Pro Java™ EE Spring Patterns: Best Practices and Design Strategies Implementing Java™ EE Patterns with the Spring Framework

ISBN-13 (pbk): 978-1-4302-1009-2

ISBN-13 (electronic): 978-1-4302-1010-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Steve Anglin, Tom Welsh
Technical Reviewer: Prosenjit Bhattacharyya
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editor: Kim Wimpsett
Associate Production Director: Kari Brooks-Copony
Production Editors: Laura Cheu, Liz Berry
Compositor: Dina Quan
Proofreader: Linda Seifert
Indexer: Ron Strauss
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 1

Introducing Enterprise Java Application Architecture and Design

For a long time, Java Enterprise Edition (Java EE) has been the platform of choice across industries (banking, insurance, retail, hospitality, travel, and telecom, to name a few) for developing and deploying enterprise business applications. This is because Java EE provides a standard-based platform to build robust and highly scalable distributed applications that support everything from core banking operations to airline booking engines. However, developing successful Java EE applications can be a difficult task. The rich set of choices provided by the Java EE platform is daunting at first. The plethora of frameworks, utility libraries, integrated development environments (IDEs), and tool options make it all the more challenging. Hence, selecting appropriate technology is critical when developing Java EE–based software. These choices, backed by sound architectural and design principles, go a long way in building applications that are easy to maintain, reuse, and extend.

This chapter takes a tour of the fundamental aspects of Java EE application architecture and design. They form the foundation on which the entire application is developed.

The journey starts with a review of the evolution of distributed computing and n-tier application architecture. I will then show how the Java EE platform architecture addresses the difficulties in developing distributed applications. You will also learn about the Model-View-Controller (MVC) architectural principle. I'll then combine MVC principles with the Java EE platform to derive multitier Java EE application architecture.

With application architecture in place, I will focus on Java EE application design based on object-oriented principles. I will also explain the use of design patterns to simplify application design and the adoption of best practices. I'll also touch on the Java EE design pattern catalog as documented by Sun's Java BluePrints and subsequently elaborated on in the book *Core J2EE Design Pattern* by Deepak Alur et al (Prentice Hall, 2003). I'll end the chapter with an introduction to Unified Modeling Language (UML) and its role in visually documenting Java EE design and architecture.

Evolution of Distributed Computing

In distributed computing, an application is divided into smaller parts that run simultaneously on different computers. This is also referred to as *network computing* because the smaller parts communicate over the network generally using protocols built on top of TCP/IP or UDP. The smaller application parts are called *tiers*. Each tier provides an independent set of services that can be consumed by the connecting or client tier. The tiers can be further divided into *layers*, which provide granular-level functions. Most applications have three distinct layers:

- The *presentation layer* is responsible for the user interfaces.
- The *business layer* executes the business rules. In the process, it also interacts with the data access layer.
- The *data access layer* is responsible retrieving and manipulating data stored in enterprise information systems (EISs).

The modern state of network computing can be better understood by analyzing the gradual transition of distributed application architecture. In the next few sections, I will examine the transition of distributed architecture with suitable examples.

Single-Tier Architecture

The single-tier architecture dates back to the days of monolithic mainframes connected by dumb terminals. The entire application comprising layers such as user interfaces, business rules, and data was collocated on the same physical host. The users interacted with these systems using terminals or consoles, which had very limited text-based processing capabilities (see Figure 1-1).

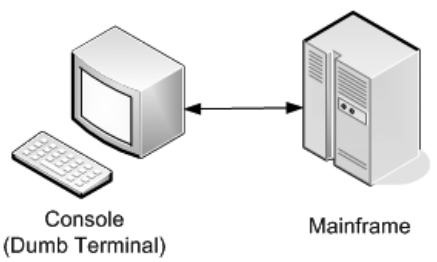

Figure 1-1. *Single-tier architecture*

Two-Tier Architecture

In the early 1980s, personal computers (PCs) became very popular. They were less expensive and had more processing power than the dumb terminal counterparts. This paved the way for true distributed, or *client-server*, computing. The client or the PCs now ran the user interface programs. It also supported graphical user interfaces (GUIs), allowing the users to enter data and interact with the mainframe server. The mainframe server now hosted only the business rules and data. Once the data entry was complete, the GUI application could optionally perform validations and then send the data to the server for execution of the business logic. Oracle Forms–based applications are a good example of two-tier architecture. The forms provide the GUI loaded on the PCs, and the business logic (coded as stored procedures) and data remain on the Oracle database server.

Then there was another form of two-tier architecture in which not only the UI but even the business logic resided on the client tier. This kind of application typically connected to a database server to run various queries. These clients are referred to as *thick* or *fat* clients because they had a significant portion of the executable code in the client tier (see Figure 1-2).

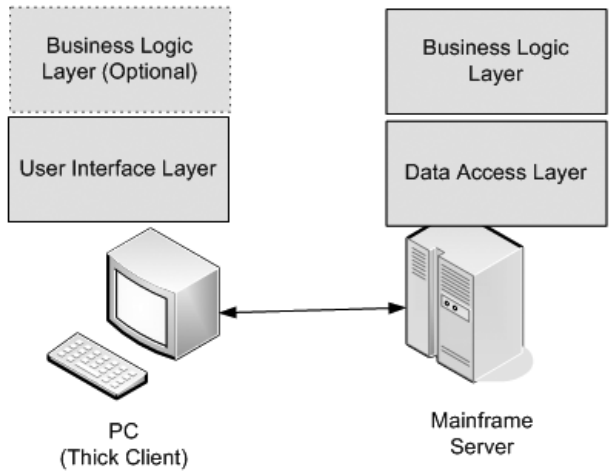

Figure 1-2. *Two-tier architecture*

Three-Tier Architecture

Two-tier thick client applications are easy to develop, but any software upgrade because of changes in user interface or business logic has to be rolled out for all the clients. Luckily, the hardware cost became cheaper and processing power increased significantly on the CPU in the mid-90s. This, coupled with the growth of the Internet and web-based application development trends, resulted in the emergence of three-tier architectures.

In this model, the client PC needs only thin client software such as a browser to display the presentation content coming from the server. The server hosts the presentation, the business logic, and the data access logic. The application data comes from enterprise information systems such as a relational database. In such systems the business logic can be accessed remotely, and hence it is possible to support stand-alone clients via a Java console application. The business layer generally interacts with the information system through the data access layer. Since the entire application resides on the server, this server is also referred to as an *application server* or *middleware* (see Figure 1-3).

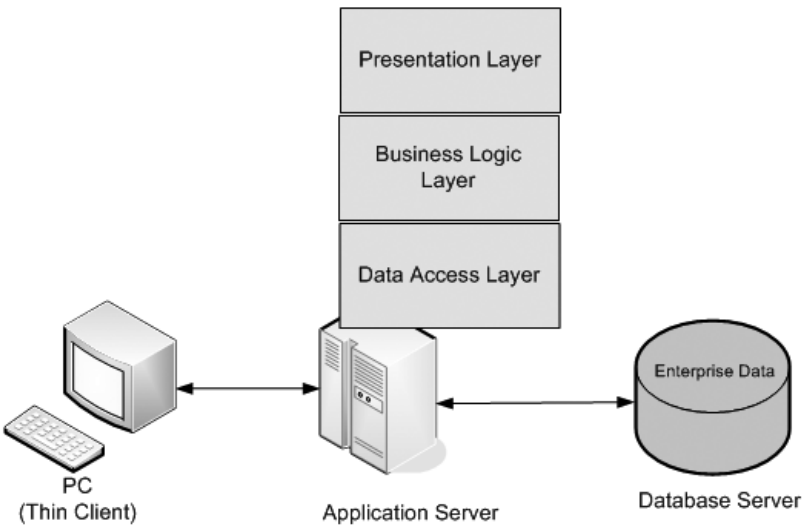

Figure 1-3. *Three-tier application*

N-Tier Architecture

With the widespread growth of Internet bandwidth, enterprises around the world have web-enabled their services. As a result, the application servers are not burdened anymore with the task of the presentation layer. This task is now off-loaded to the specialized web servers that generate presentation content. This content is transferred to the browser on

the client tier, which takes care of rendering the user interfaces. The application servers in n-tier architecture host remotely accessible business components. These are accessed by the presentation layer web server over the network using native protocols. Figure 1-4 shows the n-tier application.

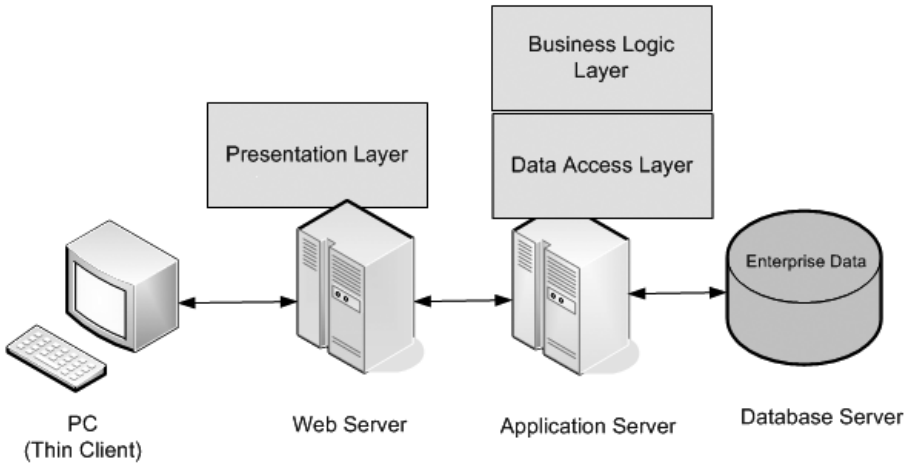

Figure 1-4. *N-tier application*

Java EE Architecture

Developing n-tier distributed applications is a complex and challenging job. Distributing the processing into separate tiers leads to better resource utilization. It also allows allocation of tasks to experts who are best suited to work and develop a particular tier. The web page designers, for example, are more equipped to work with the presentation layer on the web server. The database developers, on the other hand, can concentrate on developing stored procedures and functions. However, keeping these tiers as isolated silos serves no useful purpose. They must be integrated to achieve a bigger enterprise goal. It is imperative that this is done leveraging the most efficient protocol; otherwise, this leads to serious performance degradation.

Besides integration, a distributed application requires various services. It must be able to create, participate, or manage transactions while interacting with disparate information systems. This is an absolute must to ensure the concurrency of enterprise data. Since n-tier applications are accessed over the Internet, it is imperative that they are backed by strong security services to prevent malicious access.

These days, the cost of hardware, like CPU and memory, has gone down drastically. But still there is a limit, for example, to the amount of memory that is supported by the processor. Hence, there is a need to optimally use the system resources. Modern distributed applications are generally built leveraging object-oriented technologies. Therefore, services such as object caches or pools are very handy. These applications frequently interact with relational databases and other information systems such as message-oriented middleware. However, opening connections to these systems is costly because it consumes a lot of process resources and can prove to be a serious deterrent to performance. In these scenarios, a connection pool is immensely useful to improve performance as well as to optimize resource utilization.

Distributed applications typically use middleware servers to leverage the system services such as transaction, security, and pooling. The middleware server API had to be used to access these services. Hence, application code would be muddled with a proprietary API. This lock-in to vendor API wastes lot of development time and makes maintenance extremely difficult, besides limiting portability.

In 1999, Sun Microsystems released the Java EE 2 platform to address the difficulties in the development of distributed multitier enterprise applications. The platform was based on Java Platform, Standard Edition 2, and as a result it had the benefit of "write once, deploy and run anywhere." The platform received tremendous support from the open source community and major commercial vendors such as IBM, Oracle, BEA, and others because it was based on specifications. Anyone could develop the services as long as it conformed to the contract laid down in the specification. The specification and the platform have moved on from there; the platform is currently based on Java Platform, Standard Edition 5, and it is called Java Platform, Enterprise Edition 5. In this book, we will concentrate on this latest version, referred to officially as Java EE 5.

Java EE Container Architecture

The Java EE platform provides the essential system services through a container-based architecture. The container provides the runtime environment for the object-oriented application components written in Java. It provides low-level services such as security, transaction, life-cycle management, object lookup and caching, persistence, and network communication. This allows for the clear separation of roles. The system programmers can take care of developing the low-level services, and the application programmers can focus more on developing the business and presentation logic.

As shown in Figure 1-5, there are two server-side containers:

- The *web container* hosts the presentation components such as Java Server Pages (JSP) and servlets. These components also interact with the EJB container using remoting protocols.
- The *EJB container* manages the execution of Enterprise JavaBeans (EJB) components.

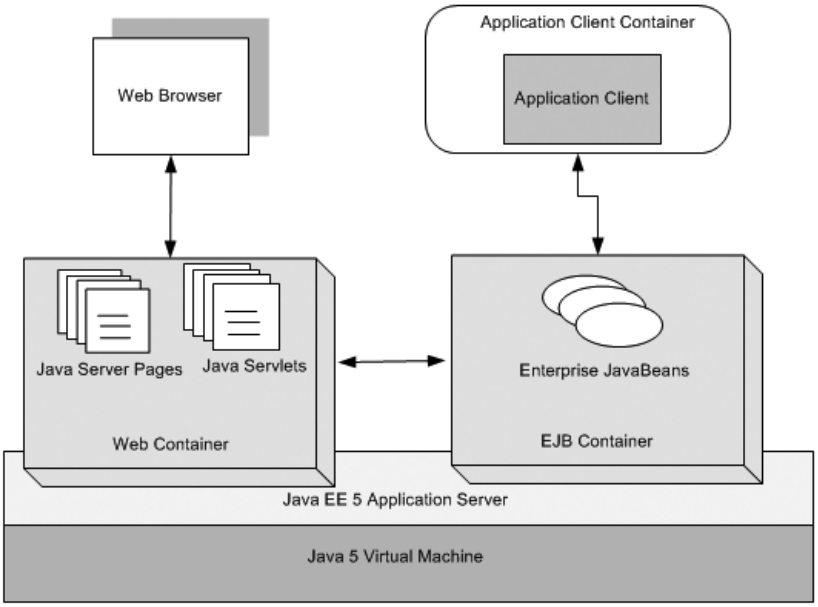

Figure 1-5. *Java EE platform architecture*

On the client side, the application client is a core Java application that connects to the EJB container over the network. The web browser, on the other hand, generally interacts with the web container using the HTTP protocol. The EJB and web containers together form the Java EE application server. The server in turn is hosted on the Java Virtual Machine (JVM).

Different containers provide different sets of low-level services. The web container does not provide transactional support, but the EJB container does. These services can be accessed using standard Java EE APIs such as Java Transaction API (JTA), Java Message Service (JMS), Java Naming and Directory Interface (JNDI), Java Persistence API (JPA), and Java Transaction API (JTA). The greatest benefit, however, is that these services can be applied transparently on the application components by mere configuration. To interpose these services, the application components should be packaged in predefined archive files with specific XML-based deployment descriptors. This effectively helps cut down on development time and simplifies maintenance.

Java EE Application Architecture

The Java EE platform makes the development of distributed n-tier applications easier. The application components can be easily divided based on functions and hosted on different tiers. The components on different tiers generally collaborate using an established architectural principle called MVC.

An MVC Detour

Trygve Reenskaug first described MVC way back in 1979 in a paper called "Applications Programming in Smalltalk-80™: How to use Model-View-Controller." It was primarily devised as a strategy for separating user interface logic from business logic. However, keeping the two isolated does not serve any useful purpose. It also suggests adding a layer of indirection to join and mediate between presentation and business logic layers. This new layer is called the *controller layer*. Thus, in short, MVC divides an application into three distinct but collaborating components:

- The *model* manages the data of the application by applying business rules.
- The *view* is responsible for displaying the application data and presenting the control that allows the users to further interact with the system.
- The *controller* takes care of the mediation between the model and the view.

Figure 1-6 depicts the relationship between the three components. The events triggered by any user action are intercepted by the controller. Depending on the action, the controller invokes the model to apply suitable business rules that modify application data. The controller then selects a view component to present the modified application data to the end user. Thus, you see that MVC provides guidelines for a clean separation of responsibilities in an application. Because of this separation, multiple views and controllers can work with the same model.

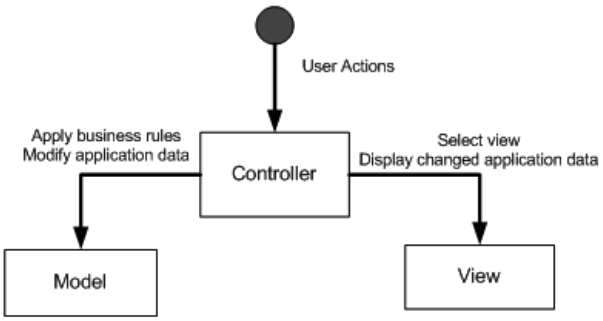

Figure 1-6. *Model-View-Controller*

Java EE Architecture with MVC

The MVC concept can be easily applied to form the basis for Java EE application architecture. Java EE servlet technology is ideally suited as a controller component. Any browser request can be transferred via HTTP to a servlet. A servlet controller can then invoke EJB model components, which encapsulate business rules and also retrieve and modify the application data. The retrieved and/or altered enterprise data can be displayed using JSP. As you'll read later in this book, this is an oversimplified representation of real-life enterprise Java architecture, although it works for a small-scale application. But this has tremendous implications for application development. Risks can be reduced and productivity increased if you have specialists in the different technologies working together. Moreover, one layer can be transparently replaced and new features easily added without adversely affecting others (see Figure 1-7).

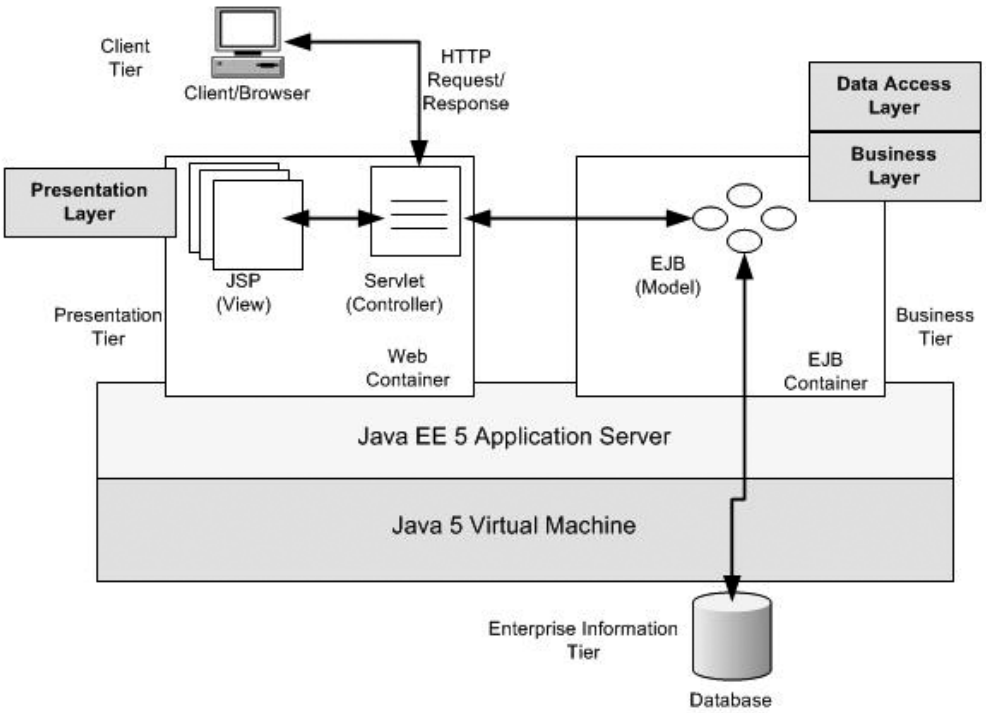

Figure 1-7. *Layered multitier Java EE application architecture based on MVC*

Layers in a Java EE Application

It is evident from Figure 1-7 that layered architecture is an extension of the MVC architecture. In the traditional MVC architecture, the data access or integration layer was assumed to be part of the business layer. However, in Java EE, it has been reclaimed as a separate layer. This is because enterprise Java applications integrate and communicate with a variety of external information system for business data—relational database management systems (RDBMSs), mainframes, SAP ERP, or Oracle e-business suites, to name just a few. Therefore, positioning integration services as a separate layer helps the business layer concentrate on its core function of executing business rules.

The benefits of the loosely coupled layered Java EE architecture are similar to those of MVC. Since implementation details are encapsulated within individual layers, they can be easily modified without deep impact on neighboring layers. This makes the application flexible and easy to maintain. Since each layer has its own defined roles and responsibilities, it is simpler to manage, while still providing important services.

Java EE Application Design

In the past few sections I laid the foundation for exploring Java EE application design in greater detail. However, the design of Java EE software is a huge subject in itself, and many books have been written about it. My intention in this book is to simplify Java EE application design and development by applying patterns and best practices through the Spring Framework. Hence, in keeping with the theme and for the sake of brevity, I will cover only those topics relevant in this context. This will enable me to focus, in the forthcoming chapters, on only those topics that are essential for understanding the subject.

Some developers and designers are of the opinion that Java EE application design is essentially OO design. This is true, but Java EE application design involves a lot more than traditional object design. It requires finding the objects in the problem domain and then determining their relationships and collaboration. The objects in individual layers are assigned responsibilities, and interfaces are laid out for interaction between layers. However, the task doesn't finish here. In fact, it gets more complicated. This is because, unlike traditional object design, Java EE supports distributed object technologies such as EJB for deploying business components. The business components are developed as remotely accessible session Enterprise JavaBeans. JMS and message-driven beans (MDB) make things even complex by allowing distributed asynchronous interaction of objects.

The design of distributed objects is an immensely complicated task even for experienced professionals. You need to consider critical issues such as scalability, performance, transactions, and so on, before drafting a final solution. The design decision to use a coarse-grained or fine-grained session EJB facade can have serious impact on the overall performance of a Java EE application. Similarly, the choice of the correct method on which transactions will be imposed can have critical influence on data consistency.

Simplifying Application Design with Patterns

Application design can be immensely simplified by applying Java EE design patterns. Java EE design patterns have been documented in Sun's Java Blueprints (`http://java.sun.com/reference/blueprints`) and also in the book *Core J2EE Design Pattern* (Prentice Hall, 2003). They are based on fundamental object design patterns, described in the famous book *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison Wesley, 1994). These patterns are also called Gang of Four (GOF) patterns because this book was written by four authors: Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The Java EE patterns catalog also takes into the account the strategies to meet the challenges of remotely accessible distributed objects besides the core object design principles.

Design patterns describe reusable solutions to commonly occurring design problems. They are tested guidelines and best practices accumulated and documented by experienced developers and designers. A pattern has three main characteristics:

- The context is the surrounding condition under which the problem exists.
- The problem is the difficult and uncertain subject area in the domain. It is limited by the context in which it is being considered.
- The solution is the remedy for the problem under consideration.

However, every solution to a problem does not qualify it as a pattern. The problem must be occurring frequently in order to have a reusable solution and to be considered as a pattern. Moreover, patterns must establish a common vocabulary to communicate design solutions to developers and designers. For example, if someone is referring to the GOF Singleton pattern, then all parties involved should understand that you need to design an object that will have only a single instance in the application. To achieve this design pattern, its description is often supplemented by structural and interaction diagrams as well as code snippets. Last but not least, each pattern description generally concludes with a benefit and concern analysis. You will take a detailed look at the constituents of a pattern when I discuss the pattern template in Chapter 2.

The Java EE Design Pattern Catalog

As stated earlier, Java EE has been the dominant enterprise development platform for nearly ten years. Over this period, thousands of successful applications and products have been built using this technology. But some endeavors have failed as well. There are several reasons for such failures, of which the foremost is inadequate design and architecture. This is a critical area because design and architecture is the bridge from requirements to the construction phase. However, Java EE designers and architects have learned their lessons from both failures and successes by drawing up a list of useful design patterns. This Java EE patterns catalog provides time-tested solution guidelines and best practices for object interaction in each layer of a Java EE application.

Just like the platform itself, the Java EE patterns catalog has evolved over time. As discussed earlier, this catalog was first formed as part of Sun's Java BluePrints and later elaborated on in the book *Core J2EE Design Pattern* (Prentice Hall, 2003). Table 1-1 presents the patterns with a brief description of each and its associated layer. I will discuss each of them in greater detail in the subsequent chapters.

Table 1-1. *Java EE Spring Patterns Catalog*

Layer	Pattern Name	Description
Presentation	View Helper	Separates presentation from business logic
	Composite View	Builds a layout-based view from multiple smaller subviews
	Front Controller	Provides a single point of access for presentation tier resources
	Application Controller	Acts as a front controller helper responsible for the coordinations with the page controllers and view components.
	Service to Worker	Executes business logic before control is finally passed to next view
	Dispatcher View	Executes minimal or no business logic to prepare response to the next view
	Page Controller	Manages each user action on a page and executes business logic
	Intercepting filters	Pre- and post-processes a user request
	Context Object	Decouples application controllers from being tied to any specific protocol
Business	Business Delegate	Acts as a bridge to decouple page controller and business logic that can be complex remote distributed object
	Service Locator	Provides handle to business objects
	Session Facade	Exposes coarse-grained interface for entry into business layer for remote clients
	Application Service	Provides business logic implementation as simple Java objects
	Business Interface	Consolidates business methods and applies compile-time checks of EJB methods
Integration	Data Access Object	Separates data access logic from business logic
	Procedure Access Object	Encapsulates access to database stored procedure and functions
	Service Activator (aka Message Facade)	Processes request asynchronously
	Web Service Broker	Encapsulates logic to access external applications exposed as web services standards

Table 1-1 is slightly altered based on the current state of Java EE. The Data Transfer Object pattern, for instance, no longer finds its place in the catalog and therefore is not listed. This pattern was used transfer data across layer and was especially useful if you used remote entity bean persistence components. But with the new Java Persistence API (part of the Java EE 5 platform) and general trend for plain old Java object (POJO) programming models, this pattern is no longer relevant.

This table is far from complete. Certain patterns can be applied across tiers. Security design patterns, for example, can be applied in the presentation layer to restrict access to web resources such as JSPs. Similarly, security patterns can be used to control method invocation on business layer EJB components. Transactional patterns, for example, can be applied at both the business and integration layers. These patterns are classified as *cross-cutting* patterns. I will explore cross-cutting patterns in detail in Chapter 6.

Java EE Architecture and Design with UML

Most modern-day applications are developed iteratively. The system grows gradually as more and more requirements become available. The core of such systems is a high-level design and architecture that evolves through iterations. It is also imperative that design and architecture are documented in both text and visual forms for the benefit of the development and maintenance teams. The visual representation is immensely useful because it helps developers understand runtime interactions and compile-time dependencies.

UML is a graphical language used for modeling and visualizing architecture and detailed design in complex enterprise systems. It is based on a specification developed by Object Management Group (OMG). I will use UML 2.0 notations (which is the latest version) available at `http://www.uml.org/`. However, UML is not limited to architecture and design but can be used in all phases of software development. UML provides a rich set of notation to depict the classes and objects and various relationship and interactions. Modern UML modeling tools such as IBM Rational XDE, Visual Paradigm, Sparx Systems Enterprise Architect, and so on, allow design patterns and best practices to be applied during system design. Moreover, with these tools, the design model can be used to generate significant portions of the application source code.

There are several kinds of UML diagram. But for analysis of Java EE design patterns, I will concentrate primarily on class and sequence diagrams and a simple extension mechanism called *stereotypes*. If you are new to UML or eager to know more, the best UML reference is *UML Distilled Third Edition* by Martin Fowler (Addison Wesley, 2005).

Class Diagram

A class diagram depicts the static relationships that exist among a group of classes and interfaces in the system. The different types of relationships that I will discuss are generalization, aggregation, and inheritance. Figure 1-8 shows the UML notation for a class used to represent the details of an insurance claim. It is represented by a rectangle with three compartments. The first compartment is the name of the class. The second compartment denotes the attributes in the class, and the last one shows the operations defined on these attributes. Note that the + and – signs before the attribute and method names are used to represent the visibility. The + sign denotes public visibility, and the – sign denotes private visibility or that the attribute is not accessible outside this class. Also note that, optionally, you can denote the data type of the attributes, method return type, and parameters.

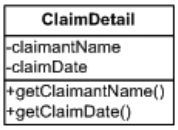

Figure 1-8. *UML class notation*

Interfaces lay down the contract that implementations must fulfill. In other words, classes that implement an interface provide a guaranteed set of behavior. An interface is represented by the same rectangular box as a class, but with a difference. The top compartment shows the class name augmented by a stereotype <<interface>>. Stereotypes are a mechanism to extend an existing notation. Some UML tools also represent interfaces with a circle with no explicit mention of the methods. Figure 1-9 shows the two different forms.

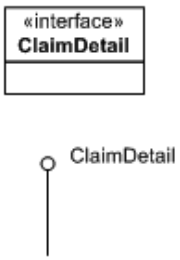

Figure 1-9. *UML interface notations*

Relationships

In the next few sections, I will examine the important relationships that exist between the classes in a software system.

Generalization

The *generalization* relation indicates inheritance between two or more classes. This is a parent-child relationship, in which the child inherits some or all of the attributes and behavior of the parent. It is also possible for the child to override some of the behaviors and attributes. Figure 1-10 shows the generalization relationship.

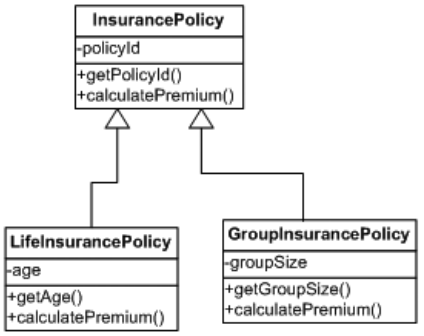

Figure 1-10. *Generalization*

Association

Association shows a general relation between two classes. In an actual class, this is shown with one class holding an instance of the other. An insurance policy always has one or more parties involved, with the most prominent being the policyholder who owns this policy. There can be an agent who helps and guides the policyholder to take this policy. Association often shows named roles, cardinality, and constraints to describe the relation in detail, as shown in Figure 1-11.

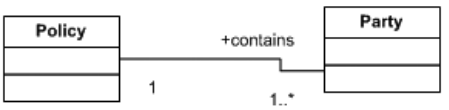

Figure 1-11. *Association*

Aggregation

Aggregation is a form of association in which one element consists of other, smaller constituents. This relationship is depicted by a diamond-shaped white arrowhead. In this case, if the parent object is deleted, the child object may still continue to exist. Figure 1-12 shows an aggregation relation between an insurance agent and the local insurance office in which he works. The local insurance office is where insurance agents carry out tasks such as policy underwriting, depositing premiums for their customers, and various other functions. So even if the local office is closed down, the agent can report to another office. Similarly, the agent can de-register from a local office and move to a different office of the same insurer.

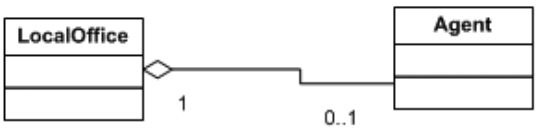

Figure 1-12. *Aggregation*

Composition

Composition is a stronger form of aggregation; as in this case, if the parent is deleted, the children will also no longer exist. This relationship is depicted by a diamond-shaped solid arrowhead. Figure 1-13 shows the composition relationship between a party involved in some policy or claim and their address. If the party is deleted from the system, its address will also be deleted.

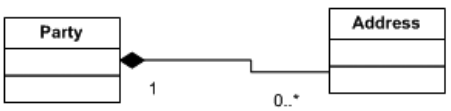

Figure 1-13. *Composition*

Sequence Diagram

A *sequence diagram* is used to model dynamic aspects of the system by depicting the message exchange between the objects in the system over a period of time. A sequence diagram is used to show the sequence of interactions that take place between different objects to fulfill a particular use case. Unlike a class diagram that represents the entire domain model of the application, a sequence diagram can show interaction details of a particular process only.

Object and Messages

In a sequence diagram, an object is shown with its name underlined in a rectangular box. The messages are represented by arrows starting on one object and ending on the other. An object can call a method on itself, which is a self-message and represented by an arrow starting and terminating on the same object, as shown in Figure 1-14.

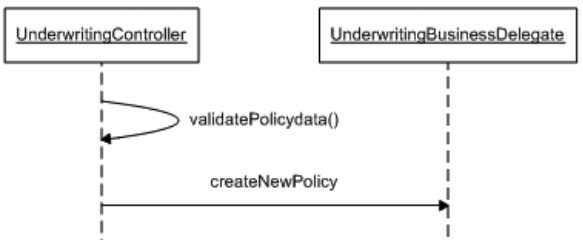

Figure 1-14. *Lifeline in a sequence diagram*

Lifeline

Each object has a *lifeline* represented by a dashed line going downward from the object box (as shown in Figure 1-14). It represents the time axis for the entire sequence diagram with time elapsed measured by moving downward on the lifeline.

Return Values

The messages in a sequence diagram can optionally have a *return value*, as shown in Figure 1-15. The `createNewPolicy` message, for instance, returns a `PolicyDetail` object.

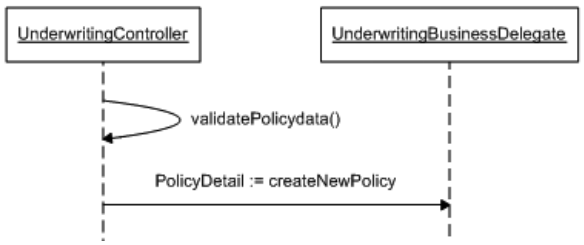

Figure 1-15. *Optional return value in a sequence diagram*

Summary

Developing distributed multitier applications is a daunting task. The Java EE platform looks to simplify this task by defining a container-based architecture. It defines a specification of the runtime environment for the application code and the low-level system services that it should provide. This allows the application developers to focus on writing business logic. The Java EE application architecture is based on the core platform architecture and established MVC principle. With this, you can clearly define specialized component layers in each tier. The web tier, for example, hosts the presentation layer of an application, whereas the business and data access layers generally reside on the application server tier.

Java EE design, on the other hand, is an extended object design. The Java EE design patterns catalog provides guidance and best practices in composing the objects and their interaction within and across the layers and tiers. The design patterns catalog documents the years of experience of designers and developers in delivering successful Java EE applications. The Java EE design and architecture can be documented using UML notations. These are graphical notations that help provide a pictorial view of the static structures and dynamic interactions of the domain objects.

In the next chapter, I'll show how the Spring Framework further simplifies Java EE application design and architecture. If you have experience with the Spring Framework already, you can jump straight to Chapter 3.