GUI where books appear. The three normal views (brief, content, and configuration) and a tree view are shown simultaneously in Figure 15.10.



Brief View: Java Golem

Tree View:
- Java Golem
  - authors
    - First Author
    - Second Author
    - Third Author

content view
- title: Java Golem
- authors: First Author, Second Author, Third Author
- ISBN: 999-888-777
- Pages: 400

Configuration View (Validation)
- [ ] Allow for empty ISBN fields
- [✓] Validate ISBN fields
  - [ ] always connect to the server
  - [ ] accept invalid ISBN fields

Figure 15.10 *Possible views of a Book object*

The Book object offers more views than OOUI objects normally do, but they will help us to better explore the proposed approach.

The interaction mechanism is as follows: the requester, usually a graphic container (a dialog or a panel) requests a view of itself from a viewable object. In practice only complex views are coded into specialized classes: default and standard views work well for simpler cases, such as brief views, or for content views of collections.

Suppose a brief view is requested of a Book instance. The Book UI is designed to provide default brief views, so it is enough to provide the Book's model for this kind of view to a specialized factory that creates a standard brief view ready to be used in the GUI container. This mechanism caters for views asking other for views, and so on – the Composite pattern.

The sequence diagram in Figure 15.11 shows this scenario: the `ViewableBook` uses a model – in the MVC meaning of 'model' – of its internal data for building its views. Models are essential because they allow for the creation of 'live' views that change as the model behind them changes, modifying the current view's appearance.

Taming complexity

When developing large and sophisticated GUIs, for example those in which the same classes provide different types of views, it is essential to tame the complexity of the resulting code.

Two problems always conflict:

- *Visibility.* Viewables and views need to be tightly coupled with their respective domain objects. This conflicts with the principle of separation of presentation from domain-related code. The three-layer architecture proposed in Chapter 7 dictates a separation of layers into presentation, application, and service layers, which helps to create high-quality code.

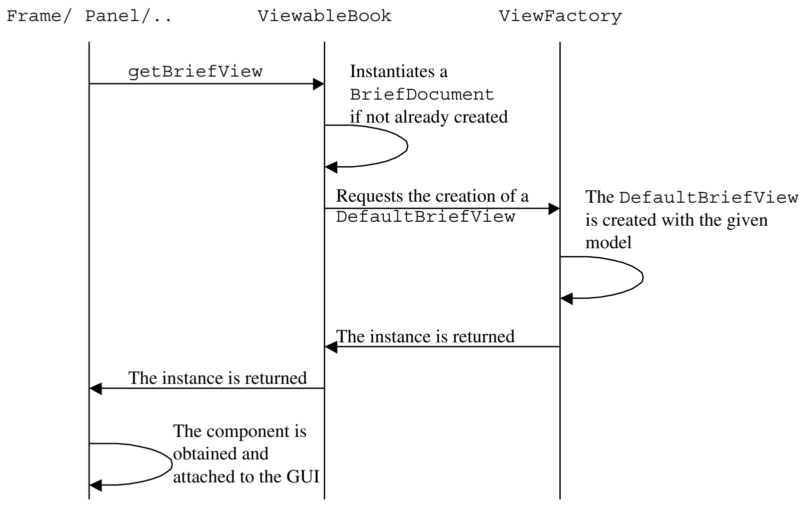

Frame/ Panel/..
ViewableBook
ViewFactory
getBriefView
Instantiates a BriefDocument if not already created
Requests the creation of a DefaultBriefView
The DefaultBriefView is created with the given model
The instance is returned
The instance is returned
The component is obtained and attached to the GUI

Figure 15.11 A Book object's views implementation

- *Code size*. Inevitably, sophisticated GUIs tend to need more code. One problem is that high-quality GUIs cannot rely on automatic, general-purpose widget layout, and have to be positioned manually by the GUI designer on a per-case basis. The OOUI mechanism described can lead to large class sizes if not properly handled.

There are several approaches to these problems. The right mix of visibility and class management is essential for long-term code maintenance. In the example here we use class inheritance to separate presentation code from domain code and to keep classes to a reasonable size. Inner classes are then used to keep view code logically organized but still visible within the class. This simplifies manipulation of domain-related fields by the presentation code.

Using inheritance is a less flexible solution than using object composition[6]. The use of inheritance depends on the particular business domain, because if presentation classes (here `ViewableBook` or `ConfigurableBook`) need to extend some other class, we cannot subclass the domain object (`Book`). Using inner classes solves many visibility problems, but tends to produce large files that could be difficult to maintain. With this approach, we need a new package to protect the domain class fields from outside, because we want only subclasses to be able to access them. This could conflict with the analysis phase, in which class packages should be designed, and on the fact that low-level implementation details should not affect the domain analysis.

6. See Chapter 6.

The classes in the example are organized as follows:

- One application class, `Book`, containing only domain-related code.
- Two presentation classes, one for managing all non-configuration views, and one to produce configuration views as well. Throughout the GUI we will always use the `ConfigurableView` class.

With this code organization long-term code maintenance is greatly improved. The class hierarchy implementing the Book object is shown in Figure 15.12.

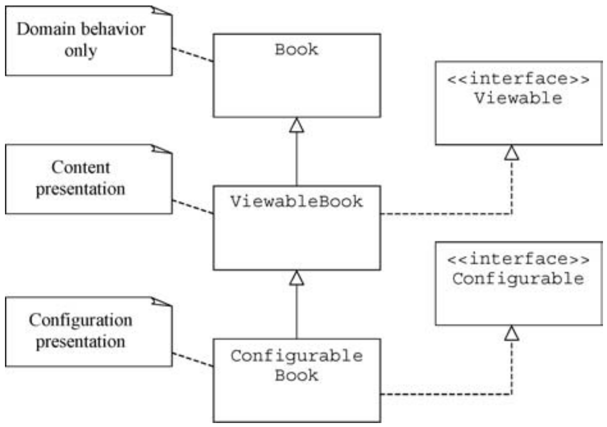


Figure 15.12 The implementation of the Book object

The following Java classes implement the Book object:

- `ViewableBook` contains the views the object releases to other classes, and the model used for the brief view. It has the following inner classes:
  - `ContentView` is the class that implements the content view for the object.
  - `TreeView` is the tree view.
  - `BriefDocument` relies on an adapter of the `PlainDocument` class, which in turn extends the `AbstractDocument` class, both provided by the Swing library.
- `ConfigurableBook` contains configuration-related code and has only one inner class:
  - `ConfigurationView` implements the book's configuration view. Given the many options available for books, we used a tabbed pane.

Unexpected views

We have added a twist to the example by providing an ad-hoc view, mainly to show the practical limits of the view/viewable framework.

Views, in the meaning of graphic representations of an OOUI object, can come in many flavors, and there is no standard, once-and-for-all way to implement them. As proof of this, the Book object releases views of itself as a tree. Using trees is quite useful, because they realize the Composite pattern (Gamma et al. 1994) nicely. For example, an object that is composed of instances of itself can be represented easily, although this is not done in this example.

From the developer's viewpoint, the general problem is that views can be implemented with classes other than `Components`, which makes our `View` interface unusable. There are a number of technical solutions to this problem, such as a `TreeView` interface that specializes `View`. In this example we choose a more straightforward solution, adding a `getTreeView` method to our `ViewableBook` class. In this way, tree views return Swing `TreeNode` instances, ready to be combined with other tree nodes to compose any hierarchical view we need.

Reducing development costs

The Book object, even if it implements more views than objects usually do, exhibits some common factors for reusability. While content and configuration views are often specialized GUIs tailored for their domain objects, brief views can be implemented using reusable components, as our example shows.

Taking advantage of the `View/Viewable` mechanism, a number of general-purpose classes can be used. For example, the `DefaultViewableList` class is used by the Book's content view to provide the representation of the Book's author list. The same component is used in other parts of the example GUI as well, such as in the `MainFrame` container. This programming style also has consequences for the GUI's usability.

The code

We are now ready to look at the final code. Listing 15.2 shows the `Book` class.

Listing 15.2. The `Book` class.

```
package com.marinilli.b1.c15.book;
import java.net.URL;
import java.io.Serializable;

/**
 * The Presentation Layer
 *
 * @author Mauro Marinilli
 * @version 1.0
 */
public abstract class Book implements Serializable {
  String title;
  String[] authors;
  String isbn;
  int pages;
  URL homePage;

  final static int BOOK_TITLE = 0;
  final static int FIRST_AUTHOR = 1;
  final static int ISBN = 2;


  public Book() {
  }

  public Book(String t, String[] a, String i, int p, URL u) {
    title = t;
    authors = a;
    isbn = i;
    pages = p;
    homePage = u;
  }

  public String toString() {
    return title;
  }




}
```

The `Book` class belongs the Application layer[7]. It embodies the domain-related data and behavior associated with the Book object: the separation of business code from its presentation is an essential benefit of such an architecture.

The business class is very simple: there are no methods, just a collection of fields. These are the book title, an array that represents the authors, and a few other data items. As we are primarily interested in the GUI-related code and its organization, we kept the domain classes as simple as possible. The `Library` class, whose implementation is not discussed here, is slightly more complex, in that it provides a business-related method for the inventory.

The ViewableBook class

The `ViewableBook` class is the main graphical class that represents Books in the GUI. Its source code can be found in the code bundle for this chapter. The class essentially implements a book suitable for display and manipulation on screen via

7. See Chapter 7, *A three-layer architecture* on page 298.

the class framework. The class contains some configuration fields, such as `currentBriefViewType`, used to decide what book data to use for the brief views, and other GUI-related details.

The `ViewableBook` class contains the views it releases to other classes and the model used for the brief view. Recapping, it has the following inner classes:

- `ContentView` is the class that implements the content view for the Book object.
- `TreeView` is added for demonstration reasons and it is not fully implemented.
- `BriefDocument` relies on an adapter of the `PlainDocument` class, which in turn implements the model for `DefaultBriefView` instances. Note that instances of this class are created only when brief views are requested. When this is not the case, the related `briefDocument` field is kept null to minimize space.

`ViewableBook` adopts a solution that is repeatedly applied in the code described in this chapter: the main class extends a less-specialized version of the same OOUI object, and all the auxiliary code required (view classes, MVC models, and the like) is provided as inner classes. This guarantees high OOP cohesion among the Java classes involved in the implementation of the Book OOUI object. As a result, the visibility of the various instances involved is highly simplified. Whenever a coupling can be relaxed, for example for views in the MVC meaning of the term, there is no need to accommodate these code objects as inner classes, as is done for `BriefDocument` within `ViewableBook`, for example.

The Book object implemented by these three Java classes is serializable. This is required to allow Book instances to be made persistent. The book template is recorded persistently so that it can be used for the initialization of books created from scratch[8].

The ConfigurableBook class

The configuration view is separated from the other views. Although there is no precise conceptual reason for this, it is handy for practical code maintenance. The `ConfigurableBook` class is the extension of the `ViewableBook` class, providing configuration views in addition to all the other views the superclass provides.

The `ConfigurableBook` class provides two implementations of `Configurable` methods. The `ConfigurationView` inner class implements the preference

8. See Chapter 7 for details of localization mechanisms and naming conventions.

setting facility for Book objects in the GUI. Physically, it is a `JTabbedPane` made up of three panels, built and added in the constructor, that correspond to the three categories into which Book object preferences were divided by the GUI designer:

- The **General** panel is created by the `createGeneralPanel()` method. It is initialized with the persistent values from the last time the application was run – see Figure 15.3 on page 546.
- The **Look** panel is created by the `createLookPanel()` method. Note that not all the widgets are actually used: the `treeLargeIconsCheckBox` is shown to the user but not bound to an actual use in the GUI, for simplicity.
- The **Validation** panel is created by the `createValidationPanel()` method – see the right-hand side of Figure 15.10 on page 552.

The `ConfigurationView` method `doCommit()` is different than usual `doCommit()` methods, because it doesn't only affect one class instance, but also persists the values through the persistence facility offered by the Service layer: when a user commits the book configuration view, its settings are intended to be used by all currently open books as well as those yet to be created. Configuration views are often a type of Singleton view, that is, only one instance is shown to the user at any one time.

Apart from the methods that take care of the GUI details, the `clone` method is employed here to support advanced object creation, as discussed in Chapter 7 in relation to the Service layer.

Libraries

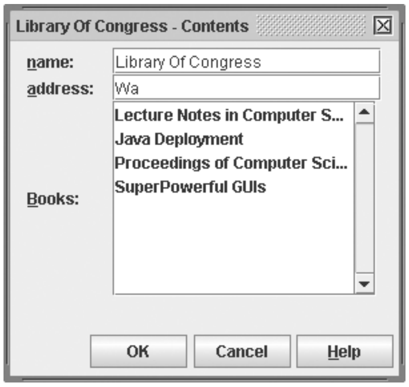
We implement libraries following the same approach used for books, with some small differences, like using accessory methods (get and set) to interface with the application layer's class, `Library`.

The implementation of the Library domain object closely resembles that of Book, so we will illustrate only the essential details:

- `ConfigurableLibrary` has only one inner class:
  - `ConfigurationView`, the configuration view of Library objects.
- `ViewableLibrary` has only one inner class:
  - `ContentView`, the content view of Library objects.

Figure 15.13 shows the Library content view.

For brevity the Java code for the classes that implement Library objects is not listed here. Nevertheless, we hope that, apart from conceptual ideas, the example code can show you many useful, practical tactics.



Library Of Congress
Wa
Lecture Notes in Computer S...
Java Deployment
Proceedings of Computer Sci...
SuperPowerful GUIs

Figure 15.13 The Library content view

Some GUI design considerations

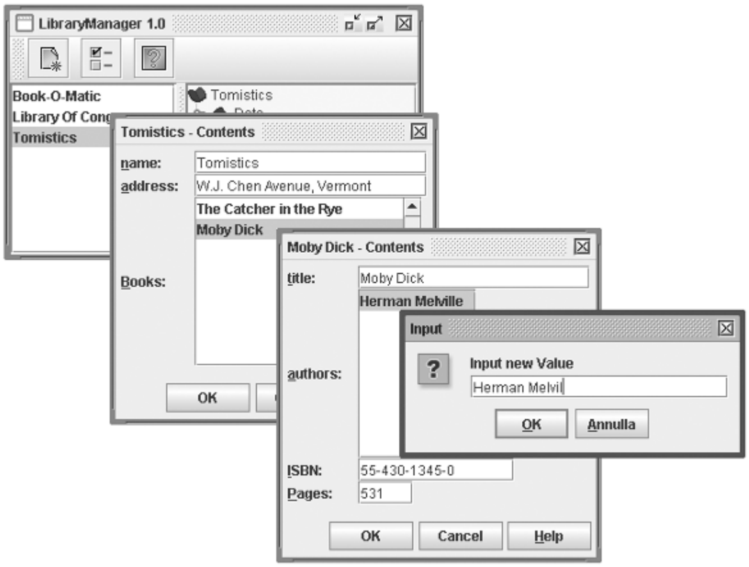
One question remains unanswered: what kind of GUI do we get with the design strategies described here?

Let's look at the Library manager application. Suppose you want to modify the author of one book in a given library. You right-click on the libraries in the main frame, selecting **Properties**, and do the same on the book list that pops up. In turn, you then select the author from the chosen book, and select the **Properties** command again, finally obtaining what you need. Figure 15.14 shows this path.

One of the reasons an example application that uses container-contained OOUI objects is chosen is to show some of the peculiarities of this kind of GUI. Using heavily OOUI-modeled relationships such as containment hierarchies in domain objects tends to produce highly 'vertical' GUI interactions, such as those shown in Figure 15.4 – many pop-up windows stacked one above the other.

This is not a bad design per se. It keeps the whole GUI highly focused by leveraging the context the user creates during an interaction – in the figure, the context of the **Herman Melville** string object is the **Moby Dick** Book object, that is in turn inherent in the **Tomistics** Library object. But precautions are needed, such as keeping the stacking level to a reasonable size, perhaps no more than three or four depending on the type of application windows and intended users, or maintaining a tight grip on the GUI by allowing only modal dialogs and hooking them to the proper container to avoid floating dialogs when users switch to another process window. Solutions to such problems are available in the literature and were also mentioned in Chapter 4.

Applying the methods described here doesn't guarantee a perfect GUI. Usability and guidelines compliance should always be kept in mind, no matter of how sophisticated an implementation may be.



name: Tomistics
address: W.J. Chen Avenue, Vermont
Books: The Catcher in the Rye; Moby Dick

title: Moby Dick
authors: Herman Melville
ISBN: 55-430-1345-0
Pages: 531

Input new Value: Herman Melvil

Figure 15.14 Using the LibraryManager GUI

Control issues

The example implementation follows both the centralized approach using the Mediator design pattern (Gamma et al. 1994), and scattered control, both discussed in Chapter 6.

Centralized control

The director class employed in the example application implementation supervises command management and other classes that need centralized control and have a web of visibility references. In the example we kept the director simple and straightforward: a more complex version of this pattern featured in Chapter 14.

The `BookDirector` class provides the following services:

- Initialization of data.
- Execution and management of application-wide commands.
- Conceptual centralization of all GUI-related control code.

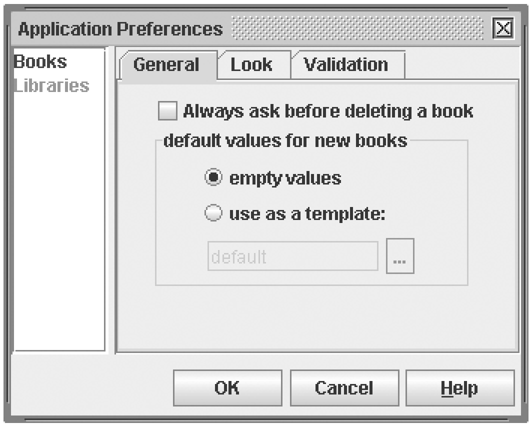
This book contains several examples of directors at work, from the simple one in the QuickText application (Chapter 7) to that presented in Chapter 16, as well as the flexible arrangement proposed in the example application in Chapter 14, with one global director coordinating two specialized ones.

Scattered control

This is related to a common situation in many GUIs. When an event takes place that is bound to some widget or domain data, something else should happen as a consequence. For example, whenever the user fills in a text field, a group of components are enabled, and vice-versa[9].

More complex interactions are possible in GUIs, making a standard and systematic approach highly desirable. This is where the Mediator design pattern comes into play: there is often no need for a fully-fledged director class. In the book's configuration dialog for the example application, some simple reactive behavior is effectively handled locally to the class, without the need for an external director.

From the book configuration dialog, users can specify how new books are created. They can decide whether empty values or template data should be used when a new book is created from scratch. In the latter case, they can inspect and modify the book template data, as shown in Figure 15.15.



Always ask before deleting a book

default values for new books

● empty values

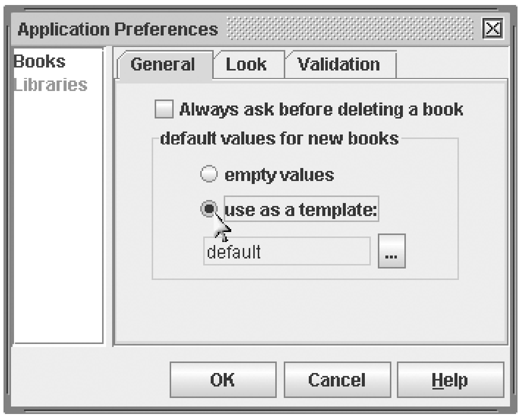
○ use as a template:

default ...

Figure 15.15 Book configuration panel, general panel

Whenever the user selects the **use as a template** option, the area corresponding to the book template is enabled, as in Figure 15.16. This area corresponds to a `DefaultBriefView` component.

9. It is always best to keep such widgets – even if disabled – always on screen, rather than making them appear magically, thus modifying content dynamically at runtime, because this confuses the user.



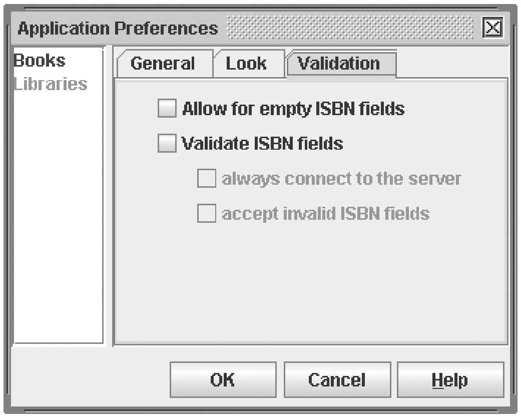
Always ask before deleting a book [ ]

default values for new books

- ○ empty values
- ● use as a template: default

Figure 15.16 Book configuration pane

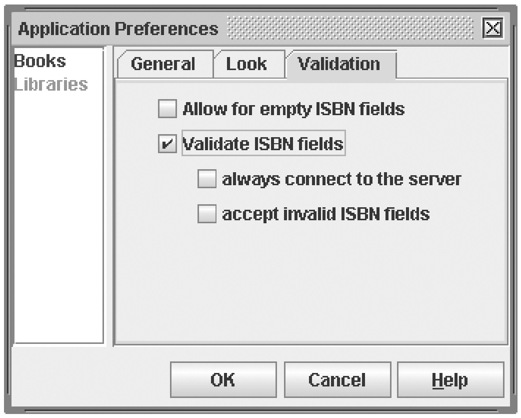
Another case in the same OOUI object's configuration panels is shown in the next two figures. The first two check boxes in Figure 15.17 are unrelated, but whenever the second is selected, the other two 'nested' checkboxes are enabled.



- [ ] Allow for empty ISBN fields
- [ ] Validate ISBN fields
  - [ ] always connect to the server
  - [ ] accept invalid ISBN fields

Figure 15.17 Book configuration panel –book validation

Figure 15.18 shows the situation in which the **Validate ISBN fields** option is selected.



Allow for empty ISBN fields [ ]
Validate ISBN fields [✓]
always connect to the server [ ]
accept invalid ISBN fields [ ]

Figure 15.18 Book configuration panel – enabling fields

In these two cases, interaction is so simple and circumscribed that setting up two, or even one, separate directors would be a needless complication. In these specific cases, using an `ItemListener` to listen for checkbox (Figure 15.18) and radio button (Figure 15.16) events suffices These are examples of the scattered control design strategy discussed in Chapter 6.

15.8 An alternative implementation using Naked Objects

This chapter concludes with an alternative implementation of the Library application, built using a specialized framework, Naked Objects.

Naked Objects is a framework for building applications by defining the business objects and their relationships only – that is, focusing on the domain model alone – and letting the framework take care of GUI details. This is the main advantage provided by this approach, but also its biggest weakness: developers have little control over GUI design details. The framework (unsurprisingly) adopts a direct manipulation, OOUI-inspired GUI design to show business objects to users 'naked' of any presentation-specific code.

A screenshot of a simplified version of the Library application implemented with this framework is shown in Figure 15.19.

This version of the application was built with only three extremely simple classes (`Book`, `Author`, and `Library`) and a framework-dependent launching class, `LibraryExploration`. To give an idea of the simplicity of this version – although providing a smaller feature set compared with the previous version – the following listing shows the implementation of the `Book` class within the Naked Objects framework.

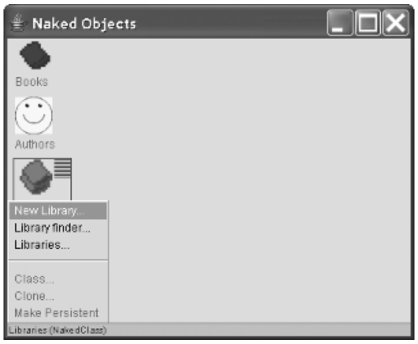


Figure 15.19 The Library GUI – Naked Objects version

Listing 15.3 The `Book` class

```
package com.marinilli.b1.c15.naked;
...

public class Book extends AbstractNakedObject {

  private final TextString isbn = new TextString();
  private final TextString title = new TextString();
  private final WholeNumber pages = new WholeNumber();
  private final InternalCollection authors =
              new InternalCollection(Author.class, this);

  public TextString getTitle() {
     return title;
  }
  public TextString getIsbn() {
     return isbn;
  }

  public Title title() {
    return title.title().append(",", pages);
  }

  public InternalCollection getAuthors() {
    resolve(authors);
    return authors;
  }
}
```

Interested readers can look at the source code of the remaining classes provided with this chapter. For more details on Naked Objects, see `http://www.nakedobjects.org/`.

Figure 15.20 shows the rather unusual GUI generated by the framework for business objects. It is possible to create new instances, assign them to their respective containers, and make them persistent, with many available alternatives. The icons on the left-hand side represent the classes, while their instances that are open for modification or inspection are represented by the various internal frames. To add an author to a book, for example, the user first has to create a new author instance by right-clicking on the **Authors** icon on the left-hand side of the window, selecting the **create new** command, filling in its data, then dragging it to the **Authors** area within a book instance.

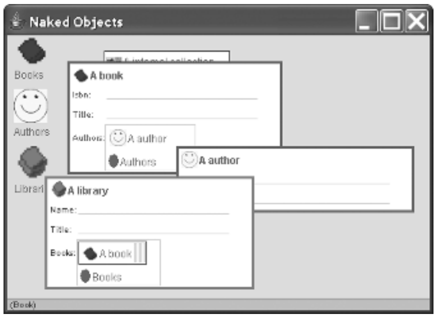


Figure 15.20 Managing data the Naked Objects way

In conclusion, it is interesting to note the amount of code employed in the first version of the Library application that is devoted purely to GUI details, compared with the second version, in which all GUI details are handled automatically by the framework. Unfortunately, GUI development is all about GUI details, and while frameworks such as Naked Objects are appealing to developers, they appear less effective to customers and GUI designers, no matter how well the GUI is automatically generated.

15.9 Summary

In this chapter we discussed the details of developing an application following an alternative approach, OOUI objects. We examined common situations that developers face when implementing non-trivial Java GUIs using the OOUI approach. We introduced several classes, some of them reusable in many different situations, and a complete example that leverages them to show how complex GUIs can be built following the OOUI approach. We also contrasted our application with a fully-fledged, OOUI-inspired framework, Naked Objects.

16 An Example Ad-Hoc Component

This chapter describes an example of the design and implementation of an *ad-hoc* component with the Java programming language, focusing on the J2SE and the Swing library only.

Chapter 2 stated that a GUI can be thought of as being composed of visual components and their auxiliary elements. From a developer's viewpoint, such components can be thought of as belonging to three main groups, depending on their development cost:

- *Standard components*, such as a 'plain' `JTree` instance.
- *Specialized components*, such as a complex `JTree` subclass that re-implements many of the `JTree` auxiliary classes.
- *Ad-hoc components*. These are visual components completely different than the standard ones provided by Swing or other GUI libraries. Ad-hoc components are much more expensive to develop, because their development effort comprises GUI design, testing, and coding.

If there were no reusable, standard libraries, we would develop components from scratch over and over, at enormous cost and without any coherence between different implementations of the same component. Luckily, the last twenty years of software engineering has provided today's developers with a wide range of tools for reusability and easy customization of their programs. See for example (McConnell 1993).

Unfortunately standard libraries don't cover all the possible components so that, although rarely, it is still possible to find yourself engaged in ad-hoc component building.

This chapter is structured as follows:

16.1, Introduction discusses the issues related to the GUI design and implementation of ad-hoc components.

16.2, The Drawing Sandbox application introduces an example ad-hoc component, showing an example of its use.

16.3, The Sandbox architecture discusses the overall architecture of the proposed component.

16.4, The Sandbox component discusses the top-down refinement of the design of the example application.

16.5, User interaction describes how user interactions are managed by the proposed implementation of the Sandbox ad-hoc component.

16.6, Control discusses how control is implemented in the proposed ad-hoc component.

16.7, The whole picture puts the various pieces together, showing a complete picture of the implementation design.

16.8, Stressing the software design discusses the proposed design critically, highlighting its strong points and its drawbacks.

16.9, Introducing JHotdraw contrasts the implementation provided so far with another implementation using the specialized library JHotdraw. After a brief introduction of the library, the alternative implementation is discussed and compared with the previous Sandbox implementation.

The chapter concludes with a summary.

16.1 Introduction

The design and development of high-quality ad-hoc components involves typical dangers that are accentuated for novice developers:

- The unnecessary effort of reinventing the wheel. When developers launch themselves into the process of creating ambitious ad-hoc components together with their support classes – thus creating a de-facto specialized small class framework – they tend to ignore the vast API that Java offers, re-implementing existing functionalities in their code.
- Given the difficulty of the implementation process, developers tend to favor coding aspects over the GUI design of the ad-hoc component being created. This is a classic problem, but it is accentuated for ad-hoc component development – the designer says: “We need drag and drop here,” and the programmer replies “No way – it’s too costly.”
- Ad-hoc components tend to be domain-specific, so developers often overlook their development in favor of other parts of the system that are judged more ‘important’ and reusable in future applications.
- All the typical risks associated with the design and development of new and complex software artifacts, such as bad cost estimates.
- The risk of a bad design, originating not from technical deficiencies, but rather the fruit of GUI design inexperience and over-ambition.

> While standard and specialized components have been properly designed and are thus harder to misuse, ad-hoc components are like a blank sheet in the hands of eager and sometimes inexperienced designers and developers.

Given the potentially large costs associated with ad-hoc component development, it's best to discourage their use in GUI design and favor wiser, less bold GUI design choices such as specialized components, as discussed in Chapter 14. Nevertheless, when one of the requirements is high GUI quality, or when the business domain dictates it, designers have to resort to ad-hoc components.

Essentially, ad-hoc components leave too much freedom to the designer. This can be a great thing, or a disastrous one, depending on the designers' experience and the context. Venturing into such a complex and time-consuming task is best avoided if it's not absolutely necessary. While it can relatively cheap in some cases, such as in the example presented here, in general it is a hazardous route rife with unforeseen problems and delays that slow down the whole development process. All the discussions in the first part of the book should be taken to heart when evaluating ad-hoc component development. When the ad-hoc component isn't a supplementary part of the GUI, but is its cornerstone, the risk is not only to at least double the development and design effort, but even to produce an unsuccessful GUI.

In the next section we look at an example of a relatively complex ad-hoc component implementation in some detail. The scenario is a familiar, interesting and intuitive domain, enhanced by some ambitious requirements that demand sound software design. Details have been reduced to a minimum to focus only on the essential aspects of the problem and produce a concrete, working example of a complex GUI cut down to its simplest implementation. The focus of this chapter is on object-oriented software design. This is often the weakest point of an ad-hoc component implementation – it is a less relatively visible aspect, but in the long term the most delicate.

16.2 The Drawing Sandbox application

Most computer users are familiar with using a graphic drawing editor into which you can drop graphic items, drag them, and modify them as you want. Commercial applications such as Adobe Illustrator, Corel Draw, or the drawing palette in Microsoft Word, provide such features.

We focus here on the mechanism of a graphic container that handles such items and provides basic operations for manipulating them – a kind of 'sandbox,' similar to Windows or MacOS folders, into which users can drop items and arrange them as desired. We will adopt something close to the direct manipulation approach in our GUI design. We are not however so much interested in GUI design issues as in the software design needed to implement them. The objective is to build just such a visual component together with all its support classes. This will be an ad-hoc component because such a visual container is not provided by the standard Java API.

The application

We'll begin with the user's experience of the final product, then explore the behind-the-scenes software design and discuss some of its aspects. Figure 16.1 shows the main application frame. For simplicity we have only one toolbar, which groups all the commands, and no menu bar.

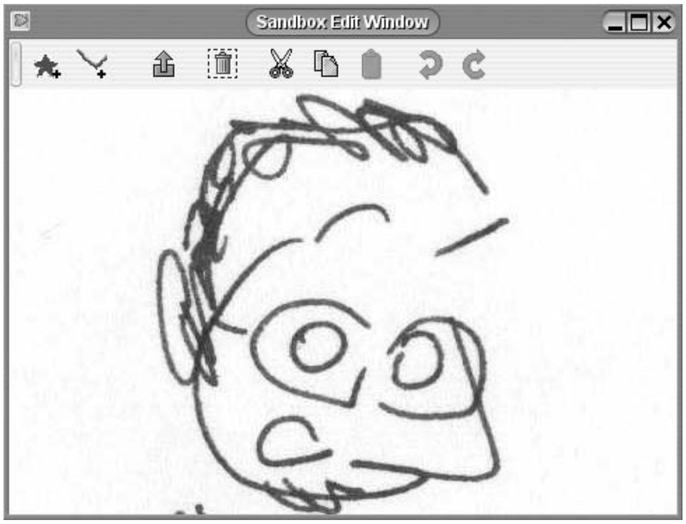


Figure 16.1 The main application frame (Liquid)

Following the order of buttons in the toolbar in Figure 16.1, you can:

- Add graphical objects[1] to the Sandbox component.
- Cut, copy and paste an object as needed.
- Undo or redo previous operations.
- Move a selected graphical object to the front, so that it lies above all others.

1. Referred to as 'symbols' in the Java code.

All other actions are object-dependent. Two graphical objects, the bitmap and the multiple-segment poly-line, have been implemented, to show how different interactions are accommodated within the same class framework. For example, the poly-line can be manipulated by mouse dragging and button combinations.

The interaction is rudimentary: you choose an object from the palette, then create it in the sandbox by clicking on it. The mouse cursor's shape changes to signal the drawing mode. Toggle buttons or more refined mechanisms like drag and drop to inform the user that the application is in 'adding' mode aren't used. In this simple implementation there is no way to exit object adding mode if you change your mind – the only way is to add the object to the sandbox and then remove it. Such niceties have been omitted to keep the code to a reasonable size for an example. In Figure 16.2 shows a bitmap object being manipulated.

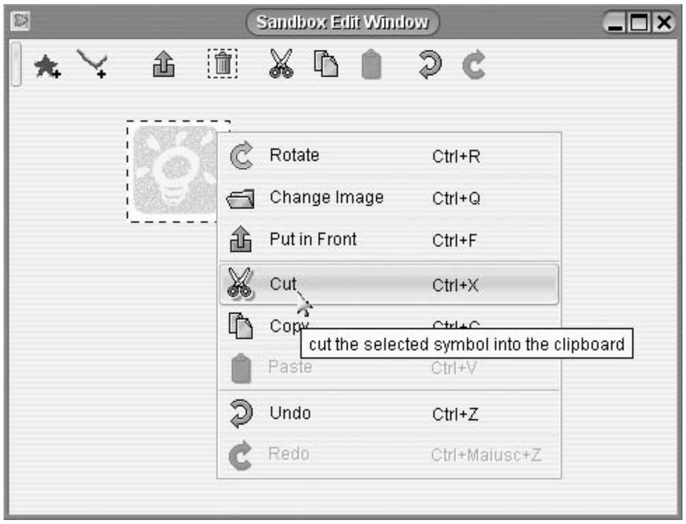


Figure 16.2 Manipulating an object via its contextual menu (Liquid)

Each object independently provides its own commands, as shown in Figure 16.3 for the bitmap.

When interacting with the GUI, actions are automatically enabled or disabled depending on context. For example, if no object is currently selected, the actions appear as in Figure 16.3: note that the pop-up contextual menu, obtained by right-clicking on the object, is coherent with the toolbar buttons.

Try launching the executable `sandbox.jar` file, and have fun with the actual program.

An example interaction

Suppose the user has already performed some manipulations on the objects shown in Figure 16.2. First undo the rotation on the bitmap using the contextual menu, as in Figure 16.3.

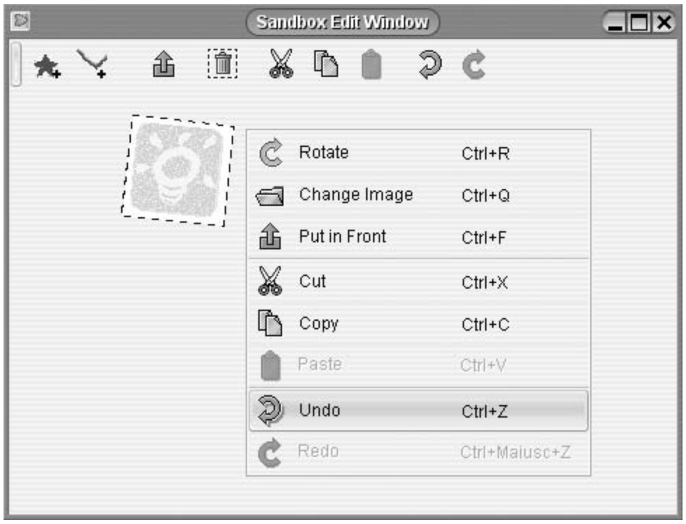


Figure 16.3 Undoing a rotation (Liquid)

Note that in the interaction example the clipboard is not used – that is, nothing was cut or copied, so the paste command remains disabled.

New lines are added just like objects of the type 'image,' by clicking on the relevant tool button, then clicking in the sandbox area where the new line is to be placed. Double-clicking on a line allows its control points to be edited, as shown in Figure 16.4. Control points are moved by dragging them with the mouse. Line editing mode is disabled by selecting another object, or by clicking somewhere else on the sandbox area. To add new control points to a line when in edit mode, the mouse right button and the control key are used together.

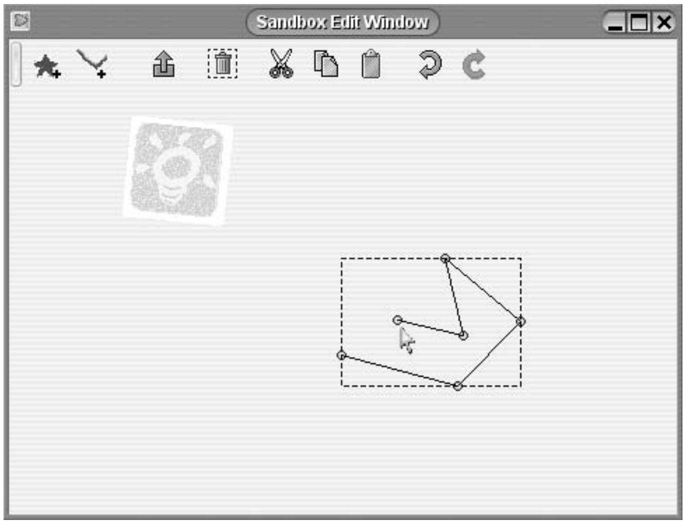


Figure 16.4 Modifying the control points of a poly-line (Liquid)

16.3 The Sandbox architecture

Now we can look at the implementation of the Sandbox component.

Chapter 2 introduced the OOUI approach, mentioning its usefulness both as a means for designing the GUI, as well as a way to organize the resulting implementation. The OOUI approach can be used as a fully-fledged composable unit strategy[2].

The software design discussed here is arranged around the development of the ad-hoc component using a top down, functional partition. Figure 16.5 shows an initial high-level division of the software design. A common approach in many architectures is to separate the domain definition from the application logic – that is, the functionalities made up of simpler, low-level features that expose the domain to users[3]. An e-mail inbox, for example, represents the generic domain of an archive of e-mails. Possible actions in such a domain depend on the purpose of the application. For example, an inbox component could be used in a customer-care application in which customer complaints are filed with e-mails that can

2. See Chapter 7.
3. From a GUI design viewpoint this reminds us of functional user interfaces – that is, application GUIs designed around a set of functions.

sorted by specific heuristics to extract a tentative subject, category, urgency, and so on. By implication, we are differentiating our ad-hoc component and the set of elementary functions it provides for interaction with the rest of the world from the commands a user will employ to interact with it.

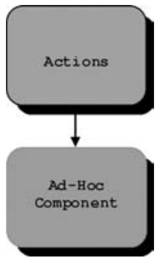


Figure 16.5 An initial high-level functional decomposition

Following this decomposition, we will build our component around a set of generic, low-level functionalities that in turn will be used by other specialized code that packs them into higher-level, useful commands for user interaction.

Figure 16.6 shows a possible refinement of the initial decomposition of Figure 16.5. The most important part is the ad-hoc component itself, which interacts with the control subsystem that is responsible for maintaining the coherency of the whole GUI. The third division accounts for the commands issued by the user, each part being composed from several Java classes. This high-level functional division focuses only on the more important portions of the implementation, and intentionally omits auxiliary code.

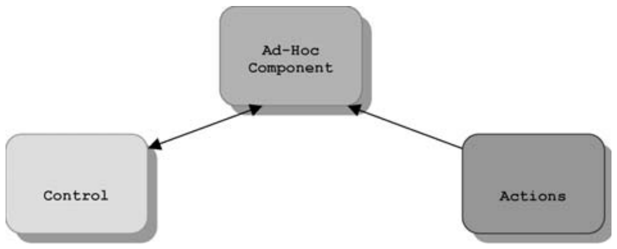


Figure 16.6 Refining the functional decomposition

The GUI will be designed by refining the simple functional organization shown in Figure 16.6 iteratively. This decomposition takes advantage of the peculiarity of the application, that of an ad-hoc component as the center of the application, but can be applied to the development of any GUI.

It would be nice to create a flexible, highly expandable framework, but the key issue here is to provide a simple, lightweight design. Features that are desirable, such as allowing for runtime palette loading – loading new graphical objects from a file and using them without having to reinstall a new version of the application – can be left for future versions.

16.4 The Sandbox component

First we concentrate on building the ad-hoc component, the core of the application, beginning software design refinement from the ad-hoc component.

Top-down refinement of functional organization

Conceptually the Sandbox component is a container of graphical objects. Both the objects and the container itself can issue commands by means of contextual menus. This functionality can be modeled using the `Commandable` interface from Chapter 15.

An initial refinement of the software design for the ad-hoc component in Figure 16.6 is shown in Figure 16.7.

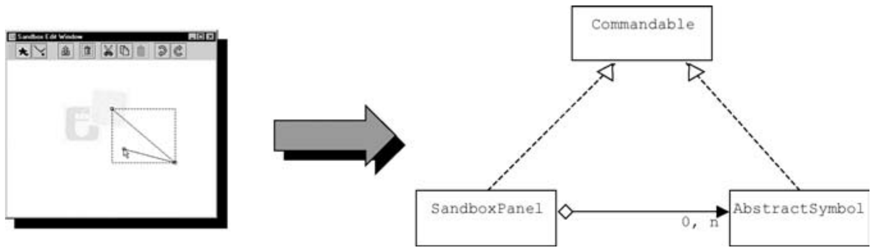


Figure 16.7 From the component to an initial high-level class diagram

So far there is the visual container, which we'll implement as a Java class called `SandboxPanel`, and the objects it contains that are manipulated by the user. Different objects will extend a base class, `AbstractSymbol`.

Organizing object communication

How should we organize the communication between the container and its objects?

One of our requirements is to guarantee flexibility to the ad-hoc component in terms of new object classes that can be used. A clean way to obtain this is to delegate responsibility to the objects themselves. The container merely holds objects and performs some common operations on them. Objects are responsible for drawing themselves on screen, for interacting with the user, and so on. This approach offers a great deal of flexibility, allowing for the use of different object types without impacting the container.

The container needs access to objects to draw them on the screen. Objects in turn now have many responsibilities delegated to them, rather than keeping them all centralized. Following this strategy, mouse input events, for example, need to be re-issued to the relevant objects, so that the objects themselves can consume the mouse event as required, transparently to the container and to other classes. Objects are black boxes as far as the other classes are concerned – the essence of object-oriented programming.

Suppose that an object is modified and the changes are committed. The container needs to be notified so that it can refresh the screen to show the changes. Decoupling responsibilities at design time is always a great idea, but generally it needs extra care when it comes to runtime object communication. One possible solution is to adopt the Observable pattern (Gamma et al. 1994). This makes the design more modular – components interested in object behavior just register themselves as observers – and allows for future expansion, with more sophisticated events being distributed by more complex objects, while keeping the overall design simple. When we need to make the director class communicate with objects, we can then just add it as another listener without making any change to the class structure. Figure 16.8 shows this arrangement.

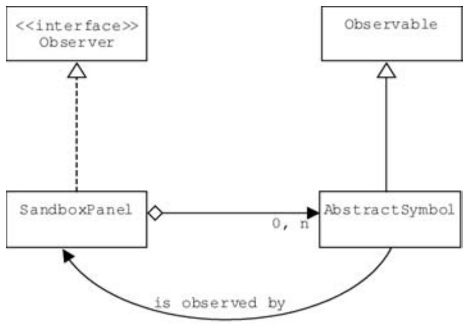


Figure 16.8 Making objects communicate with their container

Instead of creating our own event classes, we adopt the `Observer/Observable` mechanism implemented in the `java.util` package. This decision has the practical drawback that the `Observable` class must be extended, instead of an interface for objects to be observed, but this is not a problem.

Using the Observer design pattern simplifies class coupling enormously. A class interested in object behavior registers itself as an `Observer`. This application has two classes that are interested in monitoring object behavior:

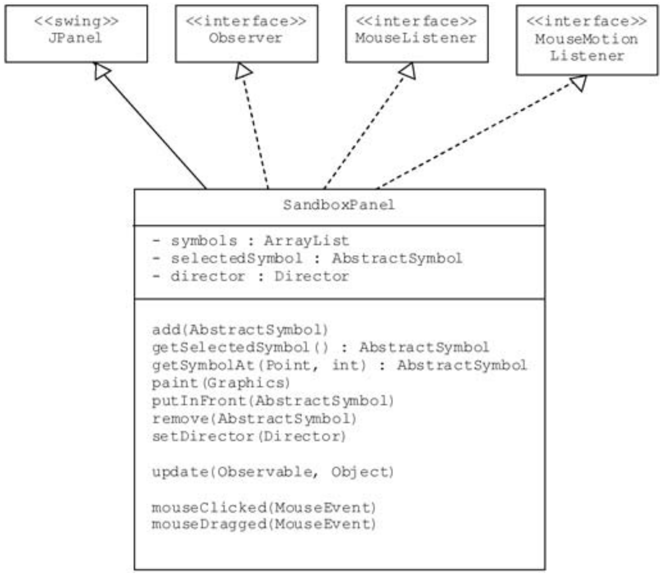
- The graphic container – when an object changes, it is time to refresh the screen.
- The director, which manages the coordination among interacting objects.

We could have mimicked the Swing API, for example for screen area invalidation, but we prefer to explore and illustrate alternative mechanisms.

We avoid an MVC architecture for our ad-hoc component to show that similar designs can solve the same problems. We don't need the flexibility of multiple views of the same model, and such a complex arrangement might make it difficult to focus on the important ideas.

The SandboxPanel class

The `SandboxPanel` class is the graphical container of `AbstractSymbol` instances. It is implemented as a specialization of the `JPanel` class, as shown in Figure 16.9.



<<swing>> JPanel
<<interface>> Observer
<<interface>> MouseListener
<<interface>> MouseMotion Listener
SandboxPanel
- symbols : ArrayList
- selectedSymbol : AbstractSymbol
- director : Director
add(AbstractSymbol)
getSelectedSymbol() : AbstractSymbol
getSymbolAt(Point, int) : AbstractSymbol
paint(Graphics)
putInFront(AbstractSymbol)
remove(AbstractSymbol)
setDirector(Director)
update(Observable, Object)
mouseClicked(MouseEvent)
mouseDragged(MouseEvent)

Figure 16.9 The SandboxPanel class

The `SandboxPanel` class is responsible for showing objects to the user and making them available for direct manipulation. Apart from this, the graphical container is essentially a passive object, manipulated by the director class. To ease the interaction between the two classes we allow the graphical container to invoke the director directly.

Objects are drawn on the screen by means of the `paint()` method, which has been customized to support our logical model. Interested users can find more information on the `paint()` method in (Fowler 2000).

The container scans the ordered list of objects, drawing each of them by invoking their `draw()` method. The order of the list defines the graphical layering of the objects, so that the method `putInFront()` simply moves the given object to the end of the ordered list, ensuring that it will be painted last and thus lie graphically 'above' all the others. The graphical container provides other methods for manipulating objects, such as `remove()` for eliminating an object.

Another duty of the container is to capture user input by means of mouse event listeners, decode it, and submit it to interested components.

The `mouseClicked()` method handles the following events:

- It adds new objects to the graphical container.
- It manages double clicks for placing objects into edit mode.
- It handles contextual menu pop-up.
- It selects objects.
- It issues events to the object itself.

The `mouseDragged` method simply reissues the event to the relevant object, so that the object can manage the event accordingly.

Following a top-down approach, we now turn to the objects themselves. At this level of detail we manage the `AbstractSymbol` class as the most general class representing any object able to be added to the container. Now we refine the part of the ad-hoc component that represents objects.

The features implemented in this example are chosen for illustrative reasons, and the application has been simplified in a variety of ways. Only single selection is available, for example, so its clipboard can contain only one object at a time. This avoids the implementation of multiple-selection interaction with the mouse, dragging a rectangle on the screen or holding the shift key while selecting more objects. This could be implemented easily using an array of objects in the clipboard. In addition, only two object types are implemented, but the design can allow for a much larger object palette. The number of different object classes is limited to focus on the more interesting aspects. There are no save or load features, although they could be added relatively easily by means of serialization extended to all the involved classes. There is no way to delete all the objects at once, or to create a new, blank sandbox.

Graphical objects

The `AbstractSymbol` class is the most generic object to be contained and manipulated by the Sandbox component. It represents the behavior of any object that can be added to the framework, and is shown in Figure 16.10.

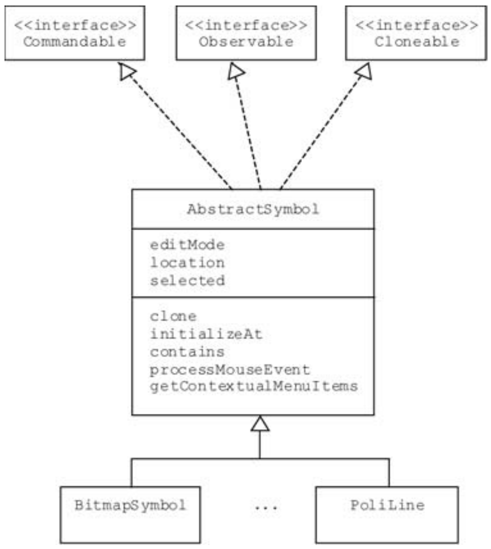


Figure 16.10 The AbstractSymbol class

Users can select an object, move it within the container, and open it for editing by double-clicking on it. These features translate into three properties of the `AbstractSymbol` class shown in Figure 16.10, respectively: `selected`, `editMode`, and `location`. There are a small number of methods that apply to any object, both for convenience:

- The `initializeAt()` method for creating an object at a given point
- The `contains()` method, which verifies whether a given point lies within that object area

and for interacting with the rest of the world:

- The `processMouseEvent()` method manages mouse input
- The `getContextualMenuItems()` method implements the `Commandable` interface for publishing user commands).

Finally, we need a `clone()` method for creating new objects.

From a practical viewpoint, the `contains()` method is used for picking an object, given its screen coordinates. This method in turn uses `getRectBounds()`, so that simple-shaped object classes don't have to implement the `contains()` method. This is helpful for objects that have a rectangular shape. For other objects types, it will be necessary to override the `contains()` method. When the user clicks on an object to select it, the container class scans the list of all objects to find the first one whose `contains()` method returns true. Note that the first one in the list will also appear to the user as the visually top-most one.

In real applications when several dozen or more different objects can be employed in the same palette, it is crucial to design the object class hierarchy carefully, both to maximize code reuse among different object classes, and to keep the design easy to maintain and expand in the future. Such considerations are out of our scope here, but (Marinilli 2000) gives more details.

The BitmapSymbol class implementation

The bitmap object represents an image in the container, as shown in Figure 16.1 on page 570. The `BitmapSymbol` class implements this simple type of object. The class is shown in Figure 16.11.

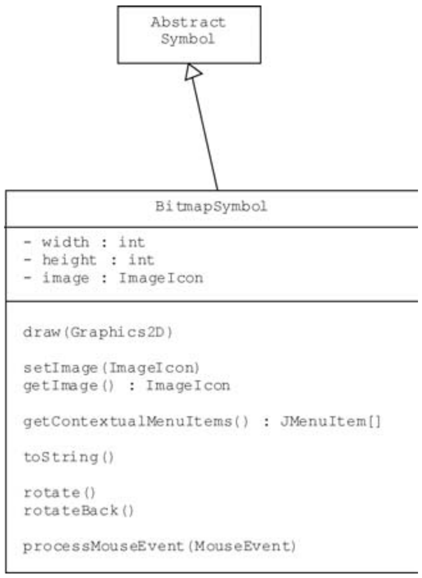


Figure 16.11 The BitmapSymbol class

The bitmap object exposes the following two commands to the user via the `getContextualMenuItems()` method of the `Commandable` interface:

- Rotate the object, performed by the `rotate()` method.
- Change the bitmap image, implemented by the `ChangeImage` action class. This is described in *The Actions class* on page 584.

A default image is used when creating a new bitmap object – see the related code in the `BitmapSymbol` class.

The `draw` method is invoked to paint an object onto the screen. The image is rotated at the current rotation angle, depending on the rotate commands previously performed on the object.

The `setImage()` accessory method implementation is worth mentioning. After updating the internal data with the new bitmap image, the method invokes the observer/observable mechanism. This will invoke all its listeners, giving them a chance to react to the event. The director checks for logical – business domain – coherence, while the graphical container repaints itself, refreshing the screen with the new object image.

Open-ended communication via events

The `BitmapSymbol` class is very simple and doesn't use all the flexibility the design allows. The sequence diagram in Figure 16.12 shows a representation of the `PoliLine` class' `mouseEvent()` implementation.

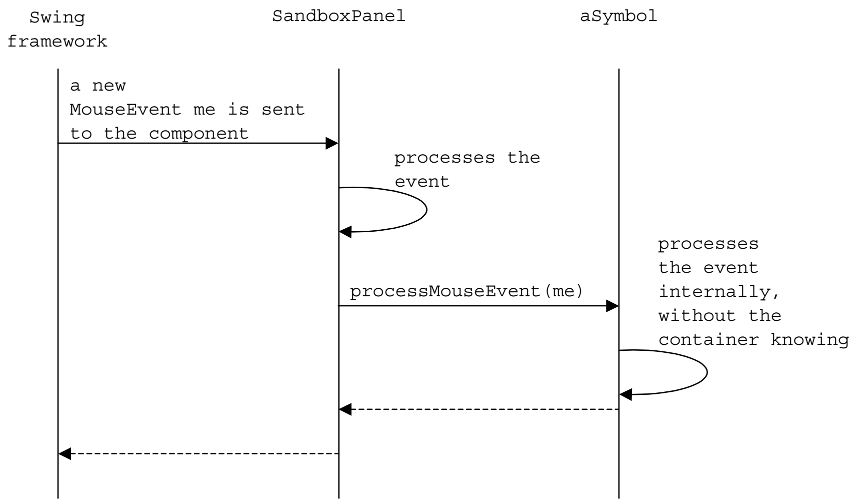

Swing framework
SandboxPanel
aSymbol
a new MouseEvent me is sent to the component
processes the event
processMouseEvent(me)
processes the event internally, without the container knowing

Figure 16.12 Objects independently process mouse events

The `mouseEvent()` method in the `PoliLine` class implements the GUI interaction style used by poly-line objects. By dragging the control points that appear when the line is being edited, the appearance of the poly-line changes. While standard dragging is implemented for all object classes, when moving them around in the container area, this class processes mouse events in a specialized way. Adding a new control point to the line is achieved through the same method as is used to process mouse events. We could also have used a custom action, such as those that are provided by the `Commandable` interface.

An abstract poly-line shape, composed of a sequence of lines, is represented as a sequence of control points – see the `generalPath` variable in the source code. Straight lines connecting pairs of control points are drawn directly in the sandbox graphical container by the `draw` method. If an object is selected, a bounding box is drawn around it.

16.5 User interaction

This section discusses how user interactions are handled by the example application.

Command composition

The creation of command menus uses a typical command-composition scheme[4]. The list of available commands for a given object is built by composing the commands available to the object container and to the object itself. We could equally have chosen a more general and powerful mechanism to implement contextual menus for objects.

The container forwards all mouse events to listening objects. Objects are themselves left with the responsibility of interpreting low-level mouse events, as well as the right-click for contextual menu pop-up. The code for commands common to all objects is kept in the root class of the object hierarchy, `AbstractSymbol` (see Figure 16.10). The approach of composing menus is chosen because it leads to a simpler class arrangement, especially as regards visibility. Note that our solution doesn't limit the freedom of `AbstractSymbols` to handle mouse events. Although programmatically feasible, it can be confusing for users to have the same input event (a mouse right-click) associated with context-dependent actions, such as for example both displaying a contextual menu and modifying control points in a curve.

Figure 16.13 shows the sequence diagram for contextual menu composition.

4. See Chapter 4 for the GUI design for composing commands, and Chapter 6 for the software design.

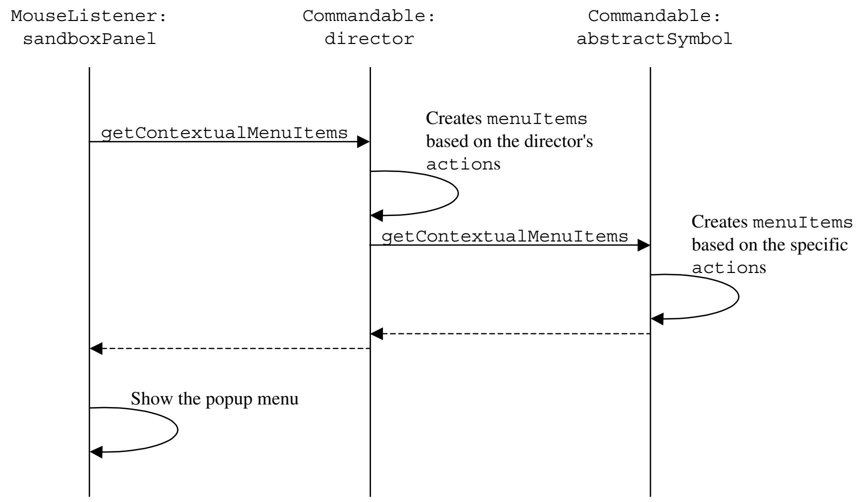


Figure 16.13 Creating the contextual menu for an AbstractSymbol instance.

The Action framework

The higher-level user commands are shown as one of the three main parts of our initial decomposition in Figure 16.6. We used ‘shallow actions’ in the Library application in Chapter 15 for handling user commands. The Sandbox application instead demonstrates the Command pattern at work, employing what we referred to as ‘deep actions’ in Chapter 6. We defined deep actions as the proper way to use Swing’s `Action` class to implement the Command design pattern fully.

Figure 16.14 restates the Command design pattern from Chapter 6.

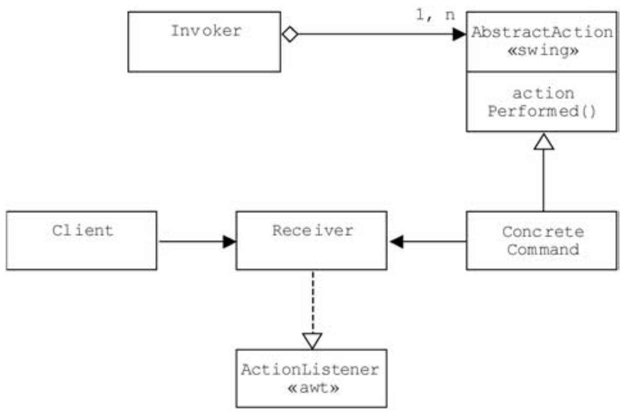


Figure 16.14 The Command design pattern

Using the Swing library, the `Invoker` is usually a `JMenuItem`, a `JButton` instance, or similar. The `ConcreteCommand` is the command instance that is set up by the `Client` class, usually the main frame or the director. The `Receiver` is the class that actually carries out the action execution. Following Java conventions, the latter class implements the `ActionListener` interface.

This approach enables several features, at the affordable price of a little additional complexity. The main benefits are:

- The whole implementation becomes closer to the domain representation than with command code centralization.
- Behavior specific to a single command is logically localized within an `Action` subclass.
- Undo and redo features stem naturally from this approach.
- The class organization that results is clearer and more systematic than that when using a centralized mechanism for commands. This is especially true for large and complex applications.
- This pattern has been adopted extensively in Java APIs, both in Swing and SWT and in other toolkits.

On the other hand, such an approach has some drawbacks, and these more evident in smaller projects. Mainly, it produces more and smaller classes, the commands themselves. This means additional complexity that has to be tamed with extra effort at design time, primarily with class interaction and management.

One solution is to use a static repository, a cluster of many, small classes obtained statically at design-time, or a dynamic one, using a runtime container such as a hash table, for example. The Sandbox application uses both strategies in the `Actions` class, which acts as a container class to rationalize action-related code maintenance. Action classes are grouped statically in the `Actions` class as inner classes, and held at runtime in a collection object managed by the `Actions` class itself. This is an implementation trick to avoid a proliferation of small classes. The `Actions` class is not a proper factory class because it doesn't create new actions, but merely provides the same action instances to interested clients.

The Actions class

The implementation of the `Actions` class is interesting for several reasons. It can be seen essentially as a static design-time container of actions. This arrangement has been adopted to ease code maintenance, as discussed in Chapter 15, where the analogous class was `ActionRepository`. Apart from grouping the code for the `Action` instances used in the application, the `Actions` class serves as a dynamic repository as well – that is, live instances are kept in memory at runtime. This is achieved using a hash map that stores the instances of the required actions.

The class name string is used to retrieve such instances when they are needed. This type of organization works nicely for this particular application, in which we need only one instance of each command. When new instances are needed, you can resort for example to the Prototype design pattern (Gamma et al. 1994), as we do for objects.

Within each `Action` subclass is the related `Edit` inner class, needed for supporting non-trivial undo/redo operations. The `Actions` class, and some of its inner command classes, is shown in Figure 16.15.

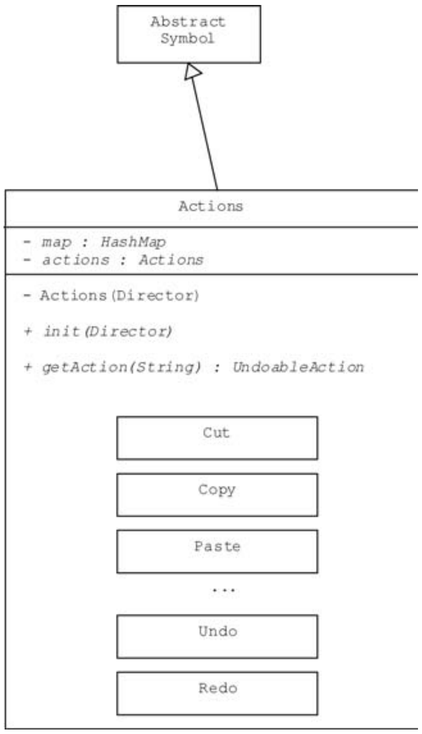


Figure 16.15 The Actions class

The constructor is kept private to avoid instantiation by other objects, which in turn is done through the `init` static method. The latter method is used to pass the `Director` instance needed for correct `Actions` initialization. The constructor initializes the dynamic repository (the static instance variable `map`) statically – that is, it is hardwired into the code. The `getAction()` method is used to query the dynamic repository.

This class is not a factory class, in that it doesn't create new object instances, but rather keeps a predefined set of them available to other classes as needed.

The `Action` subclasses used in the simple class framework are subclasses of the `UndoableAction` class, which in turn specializes `Action` for undo behavior. We adopt two slightly different designs for the Command pattern: the first one deals with classes such as `ChangeImage` or `Rotate`, while the second arrangement is a simplified version of the first and is discussed later.

The `ChangeImage` class implements the command for placing an image as a bitmap object. The implementation could have employed a content view of the bitmap object to gather all the properties of the object into one command with a single dialog, but such an arrangement has already been used in Chapter 6, so we've used a different approach here. All the details of the action, such as the icon, the tool-tip, the shortcut, and so on, are prepared in the constructor. The heart of this class is the `actionPerformed()` method, which executes the command, manipulating the rest of the application directly as needed.

This is a completely different approach than the 'shallow' actions illustrated in Chapter 6. Here the `actionPerformed()` method in the `Action` subclass takes charge of everything needed to carry out the command, in a distributed fashion – that is, the code related to commands is not centralized in one class, but is distributed throughout the related action classes. A file chooser dialog is displayed for selecting an image from the file system that is then substituted for the previous image. The old image is not lost, but is kept in the related `Edit` object in case it is needed for undoing the action. The proper `Edit` instance is created and stored in the history of executed commands, administered by the `Director` class. The `ChangeImage` class also employs an inner class, an `Edit` subclass, which is in turn an inner class of the `Action` class specialized for representing 'change image' commands.

Following the Command pattern, we define a class specialized in handling one particular command. To fully support undo and redo operations we need a further custom class that represents the 'edits' obtained by means of the command. Both the `Edit` and `Action` subclasses are tightly coupled, so the inner class implementation mechanism is ideal in this case. The `Edit0` inner class at lines 99–123 in the `ChangeImage` action stores all the information and the code for undoing the master class' command.

This elaborate design has several characteristics:

- From an OOP viewpoint it is natural – that is, it stems directly from the entities involved, so that even the resulting static class diagram is expressive, which could be useful if new developers are added to a team.
- It is easy to maintain, because the code is gathered systematically in well-defined areas.

- It is easy to expand without modifying the existing code.
- Drawbacks of static class clustering are scalability, the strain imposed on the runtime class loader when a lot of small objects are loaded, and the runtime occupancy when many such instances are kept in memory throughout an application's execution.

Our design is simplified by the double role we assigned to the `Director` class. The director works as a central access point for the manipulation of the Sandbox component. The use of fully-fledged `Action` classes poses the problem of the many objects that need to be visible to the action itself for 'doing' and 'undoing' its commands. This is why we pass the director to the `Action` class.

It is a coincidence that we have just one instance of each action class alive at any time in the application. We could equally have used just one `Add` class, for example, and slightly modified the dynamic repository by providing two methods, `getAction()` and `createNewAction()`, the latter employing the `clone()` method for creating new instances from those in the dynamic repository.

The design approach described above tends to produce many similar command classes. To reduce the number of classes, we employ a simplified version of the Command pattern. The simplest commands and their related specialized inner edit classes can be factored out into common classes. We adopt this tactic for the `AbstractUndoableEdit` subclasses, using the `Edit` class as the commonest edit instance that all other edit classes specialize. We are interested in its use for handling simple commands like cut, paste, and remove.

Take the `Cut` class. Here we have a different scheme than the `ChangeImage` action. The cut command passes the request for command execution to the director. We use a general-purpose edit class, `Edit`, that is manipulated by the `Cut` class' methods. This conservative design spares us code lines and unnecessary complexity. We can use it because of the nature of some of the commands employed – when there are many simple commands that resemble each other, this arrangement can make sense over the cleaner and more powerful one seen for the `ChangeImage` action class.

Undo-redo support

Another part of our command framework is undo/redo support. We need a class to store executed commands. In the swing `undo` package these are called *edits* – every action performed is stored in a new edit instance for future use[5].

5. We are only interested here in undo/redo implementation issues. For details about GUI design issues related to undo support, see (Cooper 2003).

When an object is rotated, for example, the application creates a new `Edit` object that stores the rotate action, together with its argument – the object that has been rotated. If the user asks for the operation to be undone, the `Edit` instance is picked up and the `undo` method of the `Rotate` action class is invoked, restoring the situation as it was before the command was issued.

This scenario is a simple one. In real-world cases more sophisticated mechanisms need to be employed, for example coalescing single small undos into larger ones, supporting undoable commands by means of specialized exceptions, and so on.

The Edit class

The `Edit` class represents the generic 'executed action' by the user on the system, and is recorded for future use in undo/redo operations. It is implemented as a subclass of the `AbstractUndoableEdit` class, part of the `swing` library for undo support. It is shown in Figure 16.16.

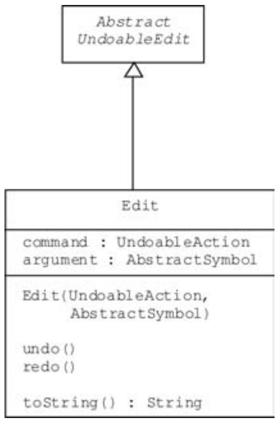


Figure 16.16 The Edit class

An `Edit` instance stores a command and its argument. Such a class can be used in two ways:

- For instantiating simpler edits, those that take only one object as an argument and that can be un/done by invoking the corresponding command class.
- As a base class for creating classes that are specialized for recording and handling more complex edits.

Recording edits

A few words about a data structure useful for managing undo and redo commands are appropriate here, but readers not interested in such implementation details can skip this section.

For simple undo support there is no problem, because a simple `Stack` instance could contain all the edits to be undone. When a redo command is to be supported, however, an additional stack can be used in which edits popped from the undo stack are pushed onto the redo stack, and vice-versa, or a specialized data structure could be used. The following shows a simple implementation of such a data structure.

Suppose the following user commands have been issued:

1. A new object `symb1` is added to the Sandbox.
2. The object is rotated.
3. A new object `symb2` is added to the Sandbox.
4. The first object is put in front of the others.
5. A new object `symb3` is added to the Sandbox.
6. The previous action (addition of `symb3`) is undone.
7. The previous action (`symb1` move front) is undone.
8. The redo command is issued, so that the `symb1` rotation is restored.

Figure 16.17 shows the state of the command history at the end of these interactions.

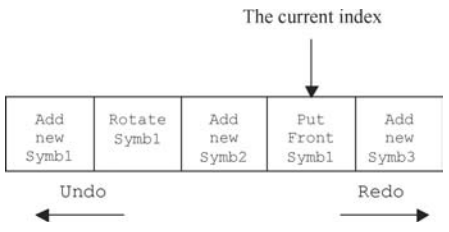


Figure 16.17 The CommandHistory implementation using an ordered list

If the user were to issue a new command, say 'add new `object4`,' it would be inserted to the right of the one pointed to by the index value, and the index incremented, moving right one position. For a redo command, the current index pointer would be moved to the right, so that it points to the next command to be undone. To keep the data structure to a manageable size and avoid expanding it indefinitely, a maximum size parameter can be enforced and the oldest edits discarded.

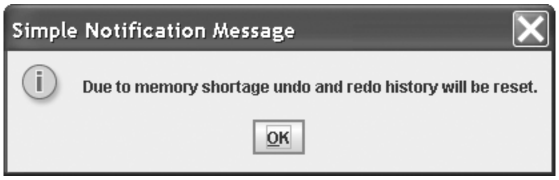
The source code for this chapter is available in the `sandbox` package.

Memory issues

Recording edits using strong references (that is, the usual normal Java references) can result in a naïve design – memory occupancy grows indefinitely with application use, so that sooner or later the heap memory is completely filled with undo records. This is especially true when working with large images. This is just the kind of situation we want to avoid.

Limiting the `CommandHistory` size to keep memory use under control circumvents this problem, but may limit the user experience – limiting the number of commands that can be undone can result in nasty surprises for users.

To provide a more flexible mechanism, we could release memory only when it is really needed. The simplest solution for this situation, from a technical viewpoint, is to use `WeakReference`s, special references that can be reclaimed by the garbage collector when the JVM is short of free heap memory. The net effect for the user is to experience a sudden reset of the undo history. Using a `ReferenceQueue` instance means that the application is notified when the most weakly referenced edits are about to be discarded. The user can then be informed about what is going on, for example by providing a pop-up message dialog like the one shown in Figure 16.18.

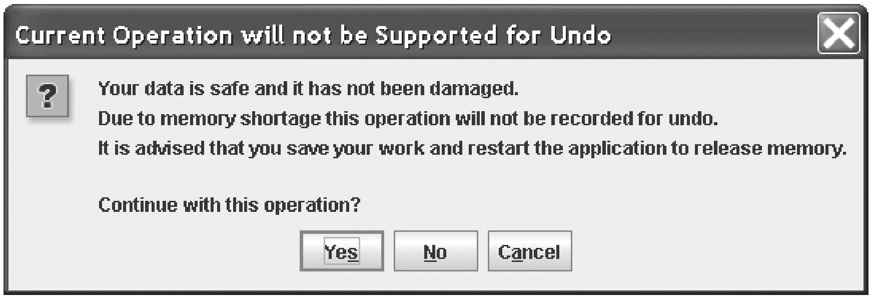


Due to memory shortage undo and redo history will be reset.

Figure 16.18 Notifying users of undo history (Ocean1.5)

From a technical perspective this solution is quite simple, as the garbage collector takes charge of all the work. From a usability viewpoint, though, this solution could cause problems, as the memory flushing happens unpredictably and ougtside of user control – and it often tends to happen during delicate, complex, and memory-consuming operations. It may therefore disrupt the user's work, and will certainly creates a vaguely unpleasant feeling.

A better solution from a usability viewpoint is to leave the user in control of the application. This can be done by checking memory occupancy before issuing an undoable command. This allows users to decide whether or not to continue with the operation, reducing the probability of their being trapped in harmful situations. In this approach the application might show a dialog like that shown in Figure 16.19 before executing the command.


Current Operation will not be Supported for Undo

Your data is safe and it has not been damaged.
Due to memory shortage this operation will not be recorded for undo.
It is advised that you save your work and restart the application to release memory.

Continue with this operation?

Figure 16.19 Preemptive control provides less intrusive notification (Ocean1.5)

Note that Figure 16.19 has a reassuring **Cancel** option – even though it is redundant, given the **No** option. This design choice is provided deliberately to ease user comprehension of the GUI in challenging situations when users are not used to this messages and become anxious about the security of their data. Look and feel design guidelines, for example those for Java and Apple Macintosh, explicitly dictate that every command has a 'cancel' option.

This type of preemptive control can be achieved as shown in the following simplified code extract:

```
if (Runtime.getRuntime().freeMemory()<MINIMUM_THRESHOLD){
     int userChoice = requirePermissionForCommand();
     if (userChoice==JOptionPane.OK_OPTION) {
          // execute command without undo support
     } else {
          // return without executing command
     }
} else {
     // execute command with undo support
}
```

Solutions that combine several techniques could of course be used. For example, we could use preemptive controls together with a policy of releasing the oldest edits explicitly, by setting their references to null and checking that they are no longer referenced using a memory profiler.

Which solution is best? It depends on the application. When developing a financial application in which the effect of every command must be tracked, the preemptive control shown in Figure 16.19 may be the simplest and most usable solution. In other cases there may be no need to record every operation exactly, and a less intrusive application could be used in which the oldest edits simply 'slip away' without users noticing.

16.6 Control

So far we have defined the container, the contained objects, and the actions that allow user interaction with the Sandbox component.

The design of the action control is left. In trivial GUIs such a feature is often not needed, and actions operate directly on the GUI objects on behalf of the user. In more complex situations, though, it is useful to add an additional layer of control, usually achieved by means of a specialized class that enforces the business logic among different parts of the GUI.

This is where the director class comes in, and has a twofold purpose. Its main duty is to implement the Mediator design pattern[6] for overseeing the interaction among different parts of the GUI. Such a class also turns out to be a good candidate for centralizing action-related code, which minimizes class coupling. Actions need only to see the director class, which in turn executes commands by manipulating the rest of the GUI. This ensures logical coherence, undo/redo support, and also rationalizes class communication. Intuitively, the director class represents the 'brains' of the application.

Figure 16.20 shows this connection scheme for the `Cut` action class.

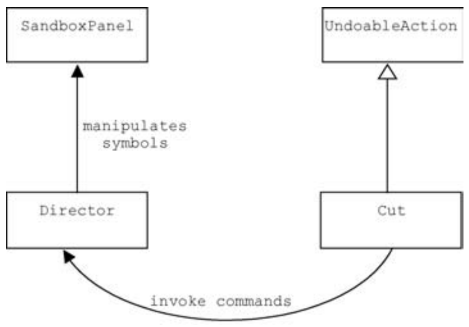


Figure 16.20 The Cut action executes the related command through the director class

6. See Chapter 6.

In this example we are going to limit control logic to action coherence, as we will see later in the implementation details.

The director is coupled to the other classes by an event-based mechanism. The director listens to object events and acts directly on the container class, as shown in Figure 16.21.

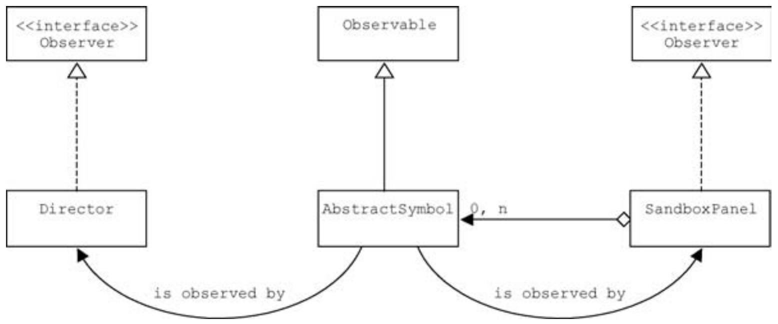


Figure 16.21 Making objects communicate with the rest of the world

The director also manages the undo/redo support by means of the `CommandHistory` class, as shown in Figure 16.22.

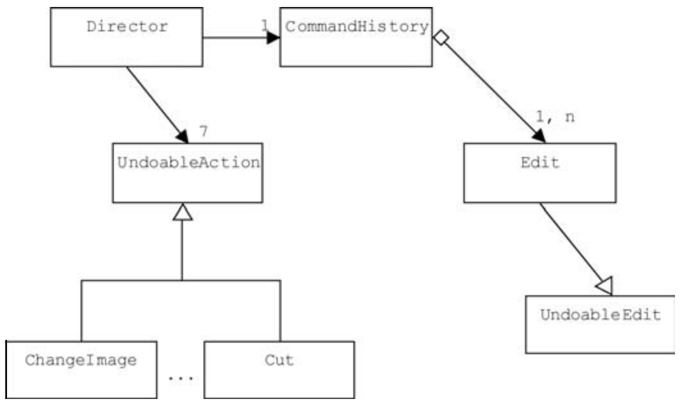


Figure 16.22 The control framework

The Director class

The `Director` class implements the Mediator design pattern, which was introduced in Chapter 6. It is shown in Figure 16.23 below.

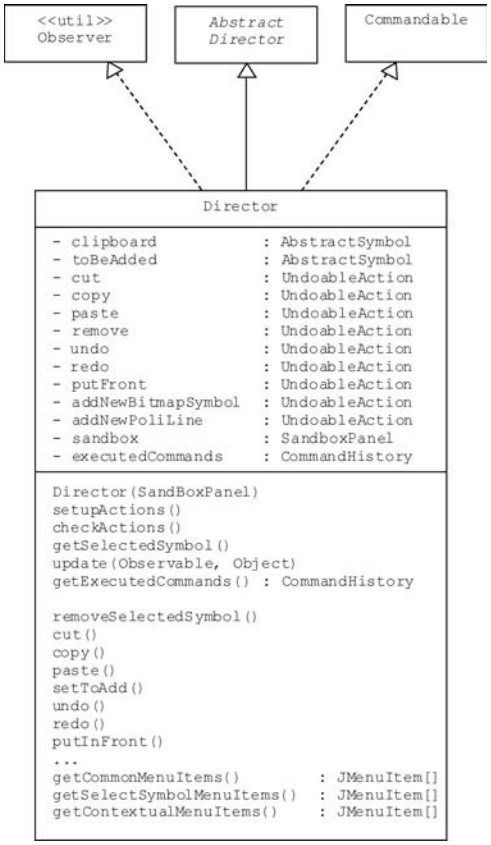

<<util>>
Observer
Abstract
Director
Commandable
Director
- clipboard : AbstractSymbol
- toBeAdded : AbstractSymbol
- cut : UndoableAction
- copy : UndoableAction
- paste : UndoableAction
- remove : UndoableAction
- undo : UndoableAction
- redo : UndoableAction
- putFront : UndoableAction
- addNewBitmapSymbol : UndoableAction
- addNewPoliLine : UndoableAction
- sandbox : SandboxPanel
- executedCommands : CommandHistory
Director(SandBoxPanel)
setupActions()
checkActions()
getSelectedSymbol()
update(Observable, Object)
getExecutedCommands() : CommandHistory
removeSelectedSymbol()
cut()
copy()
paste()
setToAdd()
undo()
redo()
putInFront()
...
getCommonMenuItems() : JMenuItem[]
getSelectSymbolMenuItems() : JMenuItem[]
getContextualMenuItems() : JMenuItem[]

Figure 16.23 The Director class

To see how this works, suppose the user invokes the cut command:

1. The `cut` method is invoked.
2. The currently-selected object is stored to the clipboard, then removed from the graphical container.
3. The whole operation is recorded for undo support.
4. The graphical container is informed of the remove operation and updates itself, making the selected object disappear.
5. An event is issued to the director to refresh command coherence.

6. If the clipboard was previously empty, after the execution of this command the paste action is automatically enabled.

The sequence diagram in Figure 16.24 illustrates this operation.

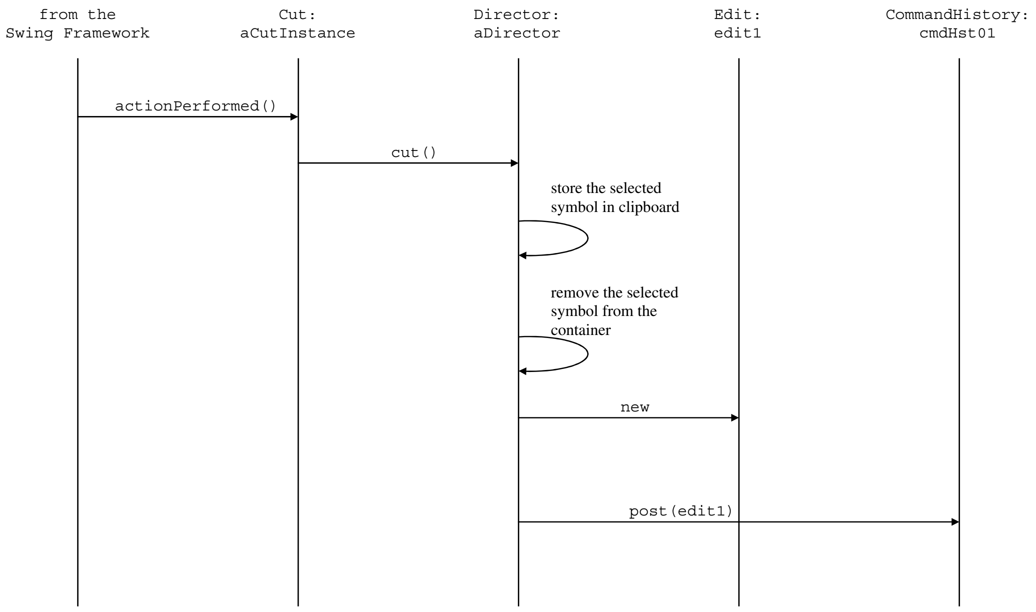


Figure 16.24 Executing the 'Cut' action

The director manages the following:

- A group of actions, described in *Managing actions* next.
- The graphical container, the `sandbox` instance variable.
- The internal clipboard, where copied or cut objects are stored.
- The set of already-performed actions, for undo/redo support.
- The object to be added to the graphical container, if any – when the user clicks the 'add new object' button, the director is informed and set to 'add' mode.

Managing actions

The director is responsible for a group of commands. These are initialized by the `setupActions()` method common to all directors, and is overridden from the `AbstractDirector` class. Such commands are packaged together for use via the `getActionToolBar()` method. Other commands are out of director's scope, such as those that are contextual to a specific object. Keeping a group of commands centralized in one class is useful in practice, because it minimizes class coupling and visibility. The latter is a classic OOP technique: lessening visibility helps to avoid cluttering the whole design and avoid misuses of public methods by other classes.

The command composition mechanism is implemented with the methods `getSelectedSymbolsMenuItems()` and `getContextualMenuItems()`. When the graphic container is queried by the user – when requesting a contextual menu – the director is invoked. The director merges the commands available from the currently-selected object, if any, with those provided by the container.

The director also takes care of the execution of some commands. Consider the `copy()` method invoked by the **Copy** action. The currently-selected object is cloned and stored in the clipboard. It is then deselected, because the standard `clone()` mechanism also copied the selected attribute value from the original, selected object, so it needs to be explicitly de-selected. Finally a check for logical coherence among actions is executed, as discussed in the next section[7].

Enforcing logical constraints on actions

The `checkActions()` method maintains coherence among all actions. This is a one of the key benefits of the Mediator pattern.

This application has three different kinds of constraints on actions:

- A group of commands (**Move front**, **Cut**, **Copy**, and **Remove**) operates on selections: if no object is selected, these actions cannot be invoked and should be disabled.
- Other action types need a non-empty clipboard: in the Sandbox application only the **Paste** action has this constraint.
- Undo/redo commands are available only when an action has been performed (undo) or when at least one old action has been undone (redo).

Abstracting from this simple example, the task of centralizing action coherence is a common one in sophisticated GUIs. Enabling or disabling actions following the application domain logic is a feature that is often dictated directly by the GUI design. From an implementation viewpoint, when the number of actions to check at any one time is non-trivial – perhaps ten or more checks, with related method calls and the like – the design suggested in the director class used in the example

7. The example code does not include undo support for copying objects.

component needs to be modified. One solution is to provide specialized events, and their related listeners, for the various types of constraint to be enforced. This avoids the frequent computation of a unique, catch-all, expensive, method. Instead, more specialized methods are invoked only when needed. This is the same idea that was used for enhancing the event delivery mechanism of the old AWT library.

As the size of applications grows, the use of a director class becomes more and more useful. You may even end up with one or more specialized classes that are solely responsible for keeping your many commands logically coherent. This reduces the maintenance costs associated with dispersing control code locally throughout the various widgets' listeners.

We are now ready to see all the different parts put together in the final application.

16.7 The whole picture

Figure 16.25 shows the overall static class diagram of the Sandbox application.

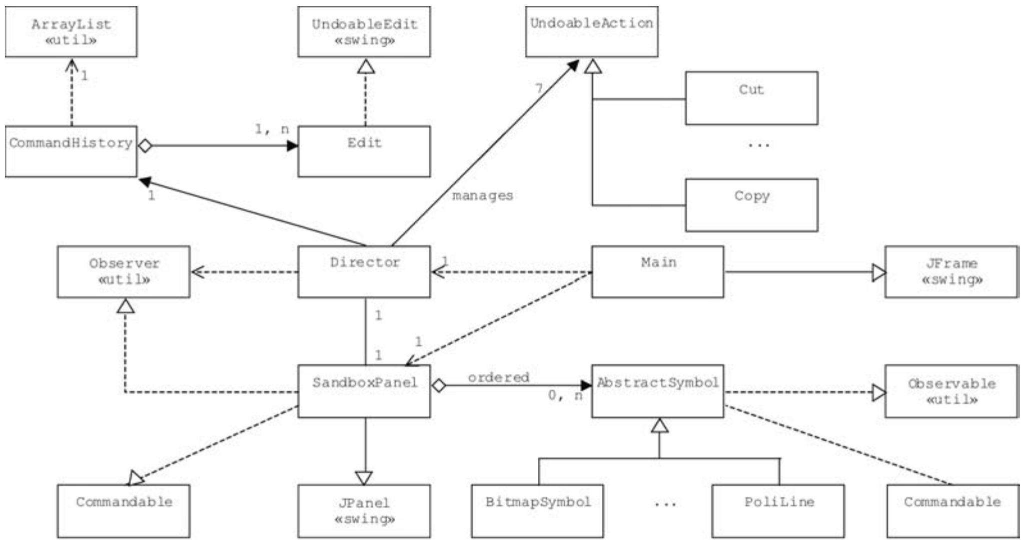


Figure 16.25 The static class diagram of the Sandbox application

We have finished the iterative refinement steps, and the application is ready to run. But what about the top-down approach initially adopted to organize the design? Figure 16.26 shows how the final classes match the initial functional partition.

Gray classes are not part of our framework. Note that the only class not contained in any of the three parts devised at the beginning is the `Main` class containing the window and the `main` method.

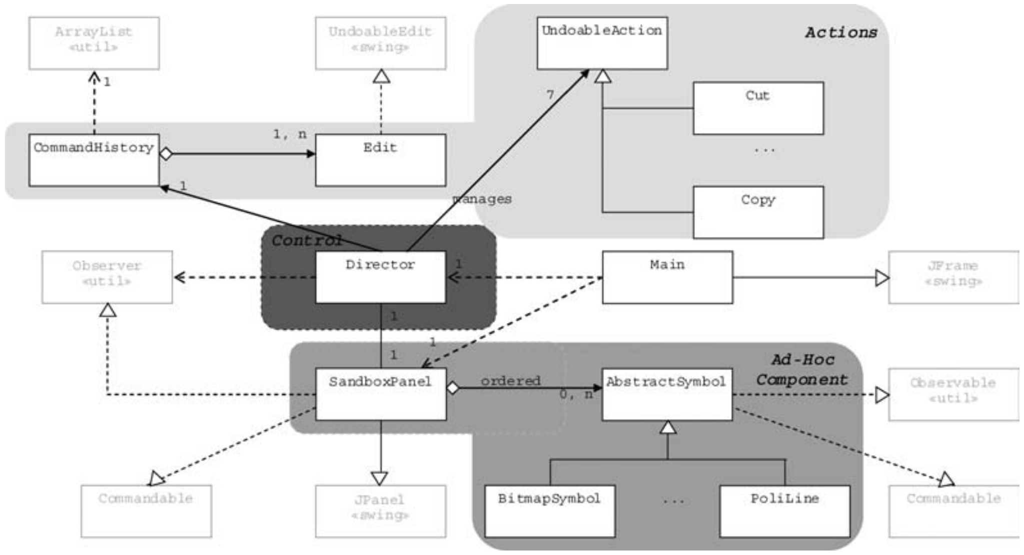


Figure 16.26 The static class diagram mapped to the functional partition

The preliminary high-level partition shown in Figure 16.6 on page 574 has been respected almost completely, apart from the communication flows among different functional parts.

16.8 Stressing the software design

At this point, some authors would hastily conclude the chapter, perhaps chanting the many virtues of design patterns, or exhorting readers to tweak the proposed source code. Instead, we take a different path, one rarely taken in technical publications in which only perfect – or supposedly perfect – solutions are presented. We already know that our design is too simple and limited to deal effectively with features like extensibility and flexibility. What we still don't know is *where* its major weak points are.

There is nothing wrong with code that works well. Code organization can of course be improved by refactoring, or by more painful design restructuring, but the real test of a design – the chances of making it better, or the risk of degradation – comes when changes are needed. Below we briefly analyze how our 'toy' Sandbox design reacts to change.

Adding objects and commands

The first experiment is to see what happens to the design when new commands such as **Print** or **Save** are added to the application. Given the design, the minimum set of steps needed to, say, add a new object class to the Sandbox application are:

1. Create a new action class, usually by extending `UndoableAction` containing the code for the relevant command.
2. Enlist the new action class in the dynamic repository held by the `Actions` class.
3. Register the new action class semantically within the Director instance. This in turn involves two steps: (i) Attaching the command to the toolbar, to make it available to users, and (ii) Writing the code that determines how the new command is going to interact with the rest of the application.

A further interesting expansion is the addition of new objects, such as text, box, or ellipse, for example. This involves defining a new object class, and the creation of an action for adding such objects to the drawing. The minimal steps we need to add a new object class to the Sandbox application are:

1. Create the new class that implements the object, extending `AbstractSymbol` or some of its subclasses.
2. Make a new action class for creating and adding objects of the new type to drawings.
3. Register the new action class in the dynamic repository held by the `Actions` class.
4. Make the director provide the new 'create object…' action to the rest of the application by attaching it to the toolbar.
5. Modify the Director class to handle the intended interactions and control for the new object.

The addition of a new object type to the design of the Sandbox application, excluding its special commands, therefore involves the creation of a new class, and adding code to other two classes, `Actions` (to add the 'create new object' command) and `Director`. The latter dependency is suspect, because the 'create new object' action doesn't involve any special control from the `Director`.

The point is that we gave the simple Director class the responsibility of creating *all* the commands, not only those that need centralized control, such as **Copy** or **Undo**. This may become a problem as the number of 'not controlled' actions outnumbers the controlled ones – the Director class may become cluttered with code unrelated to interaction control. Such code can be factored out in, for example into a `SymbolPalette` class, that is provided with mechanisms for semantic registration of specific actions with the director.

The design's weak points

Adding new objects or commands is relatively easy: the real trouble starts when we need to implement 'horizontal' features that might affect many existing classes in unforeseen ways. Adding zoom support, for example, could impact the whole design deeply. The effect of implementing other features might be more circumscribed, such as multiple views of the same drawing, for example, by adopting a fully-fledged MVC approach, or the ability to group and ungroup objects.

The relative complexity of just adding a new command to the application should ring alarms that suggest closer inspection.

Our director implementation seems to be centralizing some of other classes' responsibilities too much, such as command execution and command status (that is, whether a command should be enabled or disabled given specific external states). This stems directly from the limited use of events in our design – command classes delegate the execution of actions to the director. This simplifies many things initially, such as control: the director pulls together all the data needed to execute and control actions.

However, such a design is too tightly coupled to be maintainable in even a simple real-world situation in which tens of commands to be managed: following such a centralized design in the evolution of an application will lead to a large, convoluted Director class with a myriad of responsibility-free little classes delegating to it. To use a colorful metaphor, it's like feeding a baby monster, currently so small and cute that we don't realize its intrinsically evil nature that, once big enough to be out of control, will devour us.

Despite that it seems that sound design patterns were adopted, and what happened in reality was a misuse of them. For example, controlling whether the **Cut** action should be disabled could be done within the Command pattern approach in a decentralized fashion by letting the `Cut` action listen to clipboard events and enable or disable itself accordingly.

There is another, subtler issue with the proposed design. Commands in the design affect at least three classes: the `Action` subclass, the `Actions` repository that is queried for new commands, and the Director that executes the command itself and takes care of maintaining coherence among the actions and objects contained in the sandbox container. This can be seen as a simple, currently harmless lack of decoupling in the design. This is a well-known aspect of software. Orthogonal systems, using (Hunt and Thomas 2000) terminology, are those systems in which there are no 'effects between unrelated things.' Perhaps this is exaggerating a little, but if it is necessary to modify the implementation of the **Save** command to provide a new file format, for example, why should it also be necessary to study the implementation of three other unrelated classes as well?

16.9 Introducing JHotdraw

JHotDraw is a Java GUI framework for technical and structured graphics that is derived from an initial Smalltalk design by Ward Cunningham and Kent Beck. It has been developed as a 'design exercise' in Java by Erich Gamma and Thomas Eggenschwiler, and has a design that relies heavily on standard design patterns. It is freely available at `http://www.jhotdraw.org`.

We introduce this framework as a gallery of interesting design solutions to the problems found with our simple Sandbox application. It would make little sense to compare our toy application, with its ten classes, with such a framework, which has more than 180 classes, the many systems developed using it, and the many experienced designers who have worked on its design, but nevertheless its introduction may help further discussions.

A framework is a reusable, 'semi-complete' application that can be specialized to produce custom applications (Johnson 1998).

Frameworks are different than usual class libraries, even though sometimes the term is also used for complex class libraries. OOP frameworks usually represent a domain of interest (for example insurance) or application area (for example GUI toolkits) with OOP technology, with the explicit aim of being reused. Frameworks are intended to be reused, with client code extending them and make them concrete as required, and often define particular control flows such as inversion of control, and other devices for reuse such as hook methods, base types to be extended by client code, and so on.

The class diagram in Figure 16.27 shows only the interfaces for clarity, not the real implementations. The concrete class structure is made up of implementation classes for the interfaces shown in the following diagrams.

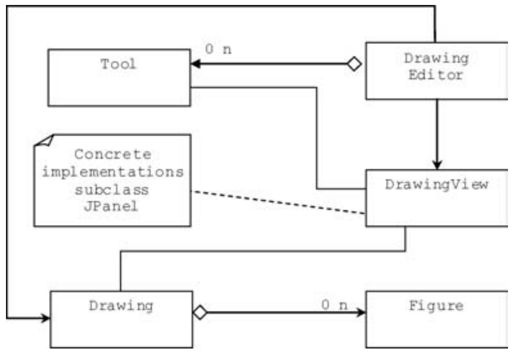


Figure 16.27 The main core class diagram of the main classes of the JHotDraw framework

The `DrawingView` implementation, such as `org.jhotdraw.standard.Standard-DrawingView`, for example, redirects user input to the installed `Tool` instances. `Drawing` implementations act as containers for `Figure` instances (the equivalent of `AbstractSymbol` in the Sandbox application) taking care of firing `Draw-ingChanged` events to registered `DrawingChangeListeners`. This type of listener is specialized for dealing with three types of occurrence:

- When an area of the `Drawing` instance is invalid.
- When the `Drawing` instance requests an update.
- When the title string changes.

Instances of the `DrawingEditor` interface coordinate the various parties involved in the editor interaction that realizes the Mediator design pattern, as does the `Director` class in the Sandbox application. To minimize decoupling with the rest of the interested classes, the main container implements the `DrawingEditor` interface, for example like `org.jhotdraw.application.DrawApplication`, a `JFrame` subclass.

It is interesting to contrast the concept of a JHotdraw tool with the way in which user input is handled in the Sandbox application. Basically, a tool defines a modality of the drawing view as perceived by the user. Tools are activated when the user clicks on a button in the palette, and deactivated when the user clicks on another button. While active, a tool consumes all input events captured by the drawing view instance and redirected to the currently-active tool. Tools inform their editor when they are finished with an interaction, for example after the creation of a new figure, by calling the editor's `toolDone()` method. Similarly to Sandbox's actions, tools are created once and then reused. Apart from this similarity, however, JHotdraw tools and Sandbox actions represent the functions available to the user in a different way. Specialized tools, subclasses of `org.jhot-draw.standard.CreationTool`, handle the creation of new figures, while in the much simpler Sandbox application, object creation is performed directly by the application.

(Christensen 2004) provides a more detailed static view of the core types of the JHotDraw framework, which still apply in Version 6.0.b1, and this is shown in Figure 16.28.

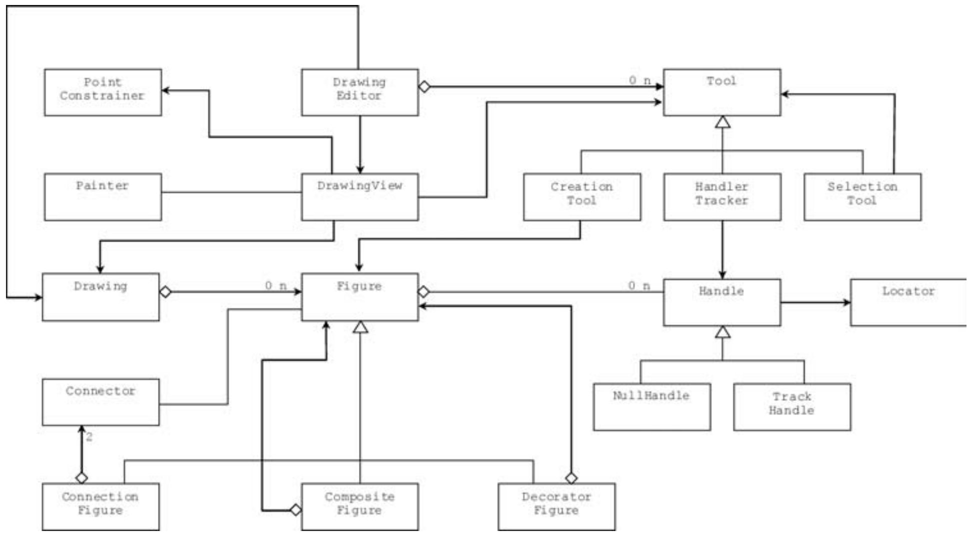


Figure 16.28 The static class diagram of the main classes of the JHotDraw framework

JHotDraw main classes can also be decomposed following the same simple decomposition that guided the development of our ad-hoc Sandbox component, as shown in Figure 16.29.

A functional decomposition like that shown in Figure 16.29 can be helpful in showing alternative organizations for the OOP implementation of our own ad-hoc component, but it might be more useful to show a decomposition based on a standard and not domain-dependent set of abstractions. The perfect candidates for these abstractions are the well-known OOP design patterns. Following (Christensen 2004) again, we have the decomposition in Figure 16.30.

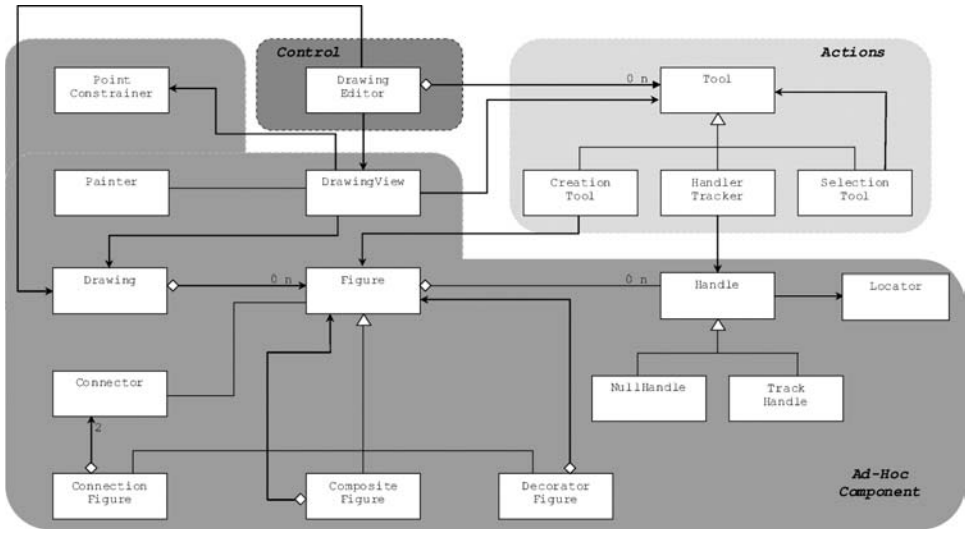


Figure 16.29 A functional decomposition of the main classes of the JHotDraw framework

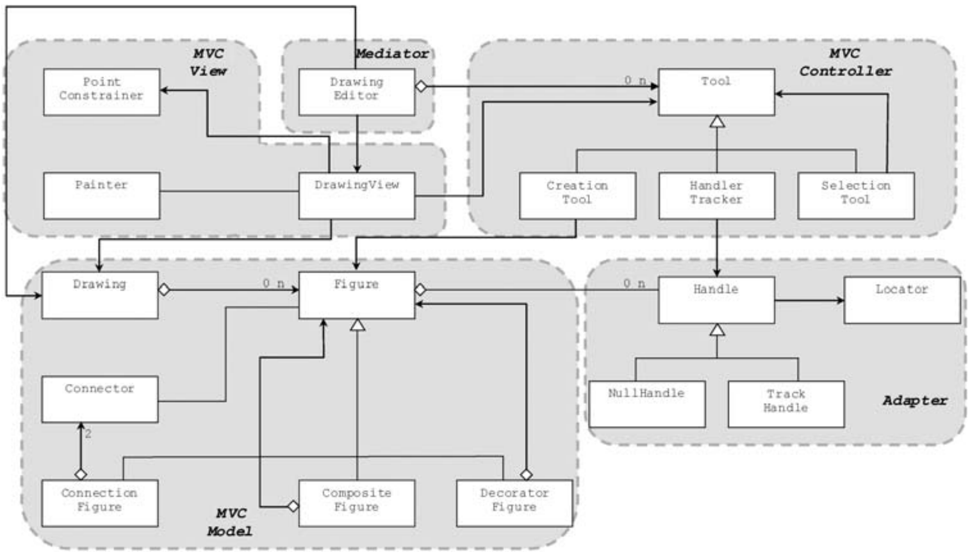


Figure 16.30 The main design patterns in the JHotDraw framework

It is easy to customize the JHotDraw framework to provide the same features as the Sandbox application. Figure 16.31 shows such a prototype.

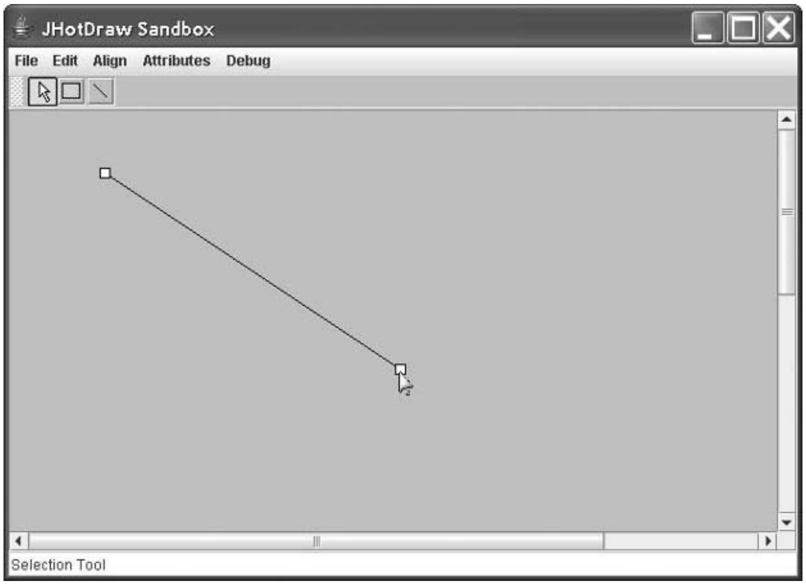


Figure 16.31 The Sandbox application with the JHotDraw framework (Ocean1.5)

16.10 Summary

In this chapter we have discussed an example of an ad-hoc component design. Such components can be quite expensive, both in terms of design and implementation, but in some cases they can make the difference between a great GUI and a mediocre product.

Key ideas

We saw many ideas at work that could also be useful in other contexts. Let's recap some of them:

- Design and use ad-hoc components only when it is really necessary. Usually, this is the case only for domain-specific and/or high-quality GUIs.
- Small classes can be gathered into container classes at compile-time, a technique that is referred to as 'static containment.' When the same arrangement is needed for runtime instances, a similar approach can be adopted,

employing a container class that releases instances as needed. Such instances can be released, or new ones created, by prototyping the stored instances.

- Adopting the fully-fledged Command pattern for handling GUI commands has many advantages, but can increase code complexity. Some remedies can be used, such as factoring out common commands or undo data to save on the number of classes used, a technique demonstrated in the Sandbox example.
- Ad-hoc components development can easily slide into the construction of specialized, fully-fledged, small class frameworks. Beware of the 'feature creep' phenomenon, which can be common among developers building ad-hoc components.

A A Questionnaire for Evaluating Java User Interfaces

This questionnaire can be used as an acceptance test at the end of a test session with users, or for a first-cut usability evaluation. It should not be used as a substitute for usability tests.

Section A: Your experience with the program

How long have you worked with the program?

1 hour or less	☐
1 hour to 1 day	☐
1 day to 1 week	☐
1 week to 1 month	☐
1 to 6 months	☐
6 months to 1 year	☐
More than a year	☐

Section B: Overall reactions

Please rate your reactions to the program:

terrible – wonderful

☐ ☐ ☐ ☐ ☐

difficult – easy

☐ ☐ ☐ ☐ ☐

frustrating – satisfying

☐ ☐ ☐ ☐ ☐

boring – stimulating

☐ ☐ ☐ ☐ ☐

rigid – flexible

☐ ☐ ☐ ☐ ☐

unhelpful – productive

☐ ☐ ☐ ☐ ☐

extremely slow– very responsive

☐ ☐ ☐ ☐ ☐

Section C: Past experience

How many other operating systems have you worked with?

1	☐
2	☐
3	☐
4 or more	☐

Please rate your familiarity with Java:

never used before – very high

☐ ☐ ☐ ☐ ☐

For the following items, tick those that you have personally used and are familiar with:

PC	Cellphone	PDA	Eclipse
☐	☐	☐	☐
Flash drive	Web browser	CD-ROM drive	Database software
☐	☐	☐	☐
Java runtime software (JRE)	Java	J2ME applet	JavaCard
☐	☐	☐	☐

Section D: Terminology

How well does the program terminology relate to the work you are doing?

not at all – very well

☐ ☐ ☐ ☐ ☐

Program terminology is used:

too frequently – appropriately

☐ ☐ ☐ ☐ ☐

The terminology is:

ambiguous – precise

☐ ☐ ☐ ☐ ☐

Messages are:

confusing – clear

☐ ☐ ☐ ☐ ☐

Message positions on the screen are:

consistent – inconsistent

☐ ☐ ☐ ☐ ☐

How often do error messages clarify the problem?

never – always

☐ ☐ ☐ ☐ ☐

Error messages seem:

annoying – constructive

☐ ☐ ☐ ☐ ☐

How often do error messages help to solve the problem?

never – always

☐ ☐ ☐ ☐ ☐

Section E: Feedback

Does the program keep you informed about what it is doing?

never – always

☐ ☐ ☐ ☐ ☐

The mouse pointer shape help in showing the current software state:

never – always

☐ ☐ ☐ ☐ ☐

While performing a task, the program freezes without showing what it is doing:

never – always

☐ ☐ ☐ ☐ ☐

Is it possible to configure the feedback level?

impossible – easy

☐ ☐ ☐ ☐ ☐

Section F: Learning the application

Learning to use the software was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Getting started with the software was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Learning advanced features was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Exploring the application by trial and error was:

risky – safe

☐ ☐ ☐ ☐ ☐

Discovering new features was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Section G: Display organization

The display organization (windows, panels, etc.) was:

confusing – clear

☐ ☐ ☐ ☐ ☐

Characters and icons were:

hard to read – very readable

☐ ☐ ☐ ☐ ☐

The command icons were:

confusing – clear

☐ ☐ ☐ ☐ ☐

The overall graphic appearance was:

annoying – pleasing

☐ ☐ ☐ ☐ ☐

Section H: Help support

Technical manuals were:

confusing – clear

☐ ☐ ☐ ☐ ☐

On-line manuals were:

confusing – clear

☐ ☐ ☐ ☐ ☐

On-line manuals were meaningfully structured:

never – always

☐ ☐ ☐ ☐ ☐

Help material covers the program features:

inadequately – completely

☐ ☐ ☐ ☐ ☐

Help support activation was:

slow – quick

☐ ☐ ☐ ☐ ☐

Help material was concise and to the point:

never – always

☐ ☐ ☐ ☐ ☐

Learning to use the program by using the help was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Section I: Deployment

Installation was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Launching the program is:

tricky – straightforward

☐ ☐ ☐ ☐ ☐

Upgrading to a newer software version was:

difficult – easy

☐ ☐ ☐ ☐ ☐

B A Questionnaire for Evaluating J2ME Applications

This questionnaire can be used as an acceptance test at the end of a test session with users, or for a first-cut usability evaluation. It should not be used as a substitute of usability tests.

Section A: Your experience with the program

How long have you worked with the program?

1 hour or less	☐
1 hour to 1 day	☐
1 day to 1 week	☐
1 week to 1 month	☐
1 to 6 months	☐
more than 6 months	☐

Section B: Your overall reactions

Please quantify your reactions to the program:

terrible – wonderful

☐ ☐ ☐ ☐ ☐

difficult – easy

☐ ☐ ☐ ☐ ☐

frustrating – satisfying

☐ ☐ ☐ ☐ ☐

boring – stimulating

☐ ☐ ☐ ☐ ☐

rigid – flexible

☐ ☐ ☐ ☐ ☐

unhelpful – productive

☐ ☐ ☐ ☐ ☐

Section C: Your past experience

How many cell similar devices have you used before?

1	☐
2	☐
3	☐
4 or more	☐

Rate your familiarity with Java applets: never used before – very high

In the following items check those that you have personally used and with which you are familiar:

PC	Smart phone	Handheld device	Portable Game Console
☐	☐	☐	☐
SMS	Web Browser	Portable MP3 Player	Digital Camera
☐	☐	☐	☐
Java	Java Virtual Machine (JVM)	J2ME applet	WiFi Network
☐	☐	☐	☐

Section D: Terminology

How close is the program's terminology to what you would expect?

not at all – very close

☐ ☐ ☐ ☐ ☐

The terminology is:

ambiguous – precise

☐ ☐ ☐ ☐ ☐

Technical terminology is used:

too frequently – appropriately

☐ ☐ ☐ ☐ ☐

Messages are:

confusing – clear

☐ ☐ ☐ ☐ ☐

Messages positions on the screen are:

consistent – inconsistent

☐ ☐ ☐ ☐ ☐

How often do error messages clarify the problem?

never – always

☐ ☐ ☐ ☐ ☐

Error messages seem:

annoying – pleasant

☐ ☐ ☐ ☐ ☐

Error messages help to solve the problem?

never – always

☐ ☐ ☐ ☐ ☐

Section E: Program feedback

The program keeps you informed about what it is doing?

never – always

☐ ☐ ☐ ☐ ☐

How frequently does the pointer shape (if any) help in showing the current application state?

never – always

☐ ☐ ☐ ☐ ☐

How often does the program freeze without showing what it is doing?

never – always

☐ ☐ ☐ ☐ ☐

Section F: Learning to use the application

Learning to use the application was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Getting started with the application was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Learning advanced features was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Exploring the features by trial and error was:

risky – safe

☐ ☐ ☐ ☐ ☐

Discovering new features was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Configuring the application's preferences was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Section G: Display organization

The display organization (screens, forms, etc.) was:

confusing – clear

☐ ☐ ☐ ☐ ☐

Characters and icons were:

hard to read – very readable

☐ ☐ ☐ ☐ ☐

The command icons were:

confusing – clear

☐ ☐ ☐ ☐ ☐

The overall graphic appearance was:

annoying – pleasing

☐ ☐ ☐ ☐ ☐

The application had the same look as other programs:

totally different – exactly the same

☐ ☐ ☐ ☐ ☐

Section H: Navigation

It was possible to cancel an operation or navigate back to a previous screen:

never – always

☐ ☐ ☐ ☐ ☐

Navigation keys and navigation commands were:

confusing – clear

☐ ☐ ☐ ☐ ☐

confusing – clear

The number of screens was:

too many – about right

☐ ☐ ☐ ☐ ☐

Reaching a given screen was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Section I: Help support

Help content was:

confusing – clear

☐ ☐ ☐ ☐ ☐

Other manuals (if any) were:

confusing – clear

☐ ☐ ☐ ☐ ☐

Help material covers the program features:

inadequately – completely

☐ ☐ ☐ ☐ ☐

Help activation was:

slow – quick

☐ ☐ ☐ ☐ ☐

Help material was easy to find:

never – always

☐ ☐ ☐ ☐ ☐

Learning to use the application by using the help support was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Section J: Deployment

Installation was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Launching the application the first time was:

confusing – clear

☐ ☐ ☐ ☐ ☐

Launching the application was:

tricky – straightforward

☐ ☐ ☐ ☐ ☐

The waiting time for launching the application was:

extremely long – reasonable

☐ ☐ ☐ ☐ ☐

Upgrading to a newer version was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Section K: Mobile experience

Switching the application on or off (pausing and restoring it) was:

difficult – easy

☐ ☐ ☐ ☐ ☐

Operations required extra attention:

never – always

☐ ☐ ☐ ☐ ☐

How many times did you have to start an operation all over again?

never – more than three times

☐ ☐ ☐ ☐ ☐

The application respected my privacy:

never – always

☐ ☐ ☐ ☐ ☐

The application handled interruptions such as phone warnings, phone calls, other external situations:

badly – very well

☐ ☐ ☐ ☐ ☐

Remote connections were signaled:

confusingly – clearly

☐ ☐ ☐ ☐ ☐

The application asked permission before making remote connections:

never – always

☐ ☐ ☐ ☐ ☐

The application respected the current phone settings, such as ringer off:

never – always

☐ ☐ ☐ ☐ ☐

References

(Advanced Java L&F Design Guidelines 2001) AA.VV. 2001. *Java Look And Feel Design Guidelines: Advanced Topics*. Reading, Massachusetts: Addison-Wesley.

(Alur, Crupi and Malks 2001) Alur, Deepak, Crupi, John and Malks, Dan. 2001. *Core J2EE Patterns*. Englewood Cliffs, New Jersey: Prentice Hall.

(Beck and Andres 2004) Beck, Kent, and Andres, Cynthia. 2004. *Extreme Programming Explained*: *Embrace Change*. Second Edition. Reading, Massachusetts: Addison-Wesley Professional.

(Bernard 2004) Golden, Bernard. 2004. *Succeeding with Open Source*. Addison-Wesley.

(Brooks 1995) Brooks, Frederick P Jr. 1996. *The Mythical Man-Month, Anniversary Edition*. Reading, Massachusetts: Addison-Wesley.

(Brown et al. 1998) Brown, William H et al. 1998. *Anti Patterns. Refactoring Software, Architectures, and Projects in Crisis*. New York: John Wiley & Sons, Inc.

(Burbeck 1992) Burbeck, Steve. 1992. *Application Programming in Smalltalk-80: How to Use the Model-View-Controller (MVC)*. Technical Report.

Available at:
http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

(Buschmann et al. 1996) Buschmann, Frank et al. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns.* New York: John Wiley & Sons, Inc.

(Buschmann et al. 2000) Buschmann, Frank et al. 1996. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects.* New York: John Wiley & Sons, Inc.

(Christensen 2004) Christensen, Henrik B. 2004. *Frameworks: Putting Design Patterns into Perspective*. In Proceedings of ITiCSE'04, June 28–30, 2004, Leeds, United Kingdom.

(Conallen 2002) Conallen, Jim. 2002. *Building Web Applications with UML*, Second Edition. Reading, Massachusetts: Addison-Wesley.

(Cooper 1995) Cooper Alan. 1995. *The Myth of Metaphor.*

http://www.cooper.com/articles/art_myth_of_metaphor.htm

(Cooper 1999)	Cooper, Alan. 1999. *The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How To Restore The Sanity.* Sams Publishing.
(Cooper 2000)	Cooper James W. 2000. *Design Patterns in Java Technology.* Presentation at the JavaOne Conference. California.
(Cooper 2003)	Cooper, Alan, and Reimann, Robert. *About Face 2.0: The Essentials of Interaction Design.* New York: John Wiley & Sons, Inc.
(Davidson 2000)	Davidson, Mark. 2000. *Using the Swing Action Architecture.* Sun Technical Article. http://www.sun.java.com/
(Daconta et al. 2000)	Daconta, Michael C. et al. 2000. *Java Pitfalls. Time-Saving Solutions and Workarounds to Improve Programs.* New York: John Wiley & Sons, Inc.
(De Marco and Lister 1999)	De Marco, Tom, and Lister, Timothy. 1999. *Peopleware. Productive Projects and Teams.* Second Edition. New York: Dorset House Publishing Co.
(Des Rivières 2000)	Des Rivières, J. *Evolving Java-based APIs* http://www.eclipse.org/eclipse/development/java-api-evolution.html
(Des Rivières 2001)	Des Rivières, J. *How to Use the Eclipse API.* http://www.eclipse.org/articles/Article-API%20use/eclipse-api-usage-rules.html
(Elkotoubi, Khriss and Keller 1999)	Eloutbi, M, Khriss, I, and Keller, R. K. 1999. *User Interface Prototyping using UML Specifications.* Université de Montreal. Technical Report.
(Evans 2004)	Evans, Eric. 2004. *Domain Driven Design.* Reading, Massachusetts: Addison-Wesley.
(Fleming 1998)	Fleming, Jennifer. 1998. *Web Navigation: Designing the User Experience.* Sebastopol, California: O'Reilly & Associates Inc.
(Fowler 1997)	Fowler, Martin. 1999. *Analysis Patterns: Reusable Object Models.* Reading, Massachusetts: Addison-Wesley.
(Fowler 1999)	Fowler, Martin, et al. 1999. *Refactoring: Improving the Design of Existing Code.* First Edition. Addison-Wesley, Boston, Massachusetts.
(Fowler 2000)	Fowler, Amy. 2000. *Painting in AWT and Swing.* Technical Article. Available at: http://java.sun.com/products/jfc/tsc/articles/painting/index.html
(Fowler 2000b)	Fowler, Amy. *A Swing Architecture Overview. The Inside Story on JFC Component Design.* Technical Report. Available at http://java.sun.com/products/jfc/tsc/articles/architecture/

(Fowler 2003) Fowler, Martin. 2003. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Third Edition.

(Fowler et al. 2000) Fowler, Martin et al. 2000. *Refactoring. Improving the Design of Existing Code.* Reading, Massachusetts: Addison-Wesley.

(Fowler et al. 2003) Fowler, Martin et al. 2003. *Patterns of Enterprise Application Architecture.* Reading, Massachusetts: Addison-Wesley.

(Gamma et al. 1994) Gamma, Erich et al. 1994. *Design Patterns*. Reading, Massachusetts: Addison-Wesley.

(Geary 1999) Geary, David M. 1999. *Graphic Java 2. Mastering the JFC.* Third Edition. Englewood Cliffs, New Jersey: Prentice Hall.

(Holub 1999) Holub, Allen. 1999. *Building Interfaces for Object Oriented Systems*. Java-World Article. http://www.javaworld.com/

(Hunt and Thomas 2000) Hunt, Andrew and Thomas, David. 2000. *The Pragmatic Programmer. From Journeyman to Master.* Addison-Wesley.

(Hutchins et al. 1986) Hutchins, Edwin, Hollan, James, and Donald Norman. *Direct Manipulation Interfaces*, in Norman, Donald, and Draper, Stephen, *User Centered System Design*. 1986. pp. 87–124.

(Java L&F Design Guidelines 2001) AA.VV. 2001. *Java Look And Feel Design Guidelines,* Second Edition. Reading, Massachusetts: Addison-Wesley.

(Johnson 1998) Johnson, Ralph and Foote, Brian. 1988. *Designing Reusable Classes*. Journal of Object-Oriented Programming. SIGS, 1, 5 (June/July. 1988), 22–35.

(Johnson 2003) Johnson, Rod. 2003. *Expert One-on-One J2EE Design and Development.* Indianapolis: Wrox; John Wiley & Sons, Inc.

(Kerievsky 2004) Kerievsky, Joshua. 2004. *Refactoring to Patterns*. Addison-Wesley Professional.

(Kruchten and Ahlqvist 2001) Kruchten, Philippe and Ahlqvist, Stefan. *User Interface Design in the Rational Unified Process*. In M. van Harmelan, Ed., *Object Modeling and User Interface Design*. Addison-Wesley, 2001.

(Larman 2003) Larman, Craig. 2003. *Agile and Iterative Development. A Manager's Guide*. Addison-Wesley Professional.

(Mandel 1997) Mandel, Theo. 1997. *The Elements of User Interface Design.* New York: John Wiley & Sons, Inc.

(Maner 1997)	Maner, Walter. 1997. *Internationalization of User Interfaces*. Available at http://web.cs.bgsu.edu/maner/uiguides/internat.htm
(Marinilli 2000)	Marinilli, Mauro. 2000. *A Java Drawing Editor*. Gamelan Article. http://www.gamelan.com/
(Marinilli Persistence 2000)	Marinilli, Mauro. 2000. *Class Semipersistence and Instance Semipersistence: Two Powerful tools in the Software Designer Toolbox*. Gamelan article. http://www.gamelan.com/
(Marinilli 2001)	Marinilli, Mauro. 2001. *Java Deployment*. Indianapolis: Sams Publishing.
(Marinescu 2002)	Marinescu, Floyd et al. 2002. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, Inc.
(Martin 2002)	Martin, Robert C. 2002. *Agile Software Development. Principles Patterns, and Practices*. Englewood Cliffs, New Jersey: Prentice Hall.
(McConnell 1993)	McConnell, Steve. 1993. *Code Complete. A Practical Handbook of Software Construction*. Redmond, Washington: Microsoft Press.
(McConnell 1996)	McConnell, Steve. 1996. *Rapid Development. Taming Wild Software Schedules*. Redmond, Washington: Microsoft Press.
(Mullet and Sano 1995)	Mullet, Kevin, and Sano, Darrel. 1995. *Designing Visual Interfaces. Communication Oriented Techniques*. Englewood Cliffs, New Jersey: Prentice Hall.
(Nielsen 1993)	Nielsen, Jakob. 1993. *Usability Engineering*. San Diego: California Academic Press.
(Norman 1990)	Norman, Donald A. 1990. *The Design of Everyday Things*. New York: Doubleday.
(Norman 1993)	Norman, Donald A. 1993. *Things That Makes Us Smart. Defending Human Attributes in the Age of the Machine*. Cambridge, Massachusetts: Perseus Books.
(Norman 1998)	Norman, Donald A. 1998. *The Design of Everyday Things*. Bantam Doubleday Dell Publishing.
(Potel 1996)	MVP: *Model-View-Presenter. The Taligent Programming Model for C++ and Java*. Technical Report. ftp://www6.software.ibm.com/software/developer/library/mvp.pdf
(Preece 1994)	Preece, Jenny. 1994. *Human Computer Interaction*. Reading, Massachusetts: Addison-Wesley.

(Reichert 2000) Reichert, Raimond. 2000. *Interact with Garbage collector to avoid memory leaks. Use Reference Objects to Prevent Memory Leaks in Application Built on the MVC Pattern*. Javaworld article. http://www.javaworld.com/

(Rosenberg and Scott 1999) Rosenber, Doug, and Scott, Kendall. 1999. *Use Case Driven Object Modeling with UML. A Practical Approach*. Reading, Massachusetts: Addison-Wesley.

(Rubin 1994) Rubin, Jeffrey. 1994. *Handbook of Usability Testing. How to Plan, Design and Conduct Effective Tests*. Wiley Technical Communication Library. New York: John Wiley & Sons, Inc.

(Selby and Porter 1998) Selby R. W. and Porter. A. A. 1988. *Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis*. IEEE Trans. on Soft. Eng., 14(12), pp. 1743–1757.

(Shirazi 2000) Shirazi, Jack. 2000. *Java Performance Tuning*. Sebastopol, California: O'Reilly & Associates Inc.

(Shirogane and Fukazawa 2002) Shirogane, Junko and Fukazawa, Yoshiaki. 2002. *GUI Prototype Generation by Merging Use Cases*. Proceedings of the IUI Conference. San Francisco, California.

(Shneiderman 1998) Shneiderman, Ben. 1998. *Designing the User Interface,* Third Edition. Reading, Massachusetts: Addison-Wesley.

(Snyder 2003) Snyder, Carolyn. 2003. *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces.* Morgan Kaufmann.

(Sundsten 1998) Sundsten, Todd. 1998. *MVC meets Swing. Explore the underpinnings of the JFC's Swing components Pattern*. Javaworld article. `http://www.javaworld.com/javaworld/jw-04-1998/jw-04-howto.html`

(Tidwell 1999) Tidwell, Jenifer. 1999. *Common Ground*. `http://mit.edu/`

(Tufte 1990) Tufte, Edward R. 1990. *Envisioning Information.* Cheshire, Connecticut: Graphic Press.

(Tufte 1997) Tufte, Edward R. 1997. *Visual Explanations.* Cheshire, Connecticut: Graphic Press.

(Tufte 2001) Tufte, Edward R. 2001. *The Visual Display of Quantitative Information.* Second Edition. Cheshire, Connecticut: Graphic Press.

(Vlissides et al. 1996) Vlissides, John et al. (Eds.) 1996. *Pattern Languages of Program Design 2*. Reading, Massachusetts: Addison-Wesley.

(Vlissides 1998) Vlissides, John. 1998. *Pattern Hatching. Design Patterns Applied*. Reading, Massachusetts: Addison-Wesley.

General advice on usability and GUI design

`http://www.acm.org/sigchi/`

CHI Conference proceedings abstracts and other academic research material.

`http://www.acm.org/~perlman/readings.html`

Suggested readings on HCI and UI development.

`http://www.asktog.com`

Bruce Tognazzini's Web site.

`http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/`

Apple Macintosh design guidelines.

`http://www.gui-designers.co.uk`

Practical examples of GUI design.

`http://www.ibm.com//ibm/hci/`

IBM's human computer interaction Web site.

A couple of 'interface hall of shame' sites are available on line:

`http://homepage.mac.com/bradster/iarchitect/shame.htm`

This is the most interesting and complete one, although now a little dated.

Other Web sites on the same subject:

`http://www.pixelcentric.net/x-shame/`
`http://www.frankmahler.de/mshame/index.html`
`http://www.rha.com/ui_hall_of_shame.htm`
`http://msdn.microsoft.com/ui`

Microsoft MSDN user interface resources.

`http://www.pegasus3d.com/apple_screens.html`

The evolution of the Macintosh interface.

`http://www.tworivers.com`

General and practical discussion on GUI design.

`http://www.usabilityfirst.com`

General advice on usability and GUI design.

`http://www.useit.com`

Jakob Nielsen's Web site, with some useful articles.

Java-specific links

`http://www.java.sun.com/products/jlf`

Java look and feel design guidelines.

`http://www.java.sun.com/products/jfc`

The Java Foundation Classes (JFC) official home page.

`http://www.java.sun.com/products/jfc/tsc`

The Swing Connection official home page.

`http://www.sun.com/access/articles/#articles`

Discussion of the Multiplexing look and feel, providing accessibility features.

`http://deyalexander.com/resources/design-guidelines.html`

A list of resources about GUI Design guidelines.

`http://www.cs.usm.maine.edu/~welty/`

A comprehensive list of useful HCI/ UI design links.

Index