1

Introducing the Spring Framework

Why Spring?

The Spring Framework is an open source application framework that aims to make J2EE development easier. In this chapter we'll look at the motivation for Spring, its goals, and how Spring can help you develop high-quality applications quickly.

Spring is an *application framework*. Unlike single-tier frameworks such as Struts or Hibernate, Spring aims to help structure whole applications in a consistent, productive manner, pulling together best-of-breed single-tier frameworks to create a coherent architecture.

Problems with the Traditional Approach to J2EE

Since the widespread implementation of J2EE applications in 1999/2000, J2EE has not been an unqualified success in practice. While it has brought a welcome standardization to core middle-tier concepts such as transaction management, many — perhaps *most* — J2EE applications are over-complex, take excessive effort to develop, and exhibit disappointing performance. While Spring is applicable in a wide range of environments — not just server-side J2EE applications — the original motivation for Spring was the J2EE environment, and Spring offers many valuable services for use in J2EE applications.

Experience has highlighted specific causes of complexity and other problems in J2EE applications. (Of course, not all of these problems are unique to J2EE!) In particular:

- **J2EE applications tend to contain excessive amounts of "plumbing" code.** In the many code reviews we've done as consultants, time and time again we see a high proportion of code that doesn't *do* anything: JNDI lookup code, Transfer Objects, try/catch blocks to acquire and release JDBC resources. . . . Writing and *maintaining* such plumbing code proves a major drain on resources that should be focused on the application's business domain.
- **Many J2EE applications use a distributed object model where this is inappropriate.** This is one of the major causes of excessive code and code duplication. It's also conceptually wrong in many cases; internally distributed applications are more complex than co-located applications, and often much less performant. Of course, if your business requirements dictate a distributed architecture, you need to implement a distributed architecture and accept the tradeoff that incurs (and Spring offers features to help in such scenarios). But you shouldn't do so without a compelling reason.
- **The EJB component model is unduly complex.** EJB was conceived as a way of reducing complexity when implementing business logic in J2EE applications; it has not succeeded in this aim in practice.
- **EJB is overused.** EJB was essentially designed for internally distributed, transactional applications. While nearly all non-trivial applications are transactional, distribution should not be built into the basic component model.
- **Many "J2EE design patterns" are not, in fact, design patterns, but workarounds for technology limitations.** Overuse of distribution, and use of complex APIs such as EJB, have generated many questionable design patterns; it's important to examine these critically and look for simpler, more productive, approaches.
- **J2EE applications are hard to unit test.** The J2EE APIs, and especially, the EJB component model, were defined before the agile movement took off. Thus their design does not take into account ease of unit testing. Through both APIs and implicit contracts, it is surprisingly difficult to test applications based on EJB and many other J2EE APIs outside an application server. Yet unit testing outside an application server is essential to achieve high test coverage and to reproduce many failure scenarios, such as loss of connectivity to a database. It is also vital to ensuring that tests can be run quickly during the development or maintenance process, minimizing unproductive time waiting for redeployment.
- **Certain J2EE technologies have simply failed.** The main offender here is entity beans, which have proven little short of disastrous for productivity and in their constraints on object orientation.

The traditional response to these problems has been to wait for tool support to catch up with the J2EE specifications, meaning that developers don't need to wrestle with the complexity noted above. However, this has largely failed. Tools based on code generation approaches have not delivered the desired benefits, and have exhibited a number of problems of their own. In this approach, we might generate all those verbose JNDI lookups, Transfer Objects, and try/catch blocks.

In general, experience has shown that *frameworks* are better than tool-enabled code generation. A good framework is usually much more flexible at runtime than generated code; it should be possible to configure the behavior of one piece of code in the framework, rather than change many generated classes. Code generation also poses problems for round-tripping in many cases. A well-conceived framework can also offer a coherent abstraction, whereas code generation is typically just a shortcut that fails to conceal underlying complexities during the whole project lifecycle. (Often complexities will re-emerge damagingly during maintenance and troubleshooting.)

A framework-based approach recognizes the fact that there is a missing piece in the J2EE jigsaw: the application developer's view. Much of what J2EE provides, such as JNDI, is simply too low level to be a daily part of programmer's activities. In fact, the J2EE specifications and APIs can be judged as far more successful, if one takes the view that they do not offer the developer a programming model so much as provide a solid basis on which that programming model should sit. Good frameworks supply this missing piece and give application developers a simple, productive, abstraction, without sacrificing the core capability of the platform.

> **Using J2EE "out of the box" is not an attractive option. Many J2EE APIs and services are cumbersome to use. J2EE does a great job of standardizing low-level infrastructure, solving such problems as *how can Java code access transaction management without dealing with the details of XA transactions*. But J2EE does not provide an easily usable view for application code.**
>
> **That is the role of an *application framework*, such as Spring.**

Recognizing the importance of frameworks to successful J2EE projects, many developers and companies have attempted to write their own frameworks, with varying degrees of success. In a minority of cases, the frameworks achieved their desired goals and significantly cut costs and improved productivity. In most cases, however, the cost of developing and maintaining a framework itself became an issue, and framework design flaws emerged. As the core problems are generic, it's much preferable to work with a single, widely used (and tested) framework, rather than implement one in house. No matter how large an organization, it will be impossible to achieve a degree of experience matching that available for a product that is widely used in many companies. If the framework is open source, there's an added advantage in that it's possible to contribute new features and enhancements that may be adopted. (Of course it's possible to contribute suggestions to commercial products, but it's typically harder to influence successful commercial products, and without the source code it's difficult to make equally useful contributions.) Thus, increasingly, generic frameworks such as Struts and Hibernate have come to replace in-house frameworks in specific areas.

The Spring Framework grew out of this experience of using J2EE without frameworks, or with a mix of in-house frameworks. However, unlike Struts, Hibernate, and most other frameworks, Spring offers services for use throughout an application, not merely in a single architectural tier. Spring aims to take away much of the pain resulting from the issues in the list we've seen, by simplifying the programming model, rather than concealing complexity behind a complex layer of tools.

> **Spring enables you to enjoy the key benefits of J2EE, while minimizing the complexity encountered by application code.**
>
> **The essence of Spring is in providing enterprise services to Plain Old Java Objects (POJOs). This is particularly valuable in a J2EE environment, but application code delivered as POJOs is naturally reusable in a variety of runtime environments.**

Lightweight Frameworks

Some parts of J2EE can properly be termed frameworks themselves. Among them, EJB amounts to a framework because it provides a structure for application code, and defines a consistent way of accessing

services from the application server. However, the EJB framework is cumbersome to use and restrictive. The work involved in implementing an EJB is excessive, given that the architects of J2EE expected that all business logic in J2EE applications would be implemented in EJBs. Developers must cope with three to four Java classes for each EJB; two verbose deployment descriptors for each EJB JAR file; and excessive amounts of code for client access to EJBs and EJB access to their environment. The EJB component model, up to and including EJB 2.1, fails to deliver on many of its goals, and fails to deliver a workable structure for business logic in J2EE applications. The EJB Expert Group has finally realized this and is attempting an overhaul of the EJB model in EJB 3.0, but we need a solution, right now, and Spring already demonstrates a far superior one in most cases.

Not merely EJB, but the majority of frameworks in the early years of J2EE, proved to have problems of their own. For example, Apache Avalon offered powerful configuration management and other services, but never achieved widespread adoption, partly because of the learning curve it required, and because application code needed to be aware of Avalon APIs.

> **A framework can only be as good as the programming model it provides. If a framework imposes too many requirements on code using it, it creates lock-in and — even more important — constrains developers in ways that may not be appropriate. The application developer, rather than framework designer, often has a better understanding of how code should be written.**
>
> **Yet a framework should provide *guidance* with respect to good practice: *It should make the right thing easy to do*. Getting the right mixture of constraint and freedom is the key challenge of framework design, which is as much art as science.**

Given this history, the emergence of a number of *lightweight* frameworks was inevitable. These aim to provide many of the services of "out of the box" J2EE in a simpler, more manageable manner. They aim to do their best to make the framework itself invisible, while encouraging good practice. Above all, they aim to enable developers to work primarily with POJOs, rather than special objects such as EJBs.

As the name implies, lightweight frameworks not only aim to reduce complexity in application code, but avoid unnecessary complexity in their own functioning. So a lightweight framework won't have a high startup time, won't involve huge binary dependencies, will run in any environment, and won't place obstacles in the way of testing.

While "old J2EE" was characterized by high complexity and a welter of questionable "design patterns" to give it intellectual respectability, lightweight J2EE is about trying to find the "simplest thing that can possibly work": wise advice from the XP methodology, regardless of whether you embrace XP practices overall.

> **While all the lightweight frameworks grew out of J2EE experience, it's important to note that none of them is J2EE-specific. A lightweight container can be used in a variety of environments: even in applets.**
>
> **For example, the *Spring Rich Client* project demonstrates the value of the Spring model outside the server environment, in rich client applications.**

Enter Spring

Spring is both the most popular and most ambitious of the lightweight frameworks. It is the only one to address all architectural tiers of a typical J2EE application, and the only one to offer a comprehensive range of services, as well as a lightweight container. We'll look at Spring's modules in more detail later, but the following are the key Spring modules:

- ***Inversion of Control* container:** The core "container" Spring provides, enabling sophisticated configuration management for POJOs. The Spring IoC container can manage fine or coarse-grained POJOs (object granularity is a matter for developers, not the framework), and work with other parts of Spring to offer services as well as configuration management. We'll explain IoC and *Dependency Injection* later in this chapter.
- **Aspect-Oriented Programming (AOP) framework:** AOP enables behavior that would otherwise be scattered through different methods to be modularized in a single place. Spring uses AOP under the hood to deliver important out-of-the-box services such as declarative transaction management. Spring AOP can also be used to implement custom code that would otherwise be scattered between application classes.
- **Data access abstraction:** Spring encourages a consistent architectural approach to data access, and provides a unique and powerful abstraction to implement it. Spring provides a rich hierarchy of data access exceptions, independent of any particular persistence product. It also provides a range of helper services for leading persistence APIs, enabling developers to write persistence framework–agnostic data access interfaces and implement them with the tool of their choice.
- **JDBC simplification:** Spring provides an abstraction layer over JDBC that is significantly simpler and less error-prone to use than JDBC when you need to use SQL-based access to relational databases.
- **Transaction management:** Spring provides a transaction abstraction that can sit over JTA "global" transactions (managed by an application server) or "local" transactions using the JDBC, Hibernate, JDO, or another data access API. This abstraction provides a consistent programming model in a wide range of environments and is the basis for Spring's declarative and programmatic transaction management.
- **MVC web framework:** Spring provides a request-based MVC web framework. Its use of shared instances of multithreaded "controllers" is similar to the approach of Struts, but Spring's web framework is more flexible, and integrates seamlessly with the Spring IoC container. All other Spring features can also be used with other web frameworks such as Struts or JSF.
- **Simplification for working with JNDI, JTA, and other J2EE APIs:** Spring can help remove the need for much of the verbose, boilerplate code that "doesn't do anything." With Spring, you can continue to use JNDI or EJB, if you want, but you'll never need to write another JNDI lookup. Instead, simple configuration can result in Spring performing the lookup on your behalf, guaranteeing that resources such as JNDI contexts are closed even in the event of an exception. The dividend is that you get to focus on writing code that *you* need to write because it relates to your business domain.
- **Lightweight remoting:** Spring provides support for POJO-based remoting over a range of protocols, including RMI, IIOP, and Hessian, Burlap, and other web services protocols.

- **JMS support:** Spring provides support for sending and receiving JMS messages in a much simpler way than provided through standard J2EE.
- **JMX support:** Spring supports JMX management of application objects it configures.
- **Support for a comprehensive testing strategy for application developers:** Spring not only helps to facilitate good design, allowing effective unit testing, but provides a comprehensive solution for integration testing outside an application server.

Spring's Values

To make the most effective use of Spring, it's important to understand the motivation behind it. Spring partly owes its success to its being based on a clear vision, and remaining true to that vision as its scope has expanded.

The key Spring values can be summarized as follows:

- **Spring is a *non-invasive* framework.** This is the key departure from most previous frameworks. Whereas traditional frameworks such as EJB or Apache Avalon force application code to be aware of the framework, implementing framework-specific interfaces or extending framework-specific classes, Spring aims to minimize the dependence of application code on the framework. Thus Spring can configure application objects that don't import Spring APIs; it can even be used to configure many legacy classes that were written without any knowledge of Spring. This has many benefits. For example:
  - Application code written as part of a Spring application can be run without Spring or any other container.
  - Lock-in to Spring is minimized. For example, you could migrate to another lightweight container, or possibly even reuse application objects in an EJB 3.0 EJB container, which supports a subset of Spring's Dependency Injection capability.
  - Migration to future versions of Spring is easier. The less your code depends on the framework, the greater the decoupling between the implementation of your application and that of the framework. Thus the implementation of Spring can change significantly without breaking your code, allowing the framework to be improved while preserving backward compatibility.

    Of course in some areas, such as the web framework, it's impossible to avoid application code depending on the framework. But Spring consistently attempts to reach the non-invasive ideal where configuration management is concerned.
- **Spring provides a consistent programming model, usable in any environment.** Many web applications simply don't need to run on expensive, high-end, application servers, but are better off running on a web container such as Tomcat or Jetty. It's also important to remember that not all applications are server-side applications. Spring provides a programming model that insulates application code from environment details such as JNDI, making code less dependent on its runtime context.
- **Spring aims to promote code reuse.** Spring helps to avoid the need to make some important hard decisions up front, like whether your application will ever use JTA or JNDI; Spring abstractions will allow you to deploy your code in a different environment if you ever need to. Thus

Spring enables you to *defer architectural choices*, potentially delivering benefits such as the need to purchase an application server license only when you know exactly what your platform requirements are, following tests of throughput and scalability.

- **Spring aims to facilitate Object Oriented design in J2EE applications.** You might be asking "How can a J2EE application, written in Java — an OO language — not be OO?" In reality, many J2EE applications do not deserve the name of OO applications. Spring aims to remove some of the impediments in place of OO in traditional designs. As one of the reviewers on this book commented, "The code I've seen from my team in the year since we adopted Spring has consistently been better factored, more coherent, loosely coupled and reusable."

- **Spring aims to facilitate good programming practice, such as programming to interfaces, rather than classes.** Use of an IoC container such as Spring greatly reduces the complexity of coding to interfaces, rather than classes, by elegantly concealing the specification of the desired implementation class and satisfying its configuration requirements. Callers using the object through its interface are shielded from this detail, which may change as the application evolves.

- **Spring promotes pluggability.** Spring encourages you to think of application objects as named services. Ideally, the dependencies between such services are expressed in terms of interfaces. Thus you can swap one service for another without impacting the rest of your application. The way in which each service is configured is concealed from the client view of that service.

- **Spring facilitates the extraction of configuration values from Java code into XML or properties files.** While some configuration values may be validly coded in Java, all nontrivial applications need some configuration externalized from Java source code, to allow its management without recompilation or Java coding skills. (For example, if there is a timeout property on a particular object, it should be possible to alter its value without being a Java programmer.) Spring encourages developers to externalize configuration that might otherwise have been inappropriately hard-coded in Java source code. More configurable code is typically more maintainable and reusable.

- **Spring is designed so that applications using it are as easy as possible to test.** As far as possible, application objects will be POJOs, and POJOs are easy to test; dependence on Spring APIs will normally be in the form of interfaces that are easy to stub or mock. Unlike the case of JNDI, for example, stubbing or mocking is easy; unlike the case of Struts, for example, application classes are seldom forced to extend framework classes that themselves have complex dependencies.

- **Spring is consistent.** Both in different runtime environments and different parts of the framework, Spring uses a consistent approach. Once you learn one part of the framework, you'll find that that knowledge can be leveraged in many others.

- **Spring promotes architectural choice.** While Spring provides an architectural backbone, Spring aims to facilitate replaceability of each layer. For example, with a Spring middle tier, you should be able to switch from one O/R mapping framework to another with minimal impact on business logic code, or switch from, say, Struts to Spring MVC or WebWork with no impact on the middle tier.

- **Spring does not reinvent the wheel.** Despite its broad scope, Spring does not introduce its own solution in areas such as O/R mapping where there are already good solutions. Similarly, it does not implement its own logging abstraction, connection pool, distributed transaction coordinator, remoting protocols, or other system services that are already well-served in other products or application servers. However, Spring does make these existing solutions significantly easier to use, and places them in a consistent architectural approach.

We'll examine these values later in this chapter and throughout this book. Many of these values are also followed by other lightweight frameworks. What makes Spring unique is that it provides such a consistent approach to delivering on them, and provides a wide enough range of services to be helpful throughout typical applications.

Spring in Context

Spring is a manifestation of a wider movement. Spring is the most successful product in what can broadly be termed *agile* J2EE.

Technologies

While Spring has been responsible for real innovation, many of the ideas it has popularized were part of the *zeitgeist* and would have become important even had there been no Spring project. Spring's greatest contribution — besides a solid, high quality, implementation — has been its combination of emerging ideas into a coherent whole, along with an overall architectural vision to facilitate effective use.

Inversion of Control and Dependency Injection

The technology that Spring is most identified with is *Inversion of Control*, and specifically the *Dependency Injection* flavor of Inversion of Control. We'll discuss this concept in detail in Chapter 2, "The Bean Factory and Application Context," but it's important to begin here with an overview. Spring is often thought of as an Inversion of Control container, although in reality it is much more.

Inversion of Control is best understood through the term the "Hollywood Principle," which basically means "Don't call me, I'll call you." Consider a traditional class library: application code is responsible for the overall flow of control, calling out to the class library as necessary. With the Hollywood Principle, framework code invokes application code, coordinating overall workflow, rather than application code invoking framework code.

Inversion of Control is often abbreviated as IoC in the remainder of this book.

IoC is a broad concept, and can encompass many things, including the EJB and Servlet model, and the way in which Spring uses callback interfaces to allow clean acquisition and release of resources such as JDBC Connections.

Spring's flavor of IoC for configuration management is rather more specific. Consequently, Martin Fowler, Rod Johnson, Aslak Hellesoy, and Paul Hammant coined the name *Dependency Injection* in late 2003 to describe the approach to IoC promoted by Spring, PicoContainer, and HiveMind — the three most popular lightweight frameworks.

Dependency Injection is based on Java language constructs, rather than the use of framework-specific interfaces. Instead of application code using framework APIs to resolve dependencies such as configuration parameters and collaborating objects, application classes expose their dependencies through methods or constructors that the framework can call with the appropriate values at runtime, based on configuration.

Dependency Injection is a form of *push* configuration; the container "pushes" dependencies into application objects at runtime. This is the opposite of traditional *pull* configuration, in which the application object "pulls" dependencies from its environment. Thus, Dependency Injection objects never load custom properties or go to a database to load configuration — the framework is wholly responsible for actually reading configuration.

Push configuration has a number of compelling advantages over traditional pull configuration. For example, it means that:

- **Application classes are self-documenting, and dependencies are explicit.** It's merely necessary to look at the constructors and other methods on a class to see its configuration requirements. There's no danger that the class will expect its own undocumented properties or other formats.
- For the same reason, **documentation of dependencies is always up-to-date**.
- **There's little or no lock-in to a particular framework, or proprietary code, for configuration management.** It's all done through the Java language itself.
- As the framework is wholly responsible for reading configuration, **it's possible to switch where configuration comes from without breaking application code**. For example, the same application classes could be configured from XML files, properties files, or a database without needing to be changed.
- As the framework is wholly responsible for reading configuration, there is usually **greater consistency in configuration management**. Gone are the days when each developer will approach configuration management differently.
- **Code in application classes focuses on the relevant business responsibilities.** There's no need to waste time writing configuration management code, and configuration management code won't obscure business logic. A key piece of application plumbing is kept out of the developer's way.

We find that developers who try Dependency Injection rapidly become hooked. These advantages are even more apparent in practice than they sound in theory.

Spring supports several types of Dependency Injection, making its support more comprehensive than that of any other product:

- **Setter Injection:** The injection of dependencies via JavaBean setter methods. Often, but not necessarily, each setter has a corresponding getter method, in which case the *property* is set to be *writable* as well as *readable.*
- **Constructor Injection:** The injection of dependencies via constructor arguments.
- **Method Injection:** A more rarely used form of Dependency Injection in which the container is responsible for implementing methods at runtime. For example, an object might define a protected abstract method, and the container might implement it at runtime to return an object resulting from a container lookup. The aim of Method Injection is, again, to avoid dependencies on the container API. See Chapter 2 for a discussion of the issues around this advanced form of Dependency Injection.

Uniquely, Spring allows all three to be mixed when configuring one class, if appropriate.

Enough theory: Let's look at a simple example of an object being configured using Dependency Injection.

We assume that there is an interface — in this case, `Service` — which callers will code against. In this case, the implementation will be called `ServiceImpl`. However, of course the name is hidden from callers, who don't know anything about how the `Service` implementation is constructed.

Let's assume that our implementation of `Service` has two dependencies: an int configuration property, setting a timeout; and a DAO that it uses to obtain persistent objects.

With Setter Injection we can configure `ServiceImpl` using JavaBean properties to satisfy these two dependencies, as follows:

```
public class ServiceImpl implements Service {
 private int timeout;
 private AccountDao accountDao;

   public void setTimeout(int timeout) {
     this.timeout = timeout;
   }

   public void setAccountDao(AccountDao accountDao) {
     this.accountDao = accountDao;
   }
```

With Constructor Injection, we supply both properties in the Constructor, as follows:

```
public class ServiceImpl implements Service {
 private int timeout;
 private AccountDao accountDao;

   public ServiceImpl (int timeout, AccountDao accountDao) {
       this.timeout = timeout;
       this.accountDao = accountDao;
   }
```

Either way, the dependencies are satisfied by the framework before any business methods on `ServiceImpl` are invoked. (For brevity, we haven't shown any business methods in the code fragments. Business methods will use the instance variables populated through Dependency Injection.)

This may seem trivial. You may be wondering how such a simple concept can be so important. While it is conceptually simple, it can scale to handle complex configuration requirements, populating whole object graphs as required. It's possible to build object graphs of arbitrary complexity using Dependency Injection. Spring also supports configuration of maps, lists, arrays, and properties, including arbitrary nesting.

As an IoC container takes responsibility for object instantiation, it can also support important creational patterns such as singletons, prototypes, and object pools. For example, a sophisticated IoC container such as Spring can allow a choice between "singleton" or shared objects (one per IoC container instance) and non-singleton or "prototype" instances (of which there can be any number of independent instances).

Because the container is responsible for satisfying dependencies, it can also introduce a layer of indirection as required to allow custom interception or hot swapping. (In the case of Spring, it can go a step farther and provide a true AOP capability, as we'll see in the next section.) Thus, for example, the container can satisfy a dependency with an object that is instrumented by the container, or which hides a "target object" that can be changed at runtime without affecting references. Unlike some IoC containers and complex configuration management APIs such as JMX, Spring does not introduce such indirection unless it's necessary. In accordance with its philosophy of allowing the simplest thing that can possibly work, unless you want such features, Spring will give you normal instances of your POJOs, wired together through normal property references. However, it provides powerful indirection capabilities if you want to take that next step.

Spring also supports *Dependency Lookup*: another form of Inversion of Control. This uses a more traditional approach, similar to that used in Avalon and EJB 1.*x* and 2.*x*, in which the container defines lifecycle callbacks, such as `setSessionContext()`, which application classes implement to look up dependencies. Dependency Lookup is essential in a minority of cases, but should be avoided where possible to minimize lock-in to the framework. Unlike EJB 2.*x* and Avalon, Spring lifecycle callbacks are optional; if you choose to implement them, the container will automatically invoke them, but in most cases you won't want to, and don't need to worry about them.

Spring also provides many hooks that allow power users to customize how the container works. As with the optional lifecycle callbacks, you won't often need to use this capability, but it's essential in some advanced cases, and is the product of experience using IoC in many demanding scenarios.

> **The key innovation in Dependency Injection is that it works with pure Java syntax: no dependence on container APIs is required.**
>
> **Dependency Injection is an amazingly simple concept, yet, with a good container, it's amazingly powerful. It can be used to manage arbitrarily fine-grained objects; it places few constraints on object design; and it can be combined with container services to provide a wide range of value adds.**
>
> **You don't need to do anything in particular to make an application class eligible for Dependency Injection — that's part of its elegance. In order to make classes "good citizens," you should avoid doing things that cut against the spirit of Dependency Injection, such as parsing custom properties files. But there are no hard and fast rules. Thus there is a huge potential to use legacy code in a container that supports Dependency Injection, and that's a big win.**

Aspect-Oriented Programming

Dependency Injection goes a long way towards delivering the ideal of a fully featured application framework enabling a POJO programming model. However, configuration management isn't the only issue; we also need to provide declarative *services* to objects. It's a great start to be able to configure our POJOs — even with a rich network of collaborators — without constraining their design; it's equally important to be able to apply services such as transaction management to POJOs without them needing to implement special APIs.

The ideal solution is Aspect-Oriented Programming (AOP). (AOP is also a solution for much more; besides, we are talking about a particular use of AOP here, rather than the be all and end all of AOP.)

AOP provides a different way of thinking about code structure, compared to OOP or procedural programming. Whereas in OOP we model real-world objects or concepts, such as bank accounts, as objects, and organize those objects in hierarchies, AOP enables us to think about *concerns* or *aspects* in our system. Typical concerns are transaction management, logging, or failure monitoring. For example, transaction management applies to operations on bank accounts, but also to many other things besides. Transaction management applies to sets of methods without much relationship to the object hierarchy. This can be hard to capture in traditional OOP. Typically we end up with a choice of tradeoffs:

- **Writing boilerplate code to apply the services to every method that requires them:** Like all cut-and-paste approaches, this is unmaintainable; if we need to modify how a service is delivered, we need to change multiple blocks of code, and OOP alone can't help us modularize that code. Furthermore, each additional concern will result in its own boilerplate code, threatening to obscure the business purpose of each method. We can use the **Decorator** design pattern to keep the new code distinct, but there will still be a lot of code duplication. In a minority of cases the **Observer** design pattern is sufficient, but it doesn't offer strong typing, and we must build in support for the pattern by making our objects observable.
- **Detype operations, through something like the Command pattern:** This enables a custom interceptor chain to apply behavior such as declarative transaction management, but at the loss of strong typing and readability.
- **Choosing a heavyweight dedicated framework such as EJB that can deliver the necessary services:** This works for some concerns such as transaction management, but fails if we need a custom service, or don't like the way in which the EJB specification approaches the particular concern. For example, we can't use EJB services if we have a web application that should ideally run in a web container, or in case of a standalone application with a Swing GUI. Such frameworks also place constraints on our code — we are no longer in the realm of POJOs.

In short, with a traditional OO approach the choices are code duplication, loss of strong typing, or an intrusive special-purpose framework.

AOP enables us to capture the cross-cutting code in modules such as interceptors that can be applied declaratively wherever the concern they express applies — *without imposing tradeoffs on the objects benefiting from the services.*

There are several popular AOP technologies and a variety of approaches to implementing AOP. Spring includes a *proxy-based* AOP framework. This does not require any special manipulation of class loaders and is portable between environments, including any application server. It runs on J2SE 1.3 and above, using J2SE *dynamic proxies* (capable of proxying any number of interfaces) or CGLIB byte code generation (which allows proxying classes, as well as interfaces). Spring AOP proxies maintain a chain of *advice* applying to each method, making it easy to apply services such as transaction management to POJOs. The additional behavior is applied by a chain of advice (usually interceptors) maintained by an *AOP proxy,* which wraps the POJO *target object*.

Spring AOP allows the proxying of interfaces or classes. It provides an extensible *pointcut* model, enabling identification of which sets of method to advise. It also supports *introduction*: advice that makes a class implement additional interfaces. Introduction can be very useful in certain circumstances (especially infrastructural code within the framework itself). Don't worry if AOP terms such as "pointcuts" and "introduction" are unfamiliar now; we'll provide a brief introduction to AOP in Chapter 4, which covers Spring's AOP framework.

Here, we're more interested in the value proposition that Spring AOP provides, and why it's key to the overall Spring vision.

Spring AOP is used in the framework itself for a variety of purposes, many of which are behind the scenes and which many users may not realize are the result of AOP:

- **Declarative transaction management:** This is the most important out-of-the-box service supplied with Spring. It's analogous to the value proposition of EJB container-managed transactions (CMT) with the following big advantages:
  - It can be applied to any POJO.
  - It isn't tied to JTA, but can work with a variety of underlying transaction APIs (including JTA). Thus it can work in a web container or standalone application using a single database, and doesn't require a full J2EE application server.
  - It supports additional semantics that minimize the need to depend on a proprietary transaction API to force rollback.
- **Hot swapping:** An AOP proxy can be used to provide a layer of indirection. (Remember our discussion of how indirection can provide a key benefit in implementing Dependency Injection?) For example, if an `OrderProcessor` depends on an `InventoryManager`, and the `InventoryManager` is set as a property of the `OrderProcessor`, it's possible to introduce a proxy to ensure that the `InventoryManager` instance can be changed without invalidating the `OrderProcessor` reference. This mechanism is threadsafe, providing powerful "hot swap" capabilities. Full-blown AOP isn't the only way to do this, but if a proxy is to be introduced at all, enabling the full power of Spring AOP makes sense.
- **"Dynamic object" support:** As with hot swapping, the use of an AOP proxy can enable "dynamic" objects such as objects sourced from scripts in a language such as Groovy or Beanshell to support reload (changing the underlying instance) and (using introduction) implement additional interfaces allowing state to be queried and manipulated (last refresh, forcible refresh, and so on).
- **Security:** Acegi Security for Spring is an associated framework that uses Spring AOP to deliver a sophisticated declarative security model.

There's also a compelling value proposition in using AOP in application code, and Spring provides a flexible, extensible framework for doing so. AOP is often used in applications to handle aspects such as:

- **Auditing:** Applying a consistent auditing policy — for example, to updates on persistent objects.
- **Exception handling:** Applying a consistent exception handling policy, such as emailing an administrator in the event of a particular set of exceptions being thrown by a business object.
- **Caching:** An aspect can maintain a cache transparent to proxied objects and their callers, providing a simple means of optimization.
- **Retrying:** Attempting transparent recovery: for example, in the event of a failed remote invocation.

See Chapter 4, "Spring and AOP," for an introduction to AOP concepts and the Spring AOP framework.

AOP seems set to be the future of middleware, with services (pre-built or application-specific) flexibly applied to POJOs. Unlike the monolithic EJB container, which provides a fixed set of services, AOP offers a much more modular approach. It offers the potential to combine best-of-breed services: for example, transaction management from one vendor, and security from another.

While the full AOP story still has to be played out, Spring makes a substantial part of this vision a reality today, with solid, proven technology that is in no way experimental.

Consistent Abstraction

If we return to the core Spring mission of providing declarative service to POJOs, we see that it's not sufficient to have an AOP framework. From a middleware perspective, an AOP framework is a way of delivering services; it's important to have services to deliver — for example, to back a transaction management aspect. And of course not all services can be delivered declaratively; for example, there's no way to avoid working with an API for data access.

The third side of the Spring triangle, after IoC and AOP, is a consistent service abstraction.

Motivation

At first glance the idea of Spring providing a consistent "service abstraction" may seem puzzling. Why is it better to depend on Spring APIs than, say, standard J2EE APIs such as JNDI, or APIs of popular, solid open source products such as Hibernate?

The Spring abstraction approach is compelling for many reasons:

- **Whether it's desirable to depend on a particular API depends more on the nature of that API than its provenance.** For example, if depending on a "standard" API results in code that is hard to unit test, you are better off with an abstraction over that API. A good example of this is JavaMail. JavaMail is a particularly difficult API to test against because of its use of static methods and final classes.
- **You may well end up building such an abstraction anyway; there's a real value in having it off the shelf, in a widely used product.** The reality is that Spring competes with in-house frameworks, rather than with J2EE itself. Most large applications will end up building helper objects and a level of abstraction over cumbersome APIs such as JNDI — as we noted early in this chapter, using J2EE out of the box has generally not proved to be a workable option. Spring provides a well-thought-through abstraction that is widely applicable and requires you to write no custom code.
- **Dependency on Spring is limited to a set of interfaces; the dependencies are simple and explicit, avoiding implicit dependencies, which can be problematic.** For example, if you write code that performs programmatic transaction management using JTA, you have a whole range of implicit dependencies. You need to obtain the JTA `UserTransaction` object using JNDI; you

thus need both a JNDI and JTA environment. This may not be available in all environments, or at test time. In some important cases, such as declarative transaction management, *no* dependencies on Spring APIs are required to use Spring's abstraction.

- **Spring's abstraction interfaces work in a wide range of environments.** If you write code that uses JTA, you are tied to an environment providing JNDI and JTA. For example, if you try to run that code in a web container such as Tomcat, you would otherwise need to plug in a third-party JTA implementation. Spring's transaction infrastructure can help you avoid this need in the majority of cases, where you are working with a single database.
- **Spring's abstraction can decouple you from heavy dependence on third-party APIs that may be subject to change, or in case you switch product.** For example, if you aren't 100 percent sure whether you want to use Hibernate or TopLink persistence, you can use Spring's data access abstraction to make any migration a relatively straightforward process. Alternatively, if you need to migrate from Hibernate 2 to Hibernate 3, you'll find that easier if you use Spring.
- **Spring APIs are written for use by application developers, rather than merely for usage behind the scenes.** This relates to one of the key points made earlier in this chapter about the value proposition of frameworks in general. Both JTA and JDBC are good counter-examples. JTA was essentially intended to work behind the scenes to enable EJB declarative transaction management. Thus, exception handling using JTA is particularly cumbersome, with multiple exceptions having no common base class and needing individual catch blocks. JDBC is a relatively successful API at providing a consistent view of any relational database, but is extremely verbose and problematic to use directly in application code because of the complex error-handling required, and because of the need to write non-portable code to try to pinpoint the cause of a particular failure and work with advanced scenarios such as working with BLOBs and calling stored procedures.

In keeping with Spring's philosophy of not reinventing the wheel, Spring does not provide its own abstraction over services unless there are proven difficulties, such as those we've just seen, relating to use of that API.

Of course it's important to ensure that the abstraction does not sacrifice power. In many cases, Spring allows you to leverage the full power of the underlying service to express operations, even using the native API. However, Spring will take care of plumbing and resource handling for you.

Exceptions

A consistent exception hierarchy is essential to the provision of a workable service abstraction. Spring provides such an exception hierarchy in several cases, and this is one of Spring's unique features. The most important concerns data access. Spring's `org.springframework.dao.DataAccessException` and its subclasses provide a rich exception model for handling data access exceptions. Again, the emphasis is on the application programming model; unlike the case of `SQLException` and many other data access APIs, Spring's exception hierarchy is designed to allow developers to write the minimum, cleanest code to handle errors.

`DataAccessException` and other Spring infrastructure exceptions are unchecked. One of the Spring principles is that infrastructure exceptions should normally be unchecked. The reasoning behind this is that:

- **Infrastructure exceptions are not usually recoverable.** While it's possible to catch unchecked exceptions when one *is* recoverable, there is no benefit in being *forced* to catch or throw exceptions in the vast majority of cases where no recovery is possible.

- **Checked exceptions lessen the value of an exception hierarchy.** If Spring's `DataAccessException` were checked, for example, it would be necessary to write a `catch (DataAccessException)` block every time a subclass, such as `IntegrityViolationException`, was caught, or for other, unrecoverable `DataAccessExceptions` to be handled in application code up the call stack. This cuts against the benefit of compiler enforcement, as the only useful catch block, for the subclass that can actually be handled, is not enforced by the compiler.
- **Try/catch blocks that don't add value are verbose and obfuscate code.** It is not lazy to want to avoid pointless code; it obfuscates intention and will pose higher maintenance costs forever. Avoiding overuse of checked exceptions is consistent with the overall Spring goal of reducing the amount of code that doesn't do anything in typical applications.

Using checked exceptions for infrastructural failures sounds good in theory, but practice shows differently. For example, if you analyze real applications using JDBC or entity beans (both of which APIs use checked exceptions heavily), you will find a majority of catch blocks that merely wrap the exception (often losing the stack trace), rather than adding any value. Thus not only the catch block is often redundant, there are also often many redundant exception classes.

To confirm Spring's choice of unchecked infrastructure exceptions, compare the practice of leading persistence frameworks: JDO and TopLink have always used unchecked exceptions; Hibernate 3 will switch from checked to unchecked exceptions.

Of course it's essential that a framework throwing unchecked exceptions must document those exceptions. Spring's well-defined, well-documented hierarchies are invaluable here; for example, any code using Spring data access functionality may throw `DataAccessException`, but no unexpected unchecked exceptions (unless there is a lower-level problem such as `OutOfMemoryError`, which could still occur if Spring itself used checked exceptions).

> **As with Dependency Injection, the notion of a simple, portable service abstraction — accompanied with a rich exception model — is conceptually so simple (although not simple to deliver) that it's surprising no one had done it before.**

Resource Management

Spring's services abstraction provides much of its value through the way in which it handles resource management. Nearly any low-level API, whether for data access, resource lookup, or service access, demands care in acquiring and releasing resources, placing an onus on the application developer and leaving the potential for common errors.

The nature of resource management differs depending on API and the problems it brings:

- JDBC requires Connections, Statements, and ResultSets to be obtained and released, even in the event of errors.
- Hibernate requires Sessions to be obtained and released, taking into account any current transaction; JDO and TopLink have similar requirements, for PersistenceManagers and Sessions, respectively.

- Correct usage of JNDI requires Contexts to be acquired and closed.
- JTA has both its own requirements and a requirement to use JNDI.

Spring applies a consistent approach, whatever the API. While in some cases JCA can solve some of the problems (such as binding to the current thread), it is complex to configure, requires a full-blown application server, and is not available for all resources. Spring provides a much more lightweight approach, equally at home inside or outside an application server.

Spring handles resource management *programmatically*, using callback interfaces, or *declaratively*, using its AOP framework.

Spring calls the classes using a callback approach based on *templates*. This is another form of Inversion of Control; the application developer can work with native API constructs such as JDBC Connections and Statements without needing to handle API-specific exceptions (which will automatically be translated into Spring's exception hierarchy) or close resources (which will automatically be released by the framework).

Spring provides templates and other helper classes for all supported APIs, including JDBC, Hibernate, JNDI, and JTA.

Compare the following code using Spring's JDBC abstraction to raw JDBC code, which would require a try/catch/finally block to close resources correctly:

```
Collection requests = jdbc.query("SELECT NAME, DATE, ...  +
                FROM REQUEST WHERE SOMETHING = 1",
         new RowMapper() {
 public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
         Request r = new Request();
         r.setName(rs.getString("NAME"));
         r.setDate(rs.getDate("DATE"));
         return r;
 }
});
```

If you've worked with raw JDBC in a production environment, you will know that the *correct* raw JDBC alternative is not a pretty sight. Correct resource handling in the event of failures is particularly problematic, requiring a nested try/catch block in the finally block of an overall try/catch/finally block to ensure that the connection and other resources are *always* closed, and there is no potential for connection leaks, which are a severe and common problem in typical JDBC code.

When using Spring JDBC, the developer doesn't need to handle `SQLException`, although she can choose to catch Spring's more informative `DataAccessException` or subclasses. Even in the event of an `SQLException` being thrown, the `Connection` (which is obtained by the framework) and all other resources will still be closed. Nearly every line of code here *does something*, whereas with raw JDBC most code would be concerned with plumbing, such as correct release of resources.

Many common operations can be expressed without the need for callbacks, like SQL aggregate functions or the following query using Spring's `HibernateTemplate`:

```
List l = hibernateTemplate.find("from User u where u.name = ?",
new Object[] { name });
```

Spring will automatically obtain a Hibernate `Session`, taking into account any active transaction, in which case a `Session` will be bound to the current thread.

This approach is used consistently within the framework for multiple APIs, such as JDBC, JTA, JNDI, Hibernate, JDO, and TopLink. In all cases, exception translation, as well as resource access, is handled by the framework.

Techniques

As important as the technologies are the techniques that they enable. As we've noted, Spring is partly a manifestation of the movement toward agile processes, and it solves many of the problems that agile practitioners otherwise encounter when working with J2EE.

For example, Test Driven Development (TDD) is one of the key lessons of XP (Extreme Programming)—although of course, the value of rigorous unit testing has long been recognized (if too rarely practiced).

Unit testing is key to success, and a framework must facilitate it. Spring does—as do other lightweight containers such as PicoContainer and HiveMind; recognition of the central importance of unit testing is not unique to Spring, nor is Spring the only product to rise to the challenge.

Unfortunately, J2EE out of the box is not particularly friendly to unit testing. Far too high a proportion of code in a traditional J2EE application depends on the application server, meaning that it can be tested only when deployed to a container, or by stubbing a container at test time.

Both of these approaches have proven problematic. The simple fact is that in-container testing is too slow and too much of an obstacle to productivity to apply successfully in large projects. While tools such as Cactus exist to support it, and some IDEs provide test time "containers," our experience is that it is rare to see an example of a well unit-tested "traditional" J2EE architecture.

Spring's approach—non-invasive configuration management, the provision of services to POJOs, and consistent abstractions over hard-to-stub APIs—makes unit testing outside a container (such as in simple JUnit tests) easy. So easy that TDD is a joy to practice, and an agile process is greatly facilitated.

> **Interestingly, while Spring plays so well with agile approaches, it can also work very well with supposedly "heavyweight" methodologies. We've seen a Spring-based architecture work far better with the Unified Software Development Process than a traditional J2EE architecture because it demands fewer compromises for implementation strategy. (Traditional J2EE approaches often have a problematic gulf between Analysis and Design Models because of the workarounds of "J2EE design patterns.")**

Spring even provides support for integration testing using a Spring context, but out of a container, in the `org.springframework.test` *package. This is not an alternative to unit tests (which should not normally require Spring at all), but can be very useful as the next phase of testing. For example, the test superclasses in this package provide the ability to set up a transaction for each test method and automatically tear it down, doing away with the necessity for database setup and teardown scripts that might otherwise slow things down significantly.*

Relationship to Other Frameworks

As we've noted, Spring does not reinvent the wheel. Spring aims to provide the glue that enables you to build a coherent and manageable application architecture out of disparate components.

Let's summarize how Spring relates to some of the many products it integrates with.

Persistence Frameworks

Spring does not provide its own O/R mapping framework. It provides an abstraction over JDBC, but this is a less painful way of doing exactly the same thing as might otherwise have been done with JDBC.

Spring provides a consistent architectural model, but allows you to choose the O/R mapping framework of your choice (or an SQL-based approach where appropriate). For the many applications that benefit from using O/R mapping, you should integrate an O/R mapping framework with Spring. Spring integrates well with all leading O/R mapping frameworks. Supported choices include:

- **Hibernate:** The leading open source O/R mapping tool. Hibernate was the first O/R mapping tool for which Spring offered integration. Spring's Hibernate integration makes Hibernate significantly easier to use, through the `HibernateTemplate` we've briefly discussed and through integration with Spring's transaction management.
- **JDO implementations:** Spring provides support for JDO 1.0 and JDO 2.0 standards. Several JDO vendors also ship their own Spring integration, implementing Spring's `JdoDialect` interface, giving application developers access to common capabilities that go beyond the JDO specification without locking into a particular JDO vendor.
- **TopLink:** TopLink is the oldest O/R mapping tool on the market, dating back to the mid-1990s. TopLink is now an Oracle product, and Oracle has written a Spring integration that enables Spring users to work as seamlessly with TopLink as with Hibernate or a JDO implementation.
- **Apache OJB:** An O/R mapping product from Apache.
- **iBATIS:** iBATIS SQL Maps is not, strictly speaking, an O/R mapping product. However, it does offer a convenient way of defining SQL statements in a declarative fashion, mapping objects to statement parameters and result sets to objects. In contrast to full-blown O/R mapping solutions, though, SQL Maps does not aim to provide an object query language or automatic change detection.

All these integrations are consistent in that Spring facilitates the use of DAO interfaces, and all operations throw informative subclasses of Spring's `DataAccessException`. Spring provides helpers such as templates for all these APIs, enabling a consistent programming style. Spring's comprehensive architectural template means that almost any persistence framework can be integrated within this consistent approach. Integration efforts from several JDO vendors — the fact that the Spring/TopLink integration was developed by the TopLink team at Oracle, and that the popular Cayenne open source O/R mapping project itself developed Spring integration for Cayenne — shows that Spring's data access abstraction is becoming something of a de facto standard for consistent approach to persistence in Java/J2EE applications.

Spring's own JDBC framework is suitable when you want SQL-based access to relational data. This is not an alternative to O/R mapping, but it's necessary to implement at least some scenarios in most applications using a relational database. (Also, O/R mapping is not universally applicable, despite the claims of its more enthusiastic advocates.)

Importantly, Spring allows you to mix and match data access strategies — for example Hibernate code and JDBC code sharing the same connection and transaction. This is an important bonus for complex applications, which typically can't perform all persistence operations using a single persistence framework.

Web Frameworks

Again, the fundamental philosophy is to enable users to choose the web framework of their choice, while enjoying the full benefit of a Spring middle tier. Popular choices include:

- **Struts:** Still the dominant MVC web framework (although in decline). Many Spring users use Struts with a Spring middle tier. Integration is fairly close, and it is even possible to configure Struts Actions using Spring Dependency Injection, giving them instant access to middle-tier objects without any Java coding.
- **WebWork:** Integration with WebWork is particularly close because of WebWork's flexible design and the strong interest in the WebWork community in using WebWork along with a Spring middle tier. It is possible to configure WebWork Actions using Spring Dependency Injection.
- **Spring MVC:** Spring's own MVC web framework, which of course integrates particularly well with a Spring middle tier.
- **Tapestry:** A component-oriented framework from Apache's Jakarta group, Tapestry integrates well with Spring because declarative page metadata makes it easy for Tapestry pages to access Spring-provided services without any code-level coupling to Spring.
- **JSF:** Spring integrates very well with JSF, given that JSF has the concept of "named beans" and does not aim to implement middle-tier services itself.

Spring's approach to web frameworks differs from that to persistence, in that Spring provides its own fully featured web framework. Of course, you are not forced to use this if you wish to use a Spring middle tier. While Spring integrates well with other web frameworks, there are a few integration advantages available only with Spring's own MVC framework, such as the ability to use some advanced features of Spring Dependency Injection, or to apply AOP advice to web controllers. Nevertheless, as Spring integrates well with other web frameworks, you should choose Spring's own MVC framework on its merits, rather than because there's any element of compulsion. Spring MVC is an appealing alternative to Struts and other request-driven frameworks as it is highly flexible, helps to ensure that application code in the web tier is easily testable, and works particularly well with a wide variety of view technologies besides JSP. In Chapter 12 we discuss Spring MVC in detail.

AOP Frameworks

Spring provides a proxy-based AOP framework, which is well suited for solving most problems in J2EE applications.

However, sometimes you need capabilities that a proxy-based framework cannot provide, such as the ability to advise objects created using the `new` operator and not managed by any factory; or the ability to advise fields, as well as methods.

To support such requirements, Spring integrates well with AspectJ and ApectWerkz, the two leading class weaving–based AOP frameworks. It's possible to combine Spring Dependency Injection with such AOP frameworks — for example, configuring AspectJ aspects using the full power of the Spring IoC container as if they were ordinary classes.

As of early 2005, the AspectJ and AspectWerkz projects are merging into AspectJ 5.0, which looks set to be the definitive full-blown AOP framework. The Spring project is working closely with the AspectJ project to ensure the best possible integration, which should have significant benefits for both communities.

Spring does *not* attempt to replicate the power of a full-blown AOP solution such as AspectJ; this would produce no benefits to Spring users, who are instead free to mix Spring AOP with AspectJ as necessary to implement sophisticated AOP solutions.

Other Frameworks

Spring integrates with other frameworks including the Quartz scheduler, Jasper Reports, and Velocity and FreeMarker template engines.

Again, the goal is to provide a consistent backbone for application architecture.

All such integrations are modules, distinct from the Spring core. While some ship with the main Spring distribution, some are external modules. Spring's open architecture has also resulted in numerous other projects (such as the Cayenne O/R mapping framework) providing their own Spring integration, or providing Spring integration with third-party products.

Architecting Applications with Spring

Let's now look at how a typical application using Spring is organized, and the kind of architecture Spring promotes and facilitates.

Spring is designed to facilitate architectural flexibility. For example, if you wish to switch from Hibernate to JDO or vice versa (or from either to the forthcoming JSR-220 POJO persistence API), using Spring and following recommended practice can make that easier. Similarly, you can defer the choice as to whether to use EJB for a certain piece of functionality, confident that you won't need to modify existing code if a particular service ends up implemented by an EJB rather than a POJO.

Let's now look at how a typical Spring architecture looks in practice, from top to bottom.

The architecture described here is referred to as the "Lightweight container architecture" in Chapter 3 of J2EE without EJB *(Johnson/Hoeller, Wrox, 2004). As such an architecture is based on OO best practice, rather than Spring, it can be implemented without Spring. However, Spring is ideally suited to making such architectures succeed.*

The Big Picture

Figure 1-1 illustrates the architectural layers in a typical Spring application. Although this describes a web application, the concepts apply to most logically tiered applications in general.

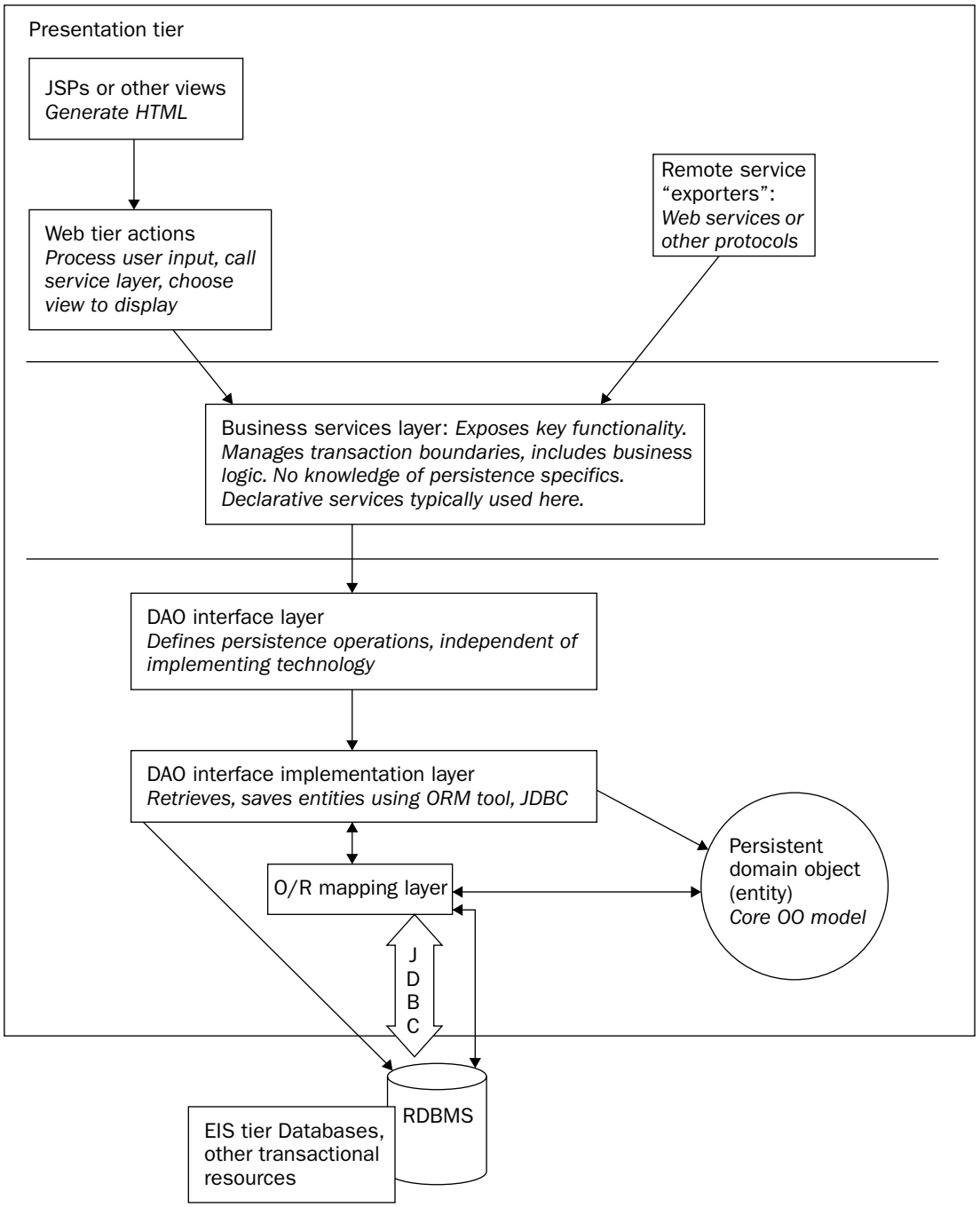
Presentation tier
JSPs or other views
Generate HTML
Web tier actions
Process user input, call service layer, choose view to display
Remote service "exporters":
Web services or other protocols
Business services layer: Exposes key functionality. Manages transaction boundaries, includes business logic. No knowledge of persistence specifics. Declarative services typically used here.
DAO interface layer
Defines persistence operations, independent of implementing technology
DAO interface implementation layer
Retrieves, saves entities using ORM tool, JDBC
O/R mapping layer
Persistent domain object (entity)
Core OO model
JDBC
RDBMS
EIS tier Databases, other transactional resources

Figure 1-1

Let's summarize each layer and its responsibilities, beginning closest to the database or other enterprise resources:

- **Presentation layer:** This is most likely to be a web tier. This layer should be as thin as possible. It should be possible to have alternative presentation layers — such as a web tier or remote web services facade — on a single, well-designed middle tier.
- **Business services layer:** This is responsible for transactional boundaries and providing an entry point for operations on the system as a whole. This layer should have no knowledge of presentation concerns, and should be reusable.
- **DAO interface layer:** This is a layer of interfaces *independent of any data access technology* that is used to find and persist persistent objects. This layer effectively consists of *Strategy* interfaces for the Business services layer. This layer should not contain business logic. Implementations of these interfaces will normally use an O/R mapping technology or Spring's JDBC abstraction.
- **Persistent domain objects:** These model real objects or concepts such as a bank account.
- **Databases and legacy systems:** By far the most common case is a single RDBMS. However, there may be multiple databases, or a mix of databases and other transactional or non-transactional legacy systems or other enterprise resources. The same fundamental architecture is applicable in either case. This is often referred to as the *EIS* (*Enterprise Information System*) tier.

In a J2EE application, all layers except the EIS tier will run in the application server or web container. Domain objects will typically be passed up to the presentation layer, which will display data they contain, *but not modify them*, which will occur only within the transactional boundaries defined by the business services layer. Thus there is no need for distinct Transfer Objects, as used in traditional J2EE architecture.

In the following sections we'll discuss each of these layers in turn, beginning closest to the database.

> **Spring aims to decouple architectural layers, so that each layer can be modified as far as possible without impacting other layers. No layer is aware of the concerns of the layer above; as far as possible, dependency is purely on the layer immediately below. Dependency between layers is normally in the form of interfaces, ensuring that coupling is as loose as possible.**

Persistence and Integration

Getting data access right is crucial for success, and Spring provides rich services in this area.

Database

Most J2EE (or Java applications) work with a relational database. Ultimately data access will be accomplished using JDBC. However, most Spring users find that they do *not* use the JDBC API directly.

Spring encourages a decoupling of business objects from persistence technology using a layer of interfaces.

Data Access Objects

In keeping with the philosophy that it is better to program to interfaces than classes, Spring encourages the use of *data access interfaces* between the service layer and whatever persistence API the application (or that part of it) uses. However, as the term Data Access Object (DAO) is widely used, we continue to use it.

DAOs encapsulate access to persistent domain objects, and provide the ability to persist transient objects and update existing objects.

By using a distinct layer of DAO objects, and accessing them through interfaces, we ensure that service objects are decoupled from the persistence API. This has many benefits. Not only is it possible to switch between persistence tools more easily, but it makes the code more coherent through separation of concerns and greatly simplifies testing (a particularly important concern when using an agile process). Imagine a business service method that processes a set of orders: If the set of orders is obtained through a method on a DAO interface, it is trivial to supply a test DAO in a JUnit test that supplies an empty set or a particular set of orders, or throws an exception — all without going near a database.

DAO implementations will be made available to objects using them using Dependency Injection, with both service objects and DAO instances configured using the Spring IoC container.

DAO interfaces will typically contain the following kinds of methods:

- **Finder methods:** These locate persistent objects for use by the business services layer.
- **Persist or save methods:** These make transient objects persistent.
- **Delete methods:** These remove the representation of objects from the persistent store.
- **Count or other aggregate function methods:** These return the results of operations that are more efficient to implement using database functionality than by iterating over Java objects.

Less commonly, DAO interfaces may contain bulk update methods.

> *While Spring encourages the use of the term "DAO" for such interfaces, they could equally well be called "repositories," as they are similar to the Repository pattern described in Eric Evans'* Domain-Driven Design *(Addison-Wesley, 2003).*

The following interface shows some of the methods on a typical DAO interface:

```
public interface ReminderDao {

  public Collection
      findRequestsEligibleForReminder()  throws DataAccessException;

  void persist(Reminder reminder) throws DataAccessException;

  void countOutstandingRequests() throws DataAccessException;

}
```

Note that all methods can throw Spring's `DataAccessException` or subclasses, thus wholly decoupling callers from the persistence technology in use. The DAO tier interfaces have become truly persistence technology agnostic.

This interface might be implemented using Hibernate, with Spring's convenience classes, as follows (some methods omitted for brevity):

```
public class HibernateReminderDao extends HibernateDaoSupport implements
ReminderDao {

  public Collection findRequestsEligibleForReminder()
                          throws DataAccessException {
          getHibernateTemplate().find("from Request r where
           r.something = 1");
  }

  public void persist(Reminder reminder)
                         throws DataAccessException {
          getHibernateTemplate().saveOrUpdate(reminder);
  }
}
```

An implementation using another O/R mapping tool, such as TopLink, would be conceptually almost identical, although naturally using a different underlying API (but also benefiting from Spring conveniences). Following the DAO pattern as facilitated by Spring and illustrated in Spring sample applications such as the Spring PetStore ensures a consistent architectural approach, whatever the chosen persistence API.

Our DAO interface could be implemented using Spring's JDBC abstraction as follows:

```
public class JdbcReminderDao extends JdbcDaoSupport
                                      implements ReminderDao {

  public Collection findRequestsEligibleForReminder()
                               throws DataAccessException {
         return getJdbcTemplate().query("SELECT NAME, DATE, ... " +
                               " FROM REQUEST WHERE SOMETHING = 1",
                new RowMapper() {
                public Object mapRow(ResultSet rs, int rowNum) throws SQLException
{
                        Request r = new Request();
                        r.setName(rs.getString("NAME"));
                        r.setDate(rs.getDate("DATE"));
                        return r;
                }
         });
  }


  public int countRequestsEligibleForReminder()
                                    throws DataAccessException {
         return getJdbcTemplate.queryForInt("SELECT COUNT(*) FROM ...");
  }
}
```

Don't worry if you don't understand the details at this point: we'll discuss Spring's persistence services in detail in Chapter 5, "DAO Support and JDBC Framework." However, you should be able to see how Spring eliminates boilerplate code, meaning that you need to write only code that actually *does something*.

Although they are typically used with relational databases and O/R mapping frameworks or JDBC code underneath, Spring DAO interfaces are not specific to RDBMS access. Spring's DAO exception hierarchy is completely independent of any persistence technology, so users have contributed support for LDAP and other technologies, and some users have implemented connectivity to products such as Documentum, within the consistent Spring data access architectural approach.

Persistent Domain Objects

Persistent domain objects are objects that model your domain, such as a bank account object. They are persisted, most often using an O/R mapping layer.

Note that the desire to isolate service objects from persistence APIs does *not* mean that persistent objects should be dumb bit buckets. Persistent *objects* should be true objects — they should contain behavior, as well as state, and they should encapsulate their state appropriately. Do not automatically use JavaBean accessor methods, encouraging callers to view persistent objects as pure storage. Likewise, avoid the temptation (as in traditional J2EE architectures using stateless session and entity beans) to move code for navigation of persistent object graphs into the business service layer, where it is more verbose and represents a loss of encapsulation.

Persistent domain models often represent the core intellectual property of an application, and the most valuable product of business analysis. To preserve this investment, they should ideally be independent of the means used to persist them.

Spring allows the application of Dependency Injection to persistent objects, using classes such as `DependencyInjectionInterceptorFactoryBean`, which enables Hibernate objects to be automatically wired with dependencies from a Spring IoC container. This makes it easier to support cases when domain objects might need, for example, to use a DAO interface. It allows this to be done simply by expressing a dependency via a setter method, rather than relying on explicit "pull" configuration calls to Spring or implementing persistence framework–specific lifecycle methods. (Another, more general, approach to this end is to use a class-weaving AOP solution such as AspectJ.) However, we advise careful consideration before allowing domain objects to access the service layer. Experience shows that it is not usually necessary in typical applications, and it may represent a blurring of the clean architectural layering we recommend.

Persistence Technology

Domain objects are typically persisted with an O/R mapping product such as Hibernate or a JDO implementation.

In certain cases, full-blown O/R mapping may be inappropriate. In this case, the DAO interfaces remain appropriate, but the implementation is likely to use JDBC — through Spring's JDBC abstraction layer, or through iBATIS SQL Maps.

In some applications — and some use cases in nearly any complex application — Spring's JDBC abstraction may be used to perform SQL-based persistence.

Business Service Objects

Depending on your chosen architecture, much of your business logic may reside in persistent domain objects.

However, there is nearly always an important role for a *service layer*. This is analogous to the layer of stateless session beans in traditional J2EE architectures. However, in a Spring-based application, it will consist of POJOs with few dependencies on Spring. The Service Layer provides functionality such as:

- **Business logic that is use case–specific:** While it is often appropriate for domain objects to contain business logic applicable to many use cases, specific use cases are often realized in the business services layer.
- **Clearly defined entry points for business operations:** The business service objects provide the interfaces used by the presentation layer.
- **Transaction management:** In general, although business logic can be moved into persistent domain objects, transaction management should not be.
- **Enforcement of security constraints:** Again, this is usually best done at the entry level to the middle tier, rather than in domain objects.

As with the DAO interface layer, the business service layer should expose interfaces, not classes.

> **The service layer also represents a valuable investment that — with good analysis — should have a long lifetime. Thus it should also be as independent as possible of the technologies that enable it to function as part of a working application. Spring makes an important contribution here through its non-invasive programming model.**
>
> **Coupling between architectural layers should be in the form of interfaces, to promote loose coupling.**

Presentation

In the recommended architecture using Spring, the presentation tier rests on a well-defined service layer.

This means that the presentation layer will be thin; it will not contain business logic, but merely presentation specifics, such as code to handle web interactions.

It also means that there can be a choice of presentation tiers — or more than one presentation layer — in an application, without impacting lower architectural layers.

> **Lower architectural layers should have no knowledge of presentation tier concerns.**

Web Tier

In a Spring application, a web tier will use a web application framework, whether Spring's own MVC framework, or an alternative framework such as Struts, WebWork, Tapestry, or JSF.

The web tier will be responsible for dealing with user interactions, and obtaining data that can be displayed in the required format.

The web tier will normally consist of three kinds of objects:

- **Controller:** Controller objects, like Spring MVC Controllers and Struts Actions, are responsible for processing user input as presented in HTTP requests, invoking the necessary business service layer functionality, and returning the required model for display.
- **Model:** Objects containing data resulting from the execution of business logic and which must be displayed in the response.
- **View:** Objects responsible for rendering the model to the response. The mechanism required will differ between different *view technologies* such as JSP, Velocity templates, or conceptually different approaches such as PDF or Excel spreadsheet generation. Views are not responsible for updating data or even *obtaining* data; they merely serve to display Model data that has been provided by the relevant Controller.

This triad is often referred to as an *MVC* (*Model View Controller*) architectural pattern, although such "web MVC" is not the same as the "classic" MVC pattern used in thick client frameworks such as Swing.

Spring's own MVC framework provides a particularly clean implementation of these three elements — the base interfaces have the names `Controller`, `Model`, and `View`. In particular, Spring MVC ensures that not only is the web tier distinct from the business services layer, but web views are completely decoupled from controllers and models. Thus a different type of view can be added without changing controller or model code.

Remote Access Facade

Spring reflects the philosophy that remote access — for example, supporting remote clients over RMI or remote clients over web services — should be viewed as an alternative presentation layer on well-defined middle tier service interfaces.

> **Unlike in traditional J2EE practice, remoting concerns should not cascade throughout an application in the form of Transfer Objects and other baggage.**
>
> **Applications should be internally Object Oriented. As remoting tends to be destructive to true Object Orientation, it should be treated as one view of an application.**

Spring provides a variety of remoting choices, based on exposing POJOs. This is less invasive than traditional J2EE EJB remoting, but it's important to remember that there are inevitable constraints around remoting, which no technology can wholly conceal. There is no such thing as a distributed object. Remoting introduces a range of concerns: not merely the need for serializability, but the depth to which object graphs should be traversed before disconnection, constraints on method granularity ("chatty" calling is ruled out for efficiency reasons), and many other issues.

If you wish to use a traditional EJB-based remoting architecture, you can use one or more Spring contexts inside the EJB container. In such an architecture, EJBs become a facade layer. (They may also be used for transaction management, although Spring's own transaction management is more flexible, and a better choice in Spring-based applications, even when using JTA.)

In such an architecture, we recommend that Transfer Objects and the other paraphernalia of remoting are added *on top of* the basic architecture, rather than used throughout it.

Spring RCP

Another important presentational choice is the Spring Rich Client Project (Spring RCP). This is a framework built around the Spring IoC container that facilitates the building of Swing applications. The use of Spring on the client side is especially compelling when it is necessary to access enterprise services, which are typically remote, as Spring's client-side, as well as server-side, remoting features may be used.

The Future

Spring is a mature, production-ready solution. However, the Spring team has demonstrated a rapid pace of innovation and is committed to further enhancements. While you can be confident of Spring offering backward compatibility, as it has done to this point, you can also expect many enhancements and new capabilities.

Release Schedule

This book covers Spring 1.2, released in early 2005. Spring 1.0 was released in March 2004, Spring 1.1 in September 2004. These two major releases have been close to 100 percent backward compatible, despite adding a significant number of features. Spring continues to evolve and innovate at a rapid rate.

Spring operates on a 4–6 week release schedule. Normally a point release contains bug fixes and minor enhancements. A major release, such as Spring 1.1, contains significant new features (in that case, some IoC container enhancements and the first phase of JMS support). Major releases are normally around 6 months apart.

> **A key proof of the value of a non-invasive framework is how Spring's implementation can evolve significantly, and its capabilities increase greatly, while preserving backward compatibility. This is emphatically different from the experience with APIs such as EJB, where every major version has resulted in a substantial migration exercise, or the maintenance of two separate code bases. As we noted earlier, Spring *is* different.**
>
> **This is also an important benefit for third-party products that integrate with Spring—and, of course, users of those products. For example, Acegi Security System, an external project, is able to provide complex value-added functionality spanning multiple layers, without needing any hooks into Spring.**

In general, you should work with the latest production release of Spring. Spring's comprehensive test coverage and excellent record on backward compatibility help make this a workable strategy.

The Evolution of Java and J2EE

J2EE itself is also evolving, so it's natural to ask how Spring will fit into the J2EE ecosystem as J2EE moves forward.

The major changes relevant to Spring development concern the EJB 3.0 specification, currently under development by the JSR-220 Expert Group, and expected to reach final release in early 2006.

EJB 3.0 introduces two major changes relevant to Spring users and Spring's positioning with respect to J2EE:

- **It introduces a simplification of the session bean programming model**, which rests partly on the use of Dependency Injection to provide simpler access to the bean's JNDI environment.
- **It introduces a new specification for POJO persistence.** Strictly speaking, this is separate from the EJB specification itself, as JSR-220 will in fact produce two deliverables: the EJB 3.0 specification and a POJO persistence specification. The JSR-220 persistence specification is likely to be supported by all major O/R mapping products, including TopLink, Hibernate, and leading JDO products. It also effectively relegates the EJB 2.*x* entity bean model to legacy status, merely formalizing the reality seen among usage trends since 2002.

The introduction of Dependency Injection features in EJB 3.0 is an interesting move from a specification group, indicating that the "lightweight" approach to J2EE popularized by Spring and other projects has become too influential to ignore. (Indeed, the influence of Spring on the EJB 3.0 session bean model is unmistakable.) However, EJB 3.0 offers a very limited form of Dependency Injection, which has different goals, and does not approach the capabilities of Spring or other leading IoC containers. For example, it can inject dependencies only on objects that come from JNDI; it cannot manage objects that are too fine-grained to be placed in JNDI; it cannot easily manage configuration properties such as ints and Strings, which are very important to externalizing configuration from code; and it cannot manage lists, maps, or other complex types that the Spring IoC container can handle. At the time of this writing, it provides no standard way to apply interception or full-fledged AOP capability to injected collaborators, meaning that it misses much of the value-adding possibilities open to a sophisticated IoC container. Finally, it seems capable only of per-*class* Dependency Injection configuration (due to reliance on Java 5.0 annotations for the recommended configuration style), rather than the per-*instance* configuration supported by Spring and other sophisticated IoC containers, which is often necessary in large applications.

After the hype dies down and more technical details emerge, it's unlikely that EJB 3.0 session beans will change things much for Spring users. If you want to use Dependency Injection, you will almost certainly end up using Spring behind an EJB 3.0 facade. The Spring team will ensure that this is easy to do.

If you want to keep your options open regarding an eventual migration to EJB 3.0, a Spring-based approach will give you a production-proven basis for you to implement now, while enabling a far easier migration path than the EJB 2.*x* model, which is effectively deprecated in EJB 3.0. (While you could continue to run EJB 2.*x* beans in an EJB 3.0 container, this isn't attractive in the long run, amounting to a substantial legacy model.) However, it's hard to see any benefit for Spring users in moving from Spring-managed business objects to EJB 3.0 session beans — this would sacrifice many capabilities, reduce the range of environments in which the relevant code could execute, and add little or no functionality.

> *The core benefit of EJB 3.0 Dependency Injection — the ability to inject objects obtained from JNDI such as* `DataSources`*, has been offered by Spring, in the form of the* `JndiObjectFactoryBean`*, since Spring's inception.*

JSR-220 persistence has more significant implications than the EJB 3.0 session bean model (which is essentially playing catch-up with features long offered by lightweight frameworks). JSR-220 persistence adds significantly to the already compelling case for Spring's unique data access abstraction. As we've seen, there is real value in being able to implement a DAO interface using, say, Hibernate or JDO, or JDBC (if it doesn't imply transparent persistence). Today, there is a range of competing O/R mapping products. With the release of the JSR-220 specification for POJO persistence ahead, this value proposition becomes still more compelling. We *know* that persistence is likely to change. Thus it is logical to try to abstract away from the details of your chosen persistence API, and isolate dependence on it behind a layer of interfaces.

Spring's sophisticated persistence infrastructure, and its DAO interface approach, is a perfect candidate to do this — indeed, the *only* candidate at present, the alternative being in-house coding.

Finally, Java is itself evolving. With the release of J2SE 5.0 in September 2004, Java has seen arguably the most significant enhancements in its history. While backward compatibility is essential *internally* for a widely used framework such as Spring, Spring 1.2 offers additional functionality for J2SE 5.0 users: for example, leveraging J2SE 5.0 annotations where appropriate (in particular, for declarative transaction management).

Technology Currents

While the J2EE specifications continue to evolve, they are not evolving fast enough. The Java specification process faces twin dangers: premature standardization, before a technology is well enough understood to choose the correct path; and glacial progress, where standards lag far behind common practice. (For example, although EJB 3.0 is a major overhaul of the session bean model, it will offer *in 2006* a subset of Dependency Injection capabilities available in a production release in Spring and other products *in early 2004*.)

As new paradigms such as AOP and IoC gain further momentum, and experience around their use accumulates, an agile framework such as Spring is far better placed than a specification committee to offer a solid implementation to its user community.

Spring is also well placed to support other emerging areas, such as:

- **Scripting:** Java is not the be all and end all of the JVM. Scripting languages — especially Groovy — are enjoying growing popularity. We expect this trend to continue. Spring can provide the same comprehensive IoC and AOP services to objects written in any scripting language as to Java objects, within a consistent programming model. The language in which an object is written — like the particular implementation of an interface — can be concealed from callers. This raises the possibility of writing objects in the most appropriate language. In some cases, language features such as closures offer a much more succinct option than Java.
- **A cross-platform programming model:** The Spring programming model is now available for .NET, with the release of Spring.NET. This is particularly interesting for organizations that have an investment in both platforms, and architects and developers who need to work on both platforms.

Standards and Open Source

Of course this touches on another challenge to J2EE orthodoxy, and the belief that Sun "standards" are always the best way forward for the industry.

Chapter 4 of *J2EE without EJB* discusses this issue in more detail. There is a strong argument that while standards have produced real benefit in the case of low level services, such as JNDI and JTA, they have largely failed where (like EJB) they have entered the domain of the application programming model. Specification committees are not best placed to deal with issues in the application programming domain; responsive open source projects with a capable development community and thousands of users are.

The consequences of failed standards, like entity beans, are far worse than the consequences of using good, non-standard, solutions. It's more important to have a working application, delivered on time and budget, than an application that uses only largely unproven standards.

Currently there doesn't seem much likelihood that the JCP will produce anything in the *application programming space* that equals the value of what Spring and other lightweight frameworks provide.

The emergence of non-invasive frameworks such as Spring has also cut against the old assumptions regarding standardization. Framework lock-in *can* be minimized. Where application code depends largely on standard Java constructs (as in the case of Setter and Constructor Injection), there isn't much that needs to be standardized.

Furthermore, it's significant that Spring and other leading lightweight containers are open source. There's an argument that open source reduces the danger of proprietary solutions, as does a non-invasive framework. There's no risk of a greedy vendor cornering the market and fleecing users; the source code is available in any event; and in the case of successful products such as Spring there are many people who understand not just how to use the framework but how it works internally.

Partly for this reason, it seems that infrastructure is inexorably moving toward open source. It's becoming increasingly difficult for commercial products to compete in this space. Spring is a key part of the brave new world of J2EE infrastructure, and application developers are the main beneficiaries.

The Spring Project and Community

Spring is perhaps the only successful software project to have evolved from a book: *Expert One-on-One J2EE Design and Development*, by Rod Johnson (Wiley, 2002). This unusual heritage has proved beneficial, providing Spring with a consistent vision from its outset. From the outset, the Spring team agreed on coding and design conventions, and the overall architecture approach that Spring should enable.

History

Spring initially grew out of Rod Johnson's experience as a consultant on various large Java/J2EE applications from 1997 through 2002. Like most experienced consultants, Rod had written a number of frameworks for various clients. The last, for a prominent global media group, began as a web MVC framework (well before Struts was available), but expanded to include what would now be called a Dependency Injection container (although the name did not exist until late 2003) and data access services.

Thus *Expert One-on-One J2EE Design and Development* included not only a discussion of the problems Rod had encountered doing extensive J2EE development, but 30,000 lines of code, in the form of the "Interface21 framework" demonstrating a practical approach to solving many of them.

The reaction to this code from readers was extremely positive, many finding that it solved problems that they'd seen over and over in their own J2EE experience. Over the next few months, numerous readers sought clarification of the terms of the license so that they could use the code in projects and — far more important — volunteered ideas and practical help to take the framework to the next level.

The most important contribution was made by Juergen Hoeller, who has been co-lead with Rod since the open source framework was founded in February 2003. Juergen immediately began to make a huge contribution to implementation and design. Other developers including co-authors Thomas Risberg, Colin Sampaleanu, and Alef Arendsen also joined early and began to make large contributions in specific areas.

Interestingly, the ideas that Spring is commonly identified with — most notably, Dependency Injection — were also developed independently by other projects. Although it was publicly announced before Spring went public, the PicoContainer project actually began several months after the Spring project.

The ATG Dynamo application server included a Dependency Injection capability in its proprietary functionality, although none of the Spring team was aware of this until Spring was already widely adopted. Proof that an idea so simple, yet powerful, cannot be invented — merely discovered.

The authors of this book include most of the core developers of Spring. Rod and Juergen remain the architects and chief developers of Spring.

The Spring community has also made a huge contribution to the evolution of Spring, through reporting problems based on experience in a wide range of environments, and suggesting many improvements and enhancements. This is a key element in the value proposition of open source infrastructure — indeed, any off-the-shelf infrastructure, as opposed to in-house frameworks. Not only can a large community of talented individuals contribute ideas to the framework itself, but a large body of understanding is available to organizations building applications using Spring.

From its outset, the Spring project has been based around respecting and listening to Spring users. Of course, as any project grows in size, it becomes impossible for any individual to participate in all discussions — even to listen attentively to the details of every discussion. But the Spring developers as a group have been attentive and responsive, and this has helped to build a flourishing community.

Spring is licensed under the Apache 2.0 license — a widely used open source license that is not restrictive.

In August 2004, Rod, Juergen, Colin, and other core Spring developers (including Spring RCP lead, Keith Donald) founded Interface21, a company devoted to Spring consulting, training, and support. This helps to secure the future of Spring, by placing a viable economic model behind it. A sophisticated and wide-ranging framework such as Spring represents a huge amount of effort; it's impossible to continue that effort without a viable economic model. Spring framework itself will always remain open source and free to anybody who wishes to use it. However, the existence of a commercial entity also provides a degree of confidence to more conservative organizations, with the availability of professional services to support the development of Spring-based applications.

Even before the release of Spring 1.0 final in March 2004, Spring had been widely adopted. Today, Spring is one of the most widely used and discussed products in J2EE development, and Spring skills are widely available and sought after on the job market.

Module Summary

Let's now examine the functionality that Spring offers in more detail. It is divided into a number of separate modules.

There are two main categories of functionality in Spring:

- An IoC container and AOP framework, which handle configuration and application of services to objects.
- A set of services that can be applied to application objects. These services can be used as a class library, even in a different runtime container.

Each of the modules discussed here fits into one or both of these categories.

Spring is a layered framework with the following core modules:

- **IoC container:** The core of Spring is the IoC container. This configures objects by Dependency Injection, but also supports a variety of optional callbacks for application objects that have more complex requirements of their environment.

- **Proxy-based AOP framework:** Spring's AOP framework is the other essential in delivering a non-invasive framework that can manage POJOs. Spring AOP can be used programmatically, or integrated with the IoC container, in which case any object managed by a Spring IoC container can be "advised" transparently.
- **Data access abstraction:** This includes:
  - The DAO exception hierarchy and consistent architectural approach discussed earlier in this chapter.
  - Template and other integration classes for Hibernate, JDO, TopLink, iBATIS, and other data access APIs.
  - Spring's own JDBC abstraction framework. While in general Spring's approach to persistence is consistent integration, Spring does provide one persistence API of its own: its JDBC framework. This is an abstraction layer over JDBC that simplifies error handling, improves code portability, and eliminates the potential for many common errors when using JDBC, such as failure to close connections and other valuable resources in the event of errors. Spring's JDBC abstraction is suitable when you want SQL-based persistence, and not O/R mapping: for example, when you have a legacy database that doesn't lend itself to O/R mapping, or you need to make extensive use of stored procedures or BLOB types. Spring's JDBC framework is also a good choice if the persistence needs are limited and you don't want to introduce the learning curve/maintenance cost of an additional persistence framework such as JDO or Hibernate.
- **Transaction abstraction:** Spring provides a transaction abstraction that can run over a variety of underlying transaction APIs, including JTA, JDBC, Hibernate, and JDO transactions. Naturally, JTA is the only option for working with multiple transactional resources, as Spring does not provide its own distributed transaction manager. However, many applications work with a single database, and do not require distributed transactions for any other reason (such as working with JMS). However, Spring's transaction abstraction provides a consistent programming model in any environment, from high-end J2EE application servers down to simple web containers and standalone clients: a unique value proposition that does away with the traditional need to choose between "global" and "local" transaction management and commit to one or another programming model early in each J2EE project. Spring provides programmatic declarative management that is much more usable than JTA, which is a fairly cumbersome API, not ideally suited for use by application developers. But most users prefer Spring's sophisticated declarative transaction management, which can provide transaction management for any POJO. Spring transaction management is integrated with all of Spring's supported data access integrations.
- **Simplification for working with JNDI, EJB, and other complex J2EE APIs and services:** If you're an experienced J2EE developer, you will have written many JNDI lookups. You may have chosen to move the lookup code to a helper class, but you can't completely conceal the complexity, or hide the implications for testing. If you've worked with EJB, you've needed to write code to look the EJB home up in JNDI before calling the `create()` method to obtain an EJB reference you can use to do work. Chances are, you've done essentially the same thing, over and over, and worked on applications where many developers have done essentially the same thing over and over, but in various different ways. These are routine tasks that are much better done by a framework. Spring's support for JNDI and EJB can eliminate the need to write boilerplate code for JNDI lookups and EJB access or implementation. By eliminating JNDI and EJB API dependencies in application code, it also increases the potential for reuse. For example, you don't need to write code that depends on an EJB home interface or handles EJB exceptions; you merely need to express a dependency on the relevant EJB's *Business Methods Interface*, which is a

plain Java interface. Spring can do the necessary JNDI lookup and create an AOP proxy for the EJB that hides the need to work with the EJB APIs. Your code is no longer dependent on EJB — you could choose to implement the Business Methods Interface without using EJB — and another bit of tedious boilerplate is gone.

- **MVC web framework:** Spring's own request-driven MVC framework. This is closely integrated with Spring's middle-tier functionality, with all controllers being configured by Dependency Injection, providing the ability to access middle-tier functionality from the web tier without any code. Spring MVC is similar in how it approaches controller lifecycle to Struts, but provides some key benefits over struts, such as:
  - Better support for view technologies other than JSP, such as Velocity and PDF generation libraries
  - Interceptors, in addition to "actions" or "controllers"
  - Ability to use domain objects to back forms, rather than special objects such as Struts `ActionForm` subclasses
  - Interface-based design, making the framework highly customizable, and avoiding the need to subclass framework classes, which is a convenience rather than a necessity
- **Integration with numerous third-party products:** This fits into Spring's role as architectural glue. As the Spring IoC container is extensible, it's also easy to "alias" additional services into Spring IoC. The `JndiObjectFactoryBean`, which looks up a named object in JNDI, is a good example.
- **Remoting:** Spring provides lightweight remoting support over a variety of protocols, including web services, RMI, RMI over HTTP, IIOP, and Caucho's Hessian and Burlap protocols. Remoting is available for POJOs, in keeping with Spring's emphasis on providing a rich environment for POJOs.
- **Simplification for working with EJBs:** This consists of:
  - Support for *implementing* EJBs: EJB 2.1 and earlier have no built-in configuration management. EJB 3.0 looks set to offer a simplistic Dependency Injection capability. In either case, there is a compelling value proposition in using a lightweight container behind an EJB facade. Spring provides support for implementing EJBs that serve as a facade in front of POJOs managed by a Spring application context.
  - Support for *accessing* EJBs, via the "codeless proxying" described earlier.
- **Message publication using JMS:** Spring's callback template approach is ideally suited to minimizing the complexity of application code required to publish JMS messages.
- **JMX support:** Spring 1.2 provides a powerful JMX layer that can publish any Spring-managed object to JMX, enabling monitoring and management with standard tools.

When you use Spring's AOP framework, a layer of dynamic capabilities is available over the Spring IoC container in Spring 1.3:

- Support for scripting languages, with full support both ways for Dependency Injection. This means that any application object can be written in any supported scripting language, depending on other objects written in Java, or vice-versa.
- Support for objects backed by a database.

Quite a list! This broad scope sometimes leads people to the misconception that Spring is a big fat blob, which aims to do everything from scratch. It's important to emphasize the following points:

- Spring's modular architecture means that you can use any part of Spring in isolation. For example, if you want to use just the IoC container to simplify your configuration management, you can do so, without the need for any other part of Spring. If you want to use Spring AOP, you can do so without the IoC container. You might choose to use Spring's transaction interceptor in another AOP framework, or the Spring JDBC framework as a class library.
- Yet there is a consistency in the design of the different Spring modules that is very helpful if you choose to use more of the Spring stack, as you can leverage the same concepts in different areas, making your application more internally consistent and making the whole framework easier to learn than you might expect, given its scope.
- Spring is substantially about integrating best-of-breed solutions.

Although the Spring download is quite large, this results mainly from its integration with other products. To run the core IoC container, you need only the Jakarta Commons Logging JAR in addition to the Spring binary. Some advanced features of both IoC container and AOP framework require CGLIB, which is also used by Hibernate and many other products. Beyond that, Spring's dependencies depend on what third-party product you integrate it with. For example, if you use Spring and Hibernate, you need a number of JARs that are required by Hibernate, besides Hibernate's own JAR.

Spring provides a number of JARs itself, offering separate services:

- **spring.jar (1.4MB):** Contains all core Spring classes, including all classes from the following JARs except spring-mock.jar.
- **spring-core.jar (265K):** The core IoC container and utility classes used elsewhere in the framework.
- **spring-context.jar (276K):** More advanced IoC features, JNDI and EJB support, validation, scheduling, and remoting support.
- **spring-aop.jar (144K):** Spring's AOP framework.
- **spring-dao.jar (241K):** Spring's data access exception hierarchy, JDBC abstraction, and transaction abstraction, including JTA support.
- **spring-orm.jar (190K):** Integration with Hibernate, JDO, iBATIS, and Apache OJB.
- **spring-web.jar (114K):** `ApplicationContext` implementations for use in web applications (with any web application framework), servlet filters, integration with Struts and Tiles, web data binding utilities, and multipart support.
- **spring-webmvc.jar (190K):** Spring's own Web MVC framework.
- **spring-mock.jar (40K):** Mock object implementations and superclasses for JUnit testing objects in a Spring container. Includes test superclasses that can begin and roll back a transaction around each test. These superclasses enable JUnit test cases themselves to be configured by Dependency Injection. (Objects to be tested are injected.)

Most users find that using spring.jar is the simplest choice. However, as the file sizes indicate, the core IoC container comes in at little over 200K.

Supported Environments

Spring requires J2SE 1.3 or above. Spring uses J2SE 1.4 optimizations where this will produce significant benefit, but falls back to using J2SE 1.3 APIs if necessary. For example, there are a number of optimizations available to the AOP framework under J2SE 1.4, such as the use of the new `IdentityHashMap` and `StackTraceElement`, which enable Spring's AOP framework to run faster on J2SE 1.4 and above. But it still retains full functionality on J2SE 1.3.

Certain additional features are supported only in Java 5.0. For example, Spring supports Java 5.0 annotations to drive declarative transaction management. However, the implementation of the core framework will remain J2SE 1.4 compatible for the foreseeable future. Our experience is that large enterprise users, in particular, often use older versions of application servers, and tend to be slow in upgrading. (At the time of writing, one of the major users of Spring in London, a global investment bank, is still using J2SE 1.3, with an old version of WebSphere.)

The Spring core does not depend on J2EE APIs. The IoC container and AOP frameworks can be used outside the J2EE environment. Even the transaction support can be used with a full J2EE environment (with JTA).

Summary

Spring is the most popular and comprehensive of the "lightweight" J2EE frameworks that have surged in popularity since 2003.

Spring enables you to streamline your J2EE development efforts, and bring a welcome consistency to application architecture and implementation. Spring will help you deal with the complexity of J2EE APIs, but is also equally at home outside a J2EE environment.

Spring is based on supporting an application programming model where code is written in POJOs.

Spring achieves this through use of two recently proven paradigms, both of which it has played an important role in popularizing:

- **Dependency Injection:** An approach to configuration management in which an Inversion of Control container such as that provided by Spring configures application objects via JavaBean properties or constructor arguments. This removes the need for application code to implement framework interfaces, and allows the framework to add value in many areas — for example, supporting hot swapping of objects without affecting other objects that hold references to them.
- **AOP:** AOP provides a way of providing services to POJOs without them needing to be aware of framework APIs, or be unduly constrained by framework requirements.

Spring supports its Dependency Injection and AOP capabilities by providing a consistent abstraction in a number of important areas, including:

- Data access
- Transaction management
- Naming and lookup
- Remoting

We saw how Spring is designed to promote architectural good practice. A typical Spring architecture will be based on programming to interfaces rather than classes, and will facilitate choice between best of breed solutions in areas such as O/R mapping and web framework, within a consistent architecture. It ensures that architectural layers are coupled through well-defined interfaces.

Spring is also designed to facilitate best practice in development process. In particular, applications using Spring are normally easy to unit test, without the need to deploy code to an application server to conduct all tests.

There is some confusion about whether Spring competes with J2EE. It does not. J2EE is most successful as a set of specifications for low-level services; Spring is essentially an enabler of an application programming model.

Spring does compete with many services provided by *EJB*, but it's important to avoid the mistake of confusing EJB with J2EE. EJB is a component model that enters the application programming domain; J2EE is a *platform* that deals more with consistent provision of low-level services. Spring in fact allows you to use J2EE more effectively. It allows you to dispense with EJB, and the complexity it entails, in many cases, while providing support for implementing and accessing EJBs in case you do choose to use them.

In reality, Spring competes far more with in-house frameworks. As we saw at the beginning of this chapter, trying to develop applications with J2EE out of the box is cumbersome and unwise. For this reason, many companies develop their own proprietary frameworks to simplify the programming model and conceal the complexity of J2EE APIs. Spring provides a far better option, with a powerful, consistent approach that has been tested by tens of thousands of users in thousands of applications.

In the remainder of this book, we'll look in detail at how Spring realizes the vision we've described in this chapter.