PART I

Introduction to Java EE Design Patterns

1

A Brief Overview of Design Patterns

WHAT'S IN THIS CHAPTER?

- An overview of design patterns
- A short history about design patterns and why they are important
- The use of design patterns in the real world
- The history and evolution of Java Enterprise Edition
- The emergence of enterprise patterns
- How these design patterns have evolved in the enterprise environment
- Why and how patterns become anti-patterns

This book is aimed at bridging the gap between the traditional implementation of design patterns in the Java SE environment and their implementation in Java EE.

If you are new to design patterns, this book will help you get up to speed quickly as each chapter introduces the design pattern in a simple-to-understand way with plenty of working code examples.

If you are already familiar with design patterns and their implementation but are not familiar with their implementation in the Java EE environment, this book is perfect for you. Each chapter bridges the gap between the traditional implementation and the new, often easier, implementation in Java EE.

If you are an expert in Java, this book will act as a solid reference to Java EE and Java SE implementations of the most common design patterns.

This book focuses on the most common Java EE design patterns and demonstrates how they are implemented in the Java EE universe. Each chapter introduces a different pattern by explaining its purpose and discussing its use. Then it demonstrates how the pattern is implemented in Java SE and gives a detailed description of how it works. From there, the book demonstrates how the pattern is now implemented in Java EE and discusses its most common usage, its benefits, and its pitfalls. All explanations are accompanied by detailed code examples, all of which can be downloaded from the website accompanying this book. At the end of each chapter, you'll find a final discussion and summary that rounds up all you have read in the chapter. There are even some interesting and sometimes challenging exercises for you to do that will test your understanding of the patterns covered in the chapter.

WHAT IS A DESIGN PATTERN?

> *Design patterns are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."*
>
> —Gang of Four

Design patterns offer solutions to common application design problems. In object-oriented programming, design patterns are normally targeted at solving the problems associated with object creation and interaction, rather than the large-scale problems faced by the overall software architecture. They provide generalized solutions in the form of boilerplates that can be applied to real-life problems.

Usually design patterns are visualized using a class diagram, showing the behaviors and relations between classes. A typical class diagram looks like Figure 1-1.

Figure 1-1 shows the inheritance relationship between three classes. The subclasses `CheckingAccount` and `SavingsAccount` inherit from their abstract parent class `BankAccount`.

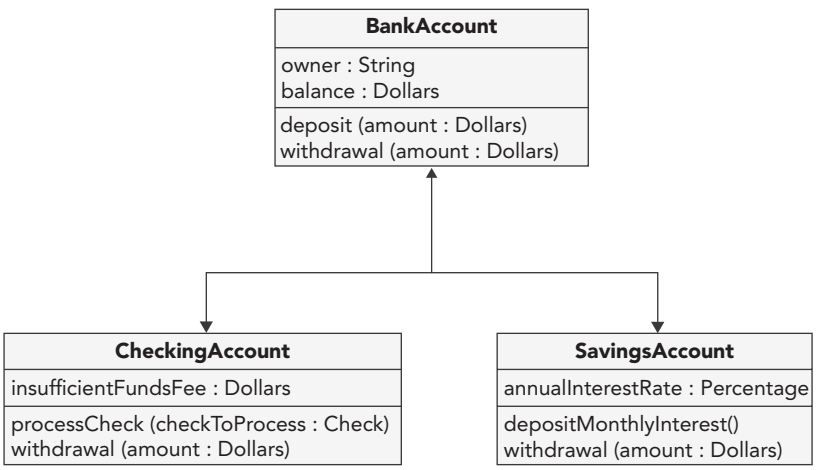

FIGURE 1-1: A class diagram showing inheritance

Such a diagram is followed by an implementation in Java showing the simplest implementation. An example of the singleton pattern, which will be described in later chapters, is shown in Figure 1-2.

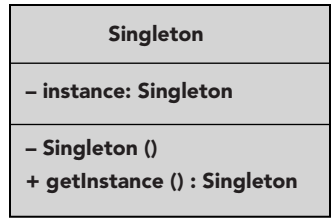

FIGURE 1-2: The singleton pattern class diagram

And here is an example of its simplest implementation.

```
public enum MySingletonEnum {
    INSTANCE;
    public void doSomethingInteresting(){}
}
```

How Patterns Were Discovered and Why We Need Them

Design patterns have been a hot topic since the famous Gang of Four (GoF, made up of Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) wrote the book *Design Patterns: Elements of Reusable Object-Oriented Software*,[1] finally giving developers around the world tried and tested solutions to the commonest software engineering problems. This important book describes various development techniques and their pitfalls and provides 23 object-oriented programming design patterns. These patterns are divided into three categories: creational, structural, and behavioral.

But why? Why did we suddenly realize we needed design patterns so much?

The decision was not that sudden. Object-oriented programming emerged in the 1980s, and several languages that built on this new idea shortly followed. Smalltalk, C++, and Objective C are some of the few languages that are still prevalent today. They have brought their own problems, though, and unlike the development of procedural programming, this time the shift was too fast to see what was working and what was not.

Although design patterns have solved many issues (such as spaghetti code) that software engineers have with procedural programming languages like C and COBOL, object-oriented languages have introduced their own set of issues. C++ has advanced quickly, and because of its complexity, it has driven many developers into fields of bugs such as memory leaks, poor object design, unsafe use of memory, and unmaintainable legacy code.

However, most of the problems developers have experienced have followed the same patterns, and it's not beyond reason to suggest that someone somewhere has already solved the issues. Back when object-oriented programming emerged, it was still a pre-Internet world, and it was hard to share experiences with the masses. That's why it took a while until the GoF formed a collection of patterns to well-known recurring problems.

Patterns in the Real World

Design patterns are infinitely useful and proven solutions to problems you will inevitably face. Not only do they impart years of collective knowledge and experience, design patterns offer a good vocabulary between developers and shine a light on many problems.

However, design patterns are not a magic wand; they do not offer an out-of-the-box implementation like a framework or a tool set. Unnecessary use of design patterns, just because they sound cool or you want to impress your boss, can result in a sophisticated and overly engineered system that

doesn't solve any problems but instead introduces bugs, inefficient design, low performance, and maintenance issues. Most patterns can solve problems in design, provide reliable solutions to known problems, and allow developers to communicate in a common idiom across languages. Patterns really should only be used when problems are likely to occur.

Design patterns were originally classified into three groups:

- **Creational patterns**—Patterns that control object creation, initialization, and class selection. Singleton (Chapter 4, "Singleton Pattern") and factory (Chapter 6, "Factory Pattern") are examples from this group.
- **Behavioral patterns**—Patterns that control communication, messaging, and interaction between objects. The observer (Chapter 11, "Observer Pattern") is an example from this group.
- **Structural patterns**—Patterns that organize relationships between classes and objects, providing guidelines for combining and using related objects together to achieve desired behaviors. The decorator pattern (Chapter 7, "Decorator Pattern") is a good example of a pattern from this group.

Design patterns offer a common dictionary between developers. Developers can use them to communicate in a much simpler way without having to reinvent the wheel for every problem. Want to show your buddy how you are planning to add dynamic behavior at run time? No more step-by-step drawings or misunderstandings. It's plain and simple; you just utter a few words: "Let's use a decorator pattern to address this problem!" Your friend will know what you are talking about immediately, no further explanation needed. If you already know what a pattern is and use it in a right context, you are well on your way to developing a durable and maintainable application.

SUGGESTED READING

It's strongly suggested that you read *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995) or *Head First Design Patterns* by Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra (O'Reilly, 2004). Both are great companions to this book and are invaluable guides for learning design patterns.

DESIGN PATTERN BASICS

One key point regarding design patterns is that overuse or unnecessary use can be troublesome. As soon as some developers learn new patterns, they show a great desire to use them whenever they can. However, doing so often results in their project being bloated with singletons or overwrapping via façades or unnecessarily complex decorators. Design patterns are answers to problems, so unless there is a problem or a chance for a problem to appear, there is no point implementing a pattern. To give an example, using the decorator pattern just because there is a slim chance that an object's behavior might change in the feature introduces development complexity today and a maintenance nightmare in the future.

ENTERPRISE PATTERNS

Java 1.0 quickly became popular after it was released in early 1996. The timing was perfect for the introduction of a new language that would remove the complexity of memory management, pointers, and the syntax of C/C++. Java offered a gradual learning curve that allowed many developers to adopt it quickly and to start programming in Java. However, there was something else that accelerated the shift: applets. An applet is a small application that runs in a website in a separate process from the web browser and adds functionality to the website that would not be possible with HTML and CSS alone. An example would be an interactive graph or streaming video feed.

With the rapid growth of the Internet, static web pages soon became archaic and uninteresting. The web user wanted a better, faster, and more beautiful surfing experience. Along came applets, which offered unbelievable interactivity, effects, and action to the then-static World Wide Web. Soon, dancing Duke (the symbol of Java) became the trend among modern websites. However, nothing remains still for long on the Internet. Users wanted even more, yet applets failed miserably at adapting to those wants, so they did not maintain their popularity.

Nevertheless, applets were the driving force behind the Java platform's fast adaptation and popularity. Today (as this book is written) Java is still among the two most popular programming languages in the world.[2]

Java to Enterprise Java

Following the release of the Standard Edition of Java, IBM introduced Enterprise JavaBeans (EJB) in 1997, which was adopted by Sun in 1999 and formed part of the Enterprise Java Platform (J2EE) 1.2. In 1998 and prior to the release of J2EE,[3] Sun released a professional version of Java labeled JPE. However, it wasn't until after EJB was released that vendors and developers became interested in adopting enterprise Java. With the release of J2EE 1.3 in 2001, Java became a key player in the enterprise world, and its position was sealed with the release of J2EE 1.4 in 2003.

Version 1.4 was one of the greatest milestones in Java's history. It was widely adopted and maintained its popularity for many years even though new versions were released. Vendors and corporations were slow to adopt the newer versions, even though many had reasons to complain about J2EE1.4. Using it was like driving a monster truck to the shops instead of a family sedan. It was definitely powerful, but it was simply too complicated and bloated with XML files, and neither the frameworks nor the containers were lightweight.

Yet J2EE became the most popular enterprise development platform. It had a set of features that made it a great choice for enterprise development.

- **Portability**—The JVM let Java code run on any operating system. Developers could develop on Windows, test on Linux, but go into production on a UNIX system.
- **Security**—J2EE offered its own role-based security model.
- **Transactions**—J2EE offered built-in transactions.
- **Language features from J2SE**—J2SE offered easy syntax, garbage collection, and great object-oriented programming features.

However, J2EE was not perfect. Soon enough, the complex structure of the platform with its heavy use of XML configurations created the perfect problem-ridden environment.

The Emergence of Enterprise Java Patterns

The complex programming models of J2EE soon led many projects into deep waters. Applications developed with J2EE technologies tended to contain excessive amounts of "plumbing" code such as JNDI lookup code, XML configuration files, and try/catch blocks that acquired and released JDBC resources. Writing and maintaining such code proved a major drain on resources and was the source of many bugs and performance issues. The EJB component model aimed to reduce complexity when implementing business logic, but it did not succeed in this aim. The model was simply too complex and often overused.

After just a few years from the first release at the JavaOne conference in 2000, Deepak Alur, John Crupi, and Dan Malks gave a talk titled "Prototyping Patterns for the J2EE Platform," which introduced several patterns targeted at common problems experienced in the design of J2EE applications. This talk would become a book. The following year, they published a book called *Core J2EE Patterns: Best Practices and Design Strategies*.[4] In addition to the 15 already well-known patterns, the authors introduced 6 new J2EE design patterns. The new patterns included Context Object and Application Controller for the Presentation tier, Application Service and Business Object for the Business tier, and Domain Store and Web Service Broker for the Integration tier.

Some of those patterns evolved from the "classic" GoF patterns, whereas others were new and addressed the flaws of J2EE. In the following years, several projects and frameworks such as Apache Camel were released that made the life of enterprise developers easier. Even some, led by Rod Johnson,[5] made a bold step by moving away from J2EE and releasing the Spring Framework. Spring soon became popular, and its popularity influenced great changes in the new programming model behind Java EE. Today most of those patterns are still valid and in use. However, some are obsolete and no longer required thanks to the simplified programming model of Java EE.

Design Patterns Versus Enterprise Patterns

Enterprise patterns differ from design patterns in that enterprise patterns target enterprise software and its problems, which greatly differ from the problems of desktop applications. A new approach, Service Oriented Architecture (SOA), introduced several principals to follow when building well-organized, reusable enterprise software. Don Box's[6] four tenets of SOA formed the basis of these fundamental principles. That set of principles addressed common needs of enterprise projects.

DON BOX'S FOUR TENETS OF SOA

1. Boundaries are explicit. 2. Services are autonomous. 3. Services share schema and contract, not class. 4. Service compatibility is determined based on policy.

However, "classical" patterns still have something to offer. With the release of Java EE 5, Enterprise Java was back in the spotlight, something that third-party frameworks such as Spring and Struts had hogged for too long. The release of Java EE 6 was an even greater step forward and made the platform more competitive.

Today in Java EE 7, most "classic" design patterns described in the GoF book are embedded in the platform ready to be used "out of the box." Unlike the J2EE era, most of these patterns can be implemented via annotations and without the need for convoluted XML configuration. This is a huge leap forward and offers the developer a simplified programming model.

Although there are several great books on design patterns and the new features of Java EE, there seems to be a missing link on how those patterns are implemented in Java EE.

Plain Old Design Patterns Meet Java EE

Even from day zero, Java has been friendly toward design patterns. Some of the patterns have a built-in implementation that is ready to use, such as the observer pattern in Java SE. Java itself also uses many design patterns; for example, the singleton pattern is used in the system and runtime classes, and comparators are a great example of the implementation of the strategy pattern.

The tradition has continued with Enterprise Java, but especially Java EE, which has built-in implementations of many of the patterns described in the GoF book. Most of these patterns can be enabled and used with simple and easy-to-use annotations. So instead of looking at class diagrams and writing boiler plate code, any developer with experience can enable a pattern with just a few lines of code. Magical? Well, not quite. Because the Java EE run time has a more sophisticated design, it can offer many capabilities, relying on the power of the underlying platform. Most of the functionality that those patterns need would not be available without Java EE's superset of features such as EJB and Context and Dependency Injection (CDI). The Java EE container does most of the work for you as it adds many embedded services and functionality to the server. The drawback is that it has resulted in a heavyweight server runtime environment, especially compared to basic web servers such as Apache Tomcat. However, this has improved, and the latest runtime builds on Java EE 7 are more lightweight.

Still, why do people continue to need design patterns in enterprise applications? Well, patterns are needed now more than ever before. Most of the enterprise applications are built for corporations by different teams of developers, and different parts need to be reused often. Unlike solving a common problem pattern on your own or in a small team, your solutions are now exposed to the whole corporation and beyond to potentially the whole world (if your project is open source). It is easy to introduce a poorly designed application and let it become a corporate tradition or development strategy. Because libraries, utility classes, and application programming interfaces (APIs) are exposed to more developers, it has become even harder to break compatibility and make radical changes. Changing one return type or even adding a new method to an interface may break all projects relying on that piece of code.

It is clear that enterprise software development requires a higher level of discipline and coordination between developer teams. Design patterns are a good way to approach this problem. However, most enterprise developers still do not make good use of classical design patterns even though they have been in Java EE since version 5.0. Although enterprise patterns can solve many issues, the original design patterns continue to have much to offer. They are well worn and proven solutions, they have stood the test of time, and they have been implemented in almost all object-oriented languages.

Finally, because most of those patterns are already integrated in the Java EE platform, there is no need to write the full implementation code. Some may require a little XML configuration, but most of the patterns can be implemented by applying an annotation to the class, method, or member

variable. Want to create a singleton? Just add the `@Singleton` annotation to the top of your class file. Need to implement the factory pattern? Just add the `@Produces` annotation, and the method will become the factory of the given return type.

Java EE also sets the standards. The `@Inject` annotation serves as a default implementation and can be mixed and matched with almost any other framework (the Spring Framework) because they use the same annotation.

When Patterns Become Anti-Patterns

Design patterns represent collected wisdom, but this doesn't mean you have to use them all the time. Just like the famous American psychologist Abraham Maslow[7] so aptly stated, "If the only tool you have is a hammer, you tend to see every problem as a nail." If you try to address all problems with only the patterns you know, they simply won't fit, or worse, they'll fit badly and cause more problems. Even more, unnecessary use of patterns tends to overcomplicate the system and result in poor performance. Just because you like the decorator pattern does not mean you need to implement the pattern on every object. Patterns work best when the conditions and the problems require their use.

SUMMARY

Java and design patterns have had a long journey to arrive at where they are now. Once they were separate, with no knowledge of each other, but now they are together, to be forever integrated in the Java Enterprise Edition. To understand this intimate paring, you must know their history. Already you have discovered the roots of your favorite couple and how they found each other. You've read about J2EE's rocky beginnings and how the GoF gave light to 23 design patterns. You've seen how frameworks like Spring came up behind Java and took over and how the reinvented Java EE is now fighting back and gaining ground. The knowledge contained in this book will prepare you to tackle with confidence the majority of the design issue that you will face during your development career. You can rest easy knowing that the years of struggle the Java Enterprise Edition has endured combined with the inherent wisdom of design patterns have resulted in an endurably strong and flexible programming language and environment.

Enjoy this invaluable guide to design patterns in Java EE, and use the wisdom gained here in every project you're involved in.

NOTES

1. *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994): Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
2. According to the TIOBE index, Java appears at number two after C: `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.
3. Before version 5, Java EE used to be called J2EE. From this point J2EE will be used to refer to pre-Java EE 5.

4. *Core J2EE Patterns: Best Practices and Design Strategies* (Prentice Hall 2003, 2nd Edition): Deepak Alur, Dan Malks, John Crupi.
5. Rod Johnson (`@springrod`) is an Australian computer specialist who created the Spring Framework and cofounded SpringSource. `http://en.wikipedia.org/wiki/Rod_Johnson_(programmer)`.
6. Don Box (`@donbox`) is a distinguished engineer: `http://en.wikipedia.org/wiki/Don_Box`.
7. Abraham Maslow (1908–1970) American Psychologist.