Java EE 8 Design Patterns and Best Practices

Build enterprise-ready scalable applications with architectural design patterns

Packt
www.packt.com

By Rhuan Rocha and João Purificação

Java EE 8 Design Patterns and Best Practices

Build enterprise-ready scalable applications with architectural design patterns

Rhuan Rocha
João Purificação

BIRMINGHAM - MUMBAI

Java EE 8 Design Patterns and Best Practices

Commissioning Editor: Richa Tripathi
Acquisition Editor: Alok Dhuri
Content Development Editor: Akshada Iyer
Technical Editor: Mehul Singh
Copy Editor: Safis Editing
Project Coordinator: Prajakta Naik
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Graphics: Jisha Chirayil
Production Coordinator: Shraddha Falebhai

First published: August 2018

Production reference: 1080818

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78883-062-1

www.packtpub.com

To my Aunt, Vanessa Rocha, for teaching me to have a calm look and observe the facts more clearly. To my mother, Ivonete Rocha, for her sacrifices and power.
- Rhuan Rocha
To my two daughters, Carolina and Beatriz, who give me the energy to walk even further; to my father, João Lobato, for his great wisdom and intelligence; and to my mother, Dinah, for her love and affection.
- João Purificação

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

About the authors

Rhuan Rocha is from Brazil and has experience with development using the Java language and Java EE. Currently, he works as senior Middleware consultant in a partnership between Red Hat and FábricaDS, and he applies Red Hat solutions using Red Hat Middlewares. Furthermore, he has 8 years, experience with Java development and Java EE development, developing enterprise applications and government applications.

João Purificação is an electronic engineer from Brazil with a master's in systems engineering. He started working with software development as a C and C ++ programmer. He has worked on the analysis, development, and architecture of Java-based enterprise applications. As a Java/JavaEE consultant, he has participated in the development and architecture of applications for private and government companies. He currently works as a senior architect at Resource IT, a company based in São Paulo.

About the reviewer

Kamalmeet Singh got his first taste of programming at the age of 15, and he immediately fell in love with it. After spending over 14 years in the IT Industry, Kamal has matured into an ace developer and a technical architect. He is also the coauthor of a book on *Design Patterns and Best Practices in Java*. The technologies he works with range from cloud computing, machine learning, augmented reality, serverless applications, to microservices and so on.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Over time, the world of enterprise has invested more and more in technologies and applications that optimize processes and help businesses increase their profits and improve services or products. The enterprise environment has challenges that need to be faced to implement good solutions, such as the high availability of services, the capacity to change when needed, the capacity to scale services, and the capacity to process a large amount of data. With this, new applications have been created to optimize processes and increase profits. The Java language and Java EE are great tools for creating an application for the enterprise environment, because, Java language is multiplatform, open source, widely tested, and has a strong community and a strong ecosystem. Furthermore, the Java language has Java EE, which is, an umbrella of specifications that permit us developer enterprise application without depending on vendors. The development of enterprise application has some well-known problems that occur over and over. These problems involve the integration of services, the high availability of applications, and resilience.

This book will explain the concepts of Java EE 8, what its tiers are, and how to develop enterprise applications using Java EE 8 best practices. Furthermore, this book will demonstrate how we can use design patterns and enterprise patterns with Java EE 8, and how we can optimize our solutions using aspect-oriented programming, reactive programming, and microservices with Java EE 8. Throughout this book, we learn about integration patterns, reactive patterns, security patterns, deployment patterns, and operational patterns. At the end of this book, we will have an overview of MicroProfile and how it can help us develop applications using microservices architecture.

Who this book is for

This book is for Java developers who want to learn to develop and deliver enterprise applications using design patterns, enterprise patterns, and Java best practices. The reader needs to know the Java language and the basic Java EE concepts.

What this book covers

Chapter 1, *Introduction to Design Patterns*, introduces design patterns, looking at the reasons to use them, how they differ from enterprise patterns, and how they behave in the real world.

Chapter 2, *Presentation Patterns*, covers each pattern by explaining the concept and showing examples of implementations. After reading this chapter, we will know about these patterns and will be able to implement them with best practices using Java EE 8.

Chapter 3, *Business Patterns*, explores definitions of the business delegate pattern, the session façade pattern, and the business object pattern. Here, we will focus on reasons to use these design patterns, the common approach to each of them, their interaction with some other patterns, their evolution, and how they behave in the real world. We will also demonstrate some examples of these patterns' implementations.

Chapter 4, *Integration Patterns*, explains some integration patterns and looks at how they work on the integration tier of Java EE. After reading this chapter, you will be able to implement these patterns and use them to solve problems. You will also be able to work on the integration tier, as well as becoming familiar with the concepts associated with integration patterns.

Chapter 5, *Aspect-Oriented Programming and Design Patterns*, looks at the concept of aspect-oriented programming (AOP), focusing on which situations AOP should be used in, as well as how to achieve AOP with the use of CDI interceptors and decorators. Finally, we will also address some examples of implementations. By the end of this chapter, you will be able to identify a situation that requires aspect-oriented programming with the use of interceptors and decorators. Furthermore, you will also be able to identify the best approach to implementing these concepts.

Chapter 6, *Reactive Patterns*, focuses on reactive patterns, concepts, and implementations, and how they can help us implement a better application. We will also cover reactive programming concepts, focusing on how they can aid application development. After reading this chapter, you will be able to use reactive patterns using Java EE 8 best practices.

Chapter 7, *Microservice Patterns*, showcases microservice patterns. We will also compare these with the monolithic pattern, looking at what the advantages and drawbacks of a microservices-based application, are as well as focusing on when to use microservices. Furthermore, we will demonstrate how to switch from a traditional monolithic application to a microservice application, while using implementation examples throughout. We will then look at the design patterns used to compose microservices. After reading this chapter, you will be able to identify the parts of an application's code that are eligible to be microservices, and you will also know how to implement a microservice pattern-based application using Java EE8.

Chapter 8, *Cloud-Native Application Patterns*, outlines cloud-native application pattern concepts. What a cloud-native application is and what goals can be achieved with a cloud-native application will be covered, and we will look at both patterns already described in the previous chapters and new patterns that have emerged to address cloud-based applications. After reading this chapter, the reader will be able to understand the concepts and patterns that characterize cloud architecture.

Chapter 9, *Security Patterns*, discusses security pattern concepts and how these can help us implement better security applications. We will also learn about the single sign-on pattern and how this can help us provide a security application. In addition, we will learn about the authentication mechanism and authentication interceptor, focusing on how to implement these concepts. After reading this chapter, you will be able to create a security application and implement it using Java EE 8.

Chapter 10, *Deployment Patterns*, features deployment patterns, why we use them, and how they impact on the delivery of applications. We will also cover the concepts of canary deployment, blue/green deployment, A/B deployment, and continuous deployment. After reading this chapter, you will be familiar with the concepts of deployment patterns.

Chapter 11, *Operational Patterns*, dives into operational patterns, focusing on why we use them and how they impact on application projects. We will then cover performance and scalability patterns, as well as management and monitoring patterns. After reading this chapter, you will have learned all about the concepts of operational patterns.

Chapter 12, *MicroProfile*, is an overview of the eclipse MicroProfile project, covering its goals and the expectation of this project. Throughout this chapter, we will cover the real benefits of using this project to develop our application and will then actually use it. After reading this chapter, you will know about the Eclipse MicroProfile project and what the real benefits of using this project in our application are. This chapter is only an overview and will not teach readers how to implement applications using the MicroProfile project, and will not be an in-depth chapter.

To get the most out of this book

1. Before reading this book, readers need to know about the object-oriented concept, the Java language, and the basic concepts of Java EE. In this book, we assume that the reader already knows some specifications of the Java EE umbrella, such as EJB, JPA, and CDI, among others.
2. To test the code of this book, you need an application server that supports Java EE 8, such as GlassFish 5.0. Furthermore, you need to use an IDE such as IntelliJ, Eclipse, NetBeans, or any other that supports the Java language.

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Java-EE-8-Design-Patterns-and-Best-Practices. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at **https://github.com/PacktPublishing/**. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/JavaEE8DesignPatternsandBestPra

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "It is also important to bear in mind that the `@Electronic` qualifier identifies the decorated object."

A block of code is set as follows:

```
public interface Engineering {
  List<String> getDisciplines ();
}
public class BasicEngineering implements Engineering {

  @Override
  public List<String> getDisciplines() {
    return Arrays.asList("d7", "d3");
  }
}
@Electronic
public class ElectronicEngineering extends BasicEngineering {
   ...
}
@Mechanical
public class MechanicalEngineering extends BasicEngineering {
   ...
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
@Loggable
@Interceptor
public class LoggedInterceptor implements Serializable {

  @AroundInvoke
  public Object logMethod (InvocationContext invocationContext) throws
  Exception{
    System.out.println("Entering method : "
        + invocationContext.getMethod().getName() + " "
        + invocationContext.getMethod().getDeclaringClass()
        );
    return invocationContext.proceed();
  }
}
```

Any command-line input or output is written as follows:

```
creating bean.
intercepting post construct of bean.
post construct of bean
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "After the user logs in, when they access **Application 1**, **Application 2**, or **Application 3**, they will not need to log in again. "

Warnings or important notes appear like this.
Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

Introduction to Design Patterns

This chapter will introduce design patterns, looking at reasons to use them, how they differ from enterprise patterns, and how they behave in the real world.

Since we assume that you are already familiar with the Java programming language and Java EE, our goal is not to teach Java EE, but to demonstrate its most common design patterns. We will also demonstrate examples of the implementation of design patterns using Java EE 8. Furthermore, we will demonstrate the best way to implement design patterns and discuss the benefits of using design patterns and enterprise patterns. If you do not know about design patterns and enterprise patterns, then this book will be a great tool for learning about the concepts and implementations of design patterns and enterprise patterns. If you already know about design patterns and enterprise patterns, then this book will be a great point of reference to address when implementing them. We'll cover the following topics in this chapter:

- Understanding design patterns
- Understanding the advantages of design patterns
- Defining the basic design patterns of the Java world
- Explaining enterprise patterns
- Explaining the difference between design patterns and enterprise patterns

Explaining design patterns

Design patterns are sets of solutions to common design problems that occur over and over in development. They work as a solution template in which an abstract solution for a common problem is described and the user then applies it, adapting it to their problem. In object-oriented programming, the design pattern provides a way to design reusable classes and objects for a specific problem as well as defining the relationship between objects and classes. In addition, design patterns provide a common idiom among programming languages that allows architects and software developers to communicate about a common and recurring problem regardless of the programming language they are using. With this, we are able to identify a problem and its solution by the name of the pattern and thinking about a solution by a model point of view in a high abstraction level of language programming details.

The design patterns theme gained strength in 1994 after the *Gang of Four* (formed by Rich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) wrote *Design Patterns*: *Elements of Reusable Object-Oriented Software*. Here, they described 23 design patterns that were later known as GoF design patterns and are still used today.

Explaining the Gang of Four design patterns

The **Gang of Four** (**GoF**) design patterns are 23 patterns that are classified as creational patterns, structural patterns, and behavioral patterns. The creational patterns control the creation and initialization of the object and class selection; the structural patterns define the relationship between classes and objects, and the behavioral patterns control the communication and interaction between objects. As well as this, the GoF design patterns have two types of scope which define the focus of solutions. These scopes are *object scope*, which resolves problems about object relations, and *class scope*, which resolves problems about class relations.

The *object scope* works with composition and the behavior changes are done in a runtime. Thus, the object can have a dynamic behavior. The class scope works with inheritance and its behavior is static-fixed at compile-time way. Then, to change the behavior of a class-scope pattern, we need to change the class and recompile.

Patterns classified as class scope solve problems about the relationship between classes and are static (fixed at compile time and cannot be changed once compiled). However, patterns classified under the object scope solve problems about the relationship between objects and can be changed at runtime.

The following diagram shows us the three classifications, as well as their patterns and scope:

	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

In the preceding diagram, we can see the **Factory Method** pattern on the **Class** section and the **Abstract Factory** pattern on the **Object** section. This occurs because the **Factory Method** works with inheritance and the abstract method pattern works with composition. Then, the Factory Method is static-fixed at compile time and cannot be changed after compilation. However, the **Abstract Factory** is dynamic and can be changed at runtime.

GoF design patterns are generally described using a graphical notation such as a use case diagram, and an example of the implementation's code. The used notation must be able to describe the classes and objects as well as the relationship between these classes and objects.

The pattern's name is an important part of the design patterns. This is because it is what the developer uses to quickly identify the problem related to the pattern and to understand how the pattern will solve it. The name of the pattern must be brief and refer to the problem and its solution.

A design pattern is a great tool for designing software development, but its use needs to be analyzed to determine if the design pattern is really required in order to solve the problem.

The catalog of Gang of Four design patterns

Names of design patterns need be succinct, making them easy to identify. This is because design patterns create a vocabulary for communicating between developers independent of programming language, permitting developers to identify problems and solutions only by name of a design pattern.

In design patterns, a catalog is a set of pattern names which are designed to permit a better communication between developers.

The catalog of GoF's design patterns has 23 patterns, as shown in the preceding diagram. Here is a description of these patterns:

- **Abstract Factory**: This provides an interface to create objects without specifying their concrete class, making it possible to decouple the business logic and the object creation logic. With this, we can update the object creation logic in an easy way.
- **Adapter**: This provides an interface that makes it possible for two incompatible interfaces to work together. The adapter pattern works as a bridge between interfaces, adapting these interfaces to work together. Furthermore, the adapter can adopt a class or objects.
- **Bridge**: This pattern decouples an abstraction from its implementation, making them vary independently. With this, we can modify the implementations without impacting the abstractions and we can also modify the abstractions without impacting the implementations. The class of abstraction hides implementations and its complexity.
- **Builder**: This pattern separates the construction of a complex object from its representation. With this, we can construct the objects of several representations using the same process to that. Thus, we create a standard process of construction of objects that have a complex process to construct.
- **Chain of responsibility**: This pattern avoids coupling the sender and receiver of a request creating some objects that have a chance to treat the requests. These objects create a chain of receiver objects for a sender's request. Each object of this chain receives the request and verifies whether or not it will treat this request.
- **Command**: This pattern encapsulates a request for an object and creates a wrapper of requests containing their information about the request. With this, we can do a request to some object sending parameters without knowing about this operation. Furthermore, the command permits us to execute an `undo` operation.
- **Composite**: This pattern composes objects into a tree structure, which represents a part-whole hierarchy. It permits you to treat a group of objects as a single object.
- **Decorator**: This pattern permit extends a functionality of a class with flexibility, without use subclass. It allows you to dynamically attach a new responsibility to an object.
- **Facade**: This hides the complexity of the system, applying a unified interface to a set of interfaces on a subsystem. This makes the subsystem easy to use.
- **Factory Method**: This defines an interface for creating an object, and the subclass states which class to initiate.
- **Flyweight**: This uses sharing to efficiently support a large number of fine-grained objects. This pattern reduces the number of objects created.

- **Interpreter**: This pattern represents language grammar and uses it to interpret them as sentences of a language.
- **Iterator**: This pattern provides a way to sequentially access the elements of a set of objects without knowing its underlying representation.
- **Mediator**: This reduces the complexity of communication by creating an object that encapsulates all the communication and interaction between objects.
- **Memento**: This pattern captures the object's internal states without hurting encapsulated concepts, with this, the state of the object can be restored by the object. This pattern works as a backup that maintains the current state of an object.
- **Observer**: This defines a one-to-many dependency between objects. This means that if one object is modified, all of its dependents are automatically notified and updated.
- **Prototype**: This pattern permits us to create a new object using an object or instance as a prototype. This pattern creates a copy of an object, creating a new object with the same state of the object used as a prototype.
- **Proxy**: This pattern creates a surrogate object (proxy object) for another object (original object) in order to control the access to the original object.
- **State:** This permits an object to alter its behavior when its internal state changes.
- **Singleton**: This ensures that a class has only one instance in the entire project, and the same instance of the object is returned every time the creation process is performed/run.
- **Strategy**: This creates a family of algorithms, encapsulating each one and making them interchangeable. This pattern permits you to change the algorithm at runtime.
- **Template method**: This defines a skeleton for an algorithm in an operation, and the subclass defines some steps to the algorithm. This pattern algorithm structure and the subclass redefine some steps of this algorithm without modifying its structure.
- **Visitor**: This represents an operation to be performed on an object structure. This pattern permits us to add new operations to an element without modifying its class.