



# **Software Design Patterns**

for

# **Java Developers**

Expert-led Approaches to Build Re-usable Software and Enterprise Applications



**Software Design  
Patterns for  
Java Developers**

---

*Expert-led Approaches to Build Re-usable  
Software and Enterprise Applications*

---

**Lalit Mehra**



[www.bpbonline.com](http://www.bpbonline.com)

Ещё больше книг по Java в нашем телеграм канале:  
<https://t.me/javalib>

**FIRST EDITION 2022**

**Copyright © BPB Publications, India**

**ISBN: 978-93-91392-475**

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

#### **LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY**

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.



[www.bpbonline.com](http://www.bpbonline.com)

Ещё больше книг по Java в нашем телеграм канале:  
<https://t.me/javalib>

---

***Dedicated to***

*My beloved Parents*

*Shri Bhushan Mehra, Smt. Rekha Mehra*

Q

*My Beautiful Wife*

---

### ***About the Author***

**Lalit Mehra** is a software engineer with a wide range of interests, covering topics such as microservices, distributed systems, scalability and asynchronous architecture. He has worked in multiple domains, including CRM, payments and finance. He earned his Master's Degree in Computer Science from Indraprastha University, Delhi, and has 10 years of experience working with some of the most prestigious organizations (such as Salesforce and Paytm).

His LinkedIn Profile: <https://www.linkedin.com/in/mehralalit>

## ***About the Reviewers***

**Gaurav Aroraa** is a Tech enthusiast and Technical consultant with more than 23 years of experience in the industry. He has a Doctorate in Computer Science. Gaurav is a Microsoft MVP award recipient. He is a lifetime member of the Computer Society of India (CSI), an advisory member and senior mentor at IndiaMentor, certified as a Scrum trainer and coach, ITIL-F certified, PRINCE-F and PRINCE-P certified. Gaurav is an open-source developer and contributor to the Microsoft TechNet community. He has authored ten books, including Microservices by Examples Using .NET Core (BPB Publications).

Your Blog links: <http://gaurav-arora.com/blog/>

Your LinkedIn Profile: <https://www.linkedin.com/in/aroragaurav/>

**Amandeep** has been working as an Engineering Manager in the field of software development at an Indian AI-based online travel company at the time of reviewing this book. He has worked for more than 9 years in multiple roles in different domains with some of the best organizations including top MNC's (such as RBS and Orange). He owns a wide horizon of interests in coding in Java and Python with an inclination towards solving business problems via software engineering. He has worked in numerous data science fields, especially Natural Language Processing. He has earned his Master's Degree with specialization in Data Analytics

from the Birla Institute of Technology and Science, Pilani and has reviewed a few research papers under “IEEE Transactions on Neural Networks and Learning Systems”. He has earned certifications from multiple MOOCs on data science, machine learning, deep learning, image processing, natural language processing, artificial intelligence, algorithms, statistics, mathematics and related courses apart from his open-source contribution in Spark-NLP, Doccano and has hosted a library aiops on pypi.org.

## ***Acknowledgement***

There are a few people whom I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank my parents and my wife for continually encouraging me for writing the book — I could have never completed this book without their support. My wife has been a continuous source of inspiration throughout the time I was involved with the book. She made sure I got enough time to work on the book after the office and took care of almost everything that needed my presence.

My gratitude also goes to the team at BPB Publications for being supportive and providing me enough time to complete the book. There are so many object-oriented design patterns that it becomes almost impossible to write about all of them in a single book. The team at BPB Publications trusted me to choose a selected few of them to write about in this book and I am very grateful for the confidence they have shown in me.

## *Preface*

This book covers the three types of object-oriented design patterns, i.e., creational, structural and behavioral design patterns. The book has five sections, the first section has a single chapter dedicated to the whats and whys of the design patterns and their types. The other three sections have four chapters each and focus on these design patterns in a one chapter one pattern formation. The last section focuses on the design principles and anti-patterns.

In this book, we will discuss the basics of software design and some of the standard design patterns that are used across the globe in many different software applications. Almost every software application that is functional today uses one or more of these design patterns to accomplish variety of tasks they are built for.

This book focuses on some of the most used object-oriented design patterns and their implementation using Java. Each chapter in this book is dedicated to only one of the design patterns to ensure better readability and understanding. The individual chapters take the problem-oriented approach and discuss the design problem that could be solved by utilizing the design pattern discussed in that chapter. The chapters emphasize on understanding the core of these design patterns and provide close to real life examples to directly connect the reader with the scenarios where these design patterns can be utilized.

This book will benefit those who have a fair understanding of software development and are familiar working with medium to large scale systems. The concepts mentioned in the book will add to the knowledge of those who work with multiple module systems and is also a go to guide for those who want to improve their software design understanding.

Ещё больше книг по Java в нашем телеграм канале:  
<https://t.me/javalib>

***Downloading the code bundle  
and coloured images:***

Please follow the link to download the **Code Bundle** and the **Coloured Images** of the book:

<https://rebrand.ly/4b37ea>

***Errata***

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

[errata@bpbonline.com](mailto:errata@bpbonline.com)

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

---

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.bpbonline.com](http://www.bpbonline.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [business@bpbonline.com](mailto:business@bpbonline.com) for more details.

At 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 BPB books and eBooks.

---

---

#### **BPB IS SEARCHING FOR AUTHORS LIKE YOU**

If you're interested in becoming an author for BPB, please visit [www.bpbonline.com](http://www.bpbonline.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.

The code bundle for the book is also hosted on GitHub at 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 Check them out!

#### **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 [business@bpbonline.com](mailto:business@bpbonline.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

## 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 BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

---

## ***Table of Contents***

### **1. Enlighten Yourself**

Structure

Objective

Introduction

Design patterns

Classification of design patterns

Creational design patterns

Familiar use cases

Structural design patterns

Familiar use cases

Behavioral design patterns

Familiar use cases

Why bother about software design?

Benefits of a good design

Think beyond method and classes

Conclusion

Questions

Pick the odd one out

Find the odd one out

Perfect match

Select the correct option

Answers

Pick the odd one out

Find the odd one out

Perfect match

Select the correct option

## 2. One of a Kind

Structure

Objective

Introduction

Logger

Singletons

Singleton design pattern

Variations

Lazy initialization

Multiple instances of a singleton

Eager initialization

Thread safe singleton

Reflection meets singleton

Enum as singleton

Singleton and serialization

Conclusion

Questions

Pick the odd one out

Find the odd one out

Complete the code

Select two correct options

Answers

Pick the odd one out

Find the odd one out

Select two correct options

## 3. Object Factory

Structure

Objective

Introduction

Interacting with interfaces

Building factories  
Response factory  
Response factory's first run  
The Simple factory  
Multiple factories  
Factory that speaks French  
Response updated!  
Response factory's second run  
The factory method pattern  
Family of objects  
Dependency injection  
Response factory's third run  
Abstract factory  
Conclusion  
Questions  
Pick the odd one out  
Find the odd one out  
Complete the code  
Select two correct answers  
Answers  
Pick the odd one out  
Find the odd one out  
Select two correct answers

## **4. Delegate Object Construction**

Structure  
Objective  
Introduction  
Delegating object construction  
Representational state

Implementing builders

The interface

Concrete builders

Deployment manager

Testing the builders

Builder design pattern – defined

Advantages and drawbacks

Usage

Conclusion

Questions

Pick the odd one out

Odd one out

Complete the code

Select two correct answers

Answers

Find the odd one out

Pick the odd one out

Select two correct answers

## 5. Recycle and Reuse

Structure

Objective

Introduction

Object reusability

Object pool design pattern

When to use an object pool?

Implementing an object pool

Pool initialization

Pool destruction

Acquiring objects

Releasing objects

Varying implementations

Partial initialization

Releasing objects – Who is the owner?

Advantages and drawbacks

Usage

Conclusion

Questions

Pick the odd one out

Find the odd one out

Complete the code

Select two correct answers

Answers

Pick the odd one out

Find the odd one out

Select two correct answers

## 6. Adapter

Structure

Objective

Introduction

Incompatible interfaces

Adapter design pattern

Building an adapter

Interfaces

Concrete implementations

The adapter

Testing the adapter

Adapter design pattern - defined

Benefits and drawbacks

Usage

Conclusion

Questions

Pick the odd one out

Find the odd one out

Select two correct answers

Answers

Pick the odd one out

Find the odd one out

Select two correct answers

## 7. Decorating Objects

Structure

Objective

Introduction

Wrappers

Decorator design pattern

Building a decorator

The interface

The subject

Base decorator

Concrete decorators

Testing the decorators

Decorator design pattern - Defined

Benefits and drawbacks

Usage

Conclusion

Questions

Pick the odd one out

Find the odd one out

Select two correct answers

Answers

Pick the odd one out

Find the odd one out

Select two correct answers

## 8. The Guardian

Structure

Objective

Introduction

Proxies

Proxy design pattern

Types of proxies

Remote proxy

Virtual proxy

Protection proxy

Building a proxy

The interface

The subject

The proxy

Testing the proxy

Proxy design pattern - defined

Benefits and drawbacks

Usage

Conclusion

Questions

Pick the odd one out

Find the odd one out

Select two correct answers

Answers

Pick the odd one out

Find the odd one out

Select two correct answers

## 9. Simplifying the Complexity

Structure

Objective

Introduction

Facade

The Facade design pattern

Building a facade

All for one

One for all

Facade design pattern - defined

Benefits and drawbacks

Usage

Conclusion

Questions

Pick the odd one out

Find the odd one out

Select two correct answers

Answers

Pick the odd one out

Find the odd one out

Select two correct answers

## 10. Template

Structure

Objective

Introduction

Template

Template method design pattern

Building a template method

The template method

Varying implementations

Testing the template method pattern

Template method design pattern - defined

Benefits and drawbacks

Usage

Conclusion

Questions

Pick the odd one out

Find the odd one out

Select two correct answers

Answers

Pick the odd one out

Find the odd one out

Select two correct answers

## 11. Keep a Close Eye

Structure

Objective

Introduction

Observer design pattern

Building observer design pattern

The interfaces

The subject

The observers

Testing the observer design pattern

Variations

Push mechanism

Pull mechanism

Observer design pattern - defined

Benefits and drawbacks

Usage

Conclusion

Questions

Pick the odd one out

Find the odd one out

Select two correct answers

Answers

Pick the odd one out

Find the odd one out

Select two correct answers

## 12. State and Behaviors

Structure

Objective

Introduction

State design pattern

Implementing state pattern

The interface

The concrete implementations

The stateful object

Testing the state design pattern

State design pattern - defined

Benefits and drawbacks

Usage

Conclusion

Questions

Pick the odd one out

Find the odd one out

Select two correct answers

Answers

Pick the odd one out

Find the odd one out

Select two correct answers

## 13. Executing Commands

Structure

Objective

Introduction

Command design pattern

Implementing command pattern

The receiver

The command

The invoker

The client

Command design pattern - defined

Benefits and drawbacks

Usage

Conclusion

Questions

Pick the odd one out

Find the odd one out

Select two correct options

Answers

Pick the odd one out

Find the odd one out

Select two correct options

## 14. Beyond Design Patterns

Structure

Objective

Introduction

Quick recap

Patterns - a summary.

Classification - class versus object patterns

Design principles

Don't repeat yourself

Single responsibility

Dependency inversion

Encapsulate what varies

Open closed

Favor interface over implementation

Interface segregation

Least knowledge

Anti-patterns

An interface for constants

The god object

Caching irregularities

Hard code

Boat anchor

Copy and paste

No silver bullets

Conclusion

Questions

Find the odd one out

Answers

Find the odd one out

Index

## CHAPTER 1

Enlighten Yourself

Ещё больше книг по Java в нашем телеграм канале:

<https://t.me/javalib>

## Structure

Topics that will make this journey worthwhile:

Introduction

Design patterns

Classification of design patterns

Creational design patterns

Structural design patterns

Behavioral design patterns

Why bother about software design?

Benefits of good design

Think beyond methods and classes

## Objective

Our objective is to learn about software design patterns and their role in software development. We will explore the different categories of the design patterns along with their importance and benefits.

## Introduction

Software development is not only about solving problems but about doing so in an efficient manner. Problem solving is a process that involves multiple steps. It starts with problem identification, advances towards developing an algorithm for it and ends with an implementation. A software can be implemented in many different ways; right from devising an algorithm to using a correct set of data objects and the integration among them; all of this is part of implementing a software. However, without a good design, even the best of implementations could fall apart.

Designing a software is not an easy task to accomplish; it requires a lot of effort and knowledge to come up with a design solution that is easy to implement, is feasible, adheres to the programming principles and is scalable in nature. Nowadays the amount of data a software process is not only greater in terms of storage, but is also much more complex. With so much information being kept in the software systems, it becomes important to inspect the current design of the system and to transit to a better one that will adapt to increase usage and fast processing requirements. Software design is not only about putting things in place in a defined manner, but it is also about establishing interconnected pathways that help to transfer data among multiple controllable structures to form a flow that when executed performs a task in an efficient way.

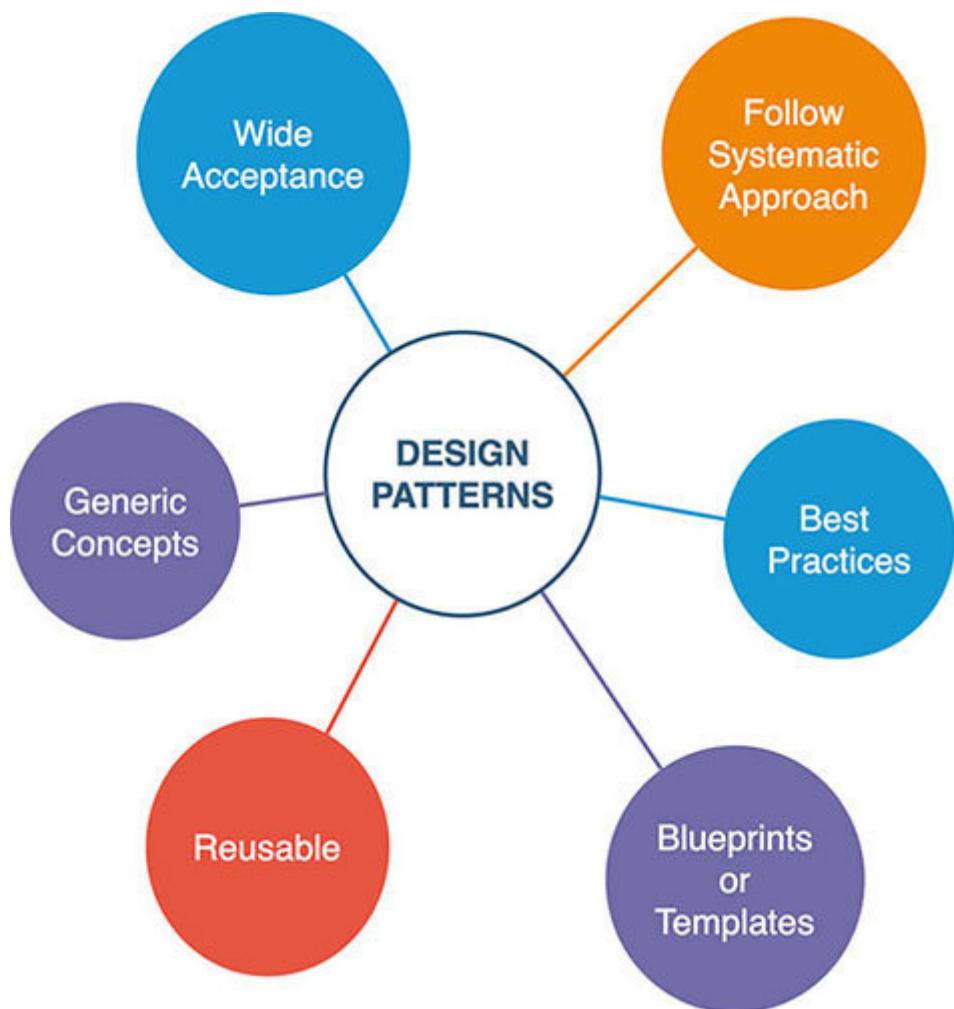
To put it in an interesting manner, the design of the software gives it life. It not only supports the underlying framework, but also makes it easy for the developers to scale the system and be ready for all the future changes, even when, at times, they are unaware of them. A good design ensures that the software is robust and can adapt to the new features easily and with lesser time that is possibly required when there is no proper design.

If you are reading this book today, it means that you have had your fair bit of experience with programming and now wish to explore further and learn about the various design techniques that will not only help you in writing better software but also help you in managing it with much ease. In this book, we will discuss the basics of software design and some of the standard design patterns that are used across the globe in many different software applications. Almost every software application that is functional today uses one or more of these design patterns to accomplish the variety of tasks they are built for. We will also discuss the pros and cons of using one design pattern over another and when to use which design pattern.

## Design patterns

Design patterns, in software engineering, are the blueprints or templates that emerge out of well-designed exercises to solve typical problems that are usually recurring in nature. Design patterns can also be considered as the best practices employed to effectively overcome software design problems:

Figure 1.1 exhibits some of the core attributes of design patterns.



***Figure 1.1: Design patterns***

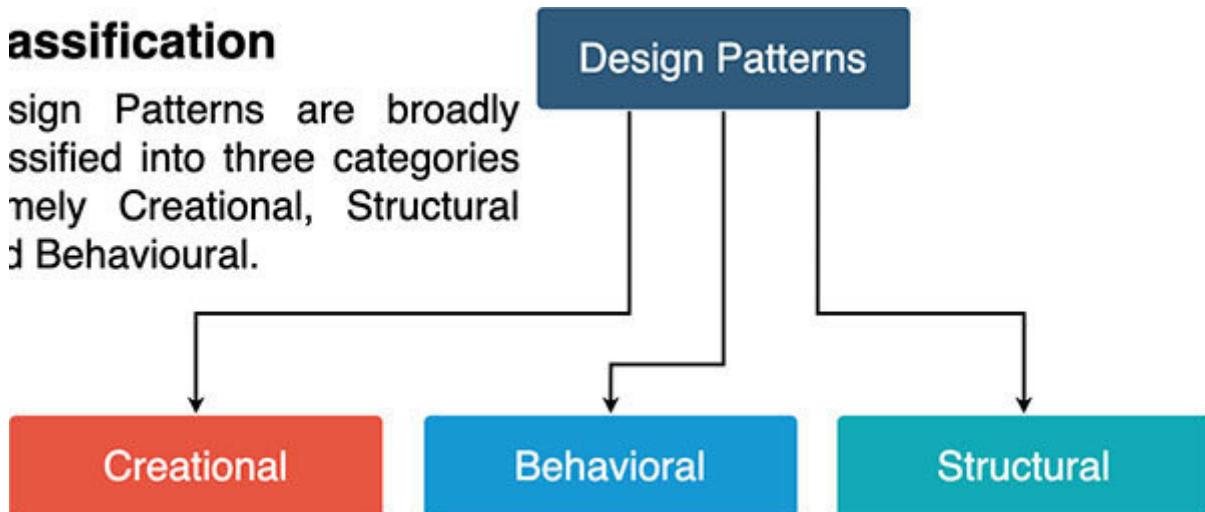
Design patterns follow a systematic approach and can be reused in a variety of situations as they provide us with an idea of how to solve a particular problem. They themselves are not the actual code solutions, but instead blueprints or general solutions that help us to solve the problem in an effective way.

A design pattern can have one or more standard objects or interfaces that are necessary for its successful implementation. These interfaces or objects are the very soul of the pattern itself and require the programmer to follow the exact practice as described in the pattern for effective utilization.

## Classification of design patterns

The design patterns help solve a variety of design problems ranging from object creation and interface development to managing the behavioral aspects of the object. They help us with various use cases centric to the application and provide solutions that are architecturally rich and easy to implement and manage.

[Figure 1.2](#) exhibits the classification of design patterns into three categories, that is, creational, behavioral, and structural:



**Figure 1.2: Design patterns classification**

An important aspect of software design is the knowledge and understanding of the person who designs it. It is therefore not just recommended but necessary to understand how the system should behave and what is the overall design expectation before

somebody chooses a design pattern. While choosing a good design is important, it is much more important to not to choose a bad design. The implication of that could be far more troublesome than expected.

Design patterns could be broadly categorized into three categories, namely, creational, structural and behavioral.

While the creational patterns are more involved with creation of objects and the interfaces that connect with those objects, the structural patterns help with designing large structures by assembling two or more systems together to provide new functionalities and the behavioral patterns are inclined towards communication between objects.

## Creational design patterns

Design patterns that relate to the idea of object creation in an efficient and controlled manner are referred to as creational design patterns.

Their working could be understood based on two defining principles:

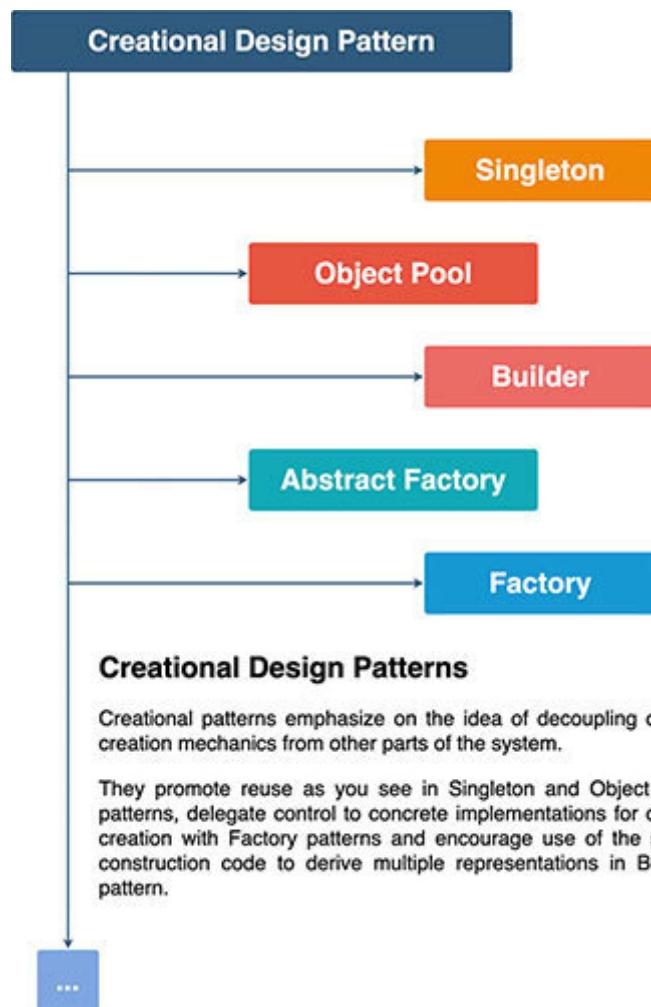
Hiding the object and its composition from the outside world

Controlling object usage

Both the principles are followed by providing an abstraction layer that ensures the only connection to these objects from the outside world is through this layer only.

Creating objects and using them without a controlled environment scatters those objects all over the place and it becomes increasingly difficult to manage and refactor the application in the future. The creational design patterns decouple the process of object creation from other parts of the application thus ensuring that the objects are created and used as per application best practices. This also helps with performance and security guidelines when the objects are created and composed in a controlled environment.

[Figure 1.3](#) explains the objective of creational design patterns and lists some of these patterns we will learn about in this book:



*Figure 1.3: Creational design pattern*

Creational design patterns make effective use of abstraction and access control mechanisms provided by the object-oriented languages. Interfaces are used to reduce the dependency of objects on each other as we see in the factory pattern and dependency injection pattern. Similarly, access control mechanisms

help to control access to an object and its state, directly, by any other object. The only way the object could be accessed is via public methods.

### **Things to Remember:**

Segregates object creation from other business logic

Promotes reusability of instances

Controls object access

Hides object composition

In this book, we will discuss some of the widely used and accepted creational design patterns, namely:

**Singleton design** This pattern ensures that only one instance of the class, which is made singleton, is exposed for access per JVM

**Builder design** The builder pattern aims to simplify the construction process of complex objects by providing a stepwise approach to instance creation. It also exhibits the capability to construct the same object with different representations

**Object pool design** This pattern creates a pool or group of objects that can be reused to process some information. This pattern is used in scenarios where the object creation cost is high and object reuse is advisable than its creation.

**Factory design** Factory design pattern provides an abstraction and delegates the control to the subclasses for the type of object to be constructed. It is the responsibility of the implementation classes to decide upon the type of object based on the supplied inputs.

### Familiar use cases

Imagine an object that converts a JSON string to some other format and requires a lot of CPU and memory resources during initialization. Whenever there is a need to perform the conversion, one way to accomplish it is to instantiate an object. The other way is to use the singleton design pattern for object creation and use that object whenever required. By using the singleton pattern, a lot of resources could be saved and hence a lot of processing time. An example of singleton in JDK is the class runtime, which has only one instance in a JVM.

Imagine a scenario where multiple objects with high cost of creation are required to perform parallel or concurrent operations. This scenario is an excellent use case for the object pool design pattern. An object pool instantiates and stores multiple objects that can be used again and again in an application that has to process multiple requests at a time. An object pool lends its resources for processing and computation and usually has a bounded queue to keep track of the unprocessed requests that wait for their turn until one of the pool objects is free. An example of an object pool in the JDK is the Executor framework.

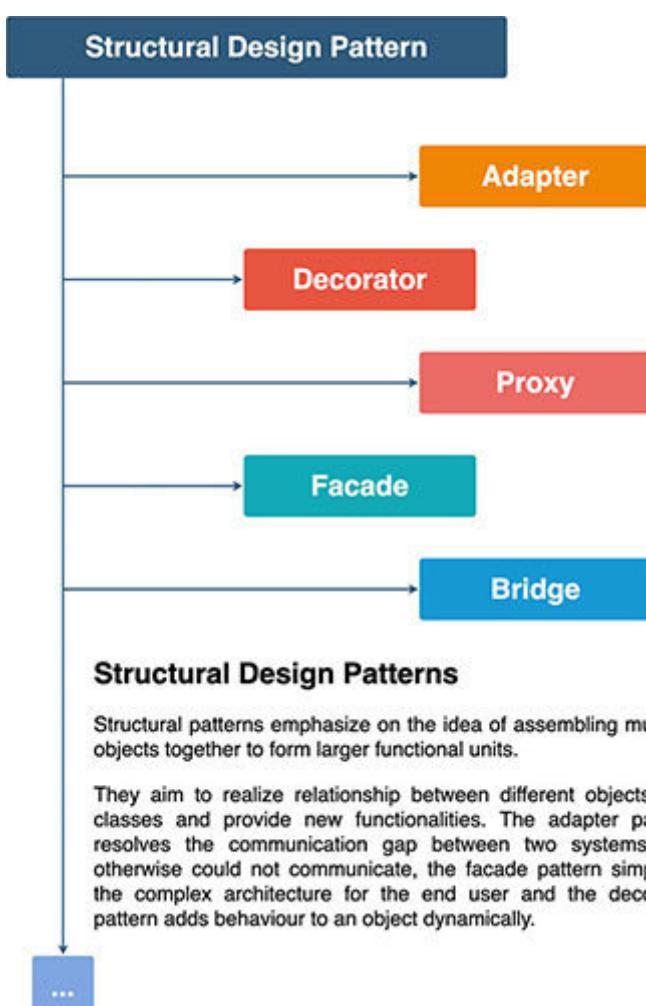
Another prominent use case is of a scenario that involves multiple implementations of the same interface and the selection of the right implementation for a particular request is governed by the

implementation themselves. Imagine that you have to calculate tax for a user and the tax calculation algorithm involves multiple factors, using the factory design pattern the calculation process could be simplified and streamlined by introducing these factors into the selection process to select the right implementation.

## Structural design patterns

The basic idea behind structural design patterns is the formation of large structures by assembling various objects or their classes together and to provide new functionalities.

[Figure 1.4](#) explains the objective of structural design patterns and lists some of these patterns we will learn about in this book:



#### ***Figure 1.4: Structural design pattern***

Structural design patterns do not modify existing classes or objects, but instead add new features on top of them. This way, in general terms, multiple business use cases can be solved effectively with minor or no architectural changes.

In other words, structural design patterns aim for the realization of relationship among objects and provide new functionalities in return. They make use of the two important pillars of object-oriented programming, that is, inheritance and composition to form the new structures. The emphasis is on the use of interfaces rather than concrete implementations so that the participating classes remain loosely coupled to each other. This brings us to the important design principles, that is, interact with interfaces rather than with concrete implementations.

Though the larger structure acts as one unified unit, the participating classes and or objects can still be improved and maintained and even replaced in isolation. This is possible because of loose coupling among the classes. Structural design patterns make use of existing classes and objects, wrap them or use them in previously unrelated or newly formed classes to create new functionalities.

#### **Things to Remember:**

Assembles classes and objects to form large structures

Expresses relationship between objects

Promotes inheritance and composition

Requires no or minimum change in participating classes

Realizes new functionality

In this book, we will discuss some of the widely used and accepted structural design patterns, namely:

**Adapter design** The adapter design pattern converts one interface into another and provides a way for classes to interact with each other without changing their source code.

**Decorator design** The decorator design pattern expands the functionality of an existing class or interface without modifying their source code. It creates wrappers that inherit the same interface and use composition to add the new functionality.

**Proxy design** Proxy, as the name suggests, acts as a placeholder for another class. The proxy pattern is used as a gateway to control access to other classes. The proxy class could be used for security, validation, data retrieval and other requirements that need controlled access to an object.

**Facade design** Facade pattern simplifies the interaction between two systems by providing an interface that hides the complex processing and data manipulation.

### Familiar use cases

Imagine a scenario where a ticket booking module confirms the booking only after receiving a response from the payment processing module. These two modules, though unrelated, are supposed to work together to ensure the functionality of the overall system. The structure of the data and the tables could be different for both the modules and might require formatting or conversion of data. The adapter design pattern is suitable for this use case and will require minimal or no changes to the existing modules.

Think of a system that uses extensive logging and the existing classes are closed for modification, but a new requirement needs additional logs. In this scenario, you can make use of the decorator design pattern by creating a wrapper class that has an instance of the log writer class. Now, instead of calling the existing log writer, you can call the instance of the wrapper class that delegates the call to the old wrapper class after writing the additional logs. The file handling capabilities in JDK make use of the decorator design pattern.

Think of an image search and rendering system that searches its large database to provide relevant images based on the input criteria. The system might be overwhelmed by the increase in the number of searches and its performance and throughput could

reduce. The proxy design pattern could be utilized to safeguard the system with minimum impact on the user experience. By using this pattern, a predefined small set of cached images could be returned in response to the search request and a database request for the actual search could be triggered in the background that when finished, its response is displayed to the user.

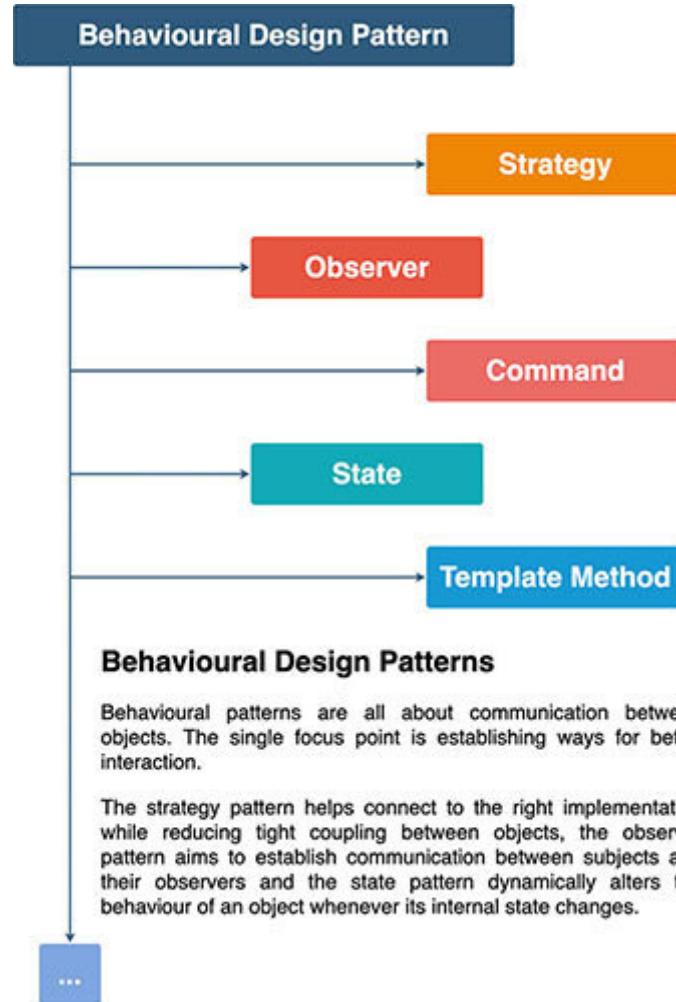
## Behavioral design patterns

Behavioral design patterns are all about the relationship between objects and how they interact with each other. They aim to reduce the communication complexity and help improve the interaction among objects. The crux of the design solutions provided by the behavioral design patterns is the use of abstraction to decouple the classes from each other and ensure a smoother communication via interfaces.

The behavioral design patterns evaluate the communication process followed within a system and realize it into a pattern that helps in streamlining the communication mechanism. By following a pattern, communication among the objects becomes smooth and easy to implement.

A system's behavior is often dependent on internal factors and external inputs. Behavioral design patterns provide solutions that aid and ease the selection of appropriate behavior or implementation while ensuring reliability and correctness.

[Figure 1.5](#) explains the objective of creational design patterns and lists some of these patterns we will learn about in this book:



**Figure 1.5:** Behavioral design pattern

The basic idea behind behavioral design patterns can be summed up as:

Evaluation of the communication mechanism to realize it into a pattern

Use of interfaces to support communication among objects

**Things to Remember:**

Focuses on improving the communication among classes and objects

Promotes interaction via interfaces

Supports multiple behaviors and supervises their selection process

In this book, we will discuss some of the widely used and accepted structural design patterns, namely:

**Strategy design** The strategy design pattern simplifies the selection process among multiple implementations by providing a common interface that all the implementations must respect.

**Observer design** The observer design pattern streamlines the communication between a subject and its observers. It relies on the push mechanism of data transfer and any change in the state of the subject is notified to the observers in near real time.

**State design** The state pattern helps to maintain internal states of an object and allows for seamless change in object's behavior with state transition.

**Command design** The command pattern encapsulates a request in the form of an object that could be invoked when required. The command can be parameterized, logged or queued for timed execution.

### Familiar use cases

Imagine a language translator that provides translation facilities among multiple languages. The application software has multiple implementations that can be triggered based on a set of input variables. For the process of translation to be effective and error-free, it is necessary to select the correct implementation. This particular use case is well suited for strategy design pattern. The implementations here are the strategies and only one of them should be followed to obtain the correct result. By using the strategy design pattern, we can develop a centralized logic that can be utilized to decide the correct implementation based on the inputs provided.

Imagine a stock trading system that keeps track of the current value of all the registered stocks. This information is very important to the stock brokers who buy and sell stocks regularly. If this information could be delivered to the brokers in real time that will immensely help them. One of the ways to establish such a mechanism is to expose APIs that are consumed by the software used by the brokers. Another way to deliver this important information to all the brokers is by introducing the observer design pattern. The observer pattern facilitates the information delivery process by registering all the interested broker clients to the stock trading system and broadcasting changes to the stock prices to all the clients whenever such a change

happens. Java JDK also exposes Observer and Subject interfaces to realize the observer design pattern.

If you have ever used the kiosk that prints boarding passes for flights at the airport, you would have noticed that it follows a series of steps to process your information and print the boarding pass. You would also have noticed that the next button does not only take you to the next step, but also processes your information in different ways at different steps. Imagine those steps as nothing more than the states of the software that is run on those kiosks. By using the state design pattern, the kiosk software could be realized in an efficient way. Change in the internal state of the object used by the kiosk system would be triggered whenever the user presses the next button and with each new state the behavior of the kiosk would change.

## Why bother about software design?

Have you ever noticed why sometimes it is very difficult to make changes to existing codebase? Why even the smallest of the changes require modifying multiple classes? Why does a change in one part of the system break something else somewhere? Why is it so difficult to read and understand the logic? Why, at times, the performance is so undesirable that the need for a new system is felt? All these question marks often end at the same answer, bad design choices.

The planning and design phase is of utmost importance for most of the questions asked above. Designing a new system or adding a new feature to the existing one comes with a lot of challenges, especially if the system's complexity is high. A bad design choice could end up in a low performance and unhealthy system that could further introduce a lot of bugs and long working hours to fix them.

A properly designed system exhibits many good properties that increase the functional and performance capabilities of the system. A good design streamlines the architecture and reduces tight coupling between modules. It helps the system to scale with time, perform better under stress and promotes plug and play.

A good design is not only relevant today, but in future as well. Having a good design ensures that the system is scalable and

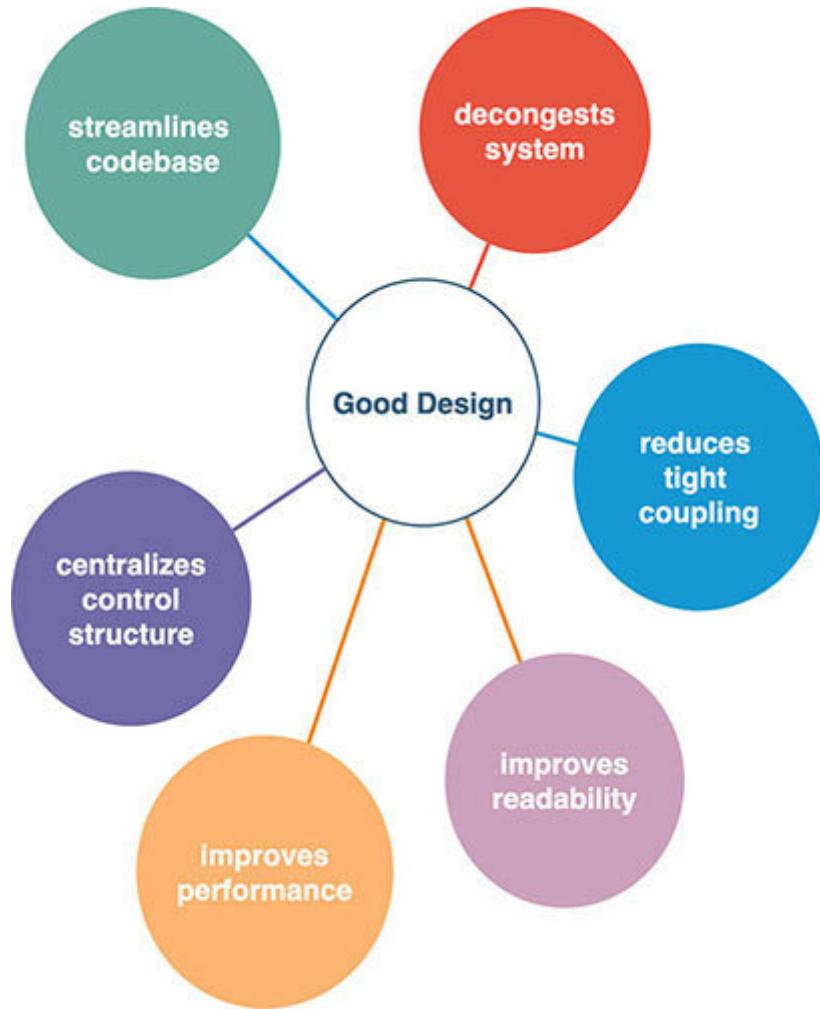
can adapt to changes easily.

### Benefits of a good design

A good design streamlines codebase, decongest system, reduces tight coupling, centralizes control structure and improves performance and readability:

When two or more classes are dependent on each other and even a small change in one affects the other, they are termed tightly coupled.

Figure 1.6 exhibits the benefits of a good design:



**Figure 1.6:** Benefits of good design

A good design solution will always promote interaction via interfaces, which introduces abstraction and the classes can be modified independently without affecting any other part of the system:

To understand the code, it is important that the code is readable, that is, you should be able to grasp the logic without much difficulty and in as less time as possible. A good design promotes readability by simplifying the architecture.

In view of a good design, it is favored to keep the important and reusable logic, for example, object access, creation or destruction in a centralized location, which allows for quick modification and system wide reflection. With the same logic scattered at multiple places, it becomes difficult to manage since even a small change will have to be made at multiple places without fail.

**Streamlines** One of the most important characteristics of a good design is that it should ease the movement of data and remove unnecessary hops.

By streamlining the communication mechanism and reducing the number of hops, the design patterns help to improve the performance of a system. A good design also reduces the possibility of architectural bugs that could be hard to trace and mitigate.

### Think beyond method and classes

Most of the things discussed in this chapter circle around the basic building blocks of any software application, that is, classes and objects and the various concepts that bind them together. While it is important to understand the concept of design patterns in the realm of these building blocks, it is also important to understand and give due attention to the overall architecture of an application.

While designing a system, one cannot simply go into the details of implementation before analyzing the various use cases and capturing the requirements the system is supposed to fulfil. This activity provides a visual description of how a system should behave and the functionalities that shall be built.

The above exercise generates a holistic view of the system and how it could be broken down into multiple modules that interact with each other to form the complete system. Development of multiple modules, each responsible for a different task presents us with the opportunity to design them separately selecting the best design patterns that perfectly fit the use case solved by that module.

Coming back to the basic building blocks, though the connection between classes could be a trivial thing to establish, it is imperative to define these connections relevant to the flow of data

and the purpose at hand. The design patterns help us with this definition by showcasing various well-tested and widely-accepted best practices to choose from.

## Conclusion

Software development is a subject of continuous evolution. With ever-changing technology and ground-breaking innovations, there will always be a need for better design and organization that facilitates and improves the development process.

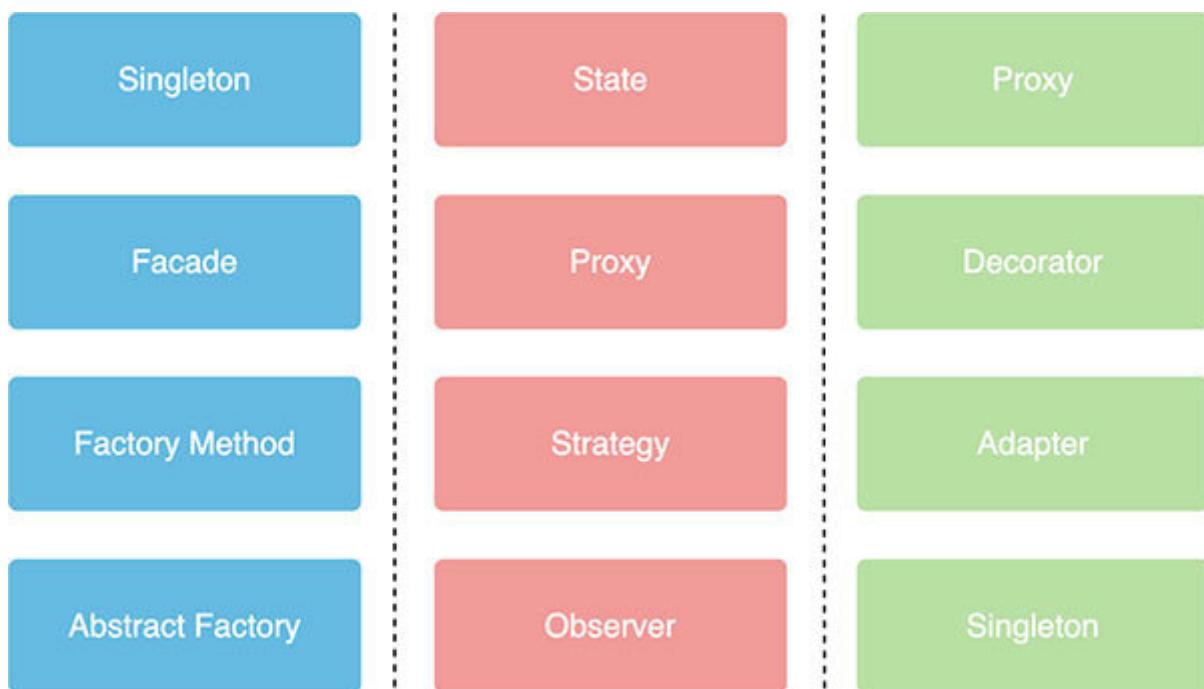
In this chapter, we discussed the basics of software design and briefly explored various patterns that should be utilized to improve the process of software development. We also discussed some of the use cases where those patterns could be used and benefitted from.

In the next chapters, we will deep dive into some of these design patterns and will discuss them thoroughly. The chapters will be bundled in three different sections, one each for creational, structural and behavioral design patterns. The chapters will take you to a tour of various patterns discussing not just the basics, but also the defining principles on which those patterns are based on. There will be coding examples to guide you throughout the chapter and to ensure that you are ready to utilize the pattern as soon as you finish the chapter.

## Questions

In this section, we will sharpen our brain by indulging in some fun brain activities. Let's get started...

Pick the odd one out



*Figure 1.7: Pick the Odd one out*

### Find the odd one out

Structural design pattern promotes

Minimal change in existing classes

Inheritance and composition

Rewriting classes to support new functionality

Assembling existing structures to form new ones

Behavioral design pattern focuses on:

Improving communication

Interaction via interfaces

Tight coupling

Support for multiple behaviors

## Perfect match

### Connect the Patterns

Revise your knowledge of which pattern belong to which type by drawing lines from left to right

Creational

Structural

Behavioural

Facade

Singleton

Decorator

State

Factory

Builder

Observer

Strategy

*Figure 1.8: Perfect Match*

Select the correct option

Loose coupling is achieved by:

Grouping classes together

Introducing abstraction in the form of interfaces

Directly calling an object from another

Creating singleton objects

What is composition?

When an object has a reference to another object

When a class inherits another class

When a method has multiple lines

When interfaces are used instead of classes

What is the benefit of using interfaces?

They help in writing better code

They provide default level access control

They promote loose coupling

They are support structures

Three things that a good design pattern provides:

Centralization, better performance, tight coupling

Readability, low performance, loose coupling

Streamlined codebase, readability, centralization

Better performance, tight coupling, readability

## Answers

Pick the odd one out

Image [Column Wise]

Facade, Proxy, Singleton

## Find the odd one out

Rewriting classes to support new functionality

Tight coupling

## Perfect match

Creational – Singleton, Factory, Builder

Structural – Facade, Decorator

Behavioral – State, Observer, Strategy

Select the correct option

Introducing abstraction in the form of interfaces

When an object has a reference to another object

They promote loose coupling

streamlined codebase, readability, centralization

## CHAPTER 2

### One of a Kind

---

You can find the editable code files for this chapter on the book's GitHub link.

<https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers/tree/main/Chapter%2002>

---

## Structure

Topics that will make this journey worthwhile:

Singletons

Singleton design pattern

Variations of singleton design pattern

Lazy initialization

Eager initialization

Thread safe singletons

Reflection meets singleton

Enum as singleton

Singleton and serialization

## Objective

Our objective is to understand the basics of a singleton object. We will learn about the singleton design pattern and the different variations of it, understand the challenges of using serialization with singleton and how to resolve them, and the effects of reflection on singleton and how to avoid that.

We will also walk you through the implementation details of different variations of the singleton design pattern, the benefits it provides, and a few drawbacks along with some of the places where this design pattern can be used.

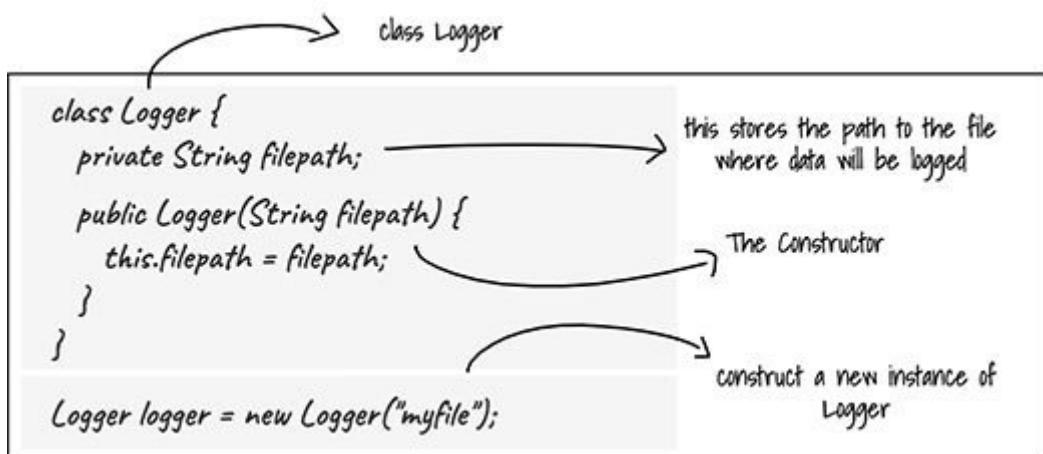
## Introduction

Objects are the center piece of any object-oriented programming language; they are the elements on which the whole application stands. In this chapter, we will discuss a very special object, a singleton and will learn about the singleton design pattern.

Singletons are unique objects as there could be only one instance of their type in the whole application.

Object creation is often a trivial task, for example, in Java, one can make use of the new keyword to construct a new object.

In Figure the class logs the information to a file:



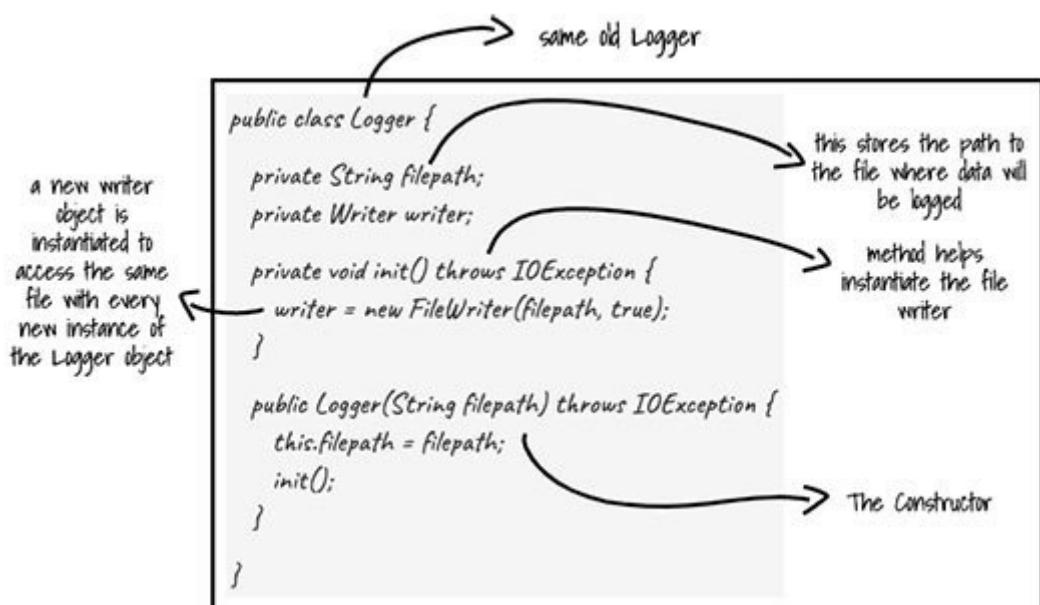
**Figure 2.1: Introduction**

It contains a public constructor and a data member that stores the path to the file where the data will be logged. The example

also demonstrates how we can construct a new instance of this class, simply by calling the constructor with the new keyword.

## Logger

In [Figure](#) the class did not have any method that can log the information for us. Let us enhance our class and add some methods to make our class realistic and useful. [Figure 2.2](#) demonstrates an updated version of the logger class, that uses a writer to write to the log files:

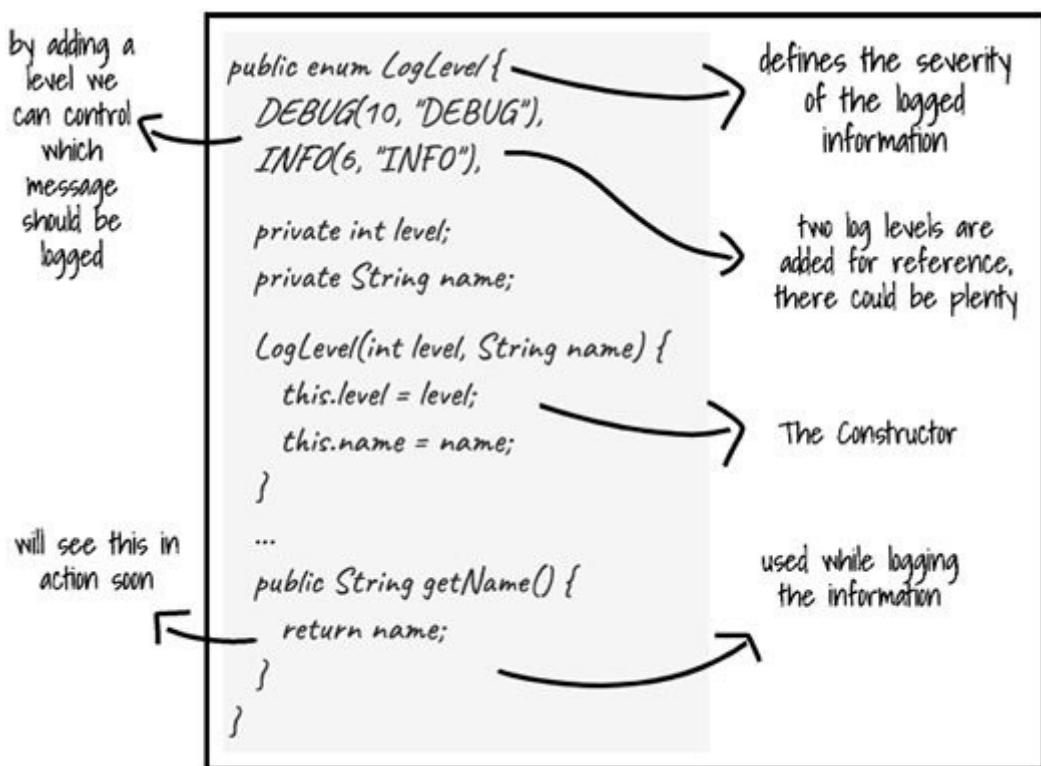


**Figure 2.2: Logger**

The class looks more promising now and initializes an actual writer object to write the data to the file whose location is passed to the constructor. For simplicity, we will use standard exception classes and will not write our own. Let's add some more methods to our class to log information to the log files. Before that, let us create an enum that will help to segregate the logged information.

Segregating logs is a good way to manage and search for the required information.

By using a log level, we can log messages based on their level of severity or importance as shown in the [Figure](#)



**Figure 2.3: Log level**

The logs can also be distributed among multiple log files using the log level, with each log level having its own file or multiple log files for the same log level as shown in the [Figure](#)

Question: What happens when every class writes to the same log file, but instantiates a new instance of logger?

Answer: With every new instance of logger a related writer instance is also spawned and the file will be shared across every writer leading to possible performance degradation.

Question: Isn't there a way to share the same instance of logger among different classes?

Answer: Precisely why we are here. We will learn how the singleton design pattern is used to instantiate a single object of type logger that is shared across the application.

**Figure 2.4:** Q&A

Though we won't be discussing these two features further, this will keep an open window for you to think and demonstrate your programming skills. [Figure 2.5](#) demonstrates the methods that we can use to log the information to the files:

```
public void log(String message) throws IOException {  
    log(LogLevel.INFO, message);  
}  
} default log level is INFO
```

*adding '\n' after every line*

```
public void log(LogLevel level, String message) throws IOException {  
    writer.append(level.getName()).append(message)  
        .append(System.lineSeparator());  
    writer.flush();  
}  
} we are flushing after each write so that when you run the code yourself you see the logs immediately
```

**Figure 2.5:** Adding Log Level to Logger

These two methods complete the implementation of the logger class demonstrated in [Figure](#). In [Figure](#) we have created a simple controller that has an instance of logger to write to a file of our choice:

```
class UserAccessController {  
    private String filepath = "access_control.log";  
    private Logger logger = new Logger(filepath); new Logger instance  
    ...  
    public boolean isAccessGranted(User user) {  
        ...  
        logger.log("Unauthorized access: " + user.getUserId()); calling log method to log information  
        ...  
    } default log level used
```

*Figure 2.6: Using Logger*

It uses the default log method demonstrated in [Figure](#). In [Figure](#) although we are logging to the same log file, we have two different instances of logger and both open separate connections to the log file:

```

class ProductSearchController {
    private String filepath = "product_catalog.log";
    private Logger logger = new Logger(filepath); → new Logger instance
    ...
    public Product search(String tag) {
        ...
        logger.log(LogLevel.DEBUG, "Product Tag: " + tag); → calling log method to log information
        ...
    }
}

class ProductListController {
    private String filepath = "product_catalog.log"; → same log file as above
    private Logger logger = new Logger(filepath); → new Logger instance
    ...
    public List<Product> fetch(Section section) {
        ...
        logger.log("Request received for section: " + section);
        ...
    }
}

```

in both the classes above we are logging to the same file but via different writer instances

**Figure 2.7: Controllers**

Imagine every class opening separate connection(s) to the file(s), soon the system will have a huge number of connections to manage. Not only does it involve a lot of unwanted management, it also impacts the performance of the system. If we could have a fixed set of connections for the whole application to use, the situation could be improved.

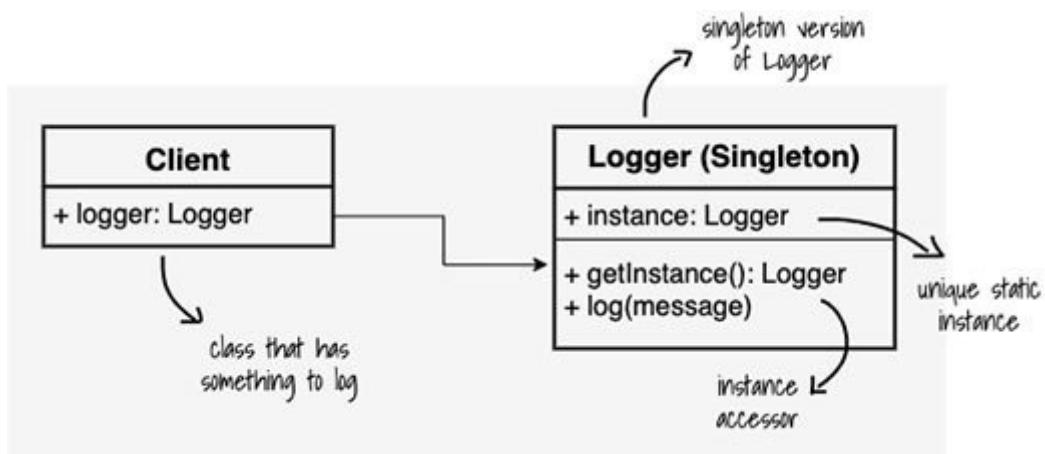
## Singletons

A singleton has only one instance throughout the application. A singleton, by design, is global and can be accessed from anywhere in the application using the exposed accessor. The accessor is designed to always return the same instance of the class no matter how many times it is called.

A singleton is useful in scenarios where only a single instance of a type is required throughout the application, making it the single point of access for that type. A singleton is often confused with global access. Though the object by definition is global, constructing a singleton to achieve application wide access defeats its purpose. It should be kept in mind that for an object to be global doesn't require it to be singleton as well. A singleton also provides some interesting add-ons; since a singleton is the single point of access for a type, it could also be utilized to restrict access to it or to provide partial access based on certain configuration. Let us now take a look at the singleton design pattern.

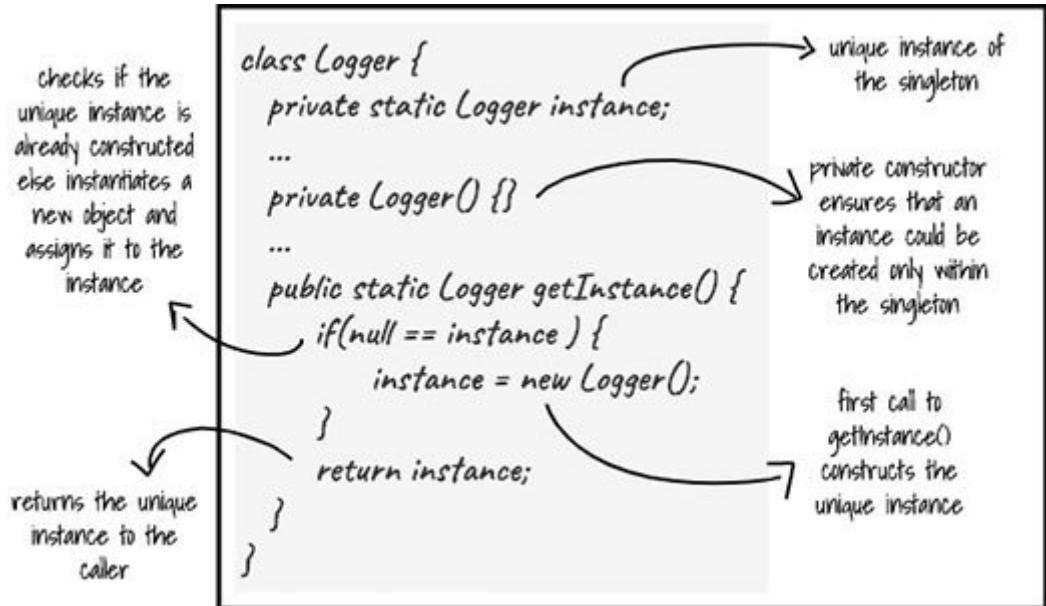
## Singleton design pattern

The singleton design pattern shown in the [Figure 2.8](#) is one of the simplest to learn and understand:



**Figure 2.8: Singleton design pattern**

It describes a singleton with a static instance of itself that is accessible by a static **getInstance** method:



**Figure 2.9: Class Logger updated**

The accessor method takes care of instantiating the instance and returns it to the caller.

In [Figure](#) we have introduced the concept of singleton to our logger class:

```

public class Logger {
    private static Logger instance;
    private Map<String, Writer> writers;
    stores writers for
    individual log files

    private Logger() {
        // read config and populate writers Map
    }

    public static Logger getInstance() {
        if(null == instance) {
            instance = new Logger();
            constructing the
            logger instance if
            it is not already
            constructed
        }
        return instance;
    }

    public void log(String log, String message) {
        log(LogLevel.INFO, logfile, message);
    }

    public void log(LogLevel logLevel, String log, String msg) {
        writers.get(log).append(logLevel.getName())
        .append(message);
        writers.get(log).flush();
    }
}

```

writer specific to the log file is selected  
and used to write message to the log

**Figure 2.10:** Class Logger complete

Instead of instantiating a new instance of the logger in each distinct class, it offers to reuse the same instance. The example is kept simple to demonstrate and focus on the application of the singleton.

The approach is to provide a configuration file that is read before constructing the singleton object. The configuration provides the information related to the log files that the application will use, Writer objects for all these log files are constructed and their references are kept in a map to support logging operations.

Let us make changes to our existing controller class to utilize our new singleton object and log some messages. The redesigned **Logger** class is a small and effective solution for immediate logging requirements, but it isn't the only one and definitely not the best, as shown in [Figure](#)

```
class UserAccessController {  
    private final String log = "access_control";  
    private Logger logger = Logger.getInstance();  
    ...  
    public boolean isAccessGranted(User user) {  
        ...  
        logger.log(log, "Access Granted" + user.getUserId());  
        ...  
    }  
}
```

accessing our new and improved, singleton version of the Logger class

calling log method to log information

the passed string will be used to select the appropriate log file and write the message to it

**Figure 2.11:** User access controller

## Variations

There are many different ways in which a singleton could be constructed; though the basic design remains the same, the algorithm could be modified to utilize mechanisms such as eager and lazy initialization or double check locking and even enums.

## Lazy initialization

The original implementation of singleton utilizes lazy initialization as described in our previous example of the **Logger** class. Refer [Figure](#)

```
class Singleton {  
    private static Singleton instance;  
    ...  
    private Singleton() {}  
    ...  
    public static Singleton get() {  
        if(null == instance) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

because of lazy instantiation,  
the first call to get() takes  
time if the object in question  
depends on system resources  
or requires processing before  
instance creation

the get() lazily  
instantiates the instance

the instance is  
constructed only when  
it is required i.e. only  
when the get() is  
called

**Figure 2.12:** Lazy Singleton

With this implementation, the singleton instances are constructed only if and when they are required, that is, on the very first call to access the singleton instance.

Since the instance is constructed when the first call to access it is received, the instantiation process could take some time and a delay could be experienced. This delay, however, is only for the first time and the instance would be immediately returned in the subsequent calls.

### Multiple instances of a singleton

The above example works well in a single threaded environment, while it could break under a multithreaded one. With a single thread accessing the singleton, there is little or no worry of accidentally constructing multiple instances of the same ‘singleton’ object. When multiple threads are active, then two or more threads can simultaneously access and construct multiple instances breaking the concept of a singleton and returning different objects of that kind instead of one as it should be.

Think of a singleton which apart from doing its primary job also counts the number of requests it received for future analysis. If there would be more than one instance of this singleton, the data would be incorrect and so will the analysis. In [Figure](#) multiple threads are accessing the singleton simultaneously:

example denotes time with  $X$  and thread with  $T$

at time  $X_1$ , both  $T_1$  and  $T_2$  are at the same position, by time  $X_2$ ,  $T_1$  is ready to construct the instance but before  $T_1$  could return at time  $X_3$ ,  $T_2$  has entered the if check and is ready to construct its own instance which it will return in  $X_4$

```
class Singleton {  
    private static Singleton _ins;  
    ...  
    private Singleton() {}  
    ...  
    public static Singleton get() {  
        X1T1 if(null == _ins) { X1T2  
            X2T1     _ins = new Singleton(); X3T2  
        }  
        X3T1     return _ins; X4T2  
    }  
}
```

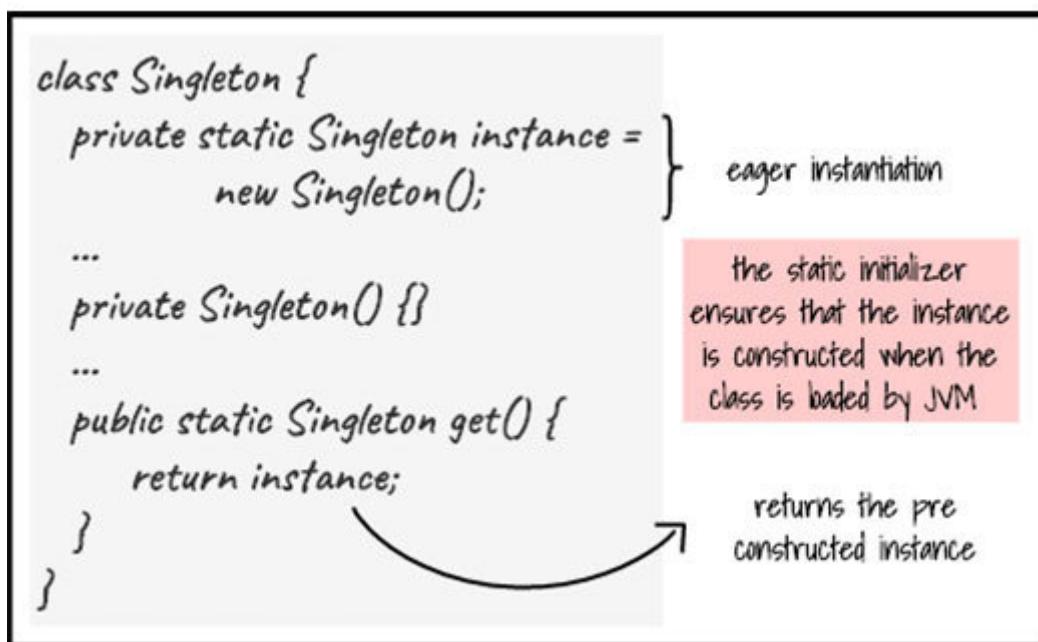
**Figure 2.13:** Multiple instances of Singleton

For any multithreaded application, in absence of a mechanism to restrict multiple threads from accessing same block of code simultaneously, multiple threads can access that same block at one time.

To conform to the principles of singleton and to ensure that multiple instances of a singleton object could not be instantiated within the same application, other variations of singleton design pattern can be utilized. There are multiple ways to construct a singleton object and all of them provide a different flavor to the construction process. We will discuss all these variants of singleton design pattern in the next pages.

## Eager initialization

With eager instantiation the object is constructed when the class is initially loaded by the java class loader. The Eager Initialization process rectifies the multiple instance problem since there is no way for more than one thread in an application to access the construction logic simultaneously. This is made possible because of static initializer that ensures the construction of instance when the class is loaded by the JVM, as shown in [Figure](#)



**Figure 2.14:** Eager initialization

The drawback of eager initialization is that the instance is always constructed irrespective of whether it will be used in runtime or not. What if the constructor of the singleton class throws an

exception? Will the application be able to recover from the exception?

The singleton object that we want to construct could throw an exception during initialization and halt execution of the application. Though we cannot complete the object construction if an exception is thrown, we can handle the exception in a desired way, if required, as shown in [Figure](#)

```
class Singleton {  
    private static Singleton instance;  
    ...  
    static {  
        try {  
            instance = new Singleton();  
        } catch(Exception e) {  
            // handle exception  
        }  
    }  
    ...  
    private Singleton() {  
        throw new RuntimeException()  
    }  
    ...  
    public static Singleton get() {  
        return instance;  
    }  
}
```

static block provides a way to support exception handling while eagerly instantiating the object instance which is not possible with inline instantiation

handle exception either by throwing it further up the stack or by consuming it

similar to inline static initiation, with static blocks the JVM ensures that the object instance is constructed only once during application startup

**Figure 2.15:** Static block eager initialization

Both these constructs initialize the singleton during class loading by the JVM and safeguard from multiple instance creation.

## Thread safe singleton

A thread safe singleton shown in the [Figure 2.16](#) makes use of synchronization to allow sequential access to threads, thus ensuring that only one instance of the singleton gets constructed, and all other threads get access to the already initialized instance of the singleton:

The diagram shows a code snippet for a `Singleton` class. The code includes a `synchronized` method `get()` which performs `lazy instantiation`. A callout points from the `get()` method's declaration to the `instance` variable.

```
class Singleton {  
    private static Singleton instance;  
    ...  
    private Singleton() {}  
    ...  
    public static synchronized Singleton get() {  
        if(null == instance) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

marking the method synchronized could have some performance issues since every call to the get() method will have to wait for the other thread finishes

**Figure 2.16: Thread safe singleton**

The thread safe singleton has a drawback when it comes to performance. Although making the accessor method synchronized serves the purpose of ensuring sequential access to threads, it

also restricts or slows down the process by limiting access to the complete method when only a specific block could be controlled by synchronization. [Figure 2.17](#) demonstrates the use of double check locking to improve the implementation of thread safe singleton:

The diagram illustrates the Double Check Locking (DCL) pattern for a Singleton class. It features a code snippet with annotations explaining its behavior.

```
class Singleton {  
    private static Singleton instance;  
    private Singleton() {}  
    ...  
    public static Singleton get() {  
        if(null == instance)  
            synchronized(Singleton.class) {  
                if(null == instance)  
                    instance = new Singleton();  
            }  
        return instance;  
    }  
}
```

**Annotations:**

- Red Box:** multiple threads could possibly enter the get() at the same time, double check locking rechecks for the null condition inside the synchronized block to ensure no new instance is instantiated if the object has already been constructed
- Yellow Box:** the first check speeds up performance by limiting access to the synchronized block
- Yellow Box:** the synchronized block is a trimmed down version of the thread safe singleton we discussed earlier
- Yellow Box:** synchronized block is required only once, when the object is first instantiated, this approach skips the sync block once the object is instantiated unlike the synchronized method approach

**Figure 2.17:** Thread safe singleton | Double check locking

By using the synchronized block, we can limit the amount of code that needs synchronization and reduce the wait time for other threads. One important point that requires mention here is the use of first null check that limits access to the synchronized block itself if the instance has already been constructed.

In the previous variants, where the whole method was synchronized every thread passes through the entire method

keeping the other threads waiting for access, but by using double check locking that is no more required.

## Reflection meets singleton

Reflection API in Java can be utilized to inspect and modify attributes of interfaces, classes, fields, and methods. We can construct instances of a type, invoke methods, and set field values in runtime. Reflection gives access to class constructors, overloaded methods. All this, and much more can be done in runtime.

Reflection can be used to break down the principle, to have only one instance of a type. In [Figure](#) we have modified our singleton class and added some methods to help us understand if reflection really can create multiple instances of a type even when it is accessible only through a singleton:

```

class Singleton {
    private static Singleton instance = new Singleton();
    private AtomicLong count;
}

private Singleton() {
    count = new AtomicLong();
}

private void incrementCount() {
    count.incrementAndGet();
}

public long getCount() {
    return count.get();
}

public static Singleton get() {
    return instance;
}

```

eager initialization

we have introduced a counter in our Singleton to check which instance is receiving the calls

AtomicLong is a thread-safe class to manipulate long value. It is a part of java.util.concurrent package

AtomicLong ensures that the wrapped value isn't effected by the use of multiple threads

we have picked eager initialization as an example, reflection effects every single variant of Singleton we have discussed so far.

**Figure 2.18:** Reflection and Singleton I

Look at the following example to understand further. In [Figure](#) we have constructed an instance of our singleton class using reflection:

```

class SingletonMeetsReflection {
    public void testReflection() {
        Singleton first = Singleton.get();
        Singleton second = null;
        Constructor[] constructors =
            Singleton.class.getDeclaredConstructors();
        constructors[0].setAccessible(true);
        try {
            second = (Singleton)
                constructors[0].newInstance();
        } catch (Exception e) {
            // handle exception
        }
        first.incrementCount();
        first.incrementCount();
        second.incrementCount();
        print(first.getCount());
        print(second.getCount());
    }
}

```

initializing an instance of our Singleton the default way

returns the constructors declared by our Singleton class

this is where the magic happens. by marking accessibility to true it suppresses the access checks applied by the java language.

increment the value of count variable

although the constructor for Singleton class is private we can still access it here just by setting this flag to true

prints 2 and 1 confirming that reflection constructed a different instance of Singleton

**Figure 2.19: Reflection and Singleton II**

Although the constructor for our class is private, we can still access it by marking it accessible via reflection. This gives us total control over the class and allows us to construct another instance of it ignoring its singleton property.

In the last lines of the example, we print the value of the count variable to confirm whether reflection does break apart the singleton design pattern. Those two lines print 2 and 1 respectively, that is, the number of calls we made with the two instances. The ‘first’ instance was created using the default accessor method, while the ‘second’ one is constructed using reflection. They should have printed 3 and 3 if the principle of singleton would have held.

## Enum as singleton

While reflection can break apart all the previously discussed variants of singleton, there is still a way to ensure that the singleton object remains unique:

```
enum Singleton {  
    INSTANCE;  
  
    private Singleton() {  
        // initialize instance  
    }  
  
    public void do() {  
        // execute operation  
    }  
}
```

- JVM ensures that enum values are instantiated only once in the application
- enums are globally accessible
- enums can have methods
- an enum cannot be constructed reflectively

```
public T newInstance(Object ... initargs) {  
  
    if ((clazz.getModifiers() & Modifier.ENUM) != 0)  
        throw new IllegalArgumentException("Cannot  
reflectively create enum objects");  
  
    ...  
}
```

**Figure 2.20:** Enum and Singleton

By using enum, one can ensure that the singleton object is unique in the entire application. In [Figure](#) the code block on the right side is taken from the **java.lang.reflection.Constructor** class, which explicitly mentions that an enum cannot be constructed reflectively.

## Singleton and serialization

Serialization is the process of converting the state of a Java object into byte stream, that could be stored in file system and later read and reverted into a copy of the serialized object. The process of converting the serialized form into a copy of the object is called deserialization.

An object is serializable if the object itself, or any of its parents implement the serializable interface or a child of the serializable interface, for example, Externalizable.

For a singleton to support serialization, it must implement the serializable interface as depicted in the [Figure](#)

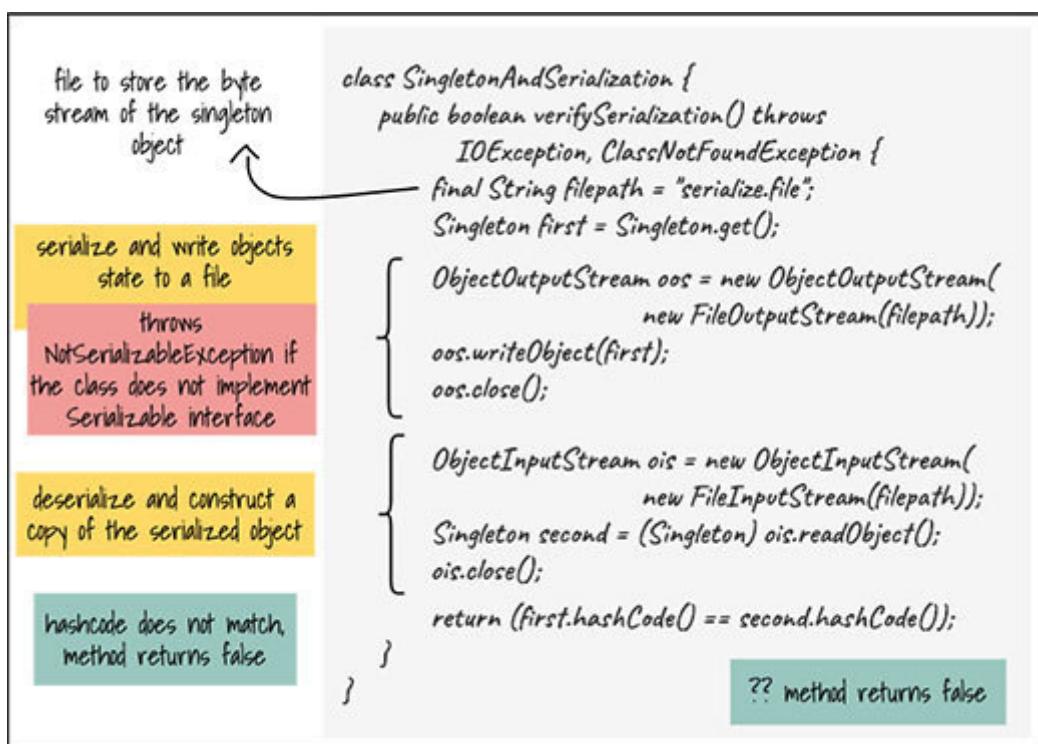
```
class Singleton implements Serializable {  
    ...  
}  
the serialization process throws NotSerializableException if the  
class does not implement the Serializable interface
```

**Figure 2.21:** Serialized Singleton

If the class is not serializable, that is, neither does the class nor does any of its parent implement serializable, then the serialization process throws **NotSerializableException** when the object is written to an output stream.

To understand how an object can be first serialized and then deserialized to regain the state of the original object, look at the following example:

In the [Figure](#) we first serialize our singleton object, save it in the filesystem and then read it from there to construct a copy of the original object that we serialized earlier:



**Figure 2.22: Singleton and Serialization**

Why does the deserialized object not match the original one? Isn't implementing `Serializable`, the only thing to do?

Deserialization constructs a copy of the original object; it is not the same object that we serialized. To preserve the state of the object, we must add some additional methods to the class that

are called during deserialization. This is the reason why the hashCode does not match in our example.

Let us update our class to construct an equivalent object during deserialization process. After making this change, the hash code of the original and the reconstructed object should match.

By adding the implementation of the **readResolve** method, in this manner, we can preserve the state of our singleton and ensure it is not broken during serialization. This is demonstrated in [Figure](#)

```
class Singleton implements Serializable {  
    private static Singleton instance =  
        new Singleton();  
    private Singleton() {}  
    ...  
    public static Singleton get() {  
        return instance;  
    }  
    protected Object readResolve() {  
        return get();  
    }  
}
```

we have added the  
readResolve() to support  
deserialization, rest of the  
class remains the same

readResolve() is called during  
deserialization to configure the  
state of the re-constructed  
object

the class is now Serializable

**Figure 2.23:** Read Resolve | Singleton and Reflection

Enums are inherently serializable and there is no need to add **readResolve** to them. All the enums by default implement the serializable interface and can be serialized and deserialized in the same way, described earlier in this chapter.

## Conclusion

Singleton design pattern is one of the highly used patterns and is easy to understand and program. It is widely used in scenarios that require a single instance throughout the application. Loggers and service locators are some of the use cases where a singleton is acceptable.

There are some concrete requirements that advocate introduction of singletons, which are as follows:

There can only be one unique instance of that type

It should be accessible from anywhere within the application

It should support concurrent access to the resources it safeguards

In this chapter, we discussed the basics of singleton design pattern, different ways it can be constructed and how it could be easily broken apart. We also discussed the remedies to safeguard the singleton from such breaks.

In the next chapter, we will discuss another creational design pattern, the builder design pattern.

## Questions

In this section, we will sharpen our brain by indulging in some fun brain activities. Let's get started.

## Pick the odd one out

Lazy Initialization	Eager Initialization	Serializable
Thread Safe Singleton	Enum as Singleton	readResolve()
Eager Initialization	Lazy Initialization	Enum
Double Check Locking	Thread Safe Singleton	NotSerializableException

*Figure 2.24: Pick the odd one out*

### Find the odd one out

Enum as Singleton:

Is thread-safe

Allows lazy initialization

Can be constructed using reflection

Do not support serialization

Singletons:

Must support multi-thread applications

Are globally accessible

Can have multiple instances in an application

Are one-of-a-kind objects

## Complete the code

```
class Singleton {  
    Singleton() {}  
    ...  
    public static {  
        synchronized( ) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}  
  
if(null == instance)  
    Singleton.class  
Singleton get()  
private static Singleton instance;  
if(null == instance)  
    private
```

```
class SingletonAndSerialization {  
    public boolean verifySerialization() throws  
        final String filepath = "serialize.file";  
        Singleton first =  
        ObjectOutputStream oos =  
            new ObjectOutputStream(  
                new FileOutputStream(filepath));  
            oos.  
            oos.close();  
  
        ObjectInputStream ois =  
            new ObjectInputStream(  
                );  
        Singleton second =  
            (Singleton) ois.  
            ois.close();  
        return  
    }  
  
(first.hashCode() == second.hashCode());  
readObject();  
new FileOutputStream(filepath)  
FileOutputStream  
writeObject(first);  
IOException, ClassNotFoundException {  
    Singleton.get();
```

**Figure 2.25:** Complete the code



Select two correct options

Eager Initialization:

Is thread-safe

Cannot be broken via reflection

Can be done via static blocks

Supports lazy loading

Reflection can be used to:

Check object inheritance

Invoke methods

Construct the instance of an enum

Change field types

Serialization:

Is a method to convert object into byte stream

Is a method to convert byte stream into an object

Is supported by enum

Can be done without implementing Serializable

What does together

Enum, serialization, eager initialization

Thread-safe, enum, lazy loading

Support concurrency, unique, globally accessible

Lazy Initialization, static blocks, enum

## Answers

Pick the odd one out

Image [Column Wise]

Thread Safe Singleton, Enum as Singleton, Enum

## Find the odd one out

is thread-safe

can have multiple instances in an application

Select two correct options

is thread-safe, can be done via static blocks

invoke methods, construct instance of an enum

is a method to convert object into byte stream, is supported by enum

[Enum, serialization, eager initialization] and [Support concurrency, unique, globally accessible]

## CHAPTER 3

### Object Factory

---

You can find the editable code files for this chapter on the book's GitHub link.

<https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers/tree/main/Chapter%2003>

---

## Structure

Topics that make this journey worthwhile:

Introduction

Interacting with interfaces

The simple factory

Factory method pattern

Dependency injection

Abstract factory pattern

## Objective

Our objective is to understand the use of abstraction and polymorphism to reduce coupling. We will learn about the factory design patterns and the application of polymorphism to bring them to life. We will learn about the simple factory, the factory method, and the abstract factory pattern.

We will also walk you through the implementation details of these three design patterns.

## Introduction

Abstraction plays an important role in the way objects interact. It provides the feasibility for introducing changes into the application without affecting the existing structure in a significant manner. Abstraction is a key component for a loosely coupled, low maintenance and adaptive architecture.

Abstraction can be achieved by using abstract classes or interfaces. Coding to interfaces or abstract classes reduces coupling among classes, which allows for dynamic polymorphism and enables runtime injection of concrete instances to their parent type references. With abstraction in place, new implementations can be easily introduced into the existing system with minimal changes.

In this chapter, we will go through examples that demonstrate the factory design pattern and the role of abstraction and polymorphism in it.

## Interacting with interfaces

Objects are the basic building blocks in an object-oriented programming language. They store data, interact with each other and process information. Interaction between objects is an essential part of the application. It is hard to think of an application that has a single object type with no interaction requirements. The way the objects interact becomes more important when the applications grow in size and introduce new type of objects. These new object types, when incorporated into the existing codebase require changes, that might be few or many depending on the way the objects interact. If the objects are tightly coupled to each other, they are hard to separate and often require a great deal of effort.

The way the objects are coupled together defines the capability of the application to adapt to changes. Tight coupling signifies interaction among or with concrete implementations. As mentioned earlier, tight coupling should be avoided to minimize the effort required to introduce changes into the application:

```
public Response prepareResponse() {  
    Response resp = new Success();  
    return resp;  
}
```

interaction with concrete implementation

### ***Figure 3.1: Interacting with Interfaces***

The code snippet demonstrated in [Figure 3.1](#) has a tight bond between the reference of type Response and its concrete implementation Success. Though it works fine, in the long run, this might be a cause of trouble when more concrete implementations are added to the application. Coding to implementation binds classes together and restricts the ability for dynamic coupling. This also means that the code is never closed for modification and will have to be reopened to incorporate changes.

An obvious question must have ventured your mind by now. Is there a way to reduce the dependency among objects? Is it possible to introduce change with minimum effect on the existing codebase? Well, there is. Coding to interfaces keeps the classes loosely coupled and unaware of their connection with other concrete classes until runtime when polymorphism comes into action and interface reference is substituted with a concrete subtype. Interfaces provide the desired abstraction and polymorphism allow concrete implementations to substitute an interface type reference with little or no change in the existing codebase.

We cannot do away with coding to implementations entirely. Eventually, there will be direct interactions with the concrete types. Such interactions can be reduced by coding to interfaces and utilizing polymorphism, leaving less code open for modification. This also means that new concrete implementations that

implement the desired interfaces can be easily incorporated into the application without much change.

## Building factories

The example that we will walk you through in this chapter is of an existing *Report Generation system* that prepare reports for a vast variety of use cases; reports that are generated after processing input data in near real time and that can be downloaded via a browser. With increase in traffic and the amount of data that should be processed, the system is under stress and the users are experiencing longer processing time and abrupt, erroneous behavior.

As a solution to this problem, the management has decided to redesign the system and introduce scheduled processing. The users will now be provided an option to schedule their reports. They can visit the application and initiate the report generation process with an additional attribute that will schedule the report for processing later period. Users will be notified of the process completion accordingly.

Our task is to build a module, that is behind a queuing mechanism, that should subscribe to messages from the queue and generate appropriate response after reading those messages, as per the configuration made by the user. The module looks a bit fat; hence, we will focus on the part that prepares the response and will leave the subscription handling and other tasks for now.

Let us start with a simple response generator and then systematically improve it so that it could be generic enough to produce responses for multiple use cases.

[Figure 3.2](#) demonstrates the response generator class that prepares one of the many response types, that is, Success. Response is an abstract class that provides steps to prepare a response, in the form of methods. It is also the parent class for the concrete type, Success.

You might be wondering why we need the response class when only a single type of response is being generated. We could simply have returned an instance of the type Success instead. To answer that, imagine other types of response being added to the application at a later stage. Creating methods for every single response type does not seem to be a good idea, is it? Instead, if we could have a parent type that is extended by all the different types of response, then returning a polymorphic reference from the method will be sufficient for all response types. Response class provides us the needed abstraction and allows for polymorphic substitution. It might not be making much sense for now, but will be of importance when we add more response types to the application:

```

public class ResponseGenerator {
    public Response prepareResponse(ProcessResult result) {
        Response resp = new Success(result);
        resp.read();
        resp.process();
        resp.format();
        return resp;
    }
}

```

being farsighted, developers introduced polymorphic references early on for easy addition of new types of response

steps to prepare response

a type of response

Response is an abstract class responsible for providing steps to prepare the final response

**Figure 3.2:** Response Generator

A response generator that always returns success looks rather lame, let us add some more response types to make it look smart and close to reality.

For starters, let us add a response type for failure as well. This should not require many changes in the code. In [Figure](#) the method now returns two different types of response, success, and failure.

switch statement  
to select among  
possible responses  
based on value of  
the code

Success and  
Failure both are  
concrete  
implementations of  
Response

return the  
constructed  
response

```
public class ResponseGenerator {  
    public Response prepareResponse(  
        ProcessResult result, String code) {  
        Response resp = null;  
        switch(code) {  
            case "200":  
                resp = new Success(result);  
                break;  
            case "400":  
                resp = new Failure(result);  
                break;  
        }  
        resp.read();  
        resp.process();  
        resp.format();  
        return resp;  
    }  
}
```

our method  
now takes a  
parameter to  
decide the  
type of  
response to  
construct

we have added  
another  
concrete  
implementation  
to the  
codebase

polymorphism in action, the actual  
method is decided in runtime based  
on the type of concrete  
implementation

prepareResponse() is open for modification, with another implementation we will  
have to reopen the method to make changes

**Figure 3.3:** Response Generator II

The method is modified to take a string as an input parameter to decide the type of response. The switch case directive selects the correct response type to be constructed, and then finally, the methods are called to prepare the response.

Every time there is a need to add another type of response, the response generation class should be revisited to introduce the changes. In short, the class is open for modification. A class that

is open for modification is vulnerable to bugs and needs more effort to test. To ensure correct and expected behavior, the whole class must be re-verified and not just the changes. As a guiding principle, a class should be open for extension but closed for modification. [Figure 3.4](#) answers some of the questions related to our response generator:

**Question:** What if we shift the code, that is vulnerable to change, to some other class?

**Answer:** By moving the vulnerable code to another class we can restrict the changes to the target class and close the current class for changes. This also helps us to follow the principle "classes should be open for extension but closed for modification"

**Question:** Will the Response Generator class be truly closed to changes? What if a new step is added to prepare response?

**Answer:** No. If a new step is added to prepare response then our ResponseGenerator class will have to be reopened and modified as it is still responsible for the response generation logic.

**Question:** Can Failure response have preparation steps different from other response types? Can we call those steps from our ResponseGenerator class?

**Answer:** It is highly unlikely to have different steps for preparing Failure response, though the possibility could not be ruled out. Since the reference type in ResponseGenerator class is Response, only those methods exposed by it are visible to ResponseGenerator. Failure response can have methods for steps specific to its generation that can be called from within the overridden ones to ensure those steps are processed. If a step is common in all the concrete types then only it should be added to the Response class otherwise not.

**Figure 3.4:** Q&A

## Response factory

So, it was decided to that the response generator class should be closed for modification and the logic to construct different types of response should be moved to another class. This new class will be responsible for constructing objects of various response types. [Figure 3.5](#) demonstrates our first impression of a factory:

```
public class ResponseFactory {  
    public Response construct(  
        ProcessResult result, String code) {  
        Response resp = null;  
        switch (code) {  
            case "200":  
                resp = new Success(result);  
                break;  
            case "400":  
                resp = new Failure(result);  
                break;  
            case "100":  
                resp = new InProgress(result);  
                break;  
        }  
        return resp;  
    }  
}
```

we have moved the response object construction logic from ResponseGenerator to ResponseFactory and closed the ResponseGenerator class for further changes until there is a change in the response generation logic itself

we have used switch, you can also use if-then-else

switch doesn't work with string parameter until java v7

a new type of response, InProgress.

ResponseFactory from now on will be responsible for constructing various kinds of response. If there will ever be a new response type it will also be constructed by the ResponseFactory

[Figure 3.5: Response Factory](#)

A factory is essentially responsible for creating objects of a particular type, and in doing so, can make use of other interfaces or classes down the line. A factory provides the necessary abstraction by hiding the creation logic from its user, relieving her from this responsibility. It also provides centralized control over the object creation process that can be utilized for data validation or permission verification or something else entirely, for example, a factory could be a place to decide whether the user has all the necessary permissions to access and use an object, which could be difficult to achieve otherwise.

Our response factory class is a basic implementation of a factory that can construct objects of type response. If in future, there is a need for a new type of response, it should be added to the factory as well. Let us now modify our response generator class and close it for further changes.

[Figure 3.6](#) demonstrates the changes done to the response generator class; it now has a reference to response factory that is responsible for constructing different types of response:

```

public class ResponseGenerator {
    private ResponseFactory factory = new ResponseFactory();
    public Response prepareResponse(ProcessResult result, String code) {
        Response resp = factory.construct(result, code);
        resp.read();
        resp.process();
        resp.format();
        return resp;
    }
}

```

the construction logic is moved to a new class  
 ResponseFactory. ResponseGenerator class is closed for changes

prepareResponse() still takes care of response creation by reading input,  
 processing it and finally formatting it in the desired format



prepareResponse()  
 now calls  
 ResponseFactory for  
 construction of  
 response object

**Figure 3.6: Response Generator and Factory**

The response factory class exposes a method called which is called by response generator for object creation. The selection of the type of response is done by passing a parameter to the factory, like it was done in the generator.

In our newly constructed factory class, the decision parameter is of type string that could be passed an invalid value, one which is not expected, failing the construction process. To resolve this, we can restrict the values that can be passed to the **construct** method by creating an enum that contains all the possible values relative to the response types.

Demonstrated in [Figure](#) the enum can be used to restrict the input values that are passed to the **construct** method:

```

public enum ResponseType {
    SUCCESS("Success!"),
    FAILURE("Failure!"),
    TIMEOUT("Time Out!"),
    IN_PROGRESS("In Progress!");
    ResponseType(String msg) {
        this.msg = msg;
    }
    String msg;
    String getMsg() {
        return msg;
    }
}

```

various response types corresponding to the possible value of response code received in processed result object

returns the message as passed in the enum constructor, like 'Success!' or 'Failure!'

ResponseType enum restricts the caller to pass a valid value to the factory for object construction. This safeguards our implementation from invalid inputs

**Figure 3.7:** Response Type Enum

To ensure that our factory is aware of any new response type and behave correctly it should be kept in sync with the enum at all times. [Figure 3.8](#) demonstrates the changes required in the response factory class:

```

public class ResponseFactory {
    public Response construct(ProcessResult result, ResponseType type) {
        Response resp = null;
        switch (type) {
            case SUCCESS:
                resp = new Success(result);
                break;
            case FAILURE:
                resp = new Failure(result);
                break;
            case TIMEOUT:
                resp = new TimeOut(result);
                break;
        }
        return resp;
    }
}

```

String value is replaced with an enum to restrict the possible inputs

care should be taken to update the factory with any new value addition to the enum to ensure correct behaviour

use of enum as input parameter restricts the possible values that can be passed to the factory. It safeguards against construction of invalid response object by limiting the number of values.

**Figure 3.8:** Response Factory II

One thing that we haven't yet discussed is the abstract class `response` and the concrete classes that implement it. Let us get familiar with those classes before we move ahead, they are important to understand the rest of this chapter.

[Figure 3.9](#) demonstrates the abstract response class. It provides the steps, in the form of methods, to prepare a response. The `Message` class stores the information after parsing the input and `response` is the final response that will be sent forward:

```

public abstract class Response {
    protected Message message;
    protected ProcessResult result;
    protected String response;
    protected Response(ProcessResult result) {
        this.result = result;
    }
    public void read() {
        print("read provided information");
        message.extract(result);
    }
    public void process() {
        print("process data before formatting");
    }
    public void format() {
        print("format data to json format");
    }
}

```

Our abstract Response class. It provides all the steps for creation of a response. These steps are arranged in correct order in our ResponseGenerator class. Any new step addition here should be added to the ResponseGenerator class as well.

The Response class can also be made concrete so that a default response can be constructed can be constructed, when there are no specific types

The methods are provided a default definition for use, they could be overridden by the concrete classes to provide a specific implementation

**Figure 3.9: Abstract response class**

Since the class is abstract, we cannot construct an object of its type directly, the class could also be made concrete if a generic response is desired, though not in our case. Notice that the class has default definition for all the methods, these generic definitions can be utilized in the concrete implementations directly or they can override them to provide their own. If you wish that the implementations should always override the methods, then keep the methods abstract and do not provide a default definition.

The order in which these steps should be called is defined in the response generator class that is responsible for preparing the final response. The response factory is utilized to construct the response. The response generator class is the only place where

this arrangement should be maintained to ensure that each response is prepared following the same steps. [Figure 3.10](#) demonstrates the concrete classes that form the actual response:

```
Both the concrete classes extend our abstract Response class and override only those methods that require specific definition.

process method processes the information received in ProcessResult object. It parses, formats and converts data to the desired style of presentation

Change in the method definition changes the way the information is processed by these classes. Polymorphism ensures call to the right implementation.

It is not necessary to override even a single method if the default definitions provide relevant results
```

```
public class Success extends Response {  
    public Success(ProcessResult result) {  
        super(result);  
    }  
    public void process() {  
        print("process success information");  
    }  
}  
  
public class Failure extends Response {  
    public Failure(ProcessResult result) {  
        super(result);  
    }  
    public void process() {  
        print("process failure information");  
    }  
}
```

**Figure 3.10:** Concrete response success

Notice that not all methods are overridden, instead only those that require a different definition are rewritten in the concrete implementations. Since we have already provided default implementation for all the methods in our response class, these definitions are used when a method is not overridden by a concrete implementation. In [Figure](#) even though the class does not override any of the parent methods, it is still a valid implementation:

TimeOut overrides none of the methods from the Response class, instead makes use of the base version of those methods as defined in the Response class itself

```
public class TimeOut extends Response {  
    // no definition, valid implementation  
}
```

**Figure 3.11:** Concrete Response TimeOut

## [Response factory's first run](#)

For all of us who have been patiently waiting, including me, to test our response generation module, we are finally ready to try our hands on our very own response factory and observe it working. Let us create a test class that touches all the aspects of our learning about abstraction, polymorphism, and factories.

Our test class demonstrates the use of response factory that is working behind the scenes to create objects of different concrete types based on the enum value passed. It also demonstrates how polymorphism picks up the correct implementation and call its methods.

In [Figure](#) the test class instantiates a response generator and calls one of its method, passing in the desired input and the type of response it should prepare. It is worth noting that we have passed the enum value explicitly to the method, while it is a part of the input itself, this is deliberately done for clarity and readability:

```

public class ResponseFactoryTester {
    public static void main(String[] args) {
        ResponseGenerator generator = new ResponseGenerator();
        Order order1 = new Order();
        ProcessResult result1 = new ProcessResult("101", order1, "200");
        Order order2 = new Order();
        ProcessResult result2 = new ProcessResult("102", order2, "400");
        Order order3 = new Order();
        ProcessResult result3 = new ProcessResult("103", order3, "100");
        Response resp1 = generator.prepareResponse(result1, result1.response());
        Response resp2 = generator.prepareResponse(result2, result1.response());
        Response resp3 = generator.prepareResponse(result3, result3.response());
    }
}

ProcessResult converts the response code to
the matching ResponseType enum value

```

extract and store data from process result process success information convert response to json format	Success Response
extract and store data from process result process failure information convert response to json format	Failure Response
extract and store data from process result process data before formatting convert response to json format	TimeOut Response

**Figure 3.12:** Response Factory Tester 1

### The Simple factory

The simple factory shown in the Figure though not considered a design pattern, is widely used in applications:

## The Simple Factory

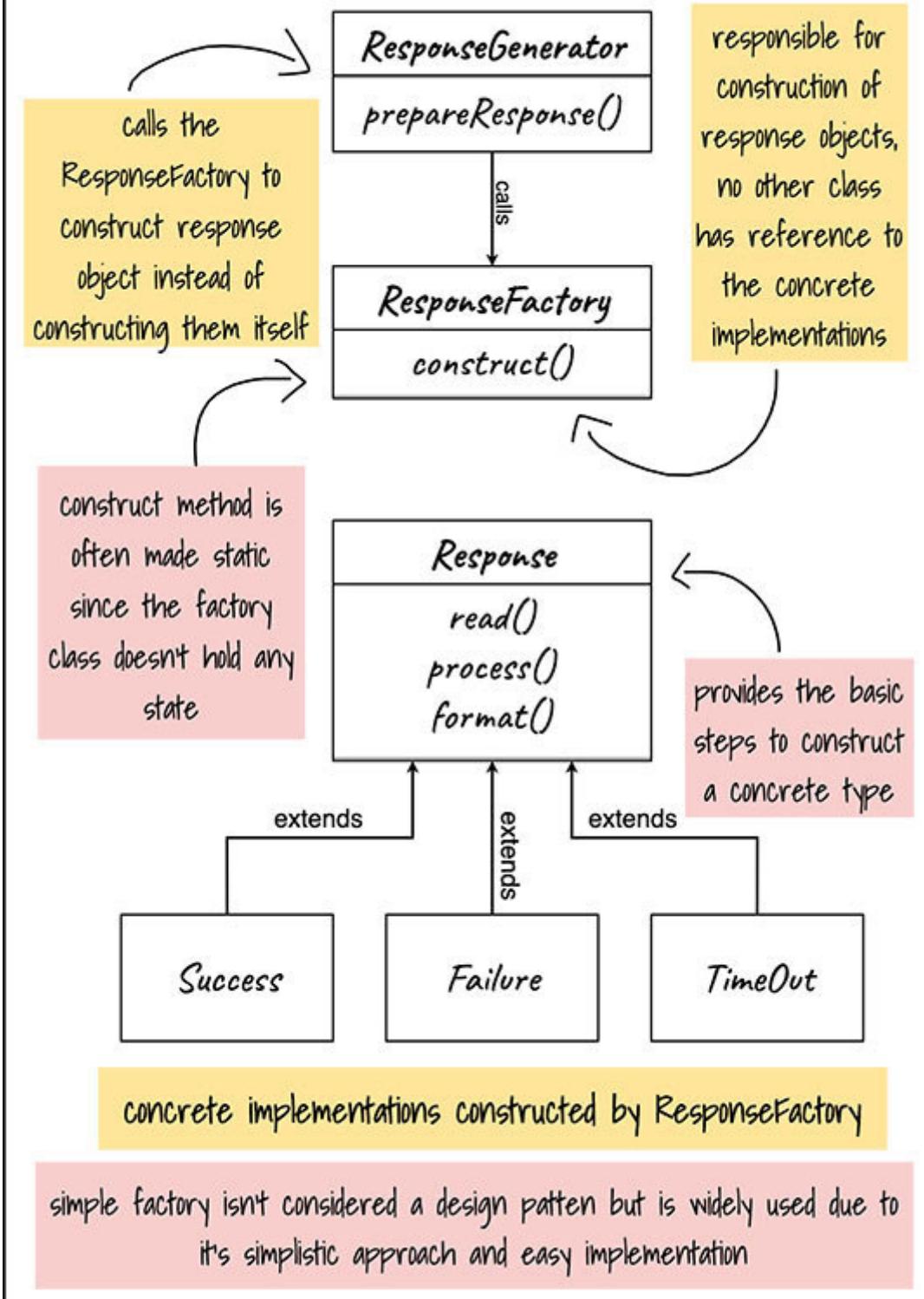


Figure 3.13: The Simple Factory

Due to its simplistic approach and easy implementation, it is often used in scenarios that require a factory like implementation. Not just that, a lot of developers often confuse it with the factory design pattern because of its high resemblance with the actual pattern.

Simple factory and the factory design pattern have some similarities, for example, both make use of abstraction to construct concrete objects utilizing a similar framework. One of the differences they have is that the factory design pattern delegates the responsibility of creation to the concrete types while that is not done in the simple factory.

The simple factory should be a basic version of the factory design pattern, but nevertheless an important one. It teaches us fundamentals of the factory design pattern and prepare us for the underlying complexity of the actual pattern. It should be noted that not all scenarios and use cases exhibit profound complexity and might not require implementing the factory design pattern; for such cases, the simple factory should be considered instead.

Stay focused, we will now step into the territory of factory design patterns. We will discuss two of its variations, the factory method pattern, and the abstract factory pattern. Both follow similar principles and make intelligent use of abstraction to achieve a loosely coupled design:

## Multiple factories

With time, *Report Generation System* witnessed huge success and the management decided to go global. Opening for the world is a huge task at hand and suggestions were invited to improve the application.

There were many features that made it to the final list and one of them was to introduce locale and region-specific changes, which meant a change in the language the reports are generated, as well. This change is necessary since there are many different languages in the world and not everyone would be comfortable in the only language, English, in which the reports are generated today.

With the introduction of multiple languages, our small module that prepares response also required changes. At first, it seemed to be a simple task in hand, our new factory could produce all types of response, all we needed to do was to add some new response types and change our factory implementation to support them. Work started on the same and a new parameter *dialect* was added to the factory to decide the region and ultimately the language in which the response should be prepared.

After a few response type additions, the team realized that the overall complexity of the current factory implementation has

increased and that it would be difficult to manage in the long run.

The development team tossed the idea of creating multiple factory classes and that each factory class should only construct objects of the types of response specific to a language:

```
public class ResponseFactory {  
    public Response construct(ProcessResult  
result, ResponseType type, String dialect) {  
        Response resp = null;  
        switch (dialect) {  
            case 'default':  
                switch (type) {  
                    case SUCCESS:  
                        resp = new Success(result);  
                        break;  
                    case FAILURE:  
                        resp = new Failure(result);  
                        break;  
                    case TIMEOUT:  
                        resp = new TimeOut(result);  
                        break;  
                }  
                break;  
            case 'german':  
                ...  
            case 'french':  
                ...  
        }  
        return resp;  
    }  
}
```

new parameter is used to select a dialect and then based on that dialect the response object is constructed

the ResponseFactory class is highly vulnerable and will remain open for changes keep coming. It could become a potential bottleneck

What if, instead of modifying the only factory class we have, we create more factories to support different dialects

With the addition of new dialect the ResponseFactory class has become more complex than ever before and this complexity will only grow as more and more dialects and response types are added to the system

ResponseFactory class is modified to support new dialects

### **Figure 3.14: Response Factory / Factory Method Pattern**

This meant that for every language a new factory class should be created. The idea was favored by many, but they also had doubts around its placement in the application.

One approach was to add a switch statement to the ‘prepare response’ method of the response generator class, introduce another input parameter and use it to select the correct factory to prepare response. But this would re-introduce the same issues that resulted in creation of a factory in the first place. The class will once again be open for changes. What looked easy earlier started to look difficult to achieve.

The team architect came up with an idea to make the response generator class abstract and introduce a new method for constructing objects of different types of response. The factory classes would then implement the abstract generator class and define the **construct** method to create the response objects. The generator class would still be responsible for preparing the response, but the responsibility of object creation would be transferred to one of its concrete implementations. This also meant that the old response factory implementation would no longer be used.

With this change, multiple factory classes can be created and plugged in to the application design without modifying the existing logic, all possible because of the loosely coupled design and the use of abstraction and polymorphism. As demonstrated in [Figure](#)

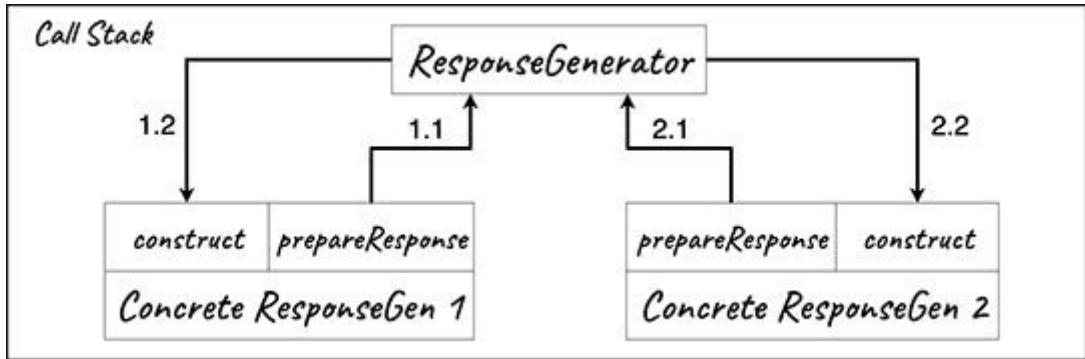
the response generator class is not dependent on any of the factories anymore:

```
public abstract class ResponseGenerator {
    public Response prepareResponse(ProcessResult result,
        ResponseType type) {
        Response resp = construct(result, type);
        resp.read();
        resp.process();
        resp.convert();
        resp.format();
        return resp;
    }
    protected abstract Response construct(ProcessResult
        result, ResponseType type);
}
```

By making ResponseGenerator abstract, the responsibility of response object construction is delegated to the concrete implementations

**Figure 3.15:** Response Generator | Factory Method Pattern

In fact, the concrete factories now extend this class and use it as a base factory. This change has allowed for the development of multiple factory classes, those could be used to prepare response in different languages. [Figure 3.16](#) demonstrates the application of polymorphism with respect to the response generator implementations:



**Figure 3.16:** Polymorphism | Factory Method Pattern

## Factory that speaks French

Let us create new factory classes that implement the abstract response generator class and add some response types. [Figure 3.17](#) demonstrates one of such factories that construct objects of types that prepare response in the French dialect:

```
public class FrenchResponseGenerator extends ResponseGenerator {  
    public Response construct(ProcessResult result, ResponseType type) {  
        Response resp = null;  
        switch (type) {  
            case SUCCESS:  
                resp = new FrenchSuccess(result);  
                break;  
            case FAILURE:  
                resp = new FrenchFailure(result);  
                break;  
            case TIMEOUT:  
                resp = new FrenchTimeOut(result);  
                break;  
        }  
        return resp;  
    }  
}
```

overrides the abstract construct() from ResponseGenerator to provide response selection logic

FrenchResponseGenerator is a concrete factory and with multiple such factories response for multiple dialects can be constructed with their own response generation logic

Nothing much looks changed here, only that the object creation is now handled by our new factory. What exactly happened?

[Figure 3.17: French Response Generator | Factory Method Pattern](#)

This new factory has many similarities with the response factory we created in the previous section. It is essentially the same logic being put in a different class. Notice how the enum is reused without any change; if the existing response factory class is used, multiple new values would have to be added to the enum as well, to ensure that every type of response, from every dialect has a value associated to it.

## Response updated!

The abstract response class is also updated as shown in the [Figure 3.18](#) to incorporate the changes required to prepare a language-specific response:

```
public abstract class Response {  
    protected Message message;  
    protected ProcessResult result;  
    protected Formatter formatter;  
    protected String response;  
  
    protected Response(ProcessResult result) {  
        this.result = result;  
        this.message = new Message();  
        this.formatter = new JsonFormatter();  
    }  
  
    public void read() {  
        message.extract(result);  
    }  
  
    public void process() {  
        print("process data before formatting");  
    }  
  
    public abstract void convert();  
  
    public void format() {  
        print("convert response to json format");  
        formatter.format(response);  
    }  
}
```

ProcessResult is the input for response generation

JsonFormatter is the default formatter

the abstract convert() converts the response from english to other dialects based on the concrete factory selected for response object generation

formatter formats the response to a standard format like json for transfer over the internet

**Figure 3.18: Abstract Response | Factory Method Pattern**

It now has a new abstract method which converts the response from English, the default language, to the required one. [Figure 3.19](#) demonstrates a concrete response type that extends the abstract response class and has a reference to a converter implementation to convert from English to French:

```
public class FrenchSuccess extends Response {  
    Converter converter = new FrenchDialectConverter();  
    public FrenchSuccess(ProcessResult result) {  
        super(result);  
    }  
    public void process() {  
        print("process success information");  
    }  
    public void convert() {  
        print("convert response to french dialect");  
        response = converter.convert(result.getOrder().getNotes());  
    }  
}
```

FrenchSuccess class generates response in french dialect

converting response to french dialect

**Figure 3.19: Concrete Response French Success | Factory Method Pattern**

The convert method is overridden and provides the logic of conversion. There could be as many factories as required, one for each dialect that support all the possible responses the system

could generate, and a different class for each dialect to support it. The updated design allows for better management and easy maintenance of all these concrete implementations.

## Response factory's second run

Let us do a test run to verify the changes we have made so far, and to finally understand the factory method pattern as shown in the [Figure](#)

```
public class ResponseTester {  
    public static void main(String[] args) {  
        Order order1 = new Order();  
        ProcessResult result1 = new ProcessResult("101", order1, "200");  
        Order order2 = new Order();  
        ProcessResult result2 = new ProcessResult("102", order2, "400");  
        Order order3 = new Order();  
        ProcessResult result3 = new ProcessResult("103", order2, "100");  
  
        DefaultResponseGenerator factory = new DefaultResponseGenerator();  
        FrenchResponseGenerator frenchFactory = new FrenchResponseGenerator();  
        Response resp1 = factory.prepareResponse(result1, result1.response());  
        Response resp2 = frenchFactory.prepareResponse(result2, result2.response());  
        Response resp3 = frenchFactory.prepareResponse(result3, result3.response());  
    }  
}
```

Not much has changed in the way the response generation logic is put to motion. Earlier, the response generator referred to a factory internally, now the factory is exposed to the end user.

subclasses decide the type of response to be prepared

generate response in french dialect

extract and store data from process result  
process success information  
conversion not required  
convert response to json format

Default Success Response

extract and store data from process result  
process failure information  
convert response to french dialect  
convert response to json format

French Dialect Failure Response

extract and store data from process result  
process data before formatting  
convert response to french dialect  
convert response to json format

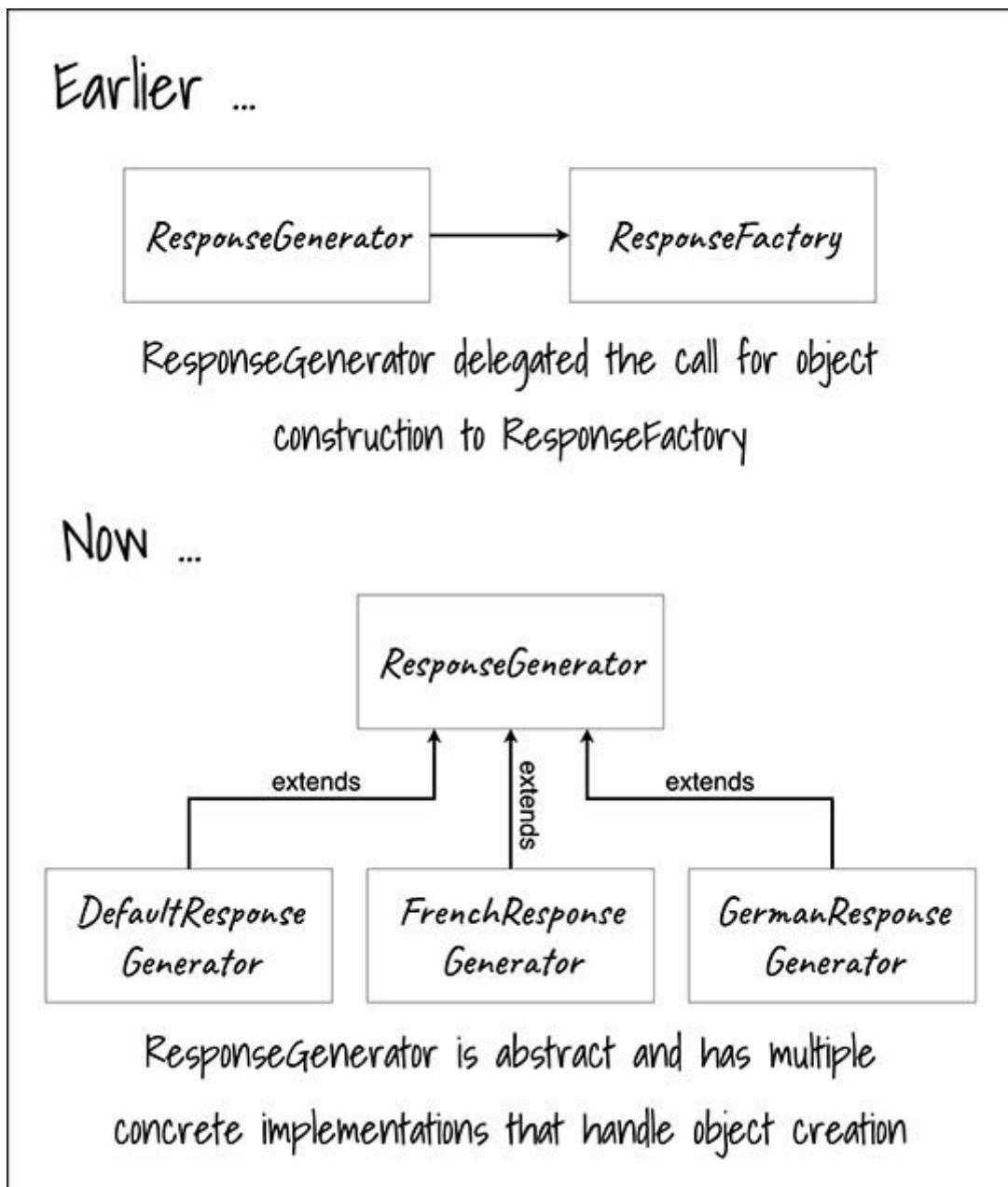
French Dialect Timeout Response

**Figure 3.20: Response Factory Tester II | Factory Method Pattern**

Earlier, the response generator referred to a factory internally, now the factory is exposed to the end user.

## The factory method pattern

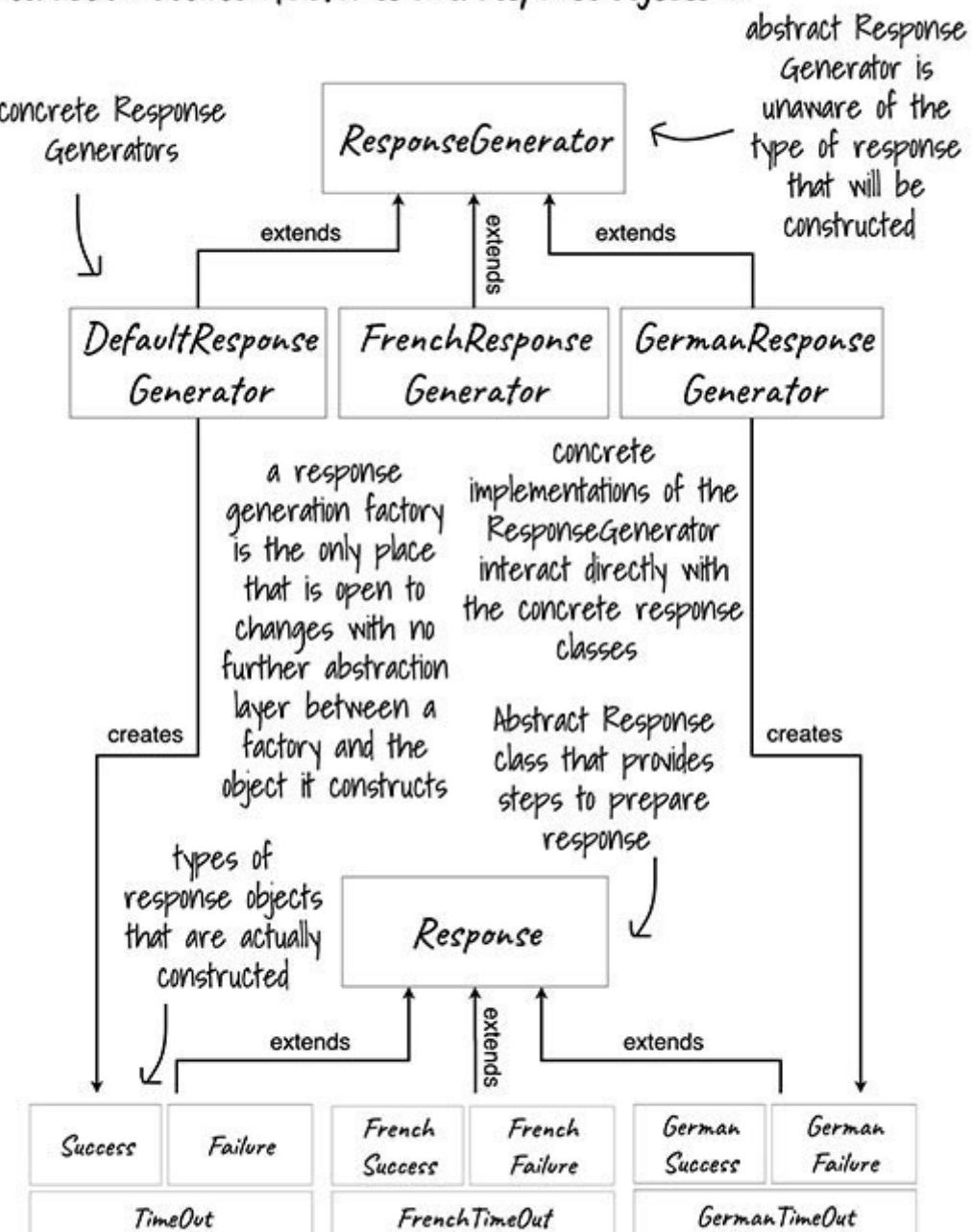
Let us now take a close look at the changes we have made and the working of the new design. In the Figure the first image depicts the changes and the second one depicts the overall working of the design:



***Figure 3.21: Factory Method Pattern I***

[Figure 3.22](#) demonstrates the interaction between the concrete factories and the respective response types:

### Interaction between factories and response objects ...

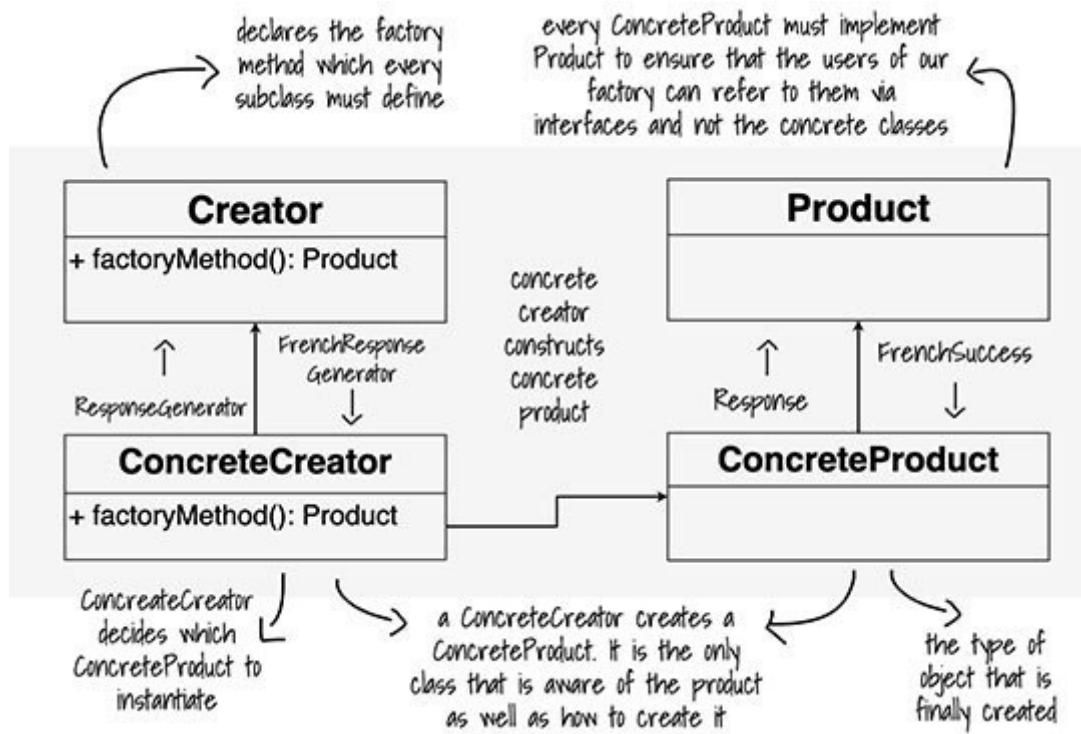


The concrete response generators act as factories in Factory Method Pattern. The responsibility of object creation is delegated to the subclasses.

**Figure 3.22: Factory Method Pattern II**

The Factory Method Pattern as shown in the [Figure 3.23](#) provides an interface to construct an object but delegates the responsibility

of construction to subclasses. The subclasses decide the class that should be instantiated:



**Figure 3.23: Factory Method Pattern Defined**

The **new** operator is deemed harmful as it always constructs an object and there's no encapsulation or abstraction involved. Factory method pattern encapsulates the process of object creation and makes it abstract for the caller; the caller does not have knowledge of the type of object until it is returned.

Factory method pattern can return the same object multiple times, or it can return a subtype of the type of object it creates.

The **creator** class know nothing about the type of product that will be created, it depends on the subclass or concrete implementation

of the creator used.

## Family of objects

The *Report Generation System* finally went global and received good response worldwide. The changes made to prepare reports in the regional languages were much appreciated. Soon, a lot of suggestions came pouring in and some of those specifically mentioned the response generation module. It was observed that the current factories supported only a single way to read, convert and format the data to prepare the response.

If a client asked for XML instead of JSON formatting of the response, a new factory had to be constructed which defeated the purpose of having limited set of factories to prepare the response. With so many factories to control, it will be difficult to continue with the design if the clients keep asking for different ways to process data and form a response:

```
public interface ResponseDialectFactory {  
    public Formatter getFormatter();  
    public Converter getConverter();  
    public Reader getReader();  
}
```

the interface provides a  
way to inject dependencies  
into our concrete response  
classes

any implementation of Formatter, Converter or Reader interface can be injected to the concrete response class to change its behaviour in runtime

by utilizing dependency injection, factories can perform tasks in multiple ways e.g.  
response could be formatted in JSON or XML without any changes in the factory or the  
concrete response classes

**Figure 3.24:** Response Dialect Factory | Abstract Factory

An interface, just like in [Figure](#) could be used to inject the desired type of formatter, converter, and reader into the concrete response classes to make them dynamic in nature and prepare the response in a variety of different ways, something that they do not currently do:

```
public class FrenchDialectFactory implements ResponseDialectFactory {  
    public Formatter getFormatter() {  
        return new XMLFormatter(); ←  
    }  
    public Converter getConverter() {  
        return new GermanDialectConverter();  
    }  
    public Reader getReader() {  
        return new DefaultReader(); ←  
    }  
}
```

implementation class for abstract dialect factory provides concrete implementations that will be injected in the response factory classes

**Figure 3.25:** French Dialect Factory | Abstract Factory

A concrete implementation of this interface could be injected in the concrete response classes:

```

public class GermanDialectFactory implements ResponseDialectFactory {
    public Formatter getFormatter() {
        return new JsonFormatter();
    }
    public Converter getConverter() {
        return new GermanDialectConverter();
    }
    public Reader getReader() {
        return new DefaultReader();
    }
}

```

multiple such dialect factories can be created to support changes in the behaviour of response factory

remember the term 'Dependency Injection'

yet another concrete implementation of the abstract dialect factory with different implementation of Formatter and Converter

**Figure 3.26:** German Dialect Factory | Abstract Factory

[Figure 3.25](#) and [Figure 3.26](#) demonstrates the concreate implementations of the response dialect factory interface.

This will introduce abstraction to the response classes and the final response will then depend on the type of response dialect factory injected into the concrete response class.

In [Figure](#) an updates response class is demonstrated that includes references to reader, converter, and formatter interfaces:

```

public abstract class Response {
    protected Message message;
    protected ProcessResult result;
    protected Formatter formatter;
    protected Converter converter;
    protected Reader reader;
    protected String response;
    protected Response(ProcessResult result) {
        this.result = result;
        this.message = new Message();
    }
    public abstract void setup();
    public void read() {
        message.extract(result);
    }
    public void process() {
        print("process data before formatting");
    }
    public abstract void convert();
    public void format() {
        print("convert response to json format");
    }
}

```

the Response class now has reference to Formatter, Converter and Reader interfaces. They are injected with correct implementation via our new abstract factory implementations

the final response that will be prepared by the concrete factory classes

setup method is where the injection of concrete formatters and converters happen

convert method is now abstract to ensure the language conversion process is implemented by the concrete classes

**Figure 3.27: Abstract Response | Abstract Factory**

These would be used in the concrete response classes to prepare the desired response. A new abstract method **setup** is introduced, which must be implemented by the concrete implementations to configure the references added to the parent response class:

```

public class FrenchSuccess extends Response {
    private ResponseDialectFactory factory;
    public FrenchSuccess(ProcessResult result,
    ResponseDialectFactory factory) {
        super(result);
        this.factory = factory;
    }
    public void setup() {
        print("setup french dialect");
        formatter = factory.getFormatter();
        converter = factory.getConverter();
        reader = factory.getReader();
    }
    public void convert() {
        print("convert response to french dialect");
        converter.convert(result.getOrder().getNotes());
    }
    public void read() {
        this.message = reader.read(result);
    }
    public void format() {
        print("convert response to " + formatter.whichFormat());
        response = formatter.format(result.getOrder().getNotes());
    }
}

```

concrete implementation for the abstract Response class

injected dialect factory

using dialect factory to provide concrete implementation to formatter, converter and reader references

using the injected objects to process response

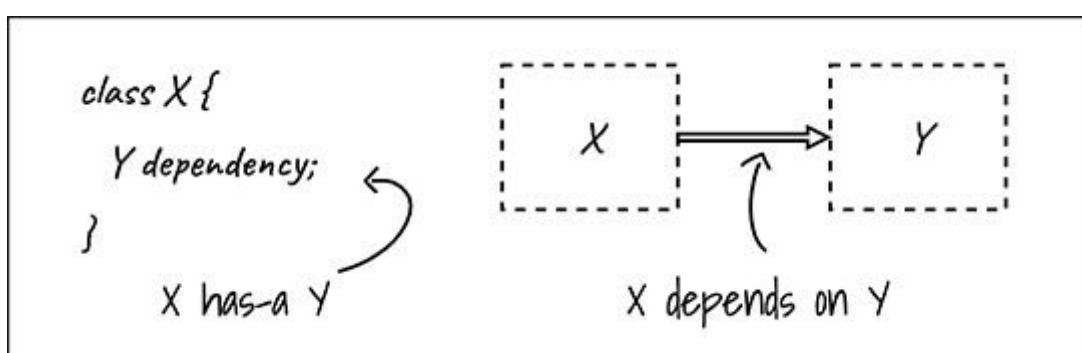
**Figure 3.28:** Concrete Response French Success | Abstract Factory

## Dependency injection

Dependency Injection is a software design concept that involve objects participating in has-a relationship. In this concept, an object is injected into another object that depends on it. If  $X$  depends on technically speaking  $X$  has-a then  $Y$  can be injected into  $X$  in a way such that both the classes are unaware of each other and can be independently developed and maintained,  $Y$  here will be termed as a dependency.

Dependency Injection allows the design to be loosely coupled and configurable. A subtype of the actual type can also be injected in its place following the principles of polymorphism. It also reduces the effort required to test an application.

An object could be injected into another object either through the constructor or through a method of the dependent class. By using dependency injection, the creation and binding of the dependent objects can be moved outside of the class that depends on them. This is demonstrated in [Figure](#)



**Figure 3.29: Dependency Injection | Abstract Factory**

In [Figure](#) the class has an object of a dialect factory that is injected into the concrete response implementation. The response generator instance could itself be injected with an instance of a dialect factory, which could later be passed to the subclasses of the response class:

```
public class FrenchResponseGenerator extends ResponseGenerator {  
    private ResponseDialectFactory dialect = new FrenchDialectFactory();  
    public Response construct(ProcessResult result, ResponseType type) {  
        Response resp = null;  
        switch (type) {  
            case SUCCESS:  
                resp = new FrenchSuccess(result, dialect);  
                break;  
            case FAILURE:  
                resp = new FrenchFailure(result, dialect);  
                break;  
            case TIMEOUT:  
                resp = new FrenchTimeOut(result, dialect);  
                break;  
        }  
        return resp;  
    }  
}
```

The diagram shows a hand-drawn style callout pointing from the line 'concrete dialect factory injected into response classes' to the line 'private ResponseDialectFactory dialect = new FrenchDialectFactory();' in the code. A yellow callout box contains the text 'different dialects can be injected in different response classes if required'.

**Figure 3.30: French Response Generator | Abstract Factory**

## Response factory's third run

Let us do a test run to verify the changes we have made so far, and to finally understand the Abstract Factory Pattern as shown in the [Figure](#)

```
public class ResponseTester {  
    public static void main(String[] args) {  
        Order order1 = new Order();  
        ProcessResult result1 = new ProcessResult("101", order1, "200");  
        Order order2 = new Order();  
        ProcessResult result2 = new ProcessResult("102", order2, "400");  
        Order order3 = new Order();  
        ProcessResult result3 = new ProcessResult("103", order2, "100");  
  
        DefaultResponseGenerator drg = new DefaultResponseGenerator();  
        FrenchResponseGenerator frg = new FrenchResponseGenerator();  
        GermanResponseGenerator grg = new GermanResponseGenerator();  
  
        Response resp1 = drg.prepareResponse(result1, result1.response());  
        Response resp2 = frg.prepareResponse(result2, result2.response());  
        Response resp3 = grg.prepareResponse(result3, result3.response());  
    }  
}
```

setup not required  
extract and store data from process result  
process success information  
conversion not required  
convert response to json format  
  
setup french dialect  
process data before formatting  
convert response to french dialect  
convert response to XML  
  
setup german dialect  
process data before formatting  
convert response to german dialect  
convert response to JSON

Default Success Response

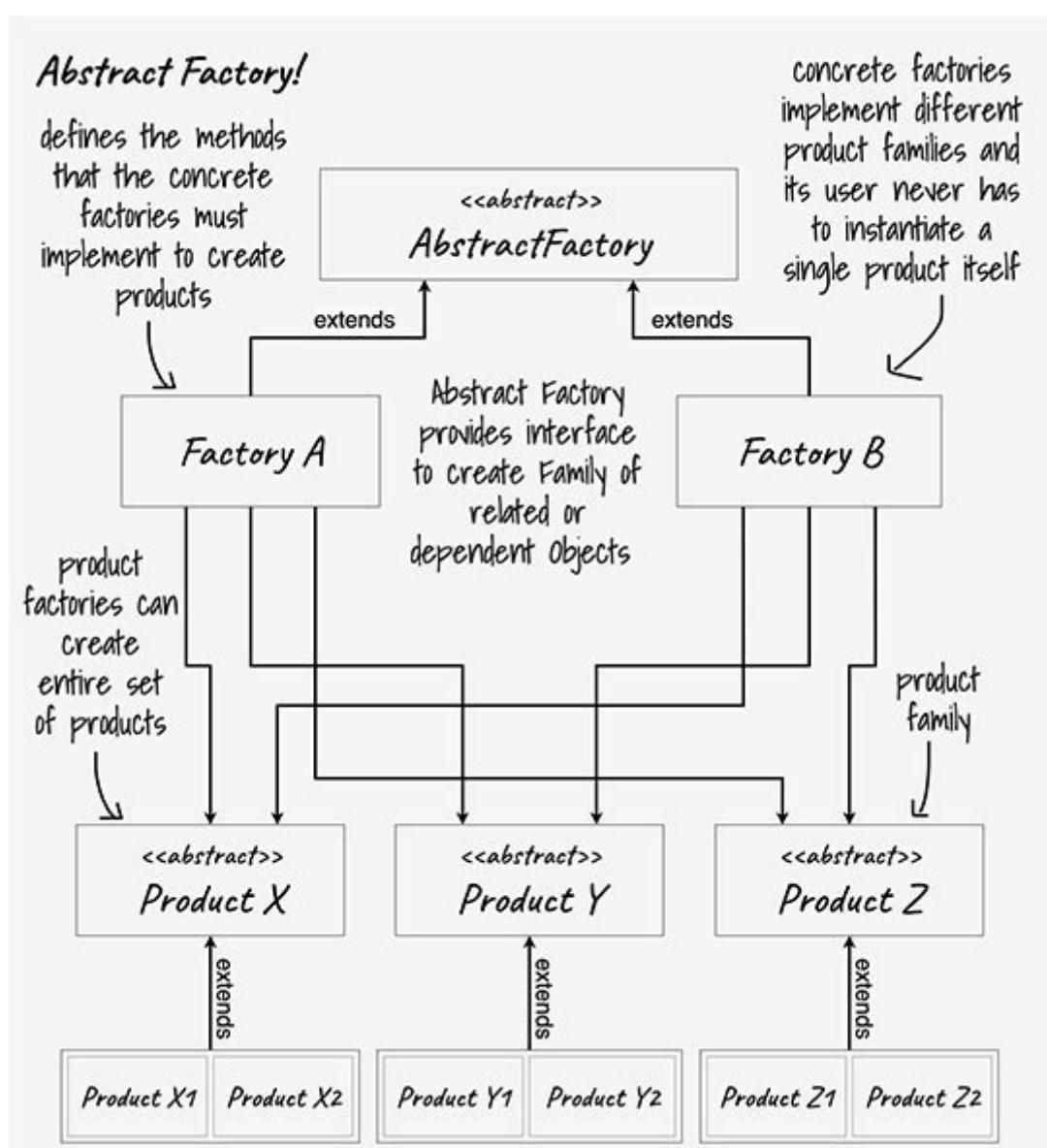
French Dialect Failure Response

German Dialect Timeout Response

**Figure 3.31:** Response Factory Tester III| Abstract Factory

## Abstract factory

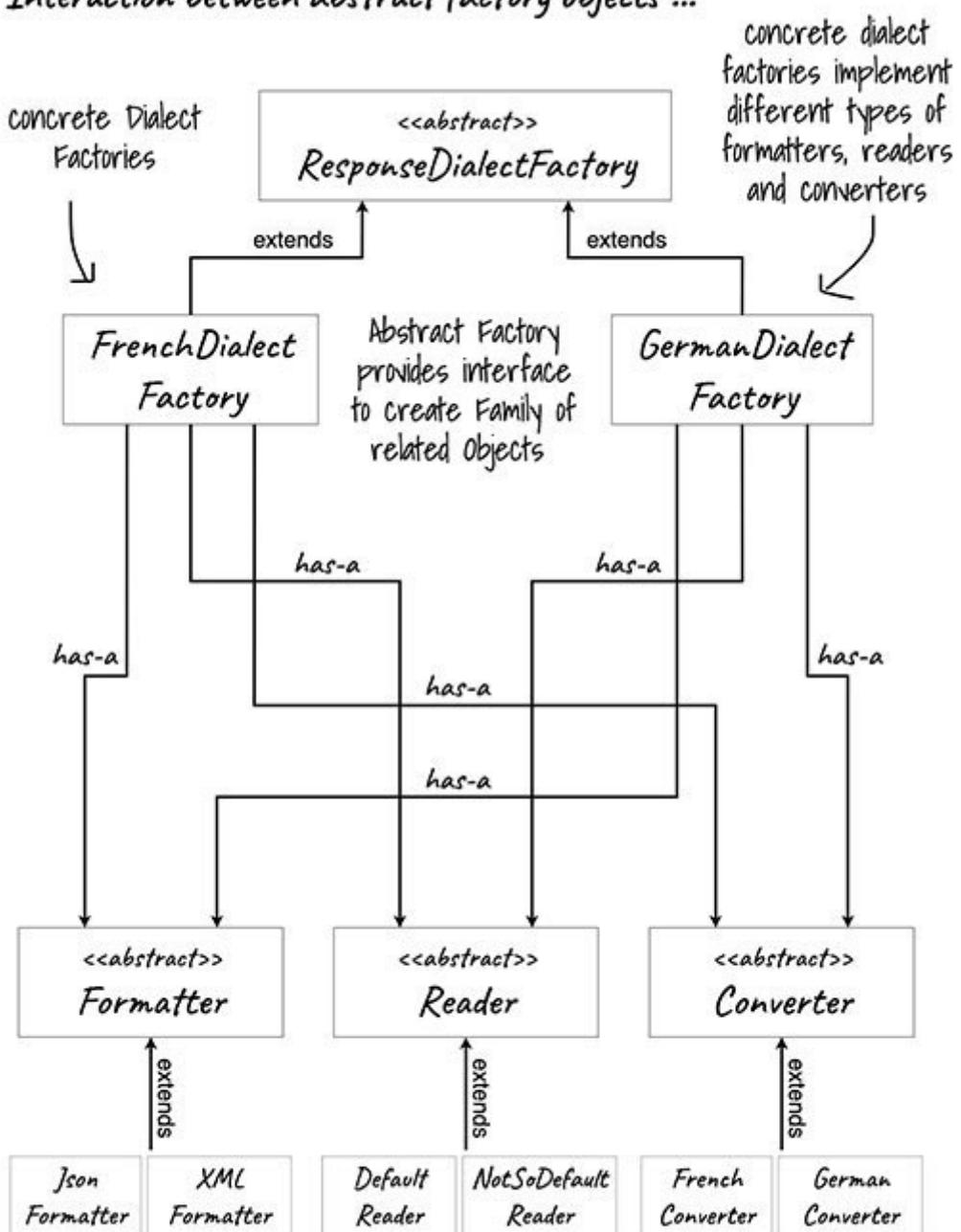
An Abstract Factory shown in the [Figure 3.32](#) provides an interface to create family of interrelated or dependent objects without specifying their concrete classes. The concrete factories implement various product families and relieve the factory to instantiate any of the product objects:



***Figure 3.32: Abstract Factory Defined***

[Figure 3.33](#) demonstrates the abstract factory implementation used in our response generator system:

### Interaction between abstract factory objects ...



subclasses of ResponseDialectFactory are used to process the input data and prepare the response for response generators

Figure 3.33: Abstract Factory Interaction

## Conclusion

Factory design patterns fall under the category of creational patterns. They hide the complexity of object creation from the user by providing an interface that can create objects for them. They encapsulate object creation, promote loose coupling by interacting with abstract types rather than the concrete ones and drive towards a flexible design approach. Factory pattern can also increase the complexity of the code if not used properly and should be avoided if techniques like a simple factory are sufficient.

They advocate the open/close principle, that is, the classes should be closed for modification and open for extension, and the dependency inversion principle, that is, depend on abstraction and not on concrete classes.

The factory method pattern provides an interface for object creation but delegates the responsibility of creation to the subclasses. The abstract factory pattern provides an interface to create family of objects without specifying the concrete classes.

In the next chapter, we will discuss about the builder design pattern and will learn about decoupling object creation from its representation.

## Questions

In this section, we will sharpen our brain by indulging in some fun brain activities. Let's get started...

## Pick the odd one out

Simple Factory	Family of Objects	Abstraction
Abstract Factory	Dependency Injection	Interface
Factory Method	X has-a Y	Encapsulation
Virtual Factory	new keyword	Concrete Class

**Figure 3.34:** Pick the odd one out

### Find the odd one out

Dependency Injection:

Involves two objects

Support polymorphic types

Can be done for unrelated objects

Require composition

Abstract Factory:

Promotes loose coupling

Require inheritance

Creates family of related objects

Can only support a handful of classes

## Complete the code

```
public {  
    protected Message message;  
    protected ProcessResult result;  
    protected formatter;  
    protected Converter converter;  
    reader;  
  
    protected String response;  
    protected Response(ProcessResult result) {  
        this.result = result;  
  
    }  
    public abstract void ();  
    public void read() {  
    }  
    public void process() {  
    }  
    public abstract void convert();  
    public void () {  
        print("convert response to json format");  
    }  
}  
  
print("process data  
before formatting");  
extract(result);  
setup  
  
protected Reader  
print("read input");  
  
Formatter  
print("format data");  
  
abstract class Response  
this.message = new  
Message();  
message.extract(result);  
  
format  
Reader
```

Figure 3.35: Complete the code

Select two correct answers

Factory method pattern:

Delegates the responsibility of object creation to subclasses

Promotes the open/close principle

Does not support subclass types as return values

Can only take one decision parameter as input

Abstract factory:

Can only have static methods

Promotes abstraction

Supports dependency injection

Is a factory of abstract classes

Interfaces

Are the pillars of factory patterns

Can only be implemented by other interfaces

Can be replaced with abstract classes in an abstract factory

Support dependency injection

Loose coupling can be achieved by utilizing:

Interfaces

Subclasses

Tight coupling

Dependency injection

## Answers

Pick the odd one out

Image [Column Wise]

Virtual Factory, new keyword, Encapsulation

## Find the odd one out

is thread-safe

can be done for unrelated objects

Select two correct answers

Delegates the responsibility of object creation to subclasses, promotes the open/close principle

Promotes abstraction, supports dependency injection

Are the pillars of factory patterns, can be replaced with abstract classes in an abstract factory

Interfaces, dependency injection

## CHAPTER 4

### Delegate Object Construction

---

You can find the editable code files for this chapter on the book's GitHub link.

<https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers/tree/main/Chapter%2004>

---

## Structure

Topics that make this journey worthwhile:

Introduction

Delegating object construction

Implementing builders

Builder design pattern – defined

Advantages and drawbacks

Usage

## Objective

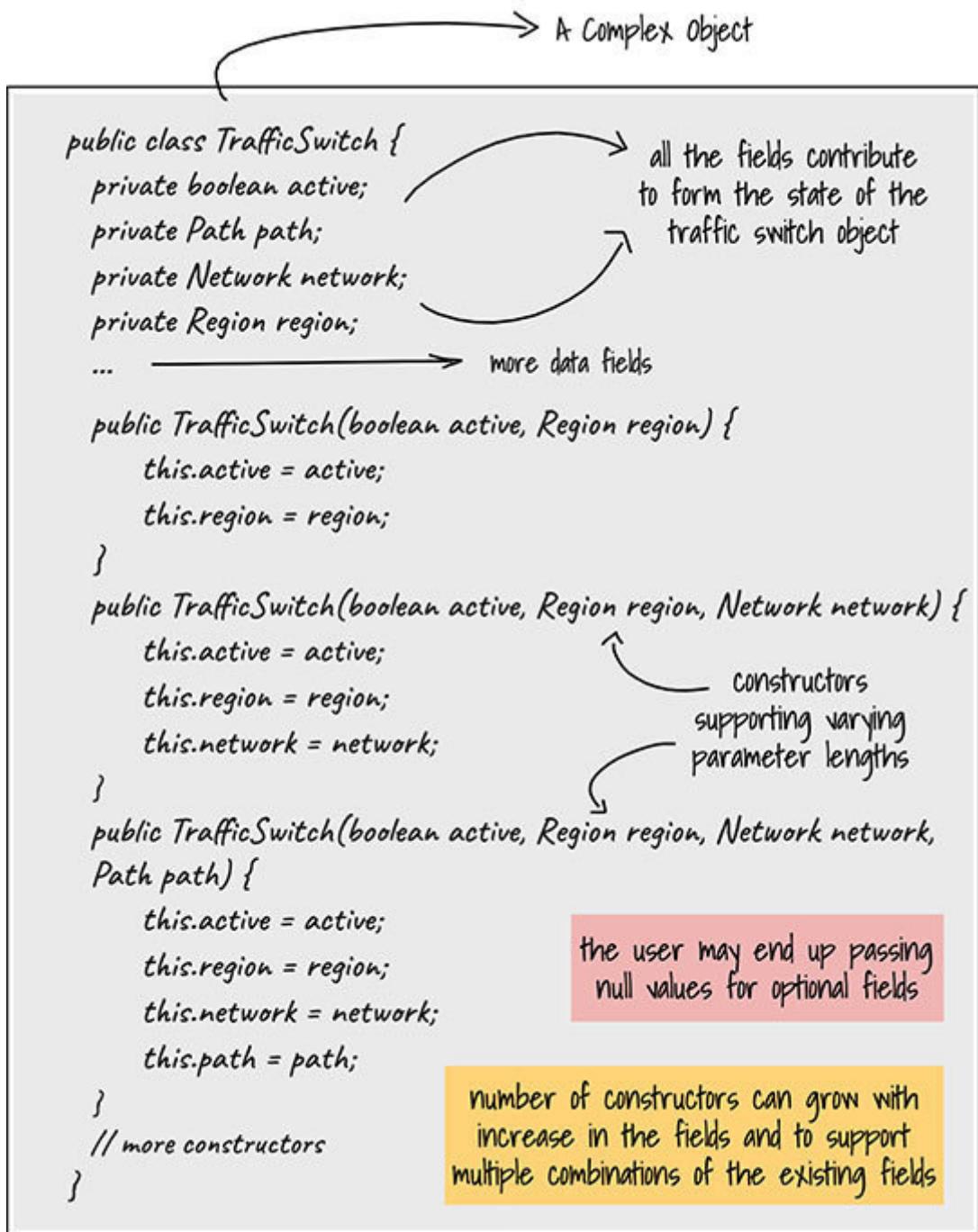
Our objective is to understand a modularized approach to build complex objects that can have multiple representations.

Constructing objects with multiple data fields using constructors can be error prone and lead to exceptions if the mandatory data members are not provided any value before using that object. The builder design pattern can be utilized to overcome this problem by providing a structured mechanism to build an object and ensuring that the mandatory data members are always available before use.

We will walk you through the implementation details of the builder design pattern, the benefits it provides, and a few drawbacks along with some of the places where this design pattern can be used.

## Introduction

An object's state is critical for an application to function properly. An incomplete or inconsistent state can be a cause of exception and faulty behavior. An object's state is defined in the form of data fields it contains and the possible values they can have:



**Figure 4.1:** Class with multiple constructors

[Figure 4.1](#) demonstrates a complex object and some of its constructors. The number of constructors can be more depending on the number of data fields and their possible combinations.

The data fields are also related to the complexity of an object, for example, a simple object has only few fields, whereas a complex object can have multiple data fields, each contributing to the increase in complexity. The more the fields, the higher the complexity.

It is often straightforward to construct a simple object using the **new** keyword, where the number of fields are limited, for example, a String or an Integer. In case of a complex object, one may end up creating multiple constructors for many different variations of an object's state where each constructor could contain a different combination of the input parameters. This could be very confusing if the complexity is really high. However, it doesn't end there, the user of that class might be equally confused with all those constructors and wondering which one to use.

Though one may contradict saying that one single constructor containing all the fields is sufficient and can take care of all the scenarios, this approach has a few drawbacks. The first drawback is that the user may end up passing a lot of null values to the constructor, for fields they assume are optional, and that could be detrimental for future operations in case a required value is passed as null, mistakenly. The second drawback is that it requires a complex validation mechanism to confirm the correctness of the passed values, which is somehow easier to manage with multiple constructor approach or the telescoping approach.

## Delegating object construction

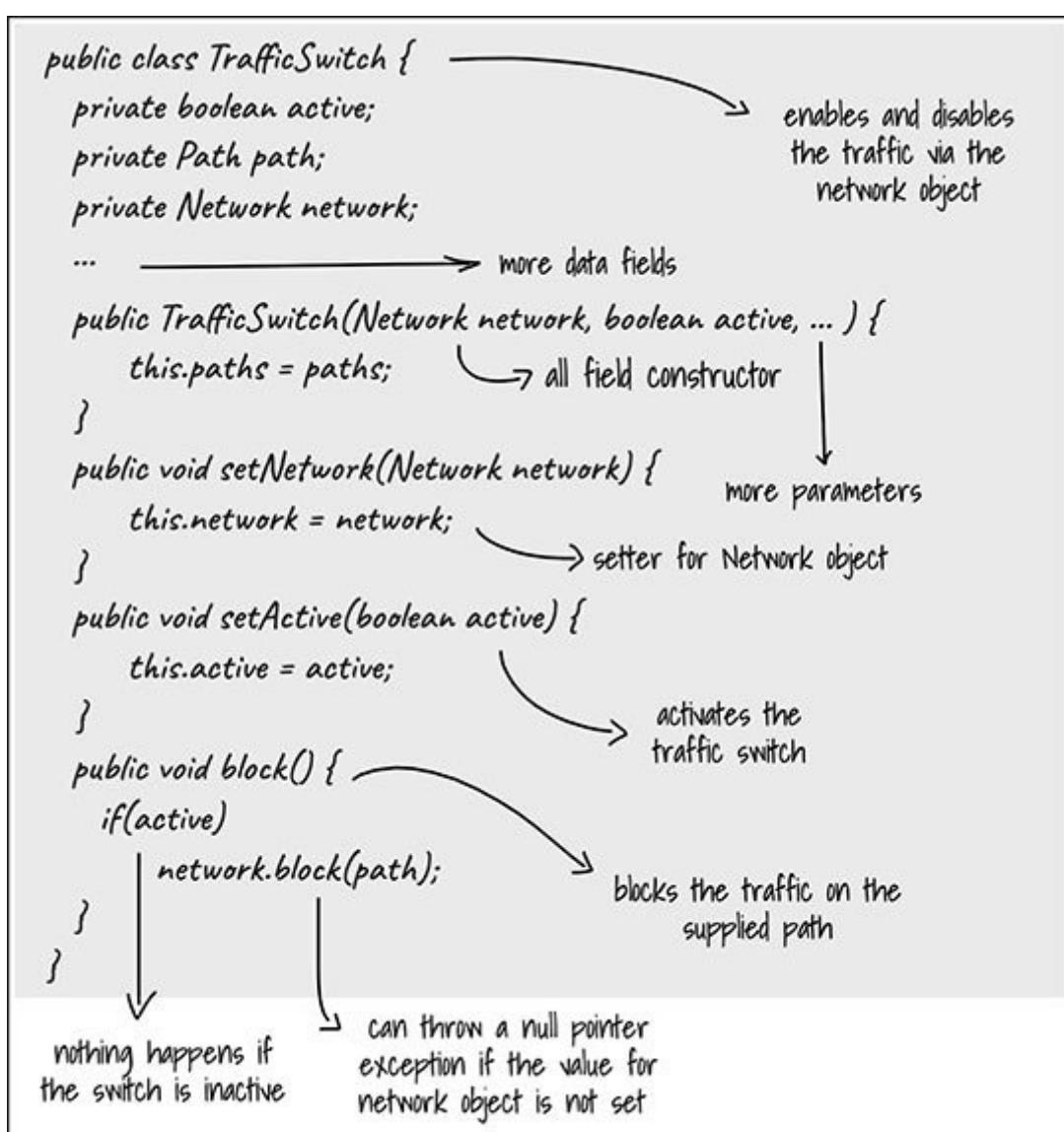
State of an object is a serious affair; it can impact the behavior of that object resulting in an unexpected output or even exceptions. An object's state can be manipulated using setter methods that can set a specific value to a data field.

Another way to construct complex objects is to use an approach that utilize methods to build the state of an object one step at a time. In this approach, the user does not directly interact with the constructor of the class, instead the task of constructing the object is achieved via method calls. Each method constructs only a part of the object state and not the entire object. Multiple such methods can be chained together to form the desired state of the object before finally calling the build method that returns the object itself.

What if, the construction of objects could be delegated to another class? A class that takes up the responsibility to construct a single type of object, validates and verifies the correctness of the input and returns an object with a consistent state. The object formed in the process could also be immutable, if required.

One can argue that by combining the setter methods or more commonly known as, the setters, with the all-field constructor approach, the desired state can be achieved at any given time.

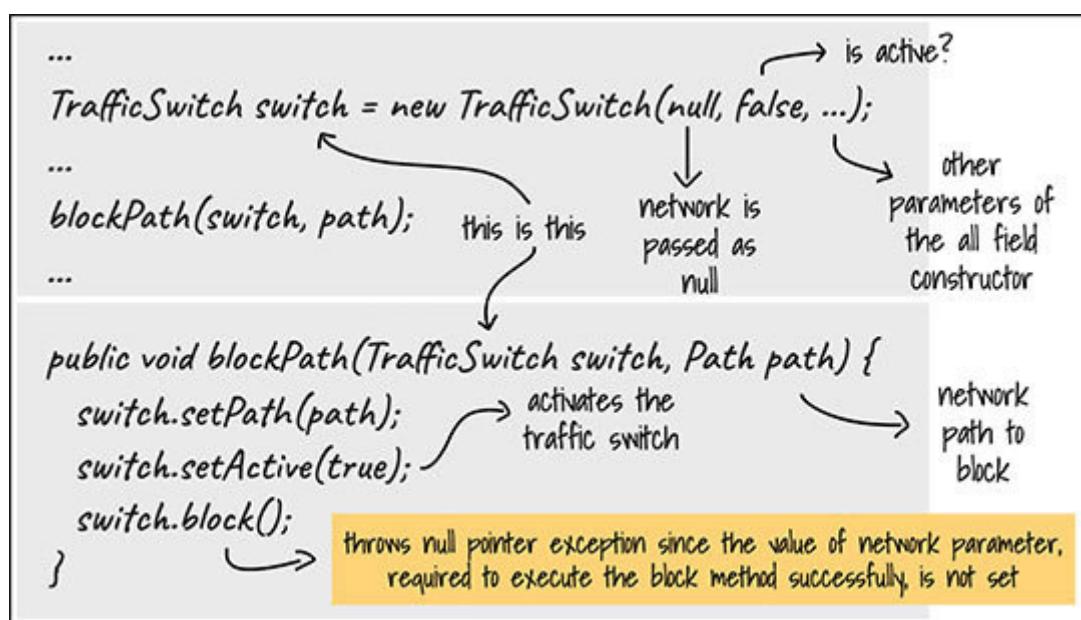
Technically, that is a valid statement as long as the setters are provided for all the fields and one does not fail to call them. The problem with that approach is that the state of the overall object is always dangling between consistent and inconsistent and its behavior cannot be guaranteed. If the values for the object fields are not set properly, the behavior is unpredictable and hence, it is always on the developer to ensure that the relevant setters are called at relevant places and to ensure that runtime exceptions are not thrown due to an improper state of the object:



**Figure 4.2: Setters and All Field Constructor**

Let's go through an example to understand and evaluate the benefits of the approach discussed above. The class in [Figure 4.2](#) is a refactored variant of the complex class we demonstrated in [Figure](#). For the 'block' method to produce the expected result, it is required that the object is in the active state, that is, the active field is set to true and that it has a valid reference to a network instance.

In [Figure](#) the network field of the traffic switch is initially passed as null and later was not set before calling the block method.



*Figure 4.3: Block traffic*

The block method will throw a null pointer exception here since the object state is inconsistent to produce the expected behavior. What is important here is to understand that the object should be properly constructed with all the necessary fields and that the

object construction process is consistent to mitigate the chances of erratic and wrongful behavior.

Revisiting our discussion on the delegation of the object construction process, here's an example of a sample class that can validate and construct objects for us while ensuring that the finally constructed object is in a consistent state.

[Figure 4.4](#) demonstrates the use of builder design pattern to construct an object by delegating the responsibility of object construction to a builder class:

```

public class TrafficSwitchBuilder {
    private TrafficSwitch switch;
    private Path path;
    private Network network;
    private boolean active;

    public TrafficSwitchBuilder setNetwork(Network network) {
        this.network = network;
        return this;           → returns this object
    }

    public TrafficSwitchBuilder setActiveState(boolean active) {
        this.active = active;
        return this;
    }

    public TrafficSwitchBuilder setPath(Path path) {
        this.path = path;
        return this;           → returns an instance of
                               type TrafficSwitch
    }

    public TrafficSwitch build() throws InvalidStateException {
        if(null != network && null != active && null != path)
            return new TrafficSwitch(network, path, active);
        throw new InvalidStateException();           → throws an exception
                                                       if the value for any
                                                       of the mandatory
                                                       field is not present
    }
}

```

**Figure 4.4:** Traffic switch builder

## Representational state

Constructors provide us the liberty to construct object instances with varying representational states, and a corresponding change in behavior. Complex objects having large number of data fields or those that are composed of other objects often have multiple constructors with a different set of input parameters. All these parameters, in various combinations, represent a different state of the object and we often see constructors corresponding to these representations.

Though using a constructor is probably the easiest of the approaches, available to construct an object, it is certainly not the most readable, especially when the constructor parameters begin to grow. As a user of that class, it is hard to figure out which constructor represents which exact state of the object and we often tend to pass in null values for one or more of the optional parameters while constructing an object.

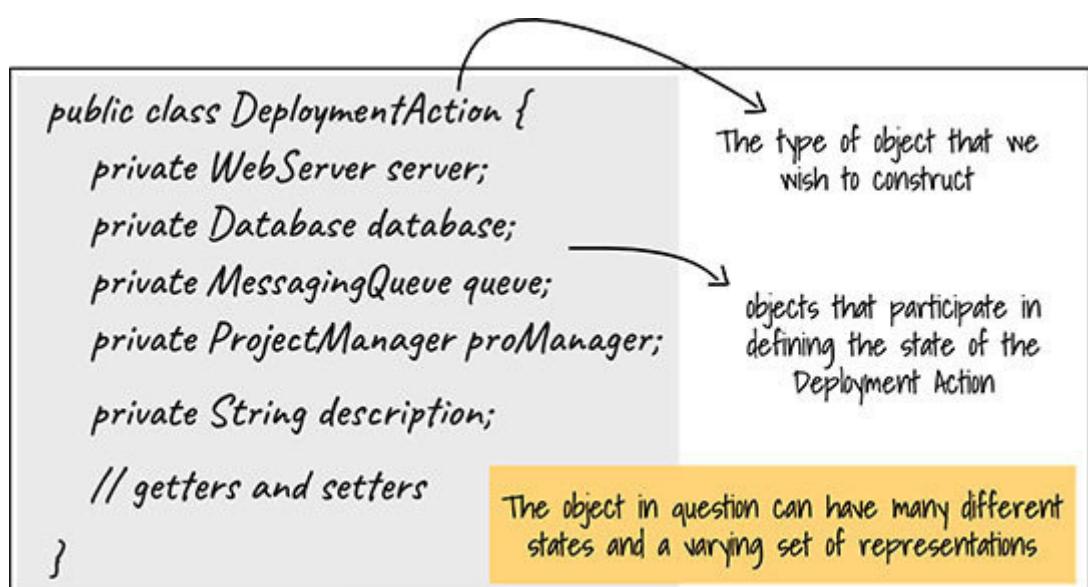
As discussed in the previous section and demonstrated in [Figure](#) we could delegate the responsibility of object creation to another class that exposes methods to set each of the data fields and a build method to finally construct an object of the actual type. The build method can also validate the mandatory parameters before constructing a valid representation of that object. There could be multiple variations of the build method to produce different representations of the same object.

In the next section, we will discuss the benefits and drawbacks of delegating the task of constructing an object to another class and will go through an example, step by step, to analyze and understand the design in question. We will also discuss some of the variations of our design and will observe how that impacts the overall architecture.

## Implementing builders

The example that we will walk you through in this chapter utilizes builders to construct objects. We have taken up this huge task of setting up an automated process to deploy virtual machines on our cloud infrastructure as and when requested by our customers. This requires us to set up a service that takes the input from our customers via a browser rendered web page and then processes it to deploy a virtual machine.

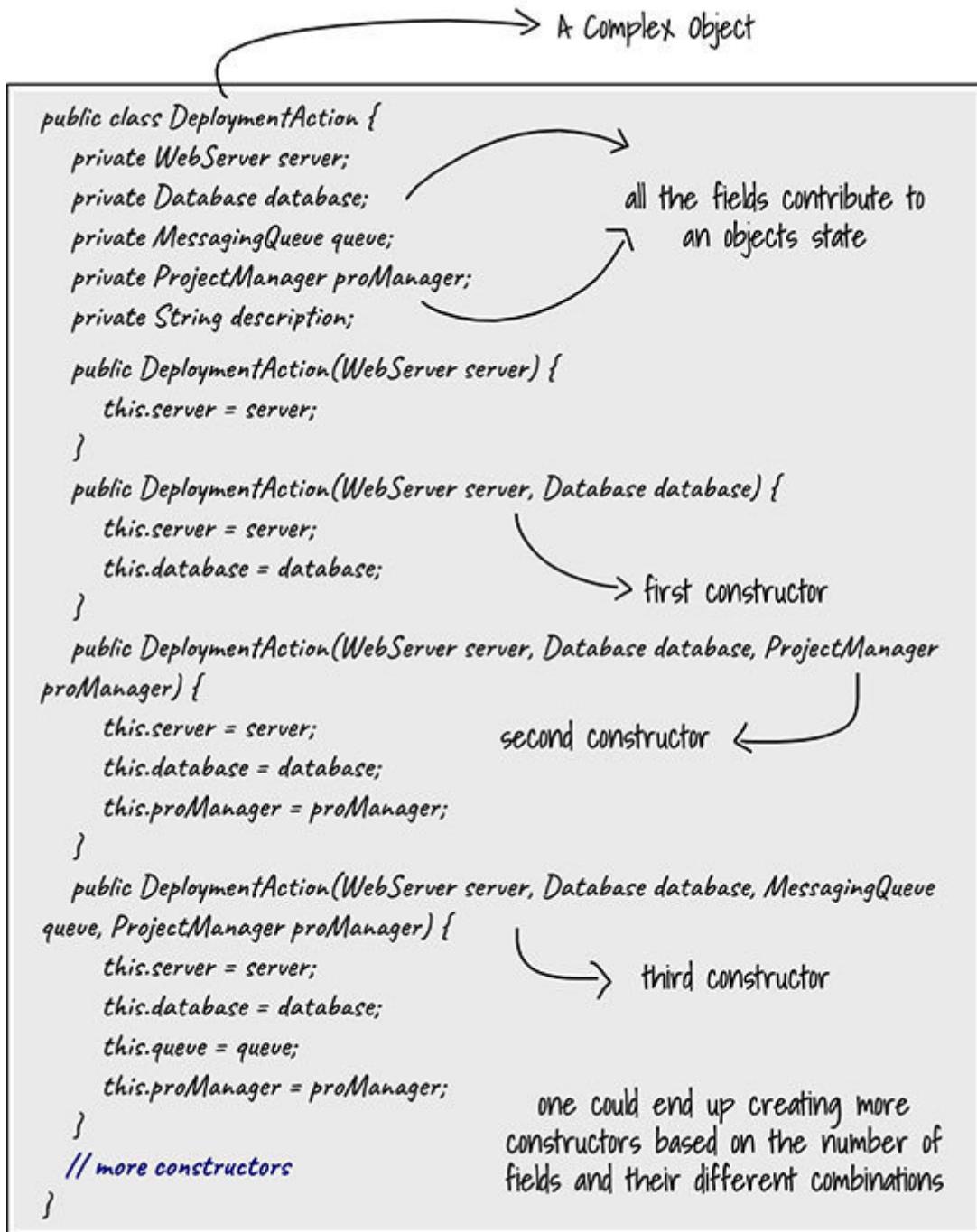
As demonstrated in [Figure](#) the class stores the information related to deployment strategy, selected by our customer to deploy a virtual machine on our cloud infrastructure, and the necessary software along with it:



**Figure 4.5:** Deployment action

The class is composed of references to many other types of objects. There could be multiple variations of the deployment strategy available to the customer and each strategy might require a different set of software. We must ensure that for each strategy the constructed object is populated with the right software that is to be installed to the virtual machine. One of the ways as we have been discussing is to have multiple constructors that support various combinations, but we won't be using it. Instead, we will create multiple classes to support our feature. Each of these classes will produce a different representation of the same object and will ensure that the constructed object has all the required data:

[Figure 4.6](#) demonstrates our class with multiple constructors. Having multiple constructors impacts the readability of the class and makes it difficult to select the right constructor for the deployment strategy:



**Figure 4.6: Multiple constructors**

The API that is exposed to the customers requires a deployment strategy for setting up virtual machines on the cloud. To simplify the selection process, we have created an enum that declares all the available strategies supported by the system. [Figure 4.6](#) demonstrates the enum that provides the deployment strategies:

```

public enum DeploymentStrategy {
    WEB("WebServer Deployment Strategy"),
    CONSOLE("Console Deployment Strategy"),
    WEBMQ("Web MQ Deployment Strategy"),
    VANILLA("Basic Deployment Strategy");
}

DeploymentStrategy(String scheme) {
    this.strategy = strategy;
}

private String strategy;
public String getStrategy() {
    return strategy;
}

```

defines the type of deployment strategy available to the user for the deployment of a virtual machine

each strategy provides a different combination of softwares that are installed in the virtual machine when it is ready for consumption

we have listed down some of the strategies, this list could grow with time, with the increase in the capabilities of our cloud infrastructure

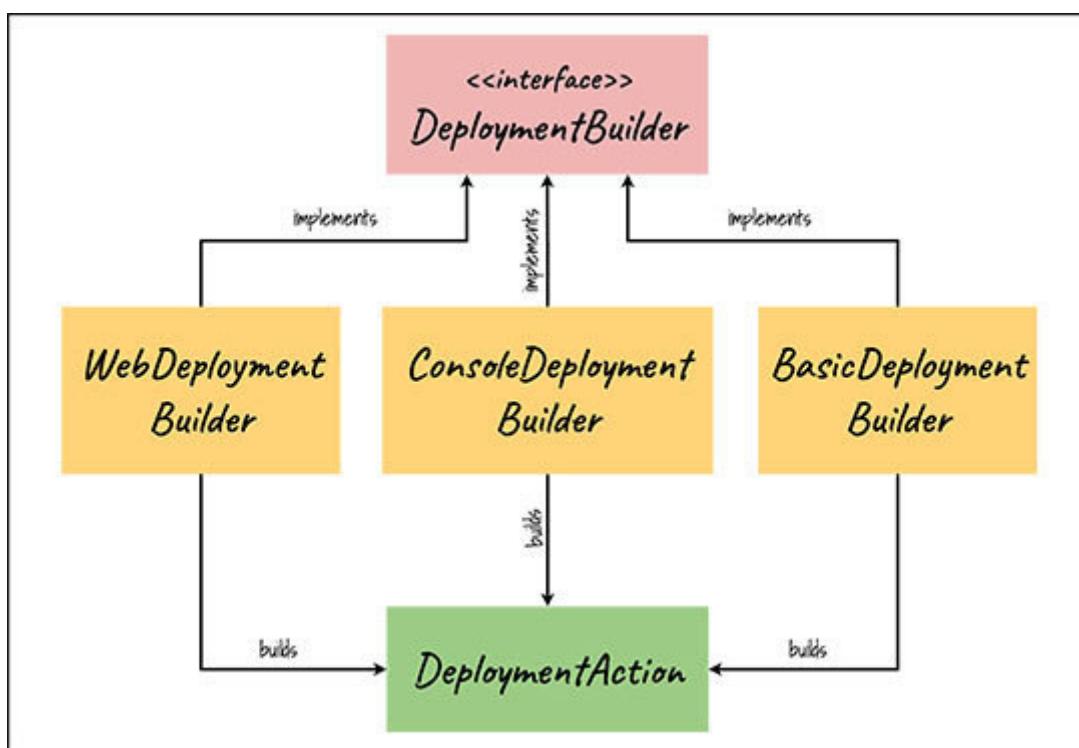
**Figure 4.7:** Deployment strategy

An entry could be added here whenever a new strategy is supported by the system and is made available to the customer.

One way to construct an object is to use one of the many constructors and use the constructed object to deploy and configure the virtual machine with the necessary software. This solution is fine for cases where the permutations on the combination of input parameters is less. In our case, there could be many different variations possible and the direct use of constructors might end up being too complex and less readable. Another way to construct an object is to make use of a similar design demonstrated in [Figure](#)

The builder design pattern aims to build an object in a stepwise approach that separates the process of construction of an object from its representation. It focuses on building an object with a complete state with all the required fields set up using method calls and eliminates the direct interaction with constructors.

[Figure 4.8](#) exhibits the use of different builder implementations to build instances of the same class but with different representational state:



**Figure 4.8: Basic design**

There could be multiple representation of the same object, we can either create multiple builder classes for each representation type or we can construct a single builder class and expose different

methods to support multiple representations. For this example, we will construct multiple deployment builders and will explore the other option later. To ensure that the different builder classes follow the same contract, we will first build an interface that will list down the behavior our builders should support.

## The interface

The deployment builder interface defines the methods that should be supported by all the concrete builders. As you can see, only the build method is truly abstract while the others aren't. Other methods are provided a default implementation, that is just to return the object itself. Wondering why have a default definition at all when all it does is just return the object? A default definition reduces the need for overriding if there is no change in implementation. Since our concrete builders return different representations of the same object, they only require overriding those methods that are relevant to them, as demonstrated in

## Figure

```

public interface DeploymentBuilder {
    public DeploymentAction build();
}

public default DeploymentBuilder installServer() {
    return this;
}

public default DeploymentBuilder installProjectManager() {
    return this;
}

public default DeploymentBuilder installDatabase() {
    return this;
}

public default DeploymentBuilder installMQ() {
    return this;
}

```

the interface that should be implemented by each of our builder classes

the implementation classes should ensure that they provide a definition to the methods

default methods return 'this'

default methods to support all the possible software installations

default methods in the interface ensure that we are future ready for any new type of software. Any new addition to the list will only require changes in this interface and the classes that have to mandatorily provide a definition

the only method that should mandatorily be a part of the Builder interface is the build method, all the other methods can be put in the concrete classes

**Figure 4.9: Deployment Builder Interface**

One important aspect of this design is that the construction of the resultant object is not complete until the build method is called on the builder. Therefore, the other methods can be chained together to prepare the object before it is finally constructed. This also means that there's only one way in which the resultant object is exposed to the outside world. The build method can also be used to validate that the object is in a completed state before constructing the subject; otherwise, throw an exception if the state of the object is incomplete. More on this in the following sections.

## Concrete builders

[Figure 4.10](#) demonstrates the simplest and most basic implementation of a concrete builder. It is mapped to the default deployment strategy and does not support installation of any type of software. It implements the builder interface and only overrides the abstract build method. It also does not perform any validation since the only mandatory parameter, that is, description, is always set to a value in the build method itself:

```
public class BasicDeploymentBuilder implements DeploymentBuilder {  
    private DeploymentAction action; —————> reference to the actual  
    object type  
    public BasicDeploymentBuilder() {  
        action = new DeploymentAction();  
    }  
    public DeploymentAction build() {  
        action.setDescription("Basic Deployment Strategy");  
        return action;  
    }  
}
```

instantiate the actual object

set the description before returning the object

current implementation is the most basic definition of a builder

**Figure 4.10: Basic Deployment Builder**

[Figure 4.11](#) demonstrates another example of a concrete implementation of the builder interface. It configures installation of **gradle** as a project management tool and **mysql** as the database engine. It also validates the completeness of the object before exposing it to the outside world. Notice that the builder class also

contains the same references as that of the subject. More on this later.

Each concrete builder builds a specific representation of the subject. Instead of selecting which constructor to call, we can just choose the specific representation and state we want to build. This is much more readable than the direct constructor approach as each builder provides methods with names that are descriptive enough for the user to understand.

The builders can also validate the state before finally constructing an object, something that is missing in the direct constructor approach:

```

public class ConsoleDeploymentBuilder implements DeploymentBuilder {
    private DeploymentAction action; —————> reference to the actual
    private ProjectManager projectManager; object type
    private Database database;

    public DeploymentBuilder installProjectManager() {
        print("Configuring Gradle");
        projectManager = new Gradle();
        return this;
    }

    public DeploymentBuilder installDatabase() {
        print("Configuring MySQL Database");
        database = new MySqlDatabase();
        return this;
    }

    public DeploymentAction build() {
        if(null == database || null == projectManager)
            throw new IllegalStateException(); —————> throw IllegalStateException
                                                if the validation fails
        action = new DeploymentAction();
        action.setDatabase(database);
        action.setProManager(projectManager);
        action.setDescription(CONSOLE.strategy()); construct the actual
                                                type of object
        return action; —————> return the properly
                                constructed object if all the
                                mandatory values are set
    }
}

```

**Figure 4.11:** Console deployment builder

[Figure 4.12](#) demonstrates another example of the deployment builder. It configures installation of the **apache tomcat** as application server, **maven** as a project management tool and **oracle** as the database engine. It too validates the completeness of the object by verifying that mandatory requirement are fulfilled, before exposing it to the outside world.

In [Figure](#) the data members are marked protected and not private. This could be useful when a similar deployment strategy is to be defined with only minor changes. In such a scenario, this builder class can be extended with relevant sections overridden:

```

public class WebDeploymentBuilder implements DeploymentBuilder {
    protected DeploymentAction action;
    protected ProjectManager projectManager;
    protected Database database;
    protected WebServer server;

    public DeploymentBuilder installServer() {
        print("Configuring Apache Tomcat");
        server = new ApacheTomcat();
        return this;
    }

    public DeploymentBuilder installProjectManager() {
        print("Configuring Maven");
        projectManager = new Maven();
        return this;
    }

    public DeploymentBuilder installDatabase() {
        print("Configuring Oracle Database");
        database = new OracleDatabase();
        return this;
    }

    public DeploymentAction build() {
        if (null == database || null == projectManager || null == server)
            throw new IllegalStateException();
        action = new DeploymentAction();
        action.setDatabase(database);
        action.setProManager(projectManager);
        action.setServer(server);
        action.setDescription(WEB.strategy());
        return action;
    }
}

```

the fields are marked protected, we plan to use them later by extending this class

overriding the default methods from the interface

installs 'maven' the builders can provide different definitions to the methods they override

validate if all the three required values are set

throw IllegalStateException if the validation fails

construct the actual type of object

set description of this strategy

**Figure 4.12:** Web Deployment Builder

### *Deployment manager*

The class demonstrated in [Figure 4.13](#) is responsible for utilizing the right deployment builder to build the action object based on the strategy selected by the user:

```

public class DeploymentManager {
    public static void deploy(DeploymentStrategy strategy) {
        DeploymentAction action = null;
        switch (strategy) {
            case WEB:
                action = new WebDeploymentBuilder()
                    .installServer()
                    .installDatabase()
                    .installProjectManager()
                    .build();
                break;
            case CONSOLE:
                action = new ConsoleDeploymentBuilder()
                    .installDatabase()
                    .installProjectManager()
                    .build();
                break;
            case WEBMQ:
                action = new WebMQDeploymentBuilder()
                    .installMQ()
                    .installServer()
                    .installDatabase()
                    .installProjectManager()
                    .build();
                break;
            default:
                action = new BasicDeploymentBuilder()
                    .build();
                break;
        }
        VMDeployer.deploy(action);
    }
}

```

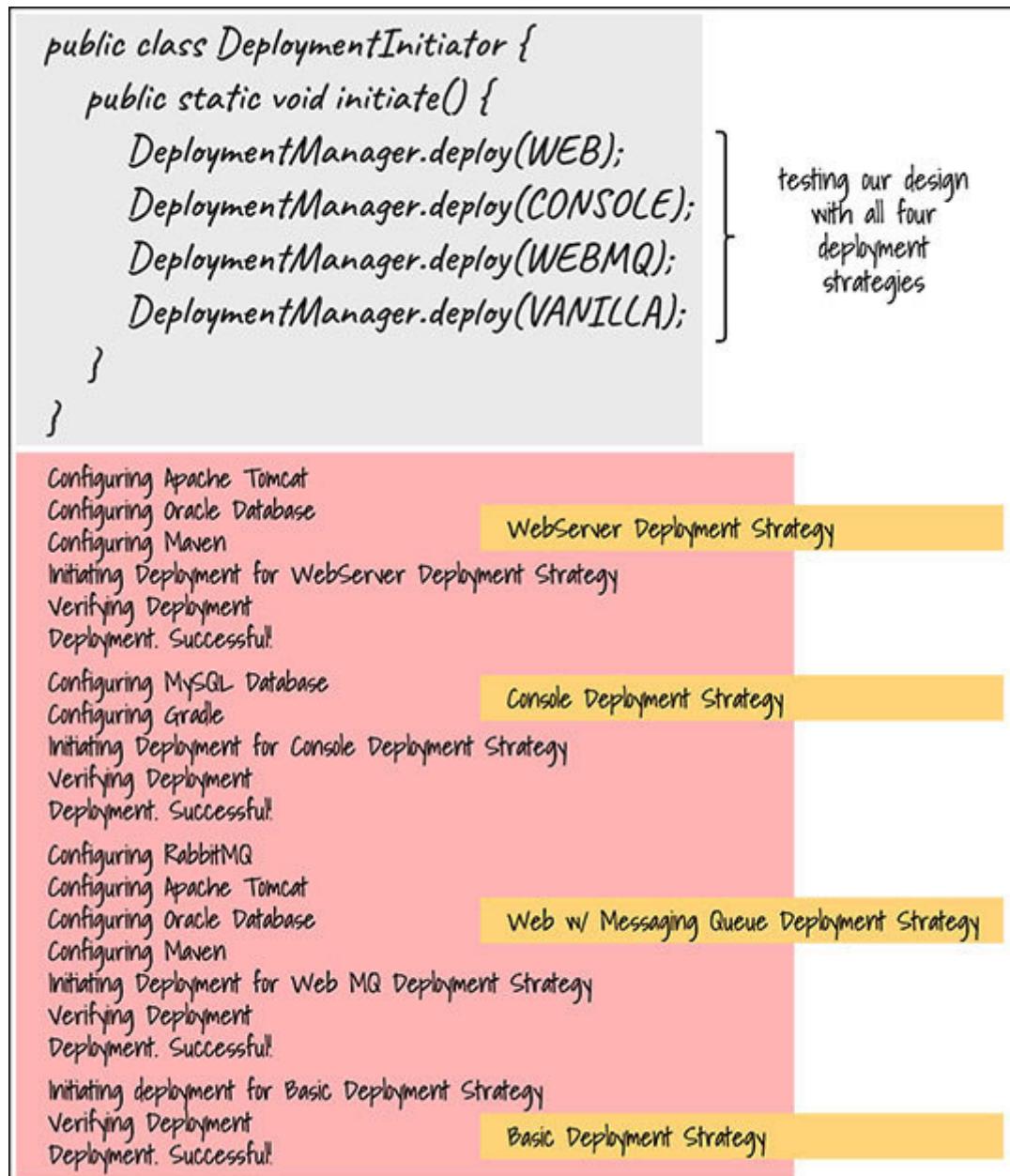
The diagram illustrates the Deployment Manager code with several annotations:

- the class that decides the type of builder to use for constructing object instances based on the deployment strategy selected by the user**: Points to the `switch (strategy)` block.
- users input, the deployment strategy**: Points to the `strategy` parameter in the `deploy` method.
- easily interpretable method names**: Points to the method names like `.installServer()`, `.installDatabase()`, etc.
- method calls chained together, missing any of the method calls will throw an exception when the build method is finally called**: Points to the chain of method calls within each `action` constructor.
- builder design pattern provides an abstraction layer that hides the construction mechanism used to construct the final object. The caller is only exposed to the method calls available to her**: Points to the overall structure of the code, emphasizing the abstraction provided by the builder pattern.
- finally deploy the virtual machine**: Points to the `VMDeployer.deploy(action);` call at the end of the `deploy` method.

Figure 4.13: Deployment manager

## Testing the builders

We have written a test class to verify that the pattern we worked on works. Let us execute the same as shown in the [Figure](#)



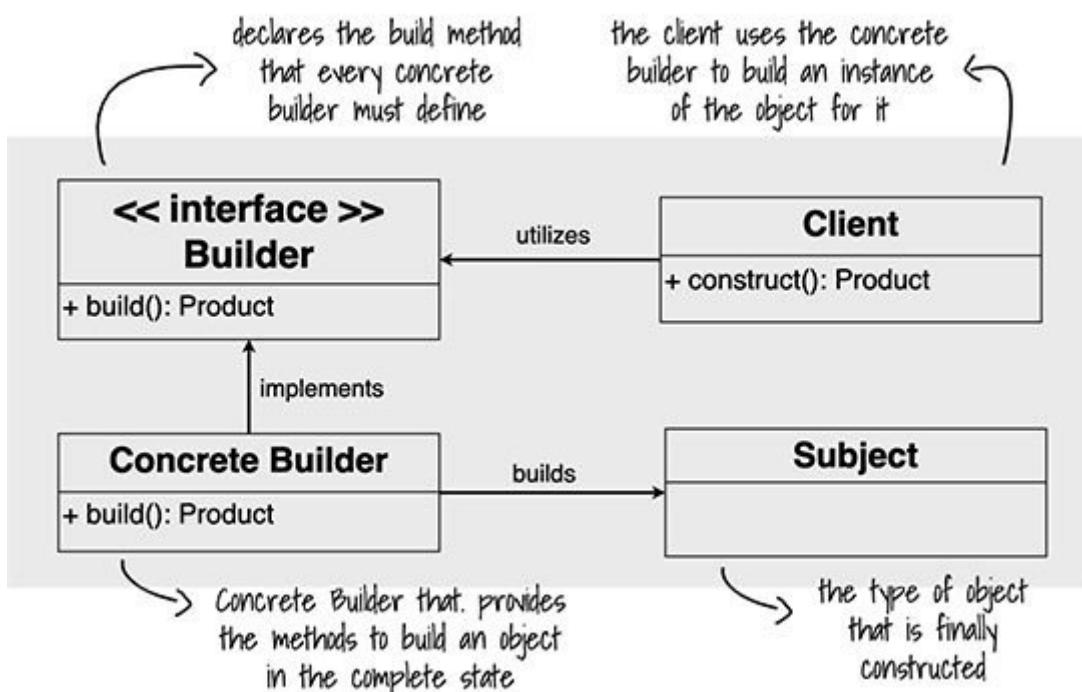
**Figure 4.14: Deployment Initiator**

In *Figure* multiple calls are made to the deploy method of the deployment manager class, passing along the strategy that should be used for deploying the virtual machine on the cloud infrastructure.

The manager chooses the builder to build the correct representation of the object, which is then passed to another class for the final deployment.

## Builder design pattern – defined

The builder design pattern as shown in the [Figure 4.15](#) is one of the widely used creational design pattern that defines a stepwise object construction process that can be reused to produce different representations of the same object. It separates the process of constructing an object from its representation:



**Figure 4.15:** Builder design pattern

In case of a large, complex object that has multiple dependencies and require processing before they can be acquired, multiple builders can be created each devoted to the task of building only one of the dependencies, and finally, one last builder can be

delegated the task of invoking those builders to gather the dependencies and build the complex object.

The builder design pattern provides a generic approach to the process of building an object with different representational states. It effectively reduces direct interaction with the constructors of the object and improves readability by providing properly named, self-descriptive methods. It can also be utilized to perform validation before exposing the constructed object.

## Advantages and drawbacks

The builder design pattern has many advantages over the approach of direct interaction with the constructors. Let us start with the benefits first before we explore the drawbacks of it:

It reduces the number of constructors in the subject class

It provides method calls to construct an object one step at a time, reducing the direct interaction with the constructors and improves readability

Restricts the creation of inconsistent state object instances by validating and restricting null arguments

Can be used to build immutable objects considering the dependent objects are also immutable, for example, string builder class in Java uses the builder design pattern

The drawbacks of the builder design pattern are:

A lot of code is duplicated since each builder is an approximate replica of the subject class

Each representational state requires a separate builder

## Usage

Some of the scenarios relevant for builder design pattern are:

When there are multiple possible representations of an object, for example, a report can have multiple datasets to display, but not every user is authorized to see it

It can be used to incrementally build an object, while respecting the validations put in place, for example, a restaurant menu builder can make use of date and time information to include or exclude certain dishes from the daily menu

## Conclusion

Builder design pattern is a widely used creational design pattern that can be utilized to build different representational states of an object. It provides an abstraction layer that hides the complexity of object construction from its users while simultaneously improving the readability by exposing methods that build the object in parts, step-by-step. It ensures that the object that is finally instantiated is in a complete state.

For a large complex object with multiple dependencies, each dependency can be constructed using a different builder and later combined to form the one complex object. The builder classes can be developed independently or as a static inner member of the subject class itself.

In the next chapter, we will continue exploring the creational design patterns and will look into the object pool design pattern that helps us construct and reuse a pool of objects.

## Questions

In this section we will sharpen our brain by indulging in some fun brain activities. Let's get started...

## Pick the odd one out

Builder	build	readable method calls
Concrete Builder	multiple constructors	immutable object
Concrete Subject	step by step	complete state
Subject	task delegation	code duplication

*Figure 4.16: Pick the odd one out*

## Odd one out

Builder design pattern

Reduces number of constructors in subject class

Builds the subject step by step

Instantiates an object in a complete state

Can build immutable objects

Concrete builders

Duplicate a lot of code

Must implement the builder interface

Build multiple type of objects

Utilize stepwise approach to construct object

Builder design pattern utilizes

Abstraction

Composition

Threading

Inheritance

## Complete the code

```
public class DeploymentManager {
    public static void deploy(          strategy) {
        DeploymentAction action = null;
        switch (strategy) {
            case      :
                action = new          ()
                    .          ()
                    .installProjectManager()
                    .build();
                break;
            case      :
                action = new          ()
                    .installDatabase()
                    .installProjectManager()
                    .build();
                break;
            case      :
                action = new          ()
                    .          ()
                    .installServer()
                    .          ()
                    .installProjectManager()
                    .          ();
                break;
            default:
                action = new          ()
                    .build();
                break;
        }
        .deploy(      );
    }
}
```

WebMQ  
Deployment  
Builder  
action  
BasicDeployment  
Builder  
Console  
Deployment  
Builder  
WEBMQ  
build  
installServer  
WebDeployment  
Builder  
VMDeployer  
Console  
installMQ  
Deployment  
Strategy  
installDatabase  
installDatabase

**Figure 4.17:** Complete the code

Select two correct answers

Builder design pattern

Duplicates code

Use multiple constructors

Can produce immutable objects

Builds incomplete object

Constructors

Can have multiple parameters

Cannot be passed final references

Construct an object

Cannot be overridden

A builder

Can be a static inner class

Can have multiple build methods

Can call other builders

Can have multiple constructors

What goes together

Code duplication, incomplete state, step-by-step

Static inner class, code duplication, multiple constructors

Step-by-step, complete state, immutable object

Single build method, step-by-step, multiple constructors

What goes together

Multiple build methods, code duplication

Multiple constructors, not null arguments

Step-by-step, complete state

Null arguments, concrete builder

## Answers

### Find the odd one out

Concrete Subject, task delegation, code duplication

### Pick the odd one out

Reduces number of constructors in subject class

Duplicate a lot of code

Threading

Select two correct answers

Use multiple constructors, can produce immutable objects

Can have multiple parameters, construct an object

Can be a static inner class, can have multiple constructors

[step-by-step, complete state, immutable object] & [single build method, step-by-step, multiple constructors]

[multiple constructors, not null arguments] & [step-by-step, complete state]

## CHAPTER 5

### Recycle and Reuse

---

You can find the editable code files for this chapter on the book's GitHub link.

<https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers/tree/main/Chapter%2005>

---

## Structure

Topics that make this journey worthwhile:

Introduction

Object reusability

Object pool design pattern

Implementing an object pool

Varying implementations

Advantages and drawbacks

Usage

## Objective

Our objective is to understand the need of reusing objects and to learn about object pools that act as containers for objects. We will learn about the object pool design pattern and how it can be utilized to manage objects and improve application performance.

We will also walk you through the implementation details of the object pool design pattern, some of its variations, the benefits it provides and a few drawbacks along with some of the places where this design pattern can be used.

## Introduction

Sometimes, a single object of the same type is just not enough, the application requires multiple of those to support parallel processing. Other times, an object is a high resource consumer and constructing the same object type repeatedly will impact the performance. In both these cases, the area of focus is object reusability, that is, to reuse an existing object to perform some operation instead of constructing a new one.

## Object reusability

Reusability is one of the important aspects of software design; it not only advocates the reuse of already defined libraries, but also promotes utilizing the existing object instances to perform recurring operations. Applications today perform millions of operations and often do so in parallel. If we must reuse our object instances, a single instance might not be enough to support parallel operations. A desirable approach here is to have a group of objects usually called an **object pool** to support such magnitude of operations.

Reusability of objects is favorable under certain scenarios:

**Cost of object** When the object consumes a lot of computing resources, for example, CPU cycles or memory, that could have an impact on application performance

**Object construction** When an object requires a large time to construct, possibly due to some IO operations, that could account for a delay in processing operations

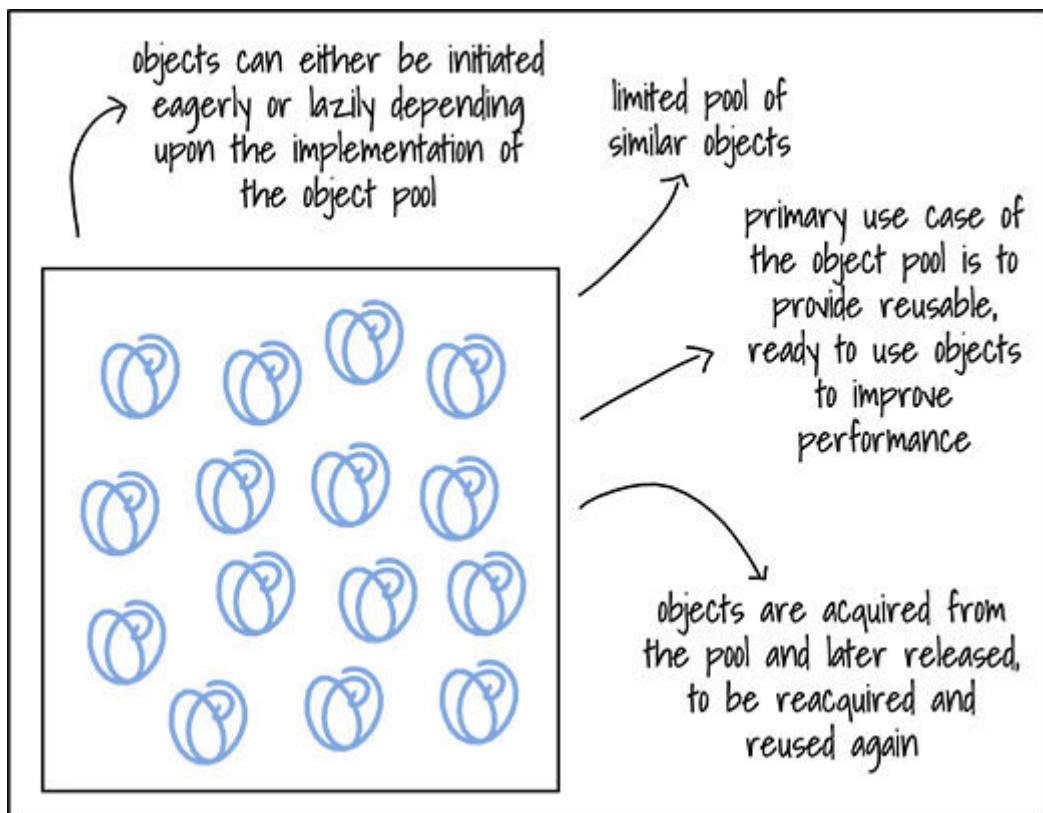
**Parallel** When the application requires the ability to support parallel processing

**Limited** When the application's memory allocation allows for a limited number of objects to be constructed

There could be additional scenarios apart from the ones mentioned earlier, those that are usually relevant to some specific programming tasks or business requirements.

## Object pool design pattern

A pool of objects represented in the [Figure 5.1](#) is essentially a group of pre-constructed object instances that can be acquired from the pool to perform some operation and then released to be reused again:

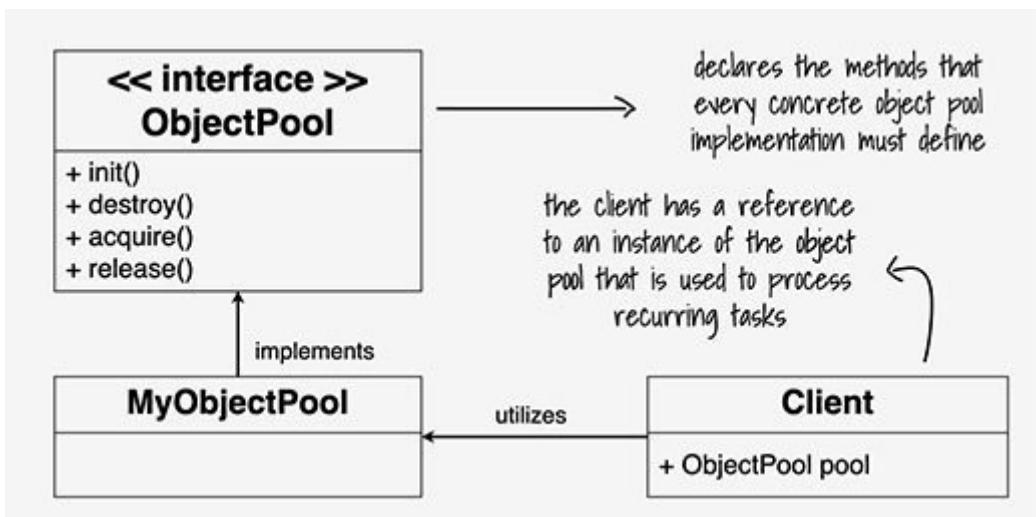


**Figure 5.1:** Pool of objects

The **object pool design pattern** is a creational design pattern that suggests a way to reuse existing object instances rather than creating new ones on-demand. It encourages reusing the same

objects to perform computation to boost performance and to limit resource consumption.

The pool contains a finite number of object instances. The objects that form the pool can be constructed either eagerly or lazily depending on the implementation of the pool. It contains similar type of objects; subtypes are also allowed depending on the nature of the pool. [Figure 5.2](#) describes the architecture of the object pool design pattern:

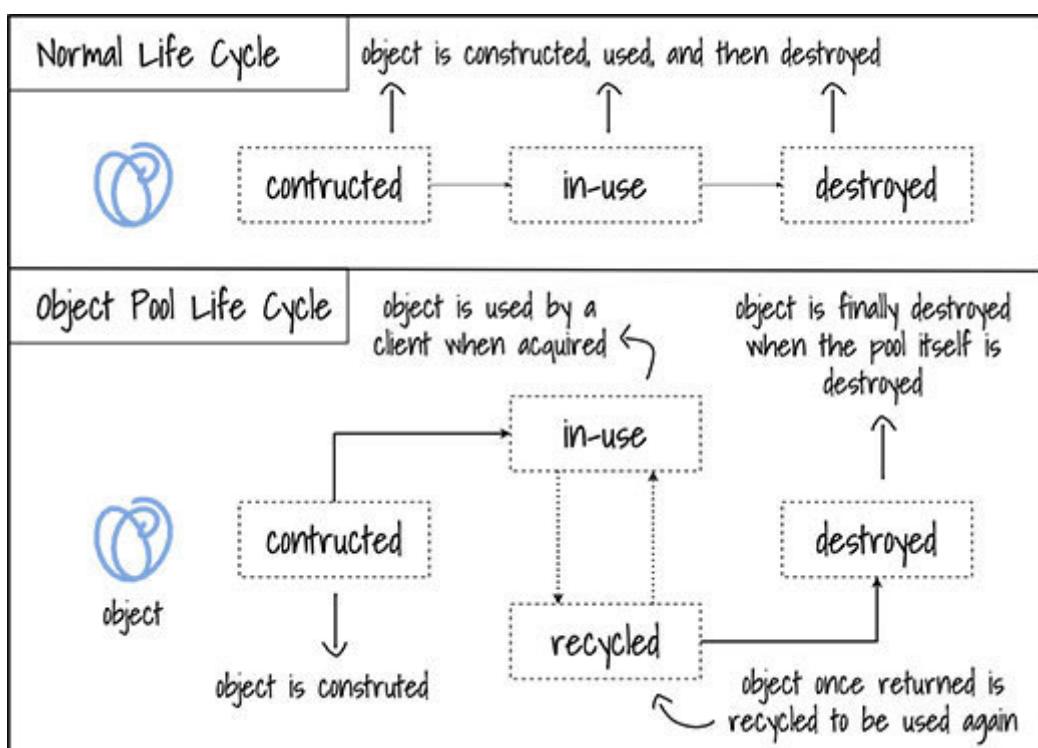


**Figure 5.2: Object pool design pattern**

The objects in the pool can be initialized well in advance so that they can be utilized by the clients to perform the functionality those objects are intended for without requiring to create new object instances. The objects from the pool can be acquired, used, and then returned to the pool to be reused again, either by the same client or some other client simultaneously accessing the pool of objects.

The object once acquired is no longer available in the pool until it is returned to it. For this reason, it is important that the object is immediately returned to the pool after its use. This ensures that the objects in the pool can be made available to other clients when required, with as less delay as possible. The client must ensure that the acquired objects are returned to the pool and not destroyed.

The objects that form the pool have an unusual life cycle, different from all other objects of the application. In layman's term, an object's life cycle has the following stages: constructed, in-use, and destroyed, in the sequence mentioned. For objects that are part of the pool, this cycle is a bit circular in nature, that is, constructed, in-use, recycled, and destroyed. This increased complexity requires additional care in implementing an object pool. [Figure 5.3](#) demonstrates the life cycle of pool objects:

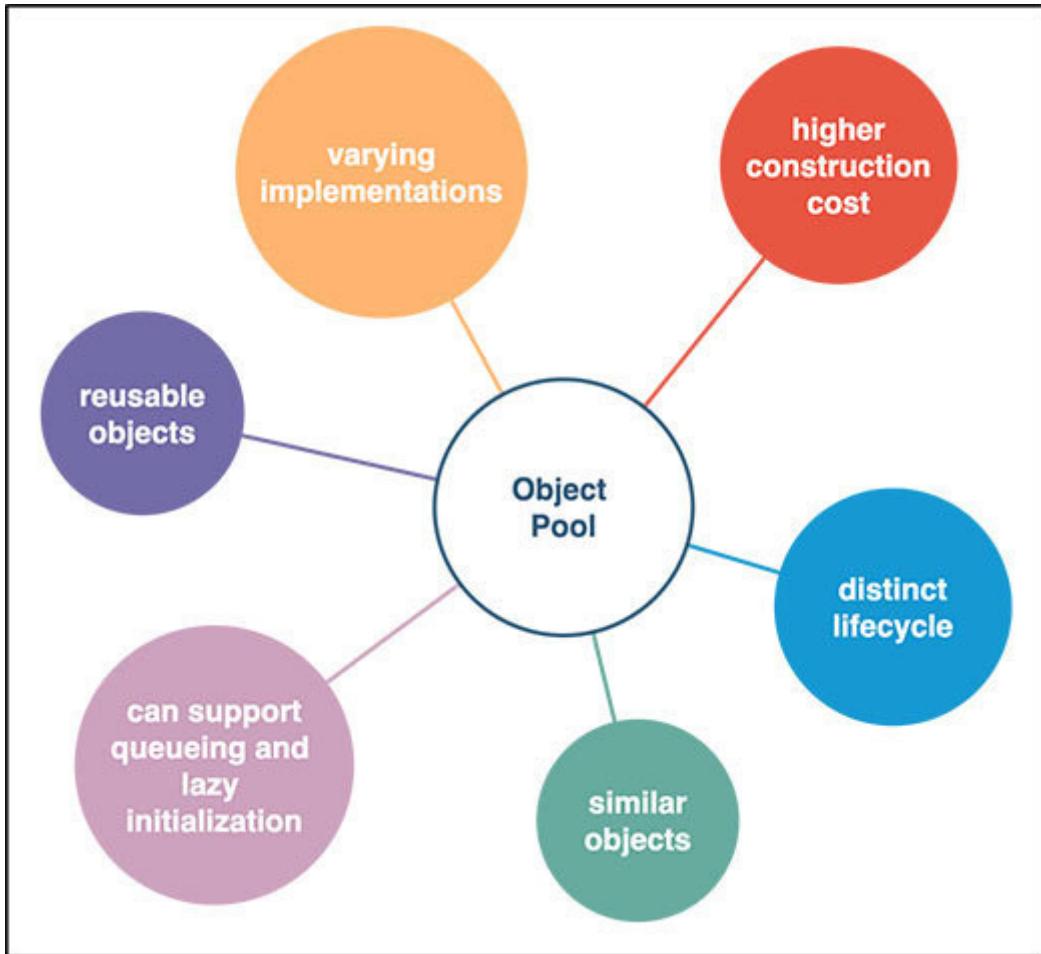


### ***Figure 5.3: Life cycle of pool objects***

Initializing pool objects at the time of pool construction ensures that the pool can be used straight away, after the application starts. This otherwise would take up time to initialize pool objects, incurring delays before it could be first used.

An implementation could only initialize a few of those objects initially and wait for them all to be acquired before initializing new objects in the pool. Others could release some of the long waiting, not-in-use objects to memory and later initialize new objects when required. The object pool design pattern suggests a way to form a pool of objects, but it does not limit the creativity of the developer.

As mentioned in [Figure](#) objects in a pool are of similar kind, reusable, have a distinct life cycle and have a higher cost of construction. The pool itself can have varying implementations and can support request queueing and lazy initialization of pool objects. Some of the characteristics mentioned above are definitive while others depend on the implementation:



**Figure 5.4:** Characteristics of Object Pool

## When to use an object pool?

A pool of objects is desirable when there is a requirement to work with a large set of objects that are expensive to initialize, consume high resources and have repeated usage. In such scenarios, an object pool design is highly suitable as it provides pre-initialized, ready-to-use objects that can be acquired, used, and later released to be reused by other clients.

An object pool acts as a container for pre-initialized objects. When there is requirement for an object, it can be acquired from the pool instead of creating a new one, saving the cost of object initialization. When requested, the pool will return an available object that is effectively recycled. Based on an implementation, if the pool reaches its limit the implementation could either queue the new requests and put the requesters on wait until an object is available to be acquired, or it can throw an exception. Once the acquired object is no more needed, it must be returned to the pool where it will be recycled to prepare it for reuse. The recycle process must invalidate all the existing references of the returned object so that it is no longer accessible to its previous acquirer. The recycle process must also reset the object's state, effectively making it a new object.

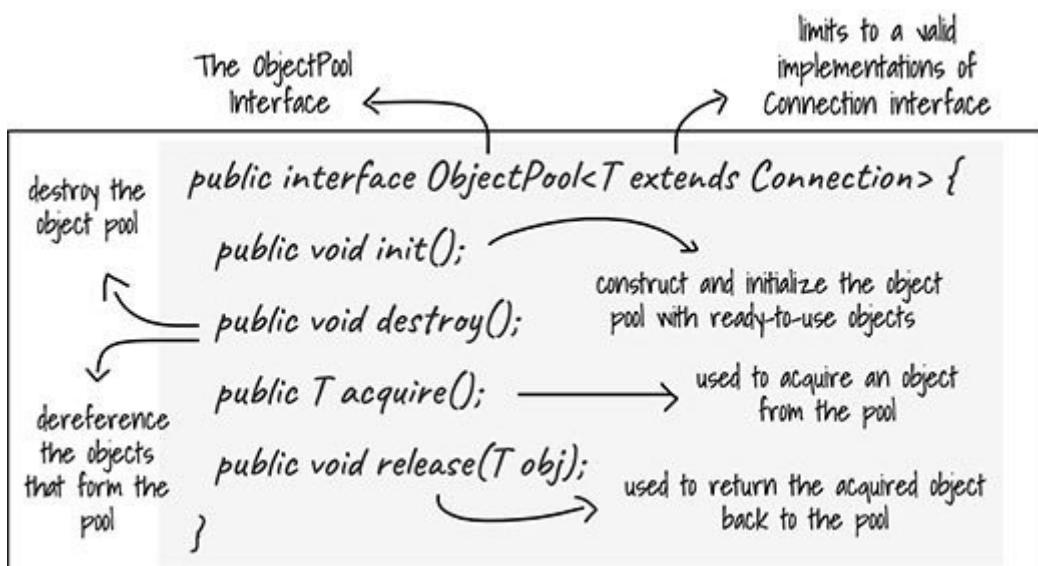
If the object's state isn't reset to default, it could lead to wrong behavior. If the object is responsible for computations, it could

result in wrong results. If the object safeguards access to a profile, it could provide access even before someone requested for a login, resulting in data theft. If the object provides access to a resource, for example, a database, it could end up providing unauthorized access. There could be many more similar scenarios, which can go wrong if the object's state isn't managed properly and therefore, it is of utmost importance to ensure that the object is properly recycled before reuse.

## Implementing an object pool

An object pool has three main components, one is the interface that defines the contract, the other is the implementation that abides by it and the third is the object that forms the pool. An implementation must ensure to provide a definition to the methods exposed by the interface.

In *Figure* the interface takes advantage of Java generics and restricts the pool to contain only a specific type of objects, subtypes are allowed. The ability to define the type of object that can be used within the implementing class allows the interface to be reused for many different implementations:



*Figure 5.5: The object pool interface*

The interface has four methods:

Constructs and initializes the object pool with ready-to-use, reusable objects that can be acquired, used, and then returned when no longer required

Destroys the object pool and de-references the objects that form the pool allowing them to be garbage collected

It is used to acquire an object from the pool

It is used to return an object to the pool

We have looked at the design of the object pool design pattern, the interface the concrete classes must implement, and the methods exposed by that interface. We also discussed when an object pool should be constructed and utilized and some of the smaller details that the developer should work on while implementing an object pool. It is time to implement an object pool and discuss the different implementation approaches.

An object pool can be implemented in many ways depending on the use case and the frequency in which the objects are acquired from the pool. Objects that form the pool, can all be instantiated at the same time when the pool is constructed or the pool could be partially filled with the objects at the time of its construction, initializing only a few of them initially and deferring the instantiation of the rest of the objects, when required. The remaining objects could be instantiated one at a time when there

is a request to acquire an object from the pool and all the existing pool objects are already acquired, but the pool size is still not reached. We will discuss these and a few other variations on the object pool implementation later in the chapter. For now, let us move ahead and implement our first object pool. [Figure 5.6](#) demonstrates the implementation of a pool of database connection objects:

```
public class DatabaseConnectionPool implements ObjectPool<Connection> {
    private static DatabaseConnectionPool _instance;
    private static final String ILLEGAL_STATE = "The state of the database
connection pool is inconsistent";
    private static final String INVALID_RELEASE_CONNECTION =
    "Request received to release invalid connection object";
    private boolean configured = false; —————→ stores if the pool is configured
    or not; used in private methods
    private int size = 10; —————→ fixed size of the pool
    private ArrayBlockingQueue<Connection> available; —————→ stores the
    private ArrayBlockingQueue<Connection> acquired; —————→ currently
    private static final String url = "localhost:3306"; —————→ available
    ... connections
    stores the connections
    acquired by clients
}

This implementation of the connection pool creates a fixed size pool and connects to a
predefined database url i.e. localhost:3306
```

**Figure 5.6: Fixed size connection pool**

The approach that we have used here is that of eager initialization, that is, instantiating all the objects in the pool at the time of pool construction itself. We will use the standard connection interface that is provided by Java so that our implementation is relevant for any of its implementations. We will also make use of some basic multi-threading to safeguard our pool objects from concurrent access. And finally, for storing the

objects in the pool, we will make use of the standard queue implementations.

In [Figure](#) we have demonstrated various data members of our object pool implementation. Our implementation uses two blocking queues, one each for available and acquired connection object, a variable to store the size of the pool, and a Boolean variable that stores the status of the pool, that is, whether it is configured or not. Once the connection pool is initialized with the connection objects, the pool's status is changed to configured.

Using two queues in the pool helps with better management of the acquired and available objects. A single queue is difficult to manage as an already acquired object could be reused even before it is released by the previous owner.

As demonstrated in [Figure](#) our implementation constructs a fixed size pool and only connects to a predefined database as mentioned in the class itself:

```

public static DatabaseConnectionPool getInstance() {
    synchronized (DatabaseConnectionPool.class) {
        if (null == _instance) {
            synchronized (DatabaseConnectionPool.class) {
                _instance = new DatabaseConnectionPool();
            }
        }
    }
    return _instance;
}

private DatabaseConnectionPool() {
    this.available = new ArrayBlockingQueue<>(size);
    this.acquired = new ArrayBlockingQueue<>(size);
}

```

constructing the singleton instance

queues that contain available and acquired connections

synchronized to safe guard against multi threaded environment

**Figure 5.7:** Connection pool constructor

This approach, which might not be suitable for all the cases considering the limitations, is the easiest one to understand the object pool and the design pattern, hence the choice. It also uses the singleton design pattern, the design pattern we discussed in [Chapter](#) to ensure that there's only a single instance of the connection pool available at any point of time. Using a singleton ensures that only desired number of connections are open, leaving no space for error. Having a single pool of database connections also make it easier to manage and use the connections.

## Pool initialization

Initializing the object pool is the first operation that should be performed on the pool. This operation fills the pool with the desired number of object instances. In [Figure](#) the **init** method initializes the pool with a fixed number of connection objects as mentioned earlier:

```
public void init() throws SQLException {
    if (!configured) {
        synchronized (DatabaseConnectionPool.class) {
            if (!configured) {
                for (int index = 0; index < size; index++) {
                    Connection con = DriverManager.getConnection(url);
                    available.offer(con);
                }
                configured = true;
            }
        }
    }
}
```

class level lock is used to synchronize all the methods

this ensures that duplicate calls to init() do not accidentally re-configure the pool

filling the queue with connection object; initially all the connections will be available

getting connection from the underlying DriverManager

configured is set to true once the pool is filled with the connection objects

all the methods use synchronization to ensure thread safety

This implementation requires a call to the init method to instantiate the pool after the very first instance of the pool is acquired via getInstance method

**Figure 5.8:** Initialize the pool

To ensure that the pool doesn't get initialized more than once or an already initialized pool is not overridden with new database connections, double-check whether the locking technique is used along with a synchronization block that safeguards the block from being accessed by more than one thread simultaneously. The

threads are only allowed to initialize the pool if the value of the variable **configured** is not `False`. The variable is set to `true` once the pool is initialized and before the thread leaves the synchronized block.

## Pool destruction

Destroying the object pool is the last operation that should be performed on the pool. This operation releases all the objects in the pool and sets its state to **not**

In [Figure](#) the **destroy** method closes all the database connections that are either available or acquired, clears both the queues, and sets the value of the configured variable to

```
public void destroy() throws SQLException {
    if (!configured) {
        throw new IllegalStateException(ILLEGAL_STATE);
    } else {
        synchronized (DatabaseConnectionPool.class) {
            if (configured) {
                for (Connection con : available) {
                    if (!con.isClosed()) {
                        con.close();
                    }
                }
                for (Connection con : acquired) {
                    if (!con.isClosed()) {
                        con.close();
                    }
                }
                available.clear();
                acquired.clear();
                configured = false;
            }
        }
    }
}
```

the check ensures that if the pool is already destroyed, no more action is necessary and throws an illegal state exception

the method is synchronized to ensure thread safety

close the available connections; also verify if the connections are already closed before closing them

close all the acquired connections as well

clear both the queues that keep track of the available as well as the acquired connections

the destroy method closes the connections, clears all the queues and finally marks the connection pool as not configured

mark the pool not configured

**Figure 5.9:** Destroy the pool

If the destroy method is called and the pool has already been destroyed, the request throws an illegal state exception. Our implementation does not wait for the acquired connections to be released. This is deliberately done to keep the example simple and to focus on the design pattern. The object pool design pattern provides a framework to create and manage a pool of objects and there are many ways to implement it. As discussed earlier, a pool of objects could be initialized eagerly or lazily, it could support many different types of objects and could possibly have many different if not unique ways to acquire an object and release it back to the pool. The implementations depend solely on the requirement, the design of the pool and the type of data structures used to support the objects that form the pool.

Each connection object could have a reference to its current owner, which could help with validating the release process to ensure the connection is released by the correct owner since the reference to an already released connection could be held up by some previous owner and an accidental request could be triggered to release it.

## Acquiring objects

Acquiring an object from the pool is a straightforward process. The mandatory requirement for it to succeed is to have an object available in the pool. [Figure 5.10](#) demonstrates a simple algorithm that allows to acquire an object from the object pool:

```
public Connection acquire() {
    verifyPoolState(); → verify if the connection pool is in configured state
    Connection con = null;
    synchronized (DatabaseConnectionPool.class) {
        if (!available.isEmpty()) {
            con = available.poll();
            acquired.offer(con);
            available.remove(con);
        }
    }
    return con; → return a connection
}
```

the acquire method checks if a connection is available, polls one from the available queue, moves it to the acquired queue and returns the connection to the client

**Figure 5.10: Acquire objects**

Similarly, in our implementation the **available** queue is checked for an object that is moved to the **acquired** queue and subsequently returned to the requestor:

```
private void verifyPoolState() {  
    synchronized (DatabaseConnectionPool.class) {  
        if (!configured)  
            throw new IllegalStateException(ILLEGAL_STATE);  
    }  
}
```

synchronized

throws an illegal state exception if the pool is not configured

**Figure 5.11:** Verify pool state

Before fulfilling a request to acquire an object, the pool is verified to be in the correct state as demonstrated in [figure](#). The pool is in the correct state if it has been properly initialized and hasn't been destroyed; otherwise, an illegal state exception is thrown.

This ensures that the request to acquire a connection does not lead to any surprise and helps in graceful handling of such requests.

## Releasing objects

To release an object back to the pool, the owner must call the release method and pass a valid connection object. [Figure 5.12](#) demonstrates a simple algorithm that allows to release an object back to the object pool. Care must be taken to ensure that the passed object was indeed acquired from the pool; otherwise, the release method will throw and illegal argument exception:

```
public void release(Connection obj) {
    verifyPoolState(); → verify if the connection pool is in a configured state
    if (null == obj)
        throw new IllegalArgumentException(INVALID_RELEASE_CONNECTION);
    synchronized (DatabaseConnectionPool.class) {
        if (acquired.isEmpty() || !acquired.contains(obj)) → verifying the
            throw new IllegalArgumentException(INVALID_RELEASE_CONNECTION);
            passed object
        acquired.remove(obj);
        available.offer(obj); } ↓
            snapping the connection object
            the release method validates the state of the
            connection pool, checks if the passed object is
            actually the one acquired from this pool and moves
            it to the available queue
```

**Figure 5.12:** Release objects

For cases, where an already released object is again passed to the release method, an exception is thrown as well. In our implementation, the released connection object is moved from the **acquired** queue to the **available** queue to make it available for reuse. It should be noted that our implementation does not provide an out-of-the-box solution to ensure that the acquired

objects are indeed released back to the pool, and therefore, the acquirer must ensure its release after its use.

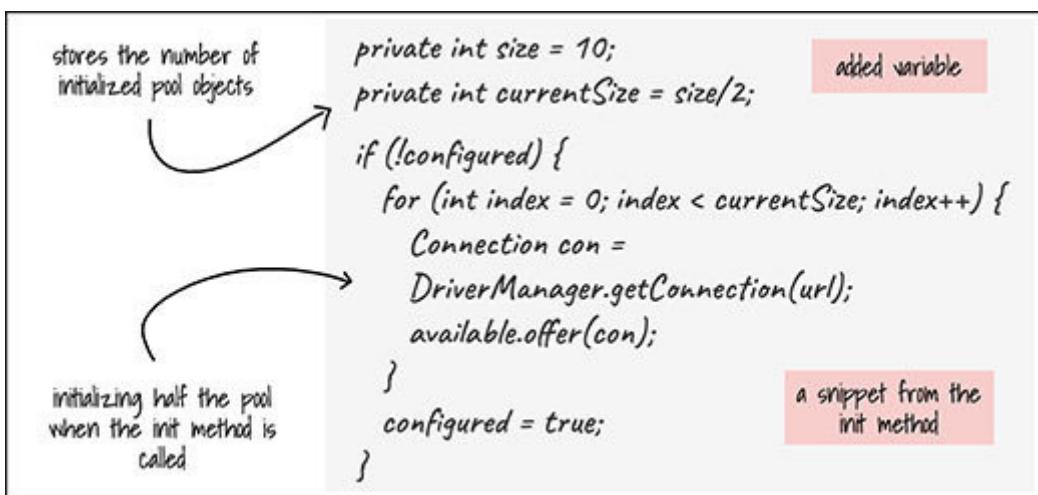
## Varying implementations

An object pool can be implemented in many ways depending on the requirement and the choice of data structures. In this section, we will explore some of those variations, for example, partial initialization or lazily created pool objects, and so on. We will also discuss some improvements to our basic pool design that will help us to create a more robust object pool.

## Partial initialization

An object pool can be filled with a smaller number of objects at the time of initialization, dynamically constructing more objects as and when required.

This allows for a lazy initialization process where all the objects do not take up space when the pool is first constructed, instead they are added to the pool as required. [Figure 5.13](#) demonstrates the changes we have made to our earlier approach of constructing and initializing an object pool:



The diagram shows a code snippet from the `init` method of an object pool. The code initializes the pool to half its total size. It includes annotations explaining the purpose of certain variables and sections of code.

```
private int size = 10;  
private int currentSize = size/2;  
  
if (!configured) {  
    for (int index = 0; index < currentSize; index++) {  
        Connection con =  
            DriverManager.getConnection(url);  
        available.offer(con);  
    }  
    configured = true;  
}
```

Annotations in the code:

- A callout points to `currentSize` with the text "stores the number of initialized pool objects".
- A callout points to the loop body with the text "initializing half the pool when the init method is called".
- A callout points to the entire code block with the text "added variable".
- A callout points to the final brace of the code block with the text "a snippet from the init method".

**Figure 5.13:** Partial Initialization I

The new implementation only constructs half the objects and adds them to the pool. The rest of the objects are constructed overtime when the need arises. [Figure 5.14](#) demonstrates the changes in the `acquire` method:

```
if(available.isEmpty() && currentSize<size) {  
    con = available.poll();  
} else if (!available.isEmpty()) {  
    con = DriverManager.getConnection(url);  
    currentSize++;  
}  
  
added logic to initialize an  
object when there is no  
object available and the  
final size of the pool is  
not yet reached
```

a snippet from the  
acquire method

**Figure 5.14:** Partial Initialization II

The **acquire** method can now construct and add a new object to the pool if there are no objects available to be acquired and the final pool size is not reached.

## Releasing objects – Who is the owner?

Releasing an object back to the object pool is an important task and should be handled with utmost care. When an object is acquired from the pool, the acquirer is termed the owner; what if the acquired object is mistakenly released by some other thread and not the owner? This could have consequences, as the object might already have been reacquired by some other thread, while the earlier owner still has its reference.

To prepare for such a use case, a simple solution is to keep a map of all the connections and their present owner threads. The map is updated each time a connection object is acquired from or released back to the pool. The same map will be used to verify whether the request to release a connection object back to the pool is made by the owner of that connection. In case of an invalid request, an exception will be thrown to safeguard the consistency of the pool.

In [Figure](#) we have introduced a map of connection objects and a corresponding string value to store the name of the thread that currently owns that connection:

```
private static final String INVALID_OWNER =  
    "Release request received from an invalid owner";  
  
private final Map<Connection, String> ownerMap;
```

added variables

map to store the  
current owner of  
each connection

**Figure 5.15:** Releasing objects I

[Figure 5.16](#) demonstrates the changes in the **acquire** method:

```
public Connection acquire() {
    verifyPoolState(); ← verifying the state of
    Connection con = null; ← the pool before acquiring
    synchronized (CustomDatabaseConnectionPool.class) {
        if (!available.isEmpty()) {
            con = available.poll();
            String currentThread = Thread.currentThread().getName();
            ownerMap.put(con, currentThread); ← updating the map with the latest
            acquired.offer(con); } snapping the
            available.remove(con); } connection
        }
    }
    return con; ← the connection will now be acquired by the requesting
}
```

the changes are identical apart from the logic to store the connection owner in a map

the connection object

owner of this connection

the connection will now be acquired by the requesting thread until it is released back to the pool

**Figure 5.16:** Releasing objects II

A synchronize block guards the connection acquirement process to ensure that no two threads can access and acquire the same connection. The queues are updated to reflect that an available connection has now been acquired.

As demonstrated in [Figure](#) every time a connection is acquired or released, the map is also updated. When a connection is acquired, the map is updated to store the owner relationship of the connection with the thread that acquires it. In case of a release,

the relationship is severed by associating the connection to a null value:

```
public void release(Connection obj) {  
    verifyPoolState();  
    if (null == obj) throw new  
        IllegalArgumentException(INVALID_RELEASE_CONNECTION);  
  
    synchronized (CustomDatabaseConnectionPool.class) {  
        if (acquired.isEmpty() || !acquired.contains(obj)) throw new  
            IllegalArgumentException(INVALID_RELEASE_CONNECTION);  
  
        String currentThread = Thread.currentThread().getName();  
        if(ownerMap.get(obj).equals(currentThread)) { ←  
            acquired.remove(obj);  
            available.offer(obj);  
            ownerMap.put(obj, null);  
        } else {  
            throw new InvalidActivityException(INVALID_OWNER);  
        }  
    }  
}
```

ensuring consistency of the object pool and safeguarding against invalid releasing of objects back to the pool

verifying that the present owner of the connection object is the thread that made the request to release it

if not then throw an invalid activity exception

Figure 5.17: Releasing Objects III

## Advantages and drawbacks

It is time to make note of some of the benefits and drawbacks of the facade design pattern. Let us start with the benefits first before we explore the drawbacks of it:

Provides ready-to-use, reusable objects that reduces the need for on-demand object creation

Improves performance of the application by reducing resource consumption<sup>2</sup>

The drawbacks of the object pool design pattern are:

If the objects are not released properly, it could adversely impact the behavior and performance of the application

## Usage

Some of the scenarios relevant for building an object pool are:

When the objects are expensive to construct and are supposed to be reused

When the application requires the ability to support parallel processing

## Conclusion

Object pool design pattern is a widely used creational design pattern that can be utilized to build pool of reusable objects. It provides a framework to pre-construct a pool of similar type of objects that can be reused to improve performance. The design pattern helps to reduce the time required for constructing objects again and again and facilitates loading objects at the time of system start-up.

An object pool is constructed in scenarios where the objects consume a lot of computing resources, or the objects require a large time to construct. An object pool is also applicable in scenarios where the objects are candidates for parallel processing, or the system has a limited memory.

In the next chapter, we will discuss the adapter design pattern and will learn how to connect two incompatible interfaces to work with each other.

## Questions

In this section we will sharpen our brain by indulging in some fun brain activities. Let's get started...

## Pick the odd one out

high resource consumption	fix sized	reusable object
reusable	partially initialized	immutable object
multi-threaded	infinitely large	singleton object
bulk initialization	variable sized	mutable object

*Figure 5.18: Pick the Odd one Out*

### Find the odd one out

Object pool design pattern

Helps to reduce time complexity

Requires large memory to implement

Provides a framework to manage a pool of objects

Requires parallel processing capabilities

Objects that form an object pool

Are of the same type

Must implement the object pool interface

May consume high resources

Can be lazily initialized by the pool

Valid methods of the object pool interface are

destruct

init

release

acquire

## Complete the code

```
public void    () throws SQLException {           destroy
    if (!configured) {
        synchronized (                                ) {
            if (!                                ) {
                for (int index = 0; index < size; index++) {
                    Connection con =
                        .getConnection(url);
                    .offer(con);
                }
            }
            configured =      ;
        }}}
```

DriverManager  
Database  
Connection  
Pool.class

```
public void    () throws SQLException {           available
    if (!configured) {
        throw new IllegalStateException(ILLEGAL_STATE);
    } else {
        synchronized (                                ) {
            if (configured) {
                for (Connection con :      ) {
                    if (!con.isClosed()) {
                        con.      ();
                    }
                }
                for (Connection con :      ) {
                    if (!con.isClosed()) {
                        con.      ();
                    }
                }
                available.clear();
                acquired.clear();
            }
            configured =      ;
        }}}
```

false  
init  
available  
true  
Database  
Connection  
Pool.class  
close

**Figure 5.19:** Complete the code

Select two correct answers

Object pool design pattern

Provides one object at a time

Can work in multi-threaded environment

Is suitable for infinite objects

Can only be used for non-immutable objects

Methods supported by object pool design pattern

construct

acquire

destruct

destroy

Which objects would you make a pool of?

Immutable objects

Private objects

Resource consuming objects

Singletons

What goes together

init, acquire, release, destroy

high resource consumption, object pool

object pool, singletons, immutable objects

destruct, get, put, initialize

## Answers

### Pick the odd one out

Image [Column Wise]

high resource consumption, infinitely large, mutable object

### Find the odd one out

Requires large memory to implement

Are of the same type

Destruct

**Select two correct answers**

Provides one object at a time, can work in multi-threaded environment

acquire, destroy

immutable objects, resource consuming objects

[init, acquire, release, destroy] & [high resource consumption, object pool]

## CHAPTER 6

### Adapter

---

You can find the editable code files for this chapter on the book's GitHub link.

<https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers/tree/main/Chapter%2006>

---

## Structure

Topics that make this journey worthwhile:

Incompatible interfaces

Adapter design pattern

Data analysis adapter

Adapter design pattern - defined

Benefits and drawbacks

Usage

## Objective

Our objective is to learn to integrate incompatible interfaces. Interfaces are not always developed to support effortless integration; in fact, the developer might not even be aware of any possible integration requirement at the time of developing an interface. Interfaces that express different functionalities are therefore often hard to integrate without making any changes. In this chapter, we will learn about the adapter design pattern and how to utilize it to integrate and reuse existing interfaces.

We will also walk you through the implementation details of the adapter design pattern, its benefits, and a few drawbacks along with some of the places where this design pattern can be used.

## Introduction

When two or more systems interact with each other, it is not unusual to have similar operations represented in different form and shape. It is often the case when the systems are developed separately and have no prior knowledge of others existence before they are integrated. The integration requires the two systems to interact, but before anything else, the systems need a way to understand each other.

The integration must ensure that there is as little change as possible in the existing systems; otherwise, we will end up modifying a lot more than required. In an ideal scenario, there should be no change required in the systems that are candidates for integration.

## Incompatible interfaces

You must have heard of this famous phrase – *no need to reinvent the* the cost of re-writing application code that already exists could be more than the estimations. Existing modules of code that have passed the test of time are usually efficient and do not require a re-write, although some improvements could be on the cards.

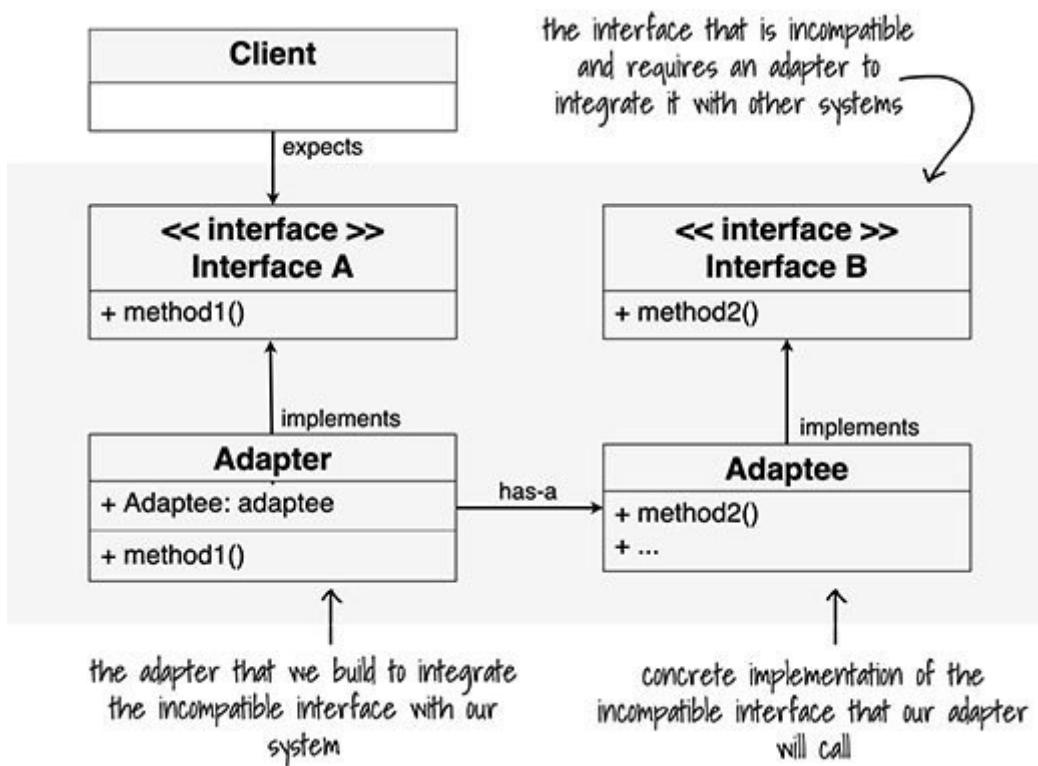
The world of software has grown exponentially; they perform a variety of tasks and are often housed with algorithms that can be utilized in multiple business use cases. These common code blocks are usually written keeping in mind the reusability principle.

The crux here is that it is not always straightforward to integrate; what this means is that the interfaces don't always match, or the methods hosted by these common code blocks have different signatures. So, does this mean we have to re-write? No, we don't. Instead, we can write adapter classes that help us integrate incompatible interfaces without modifying them.

## Adapter design pattern

The adapter design pattern integrates two incompatible interfaces. It allows an interface implementation to mimic another interface and ensures no modifications in either of the existing interfaces. The adapter pattern provides a neat way to integrate already designed interfaces with no strings attached; the adapter implementation is a standalone class that extends no obligation to the existing code, something that can be removed if required without affecting the system as such.

In [Figure](#) the client expects an interface of type but the one exposed by the existing system is of type



### **Figure 6.1: Adapter Design Pattern**

Although the system provides the required features, the incompatibility restricts its usage. To overcome this restriction, one can make use of the adapter design pattern.

Here's how it works:

The client calls **method1** of adapter implementation with the required input parameters.

The adapter converts the input request into a form expected by the adaptee.

The adapter then calls **method2** of adaptee passing in the converted parameters and receives a response.

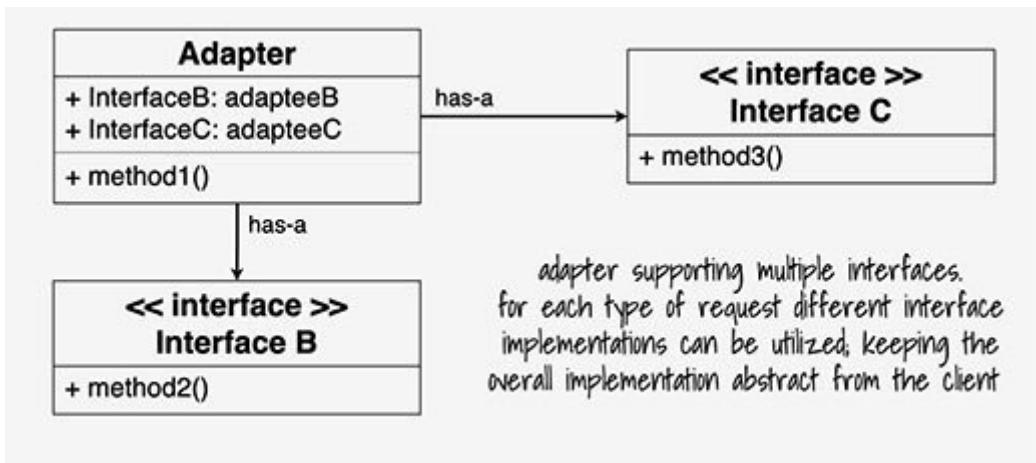
The response, if required, is then converted to a form expected by the client before it is finally returned to the client itself.

The overall request flow is abstract to the client that remains unaware of the adapter presuming that the response is received directly from the adaptee implementation, when the request is first converted and then delegated to the correct interface implementation, all with the help of an adapter. This way, the adapter solves the problem of incompatibility among the interfaces

and provides a clean approach to reuse the otherwise difficult-to-use interfaces.

The adapter can also be made to support more than one adaptee implementations with unique references pointing to various interface types if the features expected by the client are not served by a single interface. In such a case, the adapter implementation must have references to each type of interface that together provide all the features expected by the client.

[Figure 6.2](#) demonstrates the use of a single adapter to integrate multiple interfaces together:



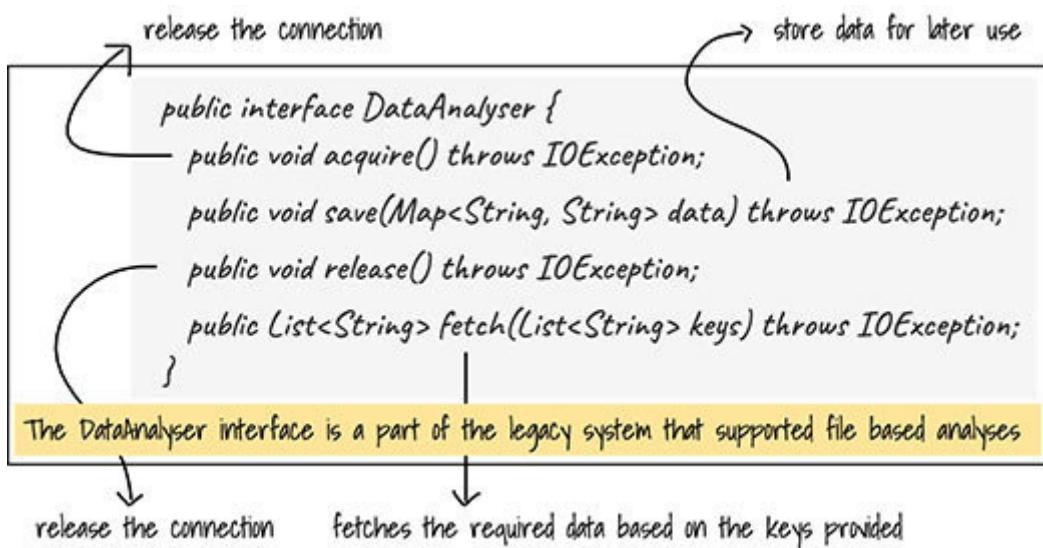
**Figure 6.2: Adapter Supporting Multiple Interfaces**

## [Building an adapter](#)

The example that we will walk you through in this chapter makes use of a data analyser that uses both modern as well legacy system to store and fetch data to perform analysis. We will learn how the adapter design pattern can be utilized to integrate a legacy implementation with a modern one to make it usable with the modern-day clients that are incompatible with the legacy systems. Let's first look at the legacy interface prior to looking into the modern interface that supports NoSQL-driven data storage mechanism.

## Interfaces

The interface demonstrated in the [Figure](#) is a part of the legacy, file-based data storage system and provides us with certain methods to support data insertion and read directly from the file system. This interface provides us with two additional methods apart from the usual **save** and **fetch** methods to store and fetch the record entries from the file system; those two methods are **acquire** and that as name suggests are used to acquire and release the file resource in a multithreaded environment:



**Figure 6.3: Interface DataAnalyser**

The interface demonstrated in [Figure](#) is a part of the new, NoSQL-based data storage system and provides us with certain methods to support data insertion and read from the distributed NoSQL database system, very similar to what the previous interface did. It

is also the interface that is expected by the client. The client understands the method signatures and has logic written based on it:

→ stores the data in the data storage, can support both modern and legacy stores

```
public interface ModernDataAnalyser {  
    public void store(DataStorage storage, Map<String, String> data)  
        throws IllegalArgumentException, DataStorageException;  
    public List<String> fetch(DataStorage storage, Set<String> keys)  
        throws IllegalArgumentException, DataStorageException;  
}
```

The ModernDataAnalyser is a part of the new upcoming system that can support a variety of data storage mechanisms for data analysis

→ fetches the data from the datastore based on the input keys

**Figure 6.4:** Interface ModernDataAnalyser

Both these interfaces perform very similar tasks but provide different set of methods. While the new interface is understood well by the client, the legacy one is not.

To make use of the legacy interface and its implementations and to make it compatible with the client, we can write a wrapper class around the legacy interface that will encapsulate the interface implementation and take care of the input and output translations for us. This wrapper class must implement the interface that is understood by the client and must also ensure that it bridges the communication gap with the legacy interface.

The wrapper class we just discussed is an adapter; it is an implementation of the type the client understands. It wraps the legacy interface implementation giving it a new face, a face that can be integrated with the client.

## Concrete implementations

Let's now explore some of the implementations for both the legacy and the modern interface. [Figure 6.5](#) demonstrates an implementation of the legacy interface and provides access to the file-based data storage:

```
public class FileIODataAnalyser implements DataAnalyser {
    public void acquire() throws IOException {
        print("Permission acquired to save data for analysis");
    }
    public void release() throws IOException {
        print("Permission released after saving data for analysis");
    }
    public void save(Map<String, String> data) throws IOException {
        FileBasedDataAnalyser.save(data);
        print("Data saved successfully for analysis");
    }
    public List<String> fetch(List<String> keys) throws IOException {
        List<String> data = FileBasedDataAnalyser.fetch(keys);
        print("Data fetched successfully for analysis");
        return data;
    }
}
```

concrete implementation of the legacy interface

↑

FileBasedDataAnalyser takes care of reading  
from and writing to the file system

**Figure 6.5: FileIODataAnalyser**

The implementation shown isn't highly productive as it doesn't really show us how exactly we can use the file system to store and fetch data. To keep it short and simple, we are only focusing on the design aspects.

[Figure 6.6](#) demonstrates an implementation of the new interface and provides access to the NoSQL database for data storage and retrieval. This implementation is compatible with the client system and does not require to be wrapped by an adapter; although in our example, we will wrap a reference to this implementation as well to showcase how a single adapter can work with multiple different implementations:

```

    concrete implementation of the modern interface
    ↓
public class ElasticDataAnalyser implements ModernDataAnalyser {
    private static ElasticDataAnalyser _instance = new
    ElasticDataAnalyser(); ←
    private ElasticConnector connector;
    private ElasticDataAnalyser() {
        // setup connection with elastic datastore
    }
    public static ElasticDataAnalyser getInstance() { ←
        return _instance;
    }

    public void store(DataStorage storage, Map<String, String> data)
    throws IllegalArgumentException, DataStorageException {
        connector.save(data); ← delegating the requests to
        print("Data stored successfully in Elastic"); elastic connector
    }

    public List<String> fetch(DataStorage storage, Set<String> keys)
    throws IllegalArgumentException, DataStorageException {
        List<String> data = connector.fetch(keys); ←
        print("Data fetched successfully from Elastic");
        return data;
    }
}

```

using the static singleton approach to initialise and reference the data analyser

delegating the requests to elastic connector

delegating the requests to elastic connector

an implementation of the ModernDataAnalyser interface that stores and fetches data

**Figure 6.6: ElasticDataAnalyser**

## The adapter

The adapter implementation acts as a bridge between two incompatible interfaces. It implements one interface and acts as a wrapper to implementation of the incompatible type. In [Figure](#) the adapter will act as a single point of interaction for the client and will wrap both the legacy and modern implementations:

The diagram shows a code snippet for the `DataAnalysisAdapter` class. A callout arrow points from the text "implements the compatible interface" to the first line of the code. A yellow box at the bottom contains the text "adapter implementation to collectively support both legacy and modern system".

```
public class DataAnalysisAdapter implements ModernDataAnalyser {  
    private ModernDataAnalyser elasticAnalyser =  
        ElasticDataAnalyser.getInstance();  
  
    public void store(StorageType storage, Map<String, String> data)  
        throws IllegalArgumentException, DataStorageException {  
        // collective implementation  
    }  
  
    public List<String> fetch(StorageType storage, Set<String>  
        searchKeys) throws IllegalArgumentException, DataStorageException {  
        // collective implementation  
    }  
}
```

adapter implementation to collectively support both legacy and modern system

**Figure 6.7: Adapter Implementation**

Do we really need to support the implementations that are compatible themselves? No, it is not required. The adapters are supposed to make incompatible interfaces compatible so that they

can be integrated with the rest of the system and are not required for already compatible interfaces.

The reason we have encapsulated multiple interface implementations within our adapter is just to showcase that one adapter can be used with multiple interface implementations, and also because the modern interface has the **StorageType** parameter that provides the type to be used for the incoming request. If there is no way to distinguish that, it won't be possible for the adapter to support multiple interface implementations.

Now, let's look into the two method definitions that perform the task of translating the data and delegate it to the actual implementations. The store method, for the elastic analyser does not perform any special action and simply delegates the call to the actual implementation; for the file-based analyser, the store method makes some additional calls to the legacy implementation apart from the call to the save method to ensure the correctness of its behavior, in turn adapting to and bridging the gap between the client and the legacy interfaces.

[Figure 6.8](#) demonstrates the implementation of the **store** method that supports both the legacy and modern system:

```

    connecting to the new system to store data in elastic datastore
    ↪
public void store(DataStorage storage, Map<String, String> data)
throws IllegalArgumentException, DataStorageException {
    if (DataStorage.ELASTIC.equals(storage)) {
        elasticAnalyser.store(storage, data);
    } else {
        FileIODataAnalyser fileIODataAnalyser = new FileIODataAnalyser();
        try {
            fileIODataAnalyser.acquire();
            fileIODataAnalyser.save(data);
            fileIODataAnalyser.release();
        } catch (IOException e) {
            throw new DataStorageException(e);
        }
    }
}

```

} connecting to the legacy system  
to store data in files, that can be  
later used for analysis

adapter implementation of the store method to  
support both the legacy and modern system

**Figure 6.8:** Adapter Implementation II

If you are thinking about the translation of the input and the output, then it is not required because the method signatures for both the interfaces expect the same input parameters. The fetch method on the other hand requires input translation from one form to another to support the method signature of the legacy interface.

[Figure 6.9](#) demonstrates the implementation of the **fetch** method that supports both the legacy and modern system. If you would have noticed, the exceptions thrown by the legacy interface methods are also different from the ones expected by the client:

```

public List<String> fetch(DataStorage storage, Set<String> searchKeys)
throws IllegalArgumentException, DataStorageException {
    List<String> data = Collections.emptyList();
    if (StorageType.ELASTIC.equals(storage)) {
        data = elasticAnalyser.fetch(storage, searchKeys);
    } else {
        FileIODataAnalyser fileIODataAnalyser = new FileIODataAnalyser();
        try {
            fileIODataAnalyser.acquire();
            List<String> searchArguments = new ArrayList<>(searchKeys);
            data = fileIODataAnalyser.fetch(searchArguments);
            fileIODataAnalyser.release();
        } catch (IOException e) {
            throw new DataStorageException(e);
        }
    }
    return data;
}

```

adapter implementation of the fetch method to support both the legacy and modern system

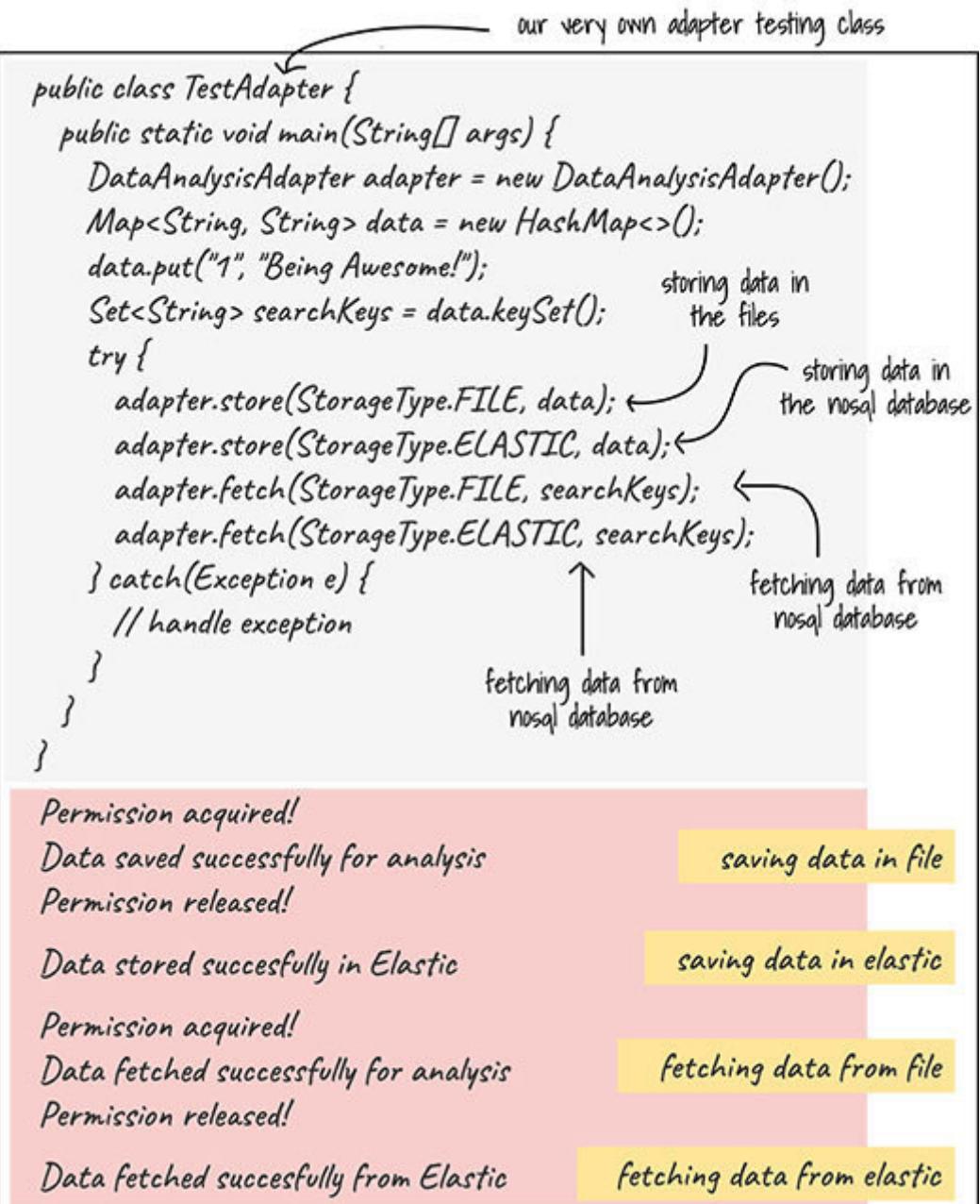
**Figure 6.9: Adapter Implementation III**

In short, the adapter is responsible for ensuring that the client is totally ignorant of the existence of the adapter.

## Testing the adapter

We have written a test class to verify that the pattern we worked on works. Let's explore the same before we move ahead with the next section.

In [Figure](#) we call the adapter implementation passing in the correct storage type and let the adapter take care of the translation and compatibility for us. Based on the provided storage type, the adapter calls the relevant implementation to perform the requested operation:



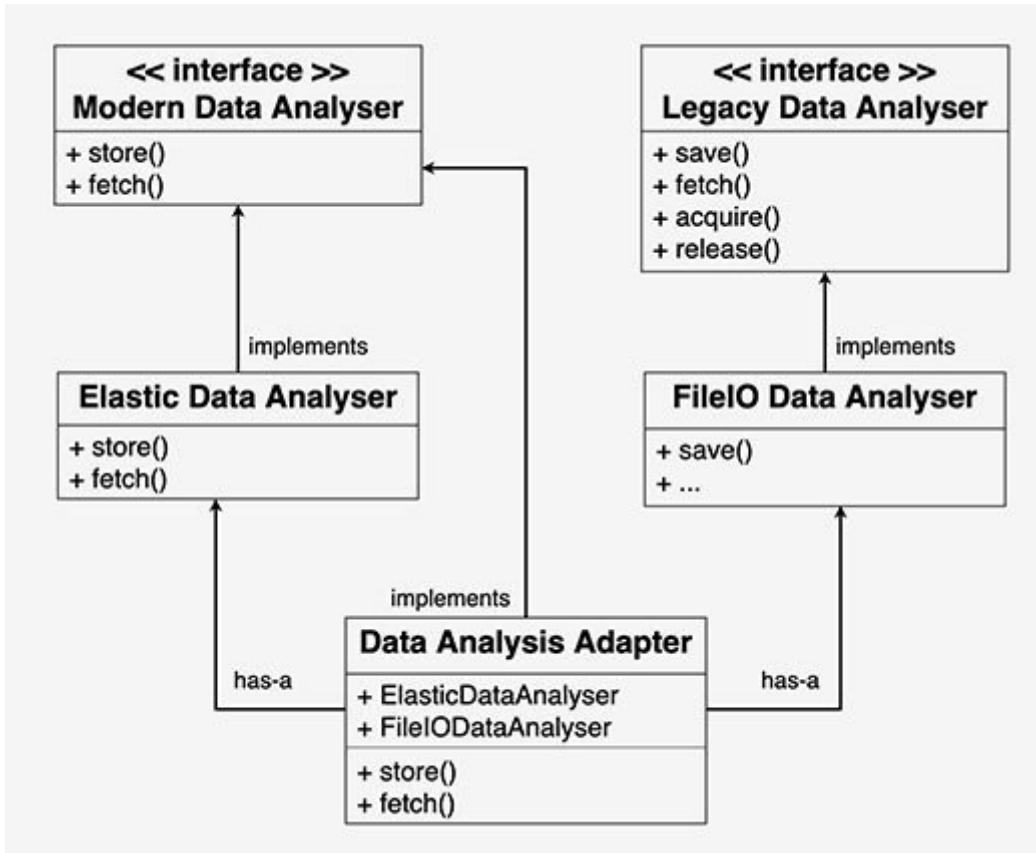
**Figure 6.10: Test Adapter Implementation**

The preceding example demonstrates how an adapter implementation can be used to work with multiple interface implementations.

## Adapter design pattern - defined

The adapter design pattern, as the name implies, acts as an adapter to make two or more interfaces compatible to each other. It allows a class to be used as another class or to mimic its behavior. The **adapter** class implements one of the interfaces and wraps the other to form a communication channel between the two interfaces so that the two can work together, which otherwise is not possible. All this is done just by introducing an adapter and without modifying any of the existing code, be it the incompatible interfaces or the client.

In [Figure](#) the **DataAnalysisAdapter** class implements the **ModernDataAnalyser** interface and wraps **FileIODataAnalyser** so that the client can directly connect to and make use of the services provided by the legacy interface:



**Figure 6.11:** Adapter Design Pattern Defined

The **adapter** class in our example has a reference to both the incompatible types and allows for converting the request from one form to another, respecting both the caller and the callee.

## Benefits and drawbacks

It is time to make note of some of the benefits and drawbacks of the adapter design pattern. Let's start with the benefits first before we explore the drawbacks of it:

It provides a way to integrate incompatible interfaces.

It allows to reuse existing functionality.

It requires little or no change in the existing interfaces and their concrete implementations.

It allows to separate the integration code from the primary business logic.

The drawbacks of the adapter design pattern are:

Adapter cannot be used with subclasses.

The overall complexity of the code is increased because of the introduction of adapter classes.

## Usage

Some of the scenarios relevant for constructing adapters are:

When the requirement is to reuse existing code that has an incompatible interface and requires additional measures for integration.

When two or more incompatible interfaces must communicate to perform a common task.

## Conclusion

Adapter design pattern is a widely used structural design pattern that can be utilized to make incompatible interfaces compatible without making many changes. The design pattern helps to reuse the existing code, which otherwise would have to be re-written every time there is a client that needs its services but couldn't because of the difference in their interfaces.

The adapter class performs these basic tasks:

It translates the input and output as required by the two adapting interfaces.

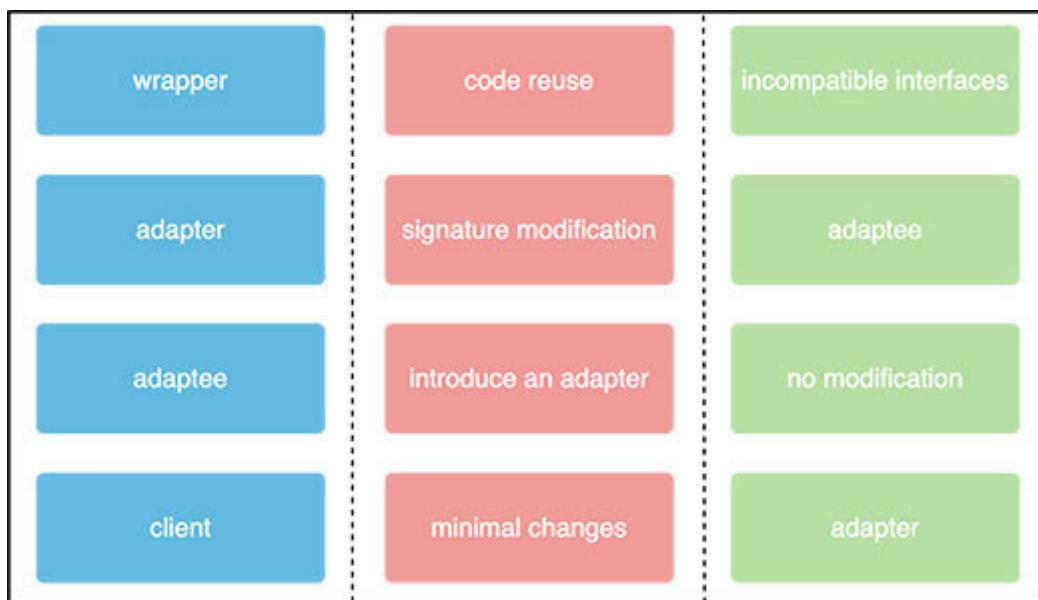
It implements the interface that is compatible with the client and wraps the implementation of the incompatible interface.

In the next chapter, we will discuss another structural design pattern known as the **decorator design pattern** and will learn another way to wrap existing code to provide new features.

## Questions

In this section, we will sharpen our brain by indulging in some fun brain activities. Let's get started...

## Pick the odd one out



*Figure 6.12: Find Odd One Out*

## Find the odd one out

Adapter design pattern

Makes incompatible interfaces compatible

Requires modifications in existing code

Introduces a wrapper class

Promotes code reuse

Adapter design pattern can be used when

There is a need to integrate incompatible interfaces

Code rewrite is not desired

Classes are wrapped for performance improvement

Existing interfaces are to be integrated

The adapter design pattern has these important actors

An adapter

An adaptee

Incompatible interfaces

Compatible interfaces

Select two correct answers

Adapter pool design pattern

Integrates incompatible interfaces

Introduces a lot of changes in the existing code

Works well with legacy architecture

Can only be used for compatible interfaces

Main actors in the adapter design pattern

Adapter

Adaptee

Singleton

Worker

What goes together

Adapter, adaptee, incompatible interfaces

Code reuse, compatible interfaces, adapter

Adapter, code reuse, adaptee

Incompatible interfaces, adapter, compatible interfaces

## Answers

Pick the odd one out

Wrapper, signature modification, no modification

## Find the odd one out

Require modifications in existing code

Classes are wrapped for performance improvement

Compatible interfaces

Select two correct answers

Integrates incompatible interfaces, works well with legacy architecture

Adapter, adaptee

[adapter, adaptee, incompatible interfaces] & [adapter, code reuse, adaptee]

## CHAPTER 7

### Decorating Objects

---

You can find the editable code files for this chapter on the book's GitHub link.

<https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers/tree/main/Chapter%2007>

---

## Structure

Topics that make this journey worthwhile:

Introduction

Wrappers

Decorator design pattern

Building a decorator

Decorator design pattern – defined

Benefits and drawbacks

Usage

## Objective

Our objective is to understand the use of wrappers to add behavior to objects, dynamically. We will learn about the decorator design pattern and how to add behavior to objects without making any changes to their structure. We will also learn about the decorator chain and the separation of concern that comes with it.

We will also walk you through the implementation details of the decorator design pattern, the benefits it provides and a few drawbacks along with some of the places where this design pattern can be used.

## Introduction

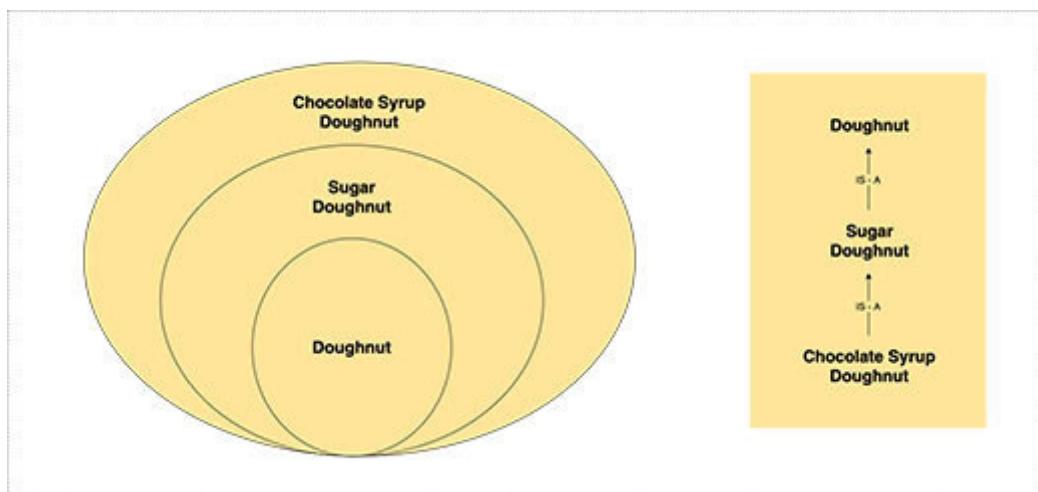
When we think of extending the functionality of a class, direct modification and sub-classing are the first two approaches that strike our mind. By altering a class structure or by extending a class, we can override the existing behavior of a type or can add additional behavior to it. Pretty straightforward, isn't it? But there are some caveats to it; with sub-classing, you cannot extend more than one class and every time there is a need for additional behavior, you end up doing the same all over again, and not all classes are open for direct modification as well.

There is another approach that we can make use of to add behavior to class objects, that too, without any modification to the existing code. The objects can be wrapped within one other in a *has-a* relationship to support additional behavior.

In this chapter, we will learn about the decorator design pattern and how it can be utilized to create decorator objects to enhance the functionality of the original object without altering its structure.

## Wrappers

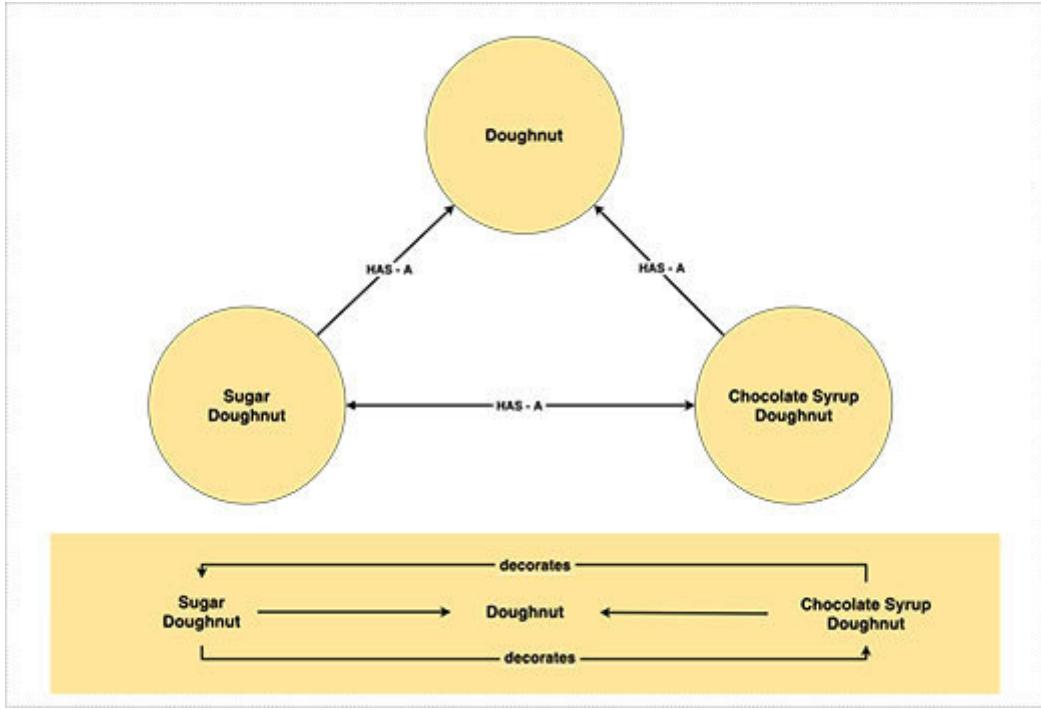
All this time we have been looking at this problem from top when we should be looking at it from the bottom:



**Figure 7.1: Class hierarchy**

[Figure 7.1](#) depicts class hierarchy in object-oriented languages.

Instead of altering or extending a type, we can simply wrap it with another class that follows the same behavioral contract. By using wrapper classes, we can provide additional behavior to a type while ensuring that the existing behavior is preserved. [Figure 7.2](#) depicts the use of composition to support additional functionality:



**Figure 7.2:** Wrapper objects

By extending a class, all its objects share the same behavior, but with wrapping an object, the behavior is added dynamically, that too, to the object that is wrapped while all other objects of its type aren't affected at all.

## Decorator design pattern

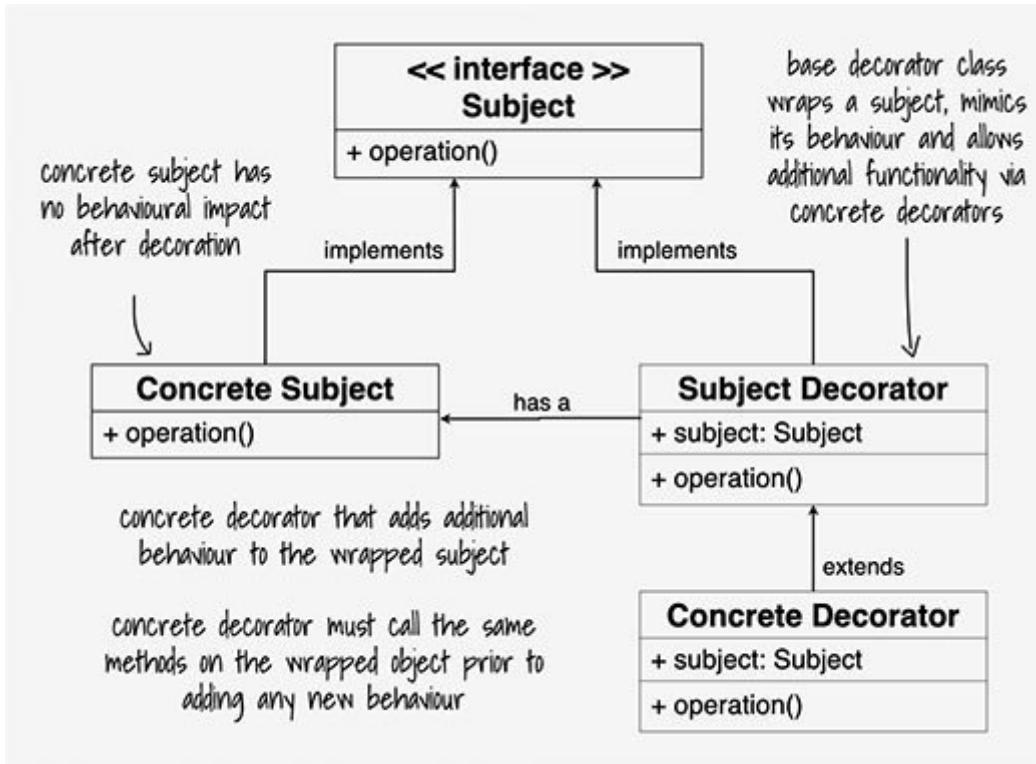
The decorator design pattern is a structural design pattern that allows additional behavior to be added to an object without altering the behavior of other objects of the same type. It makes use of decorator classes that implement the same interface as the subject class. The decorator object wraps the subject, the original object, to provide additional behavior or to enhance its current behavior.

For adding behavior to a class, the usual consideration is either to alter its structure or to subclass or extend it. The decorators are an alternative to that approach that do not require any modification in the subject class. Due to the dynamic addition of behavior, the decorator design pattern is very useful when we do not have access to the subject classes.

The decorator design pattern can also be used to enhance the behavior of objects that, by definition, are not open for extension, that is, immutable objects. Such objects can be wrapped by a decorator to provide additional behavior.

By design, a decorator can wrap another decorator and form a chain of additional behavior. This allows segregation of logic into multiple decorators and promotes the single responsibility principle and the principle of separation of concern.

In [Figure](#) the **Decorator** implements the **Subject** interface like a concrete implementation:



*Figure 7.3: Decorator design pattern*

While the concrete implementation is itself of type **Subject** and so is the the decorator class also has a reference of the interface type. This reference plays the vital role in the decorator design pattern, it could be assigned an object of that type that in-turn is decorated with the additional behavior.

Here's how it works:

An object of type **Subject** is instantiated

A decorator is instantiated, that wraps the object constructed in the first step

When a method is called on the decorator instance, it calls the same method on the wrapped object and appends any additional behavior written in the decorator

The decorator appends the additional behavior to the method call without altering the existing behavior of the wrapped object. By creating more such decorators, multiple behaviors can be written to add to the default behavior of the concrete subject implementation.

The decorators can wrap any object of the interface type, which includes the decorator themselves, opening the prospect of appending multiple behaviors on top of each other in a stack-like manner. There is no specific order in which the decorators can be used, and thus different objects can be made to behave differently following the sequence the decorators are arranged to decorate the subject.

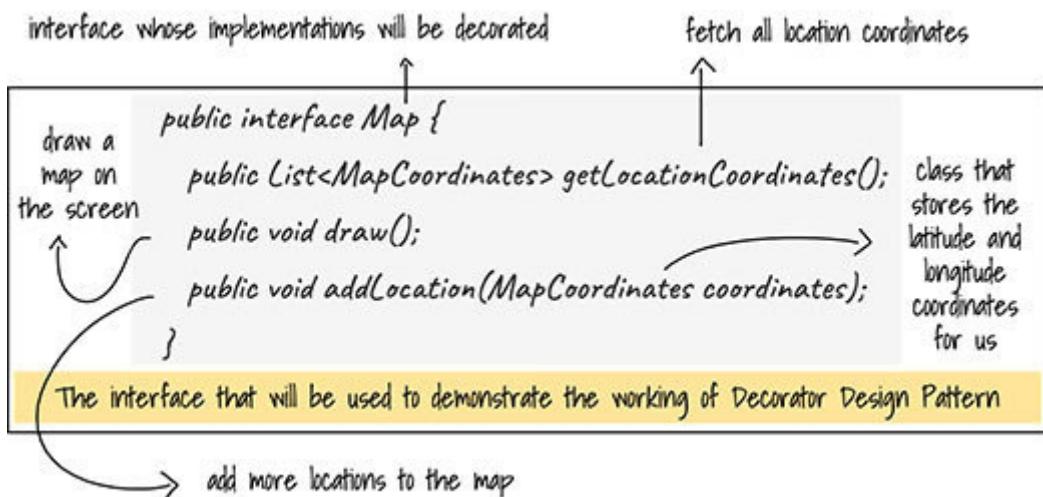
Let us walk through an example to add clarity to our understanding of the decorator design pattern.

## [Building a decorator](#)

The example that we will walk you through in this chapter makes use of a map interface, a concrete implementation of it, and a few decorators. We will learn how the decorator design pattern can be utilized to append additional behavior to existing object instances. We will also learn about the decorator chain that can be formed by wrapping decorators within decorators and the way it can be used to append behaviors in many different orders. Let us first look at the interface that sets up the common contract that both the concrete implementations and the decorators must follow for the design to work.

## The interface

The interface is the central point of the decorator design pattern. In [Figure](#) the **Map** interface provides the behavior that must be implemented by the concrete implementations as well as the decorators:



**Figure 7.4: Interface Map**

This interface provides us with three methods that can be used to draw a map, add more locations on the map and to fetch all the location coordinates.

## The subject

[Figure 7.5](#) presents a concrete implementation of the interface shown in [Figure](#). This concrete implementation is the subject in the decorator design pattern. The decorators that we will later construct in this chapter will wrap an instance of our subject to provide additional behavior on top of what is already provided by the subject:

our very own Street Map implementation

```
public class StreetMap implements Map {  
    protected MapUI mapUI = MapUI.getInstance();  
    protected List<MapCoordinates> coordinates;  
    public StreetMap(List<MapCoordinates> coordinates) {  
        this.coordinates = coordinates;  
    }  
    public void draw() {  
        mapUI.drawShortestPath(coordinates);  
    }  
    public void addLocation(MapCoordinates coordinates) {  
        this.coordinates.add(coordinates);  
        mapUI.addLocation(coordinates);  
    }  
    public List<MapCoordinates> getLocationCoordinates() {  
        return coordinates;  
    }  
}
```

class that handles the Map UI and process input requests

draw the shortest path between the coordinates

add more locations

**Figure 7.5: Street Map | Concrete Implementation**

This is just one of the implementations of the interface and there could be many others based on the rendering required for different scenarios, for example, street view, satellite view, terrain view, and so on.

### Base decorator

The base decorator and the subject implement the same interface, but the decorator is more than just a concrete implementation. It wraps an object of the same type as that of the interface it implements. It does not have any behavior of its own and delegates all the incoming requests to the object it wraps. This is the most crucial aspect of the base decorator.

Figure 7.6 showcases a base decorator. The decorator class is structured in a manner to decorate whichever object is passed to it, be it a concrete implementation of that type or another decorator that implements the same interface:

```

public class MapDecorator implements Map {
    protected Map map;
    protected MapUI mapUI = MapUI.getInstance();

    public MapDecorator(Map map) {
        this.map = map;
    }

    public void draw() {
        map.draw();
    }

    public void addLocation(MapCoordinates coordinates) {
        map.addLocation(coordinates);
    }

    public List<MapCoordinates> getLocationCoordinates() {
        return map.getLocationCoordinates();
    }
}

acts as a wrapper
to an existing object
of type Map
call methods of the
wrapped type
the base decorator
should always imitate
the behaviour of the
wrapped object
implements the same
interface as the wrapped
type, i.e. Map

```

the decorator class allows to add behaviour to the concrete implementations of the 'Map' interface without affecting their existing behaviour

**Figure 7.6: Map Decorator**

The definitions provided by the base decorator class does nothing significant, but it does ensure to mimic the behavior of subject passed to it. You must be wondering why to even have a base decorator when all it does is just to call methods of some wrapped object and when the concrete implementations could have done the same as well. The decorators are not supposed to override all the methods but only those that must support additional behavior, a base decorator helps here by providing default implementations to every method of the interface, effectively reducing the need for the concrete ones to implement them.

Apart from the methods provided by the interface, the decorators can also have their own methods that can provide features missing in the subject class, the one we wrap with decorators.

Let us now look into some of the decorators and understand the overall working of the decorator design pattern.

## Concrete decorators

[Figure 7.7](#) and [Figure 7.8](#) present two concrete decorators that extend the base decorator and override the relevant methods. The overridden methods can be seen making a call to the super implementation to make sure that the default implementation is always executed:

```
public class PinnedLocationMap extends MapDecorator {  
    public PinnedLocationMap(Map map, MapCoordinates[] coordinates) {  
        super(map);  
    }  
    public void draw() {  
        super.draw();  
        mapUI.pinLocations(map.getLocationCoordinates());  
    }  
    public void addLocation(MapCoordinates coordinates) {  
        super.addLocation(coordinates);  
        mapUI.pinLocation(map.getLocationCoordinates());  
    }  
}
```

the concrete decorator implementations extend the functionality of the wrapped object

always calls the method of the wrapped object before adding behaviour

concrete decorator

always calls the method of the wrapped object before adding behaviour

pin location of the coordinates on the map

additional behavior

**Figure 7.7:** Pinned Location Map Decorator

[Figure 7.7](#) demonstrates a decorator that pins the locations on the map both when the map is drawn and when a new location is added to it. [Figure 7.8](#) demonstrates a decorator that prints the distance between the locations as well as the time required to

travel that distance when the map is drawn. Only one of the methods is overridden and the base decorator takes care of calling the super implementation for the other one:

```
public class TravelerMap extends MapDecorator {  
    public TravelerMap(Map map, MapCoordinates[] coordinates) {  
        super(map);  
    }  
    public void draw() {  
        super.draw(); ← always calls the method  
        mapUI.printDistance(coordinates); ← of the wrapped object  
        mapUI.printTravelTime(coordinates); ← before adding behaviour  
    }  
}
```

concrete decorator

notice no additional behaviour while adding a new location

multiple additions can be made

the concrete decorator implementations extend the functionality of the wrapped object

**Figure 7.8: Traveler Map Decorator**

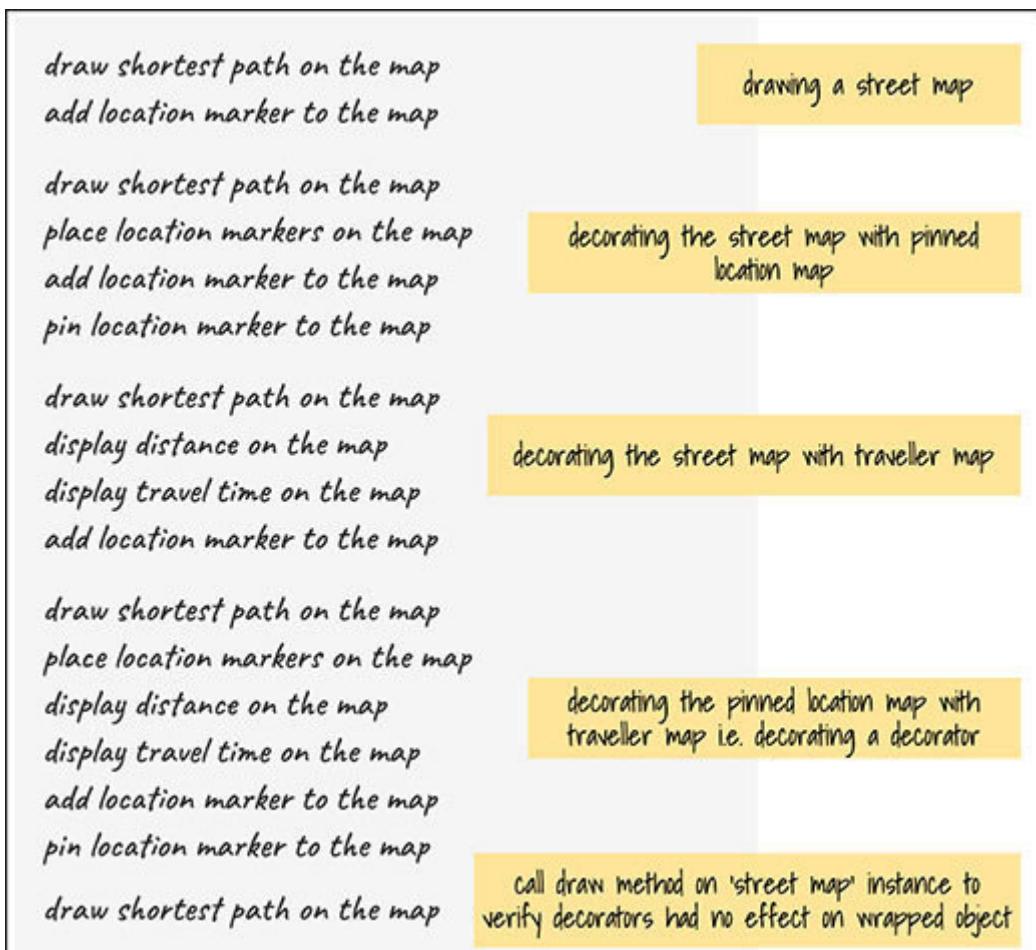
## Testing the decorators

We have written a test class to verify that the pattern we worked on works. Let us execute the same as shown in the [Figure](#)

```
public class TestMap {  
    public static void main(String[] args) {  
        List<MapCoordinates> coordinates = new ArrayList<MapCoordinates>();  
        coordinates.add(new MapCoordinates(234, 123));  
        coordinates.add(new MapCoordinates(215, 134));  
        MapCoordinates coord3 = new MapCoordinates(134, 324); ← location  
        coordinates  
        stMap = new StreetMap(coordinates); ← additional location  
        stMap.draw(); coordinate  
        stMap.addLocation(coord3);  
  
        Map plMap = new PinnedLocationMap(stMap, coordinates);  
        plMap.draw(); ← concrete Map implementation  
        plMap.addLocation(coord3); ← decorate the concrete Map implementation  
  
        Map trvlrMap1 = new TravelerMap(stMap, coordinates);  
        Map trvlrMap2 = new TravelerMap(plMap, coordinates);  
        trvlrMap1.draw(); ←  
        trvlrMap1.addLocation(coord3); ← decorate the decorator itself  
        trvlrMap2.draw(); ←  
        trvlrMap2.addLocation(coord3); ← calling methods on the decorator instance  
        stMap.draw(); ← checking if there's any change in the  
    }                                behaviour of street map object  
}
```

**Figure 7.9:** Test decorators

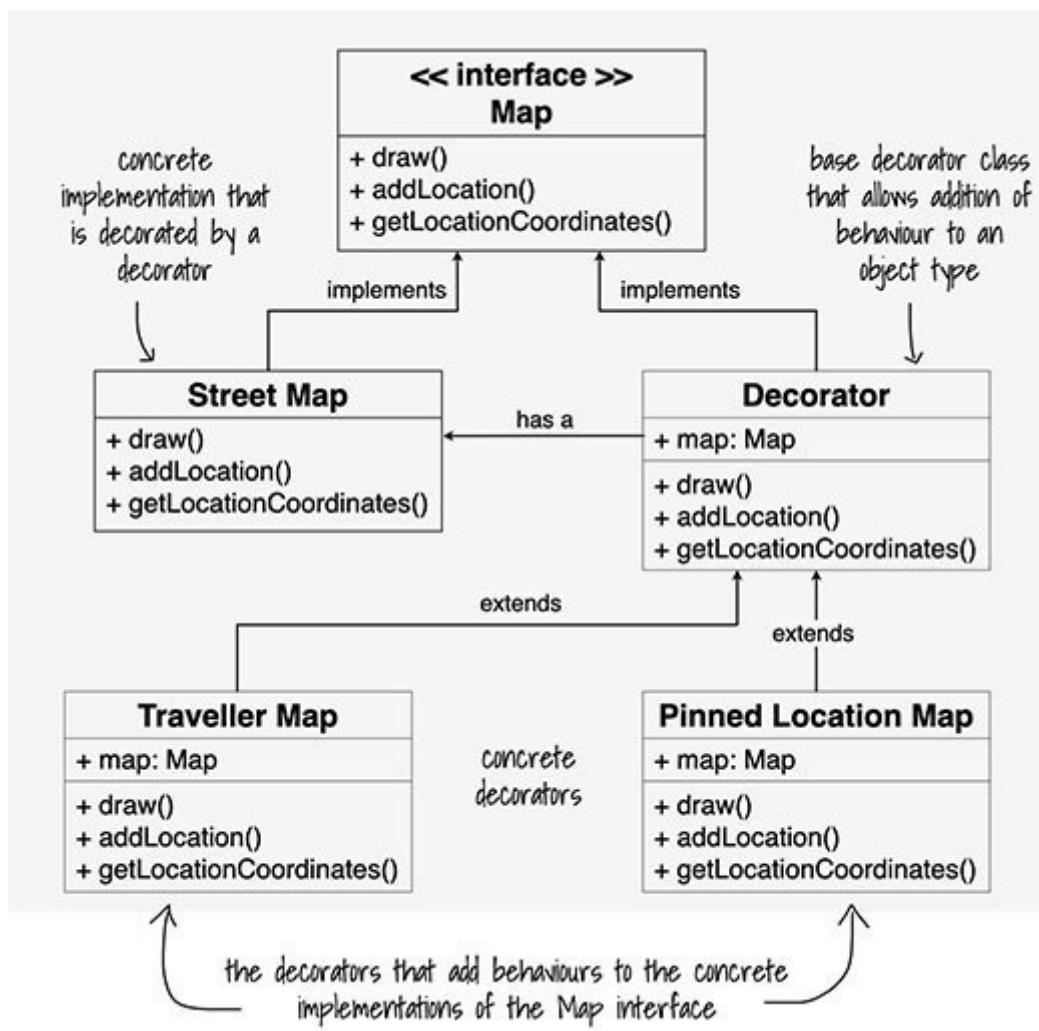
In [Figure](#) we start with setting up the map coordinates and instantiate an instance of our subject, **Street**. Then we instantiate two decorator objects, one of type **PinnedLocationMap** and another of type **Once**. Once these instantiations are complete, we execute the methods on each of these instances to verify the change in behavior when the same methods are called on the subject and then on the decorators. We have also decorated a decorator as well to establish a chain of decorators and to confirm that it works. In [Figure](#) you can see the results of calling the same methods on the subject as well as the decorator instances:



**Figure 7.10:** Test decorator result

First, we executed methods on the subject instance, then by wrapping that instance in two different decorators and then we decorated a decorator itself. In the end, the draw method is executed again on the same subject instance to verify that there is no change in its behavior and that the decorators had no effect on it.

## Decorator design pattern - Defined



**Figure 7.11:** Decorator design pattern defined

[Figure 7.11](#) exhibits the architecture of the decorator design pattern. The decorator design pattern, as the name implies, decorates another object to provide additional behavior on top of what was provided by that object. A decorator has no impact on the structure of the subject. The decorators implement the same

interface as the subject and wrap an object of the interface type to provide additional behavior.

The decorator design pattern is useful when the classes are not open for extension because they are final in nature or when multiple distinct behaviors are to be added.

## Benefits and drawbacks

It is time to make note of some of the benefits and drawbacks of the decorator design pattern. Let us start with the benefits first before we look into the drawbacks of it:

It provides a way to add behavior to an object without altering the structure or existing behavior of that object

It can be used to stack together multiple behaviors on top of each other by chaining decorators together

It is better than using static inheritance

It provides a flexible way to add and revoke behavior dynamically

The drawbacks of the decorator design pattern are:

Constructing too many proxies can be hard to maintain and must be carefully ordered when chained together

Since there is no provision to amend the interface, the parameters passed to the methods are rigid as well

## Usage

Some of the scenarios relevant for constructing decorators are

Providing unique behaviors on top of a common, generic behavior, for example, converting the output generated by the subject into different formats

For chaining behaviors together while adding some and skipping others based on conditions

## Conclusion

Decorator design pattern is a widely used structural design pattern that can be utilized to decorate existing objects without altering their structure. The design pattern helps to add additional behaviors and can be utilized to form decorator chains to append distinct behaviors on top of each other while ensuring no change to the subject.

In the next chapter, we will discuss another structural design pattern known as the **proxy design pattern** and will learn how to create proxy objects that guard the real object.

## Questions

In this section, we will sharpen our brain by indulging in some fun brain activities. Let's get started...

## Pick the odd one out

decorator	code reuse	subject
chain	base interface	decorator
additional behaviour	structure alteration	not open for extension
marker interface	behaviour addition	no available interface

*Figure 7.12: Pick the Odd one out*

### Find the odd one out

Decorator design pattern

Decorates existing objects

Requires modifications in the existing code

Introduces a decorator class

Promotes code reuse

Decorator design pattern can be used when

Additional behavior is desired

There is no base interface

Subject class is not open for change

Class is final in nature

The adapter design pattern has these important actors

A decorator

A subject

A base interface

A polymorphic reference

Select two correct answers

Decorator design pattern

Decorates existing objects

Requires modifications in the existing code

Requires a base interface to implement

Can only be used with non-final classes

Decorator design pattern is most useful when

Class is not open for extension

Subject's source code is not available

Sub-classing is the only solution

Performance is the key

What goes together

Subject, decorator, final

Decorator chain, multiple behaviors, decorator

Decorator, sub-classing, inheritance

Wrapper, subject, no interface

## Answers

Pick the odd one out

Image [Column Wise]

marker interface, structure alteration, not open for extension

## Find the odd one out

Requires modifications in existing code

Subject class is not open for change

A polymorphic reference

Select two correct answers

Decorates existing objects, require a base interface to implement

Class is not open for extension, subject's source code is not available

[subject, decorator, final] and [decorator chain, multiple behaviors, decorator]

## CHAPTER 8

### The Guardian

---

You can find the editable code files for this chapter on the book's GitHub link.

<https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers/tree/main/Chapter%2008>

---

## Structure

Topics that make this journey worthwhile:

Introduction

Proxies

Proxy design pattern

Types of proxies

Building a proxy

Proxy design pattern – defined

Benefits and drawbacks

Usage

## Objective

Our objective is to understand object proxies and how they can be utilized to control access to the objects they substitute. We will learn about the proxy design pattern and the different types of proxy objects that can be constructed to act as placeholders for the original objects.

We will also walk you through the implementation details of the facade design pattern, the benefits it provides, and a few drawbacks along with some of the places where this design pattern can be used.

## Introduction

The process of software development is a continuous one. A software provides many different functionalities that are often developed in iterations and are independent of each other, but not always. At times, we also need to add to an existing functionality. The approach to achieve the same seems straightforward by making the necessary changes directly to the existing class. This is only possible when the class is available and open for change. Also, we must consider the single responsibility principle and must not overburden the class with multiple functionalities.

If the class is not available for changes, for example, it is a part of some library we do not have direct access to, one option that opens up is to construct another class that provides the additional functionality and make changes in the sequence of calls that leads the request to the original object. The approach could be a bit difficult to manage when the structure of the new class is different from the original one, it could lead to integration issues with the classes that interact with the original class.

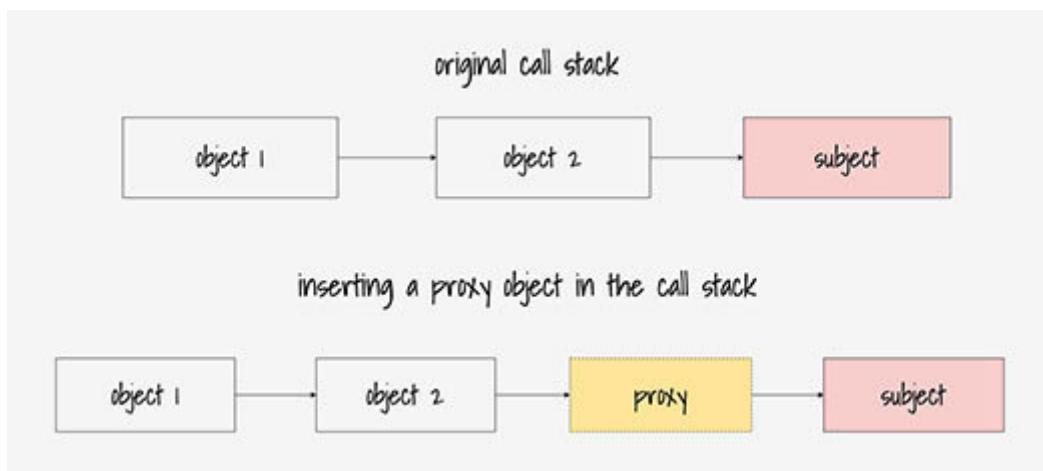
What if there is a class that resembles the same structure as that of the original one and that could be easily integrated with the rest of the code.

In this chapter, we will learn about the proxy design pattern and how it can be utilized to create surrogate objects that guard the

original object and may perform some actions before providing access to it.

## Proxies

A proxy acts much like a placeholder, a substitute in place of some other object. It resembles the original object and can be easily integrated with the existing codebase. It implements the same interface as the original object and wraps it to serve as its placeholder. It is often termed as a wrapper, a surrogate or a substitute:



**Figure 8.1:** Inserting a proxy in call stack

Inserting a proxy into the call stack enables it to filter all the requests that reach the original object and perform additional operations. The original object instance can be easily replaced with a proxy instance without effecting the existing integration or the call stack. The operations could be related to caching, logging, access check, marshalling, or something else depending upon the type of proxy.

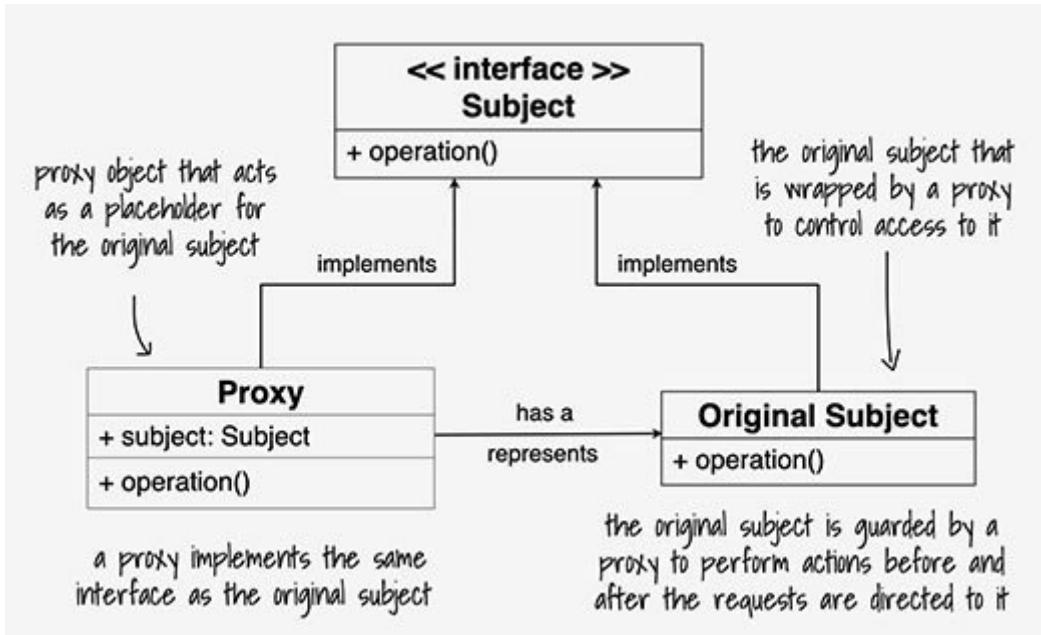
## Proxy design pattern

The proxy design pattern is a structural design pattern that allows for an object to serve as a placeholder to some other object. In general terms, as the name suggests, it acts as a proxy to some interface that the client wants to access. The proxy object encapsulates the complexity of the object it wraps. It could also be described as a filter that intercepts all the incoming requests for the subject, performs some action before directing the request to the original object, and after the request is served by it.

Multiple proxies can be chained together to operate in a sequence, with each proxy performing a specific task. The order of execution depends on how these proxy objects wrap each other. The subject is wrapped by the last proxy in the sequence.

A prominent real-life example of a proxy is a manager and his assistant. Every request to meet the manager must go through his assistant. The assistant takes care of the manager's calendar and who he would be interested to meet.

In [Figure](#) the **Proxy** implements the **Subject** interface similar to an original subject. The **proxy** class has a reference of the interface type that can be passed either the subject or another proxy object to form a chain:



**Figure 8.2:** Proxy design pattern

Here's how it works:

Instantiate an object of the original type, a subject

Construct a proxy object and pass it to the subject

Direct all requests to the proxy object instead of the original one

When the proxy object receives a request, it performs some additional logic along with delegating the request to the original object and finally, sending a response to the caller

The proxy object does not impact the behavior of the original object, though it could alter the chain of calls in some cases. By creating multiple proxies, they can be chained together to perform various tasks working together.

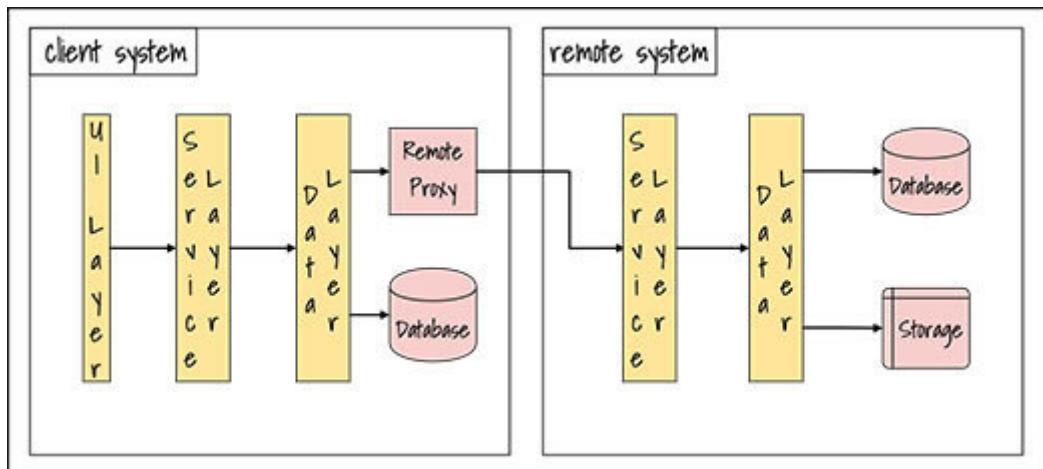
The proxy design pattern can be utilized in many ways. It can be used to control access to an object, perform lazy initialization, cache data, audit the incoming requests, and much more.

## Types of proxies

The proxy design pattern is usually described as having specific types, that is, remote proxy, control proxy, and virtual proxy.

## Remote proxy

A proxy that represents a remote object. It encapsulates the connectivity to the remote object, the serialization and deserialization required, the necessary access checks or any other task that is required to connect to that remote object:

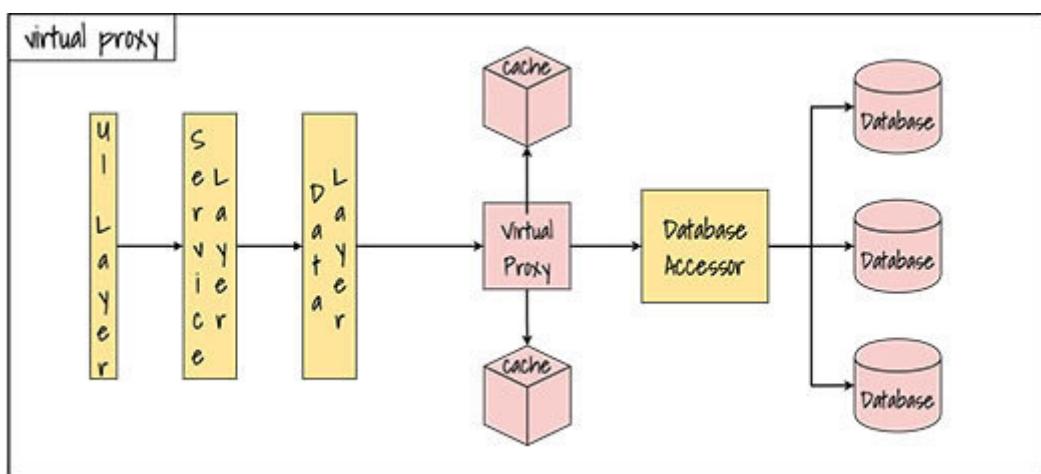


**Figure 8.3: Remote proxy**

The remote proxy object is a placeholder for the remote object and is available locally. It can be programmed to connect to an active instance of the remote object in a distributed architecture to ensure high availability.

## Virtual proxy

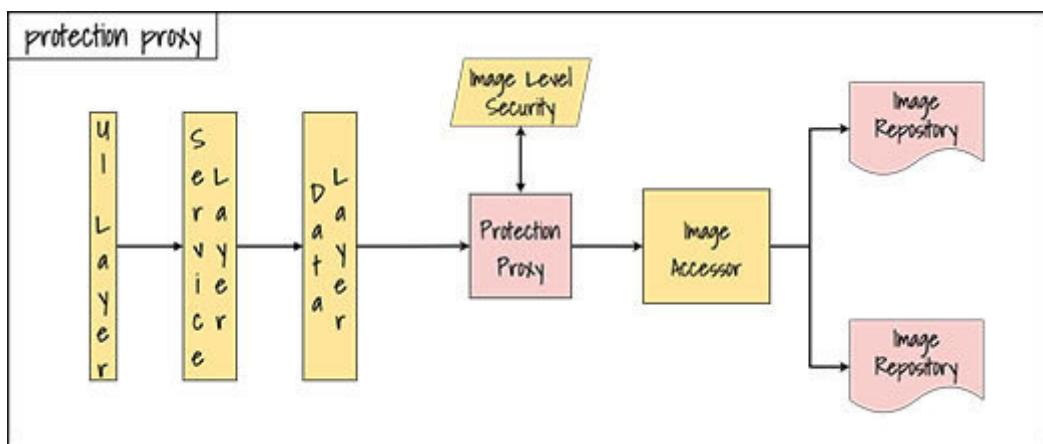
A proxy that acts as a surrogate to the original object. It is used to represent objects that interact with data or resources and consume time during their interaction with them. The virtual proxy object can be used to cache information and send it to the client while the call to the original object returns, the updates can then be pushed to the client:



**Figure 8.4:** Virtual proxy

## Protection proxy

A proxy object that guards access to another object could be used to perform access checks and improve the security for critical resources:



**Figure 8.5: Protection proxy**

The protection proxy could connect to another service to confirm that the client has access to the requested object or resource it guards. It then either provides the access or restricts it based on the response received from the service.

Let us walk through an example to add clarity to our understanding of the proxy design pattern.

## [Building a proxy](#)

The example that we will walk you through in this chapter makes use of an interface that provide methods to access weather-related information, a concrete implementation of it, and a proxy that can replace it and provide additional functionality on top of it. We will learn how the proxy design pattern can be utilized to construct proxy objects that can represent existing object instances. Let us first take a look at the interface that sets up the common contract that both the concrete implementation and the proxy must follow for the design to work.

## The interface

The interface is the central point of the proxy design pattern. In [Figure](#) the **Weather** interface provides the behavior that must be implemented by the concrete implementations as well as the proxies:

```
the subject  
interface  
public interface Weather {  
    public Map<Integer, Integer> fetchTemperature(String city);  
}
```

**Figure 8.6:** Interface Weather

This interface provides us with a single method that can be used to fetch temperature values for the current day, for a given city.

## The subject

In [Figure Weather Service](#) is a concrete implementation of the **Weather** interface. It provides the logic to fetch the weather information whenever a request is received. When an instance of this service is instantiated, it establishes a connection with the weather station and reuses it to provide the desired functionality:

```
public class WeatherService implements Weather {  
    private WeatherStation ws; → the reference to weather  
    private String md5; → information system, from where  
    { → we fetch the data  
        ws = new WeatherStation();  
        md5 = ws.connect("droid", "H6sDg6r8TfRdfv76"); → establish connection  
    } → with the weather  
    system  
  
    public Map<Integer, Integer> fetchTemperature(String city) {  
        Map<Integer, Integer> temperatures = new HashMap<>(24);  
        double token = ws.fetchToken(md5); → fetching the per request  
        temperatures = ws.fetchTemperature(token, city); → unique token, used for  
        return temperatures; → authentication  
    } → fetching the day wise  
    } → temperature information  
    for a particular city  
  
    the original subject that provides the weather information
```

**Figure 8.7: Weather Service | Original Object**

Our implementation in [Figure 8.7](#) sends out a request to the weather station every single time. With increase in the number of requests, there remains a possibility that the performance could be

degraded as there is no limit or bound on the number of requests that could be processed in a specific timeframe.

## The proxy

The proxy class implements the same interface as the subject or the original class and also wraps an object of the same interface type, a subject. The proxy class imitates the structure of the subject class and therefore could easily replace it. Once the proxy is inserted into the call stack, it is hardly noticeable and does not impact existing integrations:

```
public class WeatherServiceProxy implements Weather {
    private Map<String, WeatherInformation> temperatures = new HashMap<>();
    private Weather weatherService;
    private static double timeWindow = 600000; →
    private WeatherInfo weatherInfo;

    public WeatherServiceProxy(Weather service) {
        this.weatherService = service;
    }

    public Map<Integer, Integer> fetchTemperature(String city) {
        double currentTime = calendar.getTimeInMillis();
        WeatherInformation weinfo = temperatures.get(city);
        Map<Integer, Integer> temperature;
        if(null == weinfo) {
            print("first call / sending request to weather system");
            WeatherInformation info = new WeatherInformation();
            info.setCity(city);
            temperature = weatherService.fetchTemperature(city);
            info.setTemperature(temperature);
            info.setLastFetchedOn(currentTime);
            temperatures.put(city, info);
        } else if(currentTime > weinfo.getLastFetchedOn() + timeWindow) {
            print("time window breached / sending request to weather system");
            temperature = weatherService.fetchTemperature(city);
            weinfo.setTemperature(temperature);
            weinfo.setLastFetchedOn(currentTime);
        } else {
            print("time window intact / utilizing cached information");
            temperature = weinfo.getTemperature();
        }
        return temperature;
    }
}
```

the proxy class maintains a window of 10 minutes between calls to the weather system

the time window is there to ensure that the system is not choked with multiple requests

the proxy caches the information received in the previous call to the original object

the proxy class that acts as a placeholder for the original subject

the overridden method that fetches the weather information

the time window that defines the duration between the two calls

the proxy class that acts as a placeholder for the original subject

### **Figure 8.8: Weather Service Proxy**

The proxy class receives all the requests, which were earlier received by the subject, works on it, forwards the request to the original subject, receives the response, works on it again if required, and finally sends the response to the client.

In [Figure](#) the proxy class implements the **Weather** interface and has a reference to the original subject type as well. The proxy class is structured in a manner such that it can act as a proxy to whichever object is passed to it, be it a concrete implementation of that type or another proxy that implements the same interface. The proxy classes can also be chained together to form a chain of proxies that perform specific tasks before and after the request is received by the original subject.

Our implementation of the proxy class adds a layer of caching on top of the subject implementation. It maintains a local cache that is updated from time to time, as per the defined time window. Whenever a request lands on the proxy object, it first checks whether it could be served from the local cache. If the local cache has the required information and the time window has not lapsed, the result is served from the cache. Only when either the cache does not have the information or the time window for that information has lapsed, it makes a request to the original subject, fetches the response, and updates the local cache.

The proxy object with its implementation of a cache limits the number of requests that end up at the original subject and increases the throughput. It also reduces latency and the chances of performance degradation due to multiple requests choking the weather service.

## Testing the proxy

We have written a test class to verify that the pattern we worked on works. Let's execute the same.

In [Figure](#) we start with instantiating an original subject and a proxy object, passing in the original subject. Then we make three calls to the proxy instead of the original object. The first two calls pass a different value for the method argument, but the third call is passed an already used value. Response received from the proxy instance verifies that the third call was served using the cache and that no request was sent to the original object:

```

public class TestProxy {
    public static void main(String[] args) {
        WeatherService weatherService = new WeatherService();           ↓ the original object
        Weather wsp = new WeatherServiceProxy(weatherService);          ← utilizing the proxy object

        Map<Integer, Integer> tmpr = wsp.fetchTemperature("delhi");   ←
        print("Temperature of Delhi", tmpr.toString());                  ↓ fetch data

        Map<Integer, Integer> tmpr2 = wsp.fetchTemperature("mumbai");  ←
        print("Temperature of Mumbai", tmpr2.toString());                ↓

        Map<Integer, Integer> tmpr3 = wsp.fetchTemperature("delhi");   ←
        print("Temperature of Delhi", tmpr3.toString());                  ↓
    }
}

```

first call / sending request to weather system

... Temperature of Delhi ...

{0=0, 1=7, 2=10, 3=7, 4=3, 5=5, 6=4, 7=19, 8=5, 9=17, 10=19 ...}

first call / sending request to weather system

... Temperature of Mumbai ...

{0=18, 1=12, 2=12, 3=17, 4=6, 5=11, 6=16, 7=8, 8=15, 9=5, 10=4 ...}

time window intact / utilizing cached information

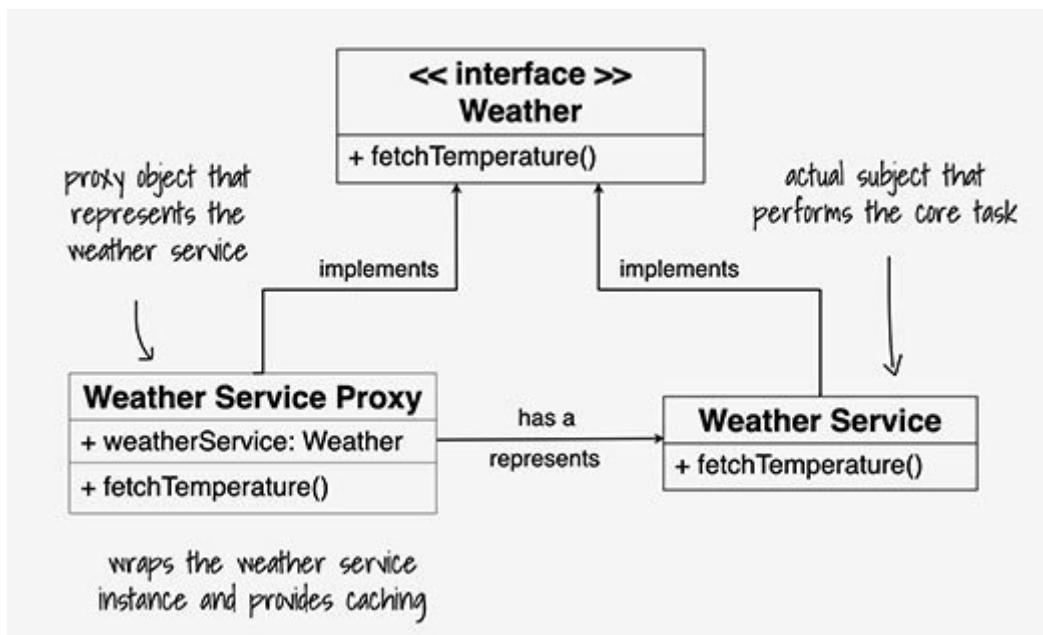
... Temperature of Delhi ...

{0=0, 1=7, 2=10, 3=7, 4=3, 5=5, 6=4, 7=19, 8=5, 9=17, 10=19 ...}

**Figure 8.9:** Test Proxy

## Proxy design pattern - defined

The proxy design pattern, as the name implies, provides a way to create proxies that can be used to represent other objects. A proxy has the same structure as the original subject, implements the same interface, and wraps a subject instance:



**Figure 8.10:** Proxy design pattern defined

## Benefits and drawbacks

It is time to make note of some of the benefits and drawbacks of the proxy design pattern. Let us start with the benefits first before we look into the drawbacks of it:

Access to an object can be controlled without changing the way the object is being accessed

Multiple proxies can be chained together in a sequence, where each proxy is responsible for a specific task

A proxy can imitate a real object when it is not ready to take requests

Proxies can be attached to the original object in runtime

Promotes separation of concern

The drawbacks of the proxy design pattern are:

If not designed properly, the proxies can end up calling themselves in a loop

Proxies can lead to bottlenecks when the incoming requests are overwhelmingly large in number



## Usage

Some of the scenarios relevant for constructing a proxy are:

Guarding or controlling access to an object

Caching or logging information every time an object is accessed

Managing access to a remote object

Massaging input before it reaches the original object

Performing cleanup operations

## Conclusion

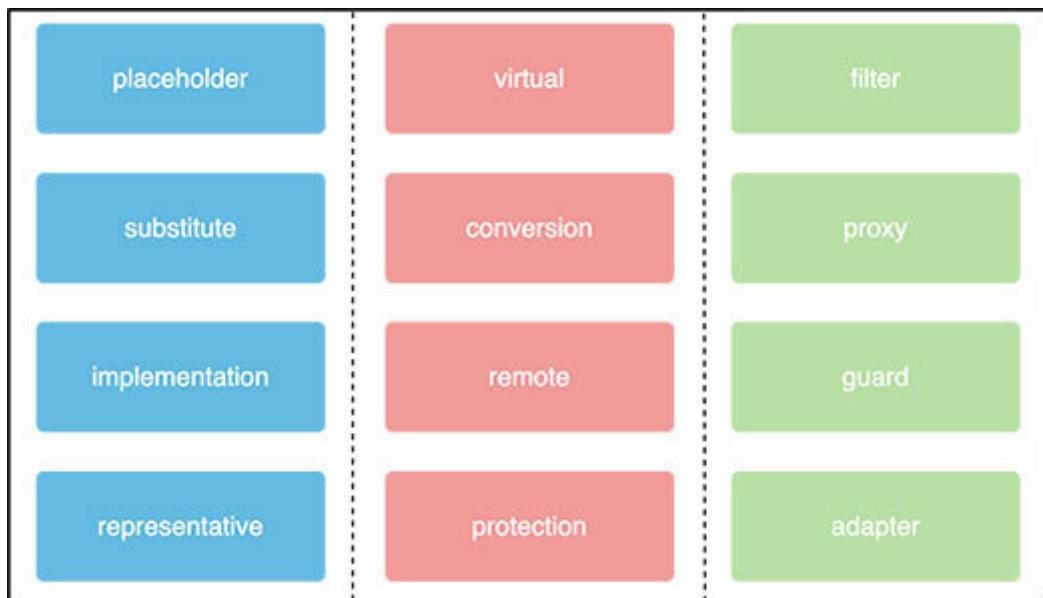
Proxy design pattern is a widely used structural design pattern that can be utilized to create proxies that represent existing objects without altering their structure. There are broadly three types of proxy objects namely remote proxy, virtual proxy and protection proxy. The proxies can be chained together to represent either the original subject or another proxy.

In the next chapter, we will discuss another structural design pattern known as the ‘facade design pattern’ and will learn how to simplify complex systems by introducing easy to use interfaces that hides the complex internals of a system.

## Questions

In this section, we will sharpen our brain by indulging in some fun brain activities. Let's get started:

## Pick the odd one out



*Figure 8.11: Pick the Odd one out*

### Find the odd one out

Proxy design pattern

Creates proxies to represent existing objects

Requires modifications in the existing code

Requires changes in existing integration

Is a structural design pattern

Proxy design pattern is broadly categorized as:

Virtual proxy

Remote proxy

Converter proxy

Protection proxy

The proxy design pattern can be used to:

Perform lazy initialization

Replace the original object

Cache information

Safeguard access to original object

**Select two correct answers**

Proxy design pattern:

Replaces existing objects

Requires modifications in the existing code

Requires a base interface to implement

Can safeguard access to objects

Proxy design pattern is most useful for representing:

Objects with high latency

Model objects

Other proxies

Objects that need to be secured

What goes together?

Security, access control, protection proxy

Chain of proxies, high coupling, remote proxy

Sub-classing, inheritance, proxy design pattern

Caching, database, direct access

## Answers

Pick the odd one out

Image [Column Wise]

implementation, conversion, adapter

## Find the odd one out

Requires modifications in existing code

Converter proxy

Replace the original object

Select two correct answers

Requires a base interface to implement, can safeguard access to objects

Objects with high latency, objects that need to be secured

[security, access control, protection proxy] and [chain of proxies, different behaviors, sequence]

## CHAPTER 9

### Simplifying the Complexity

---

You can find the editable code files for this chapter on the book's GitHub link.

<https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers/tree/main/Chapter%2009>

---

## Structure

Topics that make this journey worthwhile:

Introduction

What is a Facade?

Facade design pattern

Building a facade

Facade design pattern: defined

Benefits and Drawbacks

Usage

## Objective

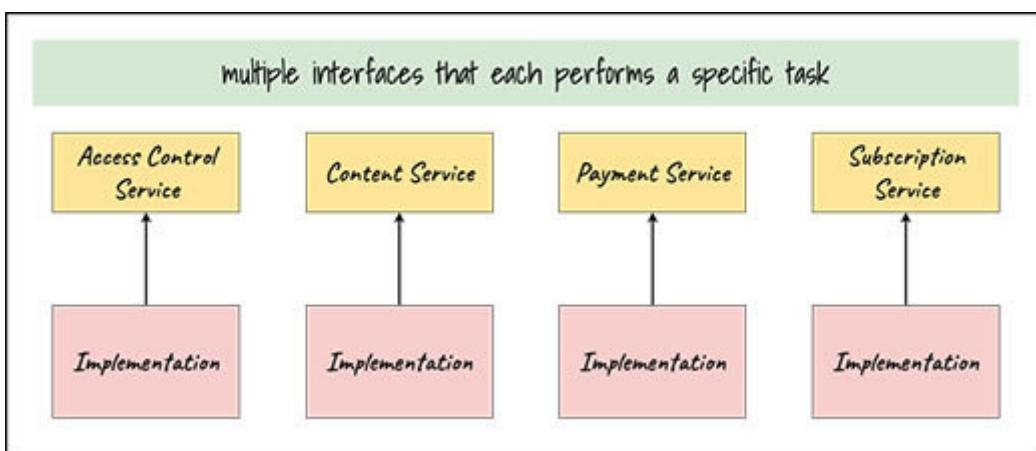
Our objective is to understand the complexity of integrating two systems together, one is the client, and other is the service provider. We will learn about the facade design pattern and how it can be utilized to provide a simple and unified interface that helps to reduce the complexity experienced by the clients during integration.

We will also walk you through the implementation details of the facade design pattern, the benefits it provides, and a few drawbacks along with some of the places where this design pattern can be used.

## Introduction

Developing interfaces for services is one of the ideal ways to design a software application. An interface describes a contract between a service and its users and provide methods to facilitate interaction. An application usually consists of multiple interfaces that each is responsible for providing methods to integrate with various services or sub-systems.

Integration involving multiple interfaces could easily become complex if the interfaces are not well defined or if the desired behavior requires interacting with many of them. To interact with a system that involves multiple sub-systems, order of execution is also important, which could add to the complexity of integrating with the overall system:



**Figure 9.1: Individual Interfaces**

[Figure 9.1](#) depicts distinct interfaces with their implementations. It symbolizes the effort required to integrate with each of them. To make a complex system easier to use and integrate with, a unified interface could be provided that minimizes the dependency of the clients on the multiple interfaces exposed by the system and reduces the complexity by exposing methods that present an abstract view of the system to the outside world.

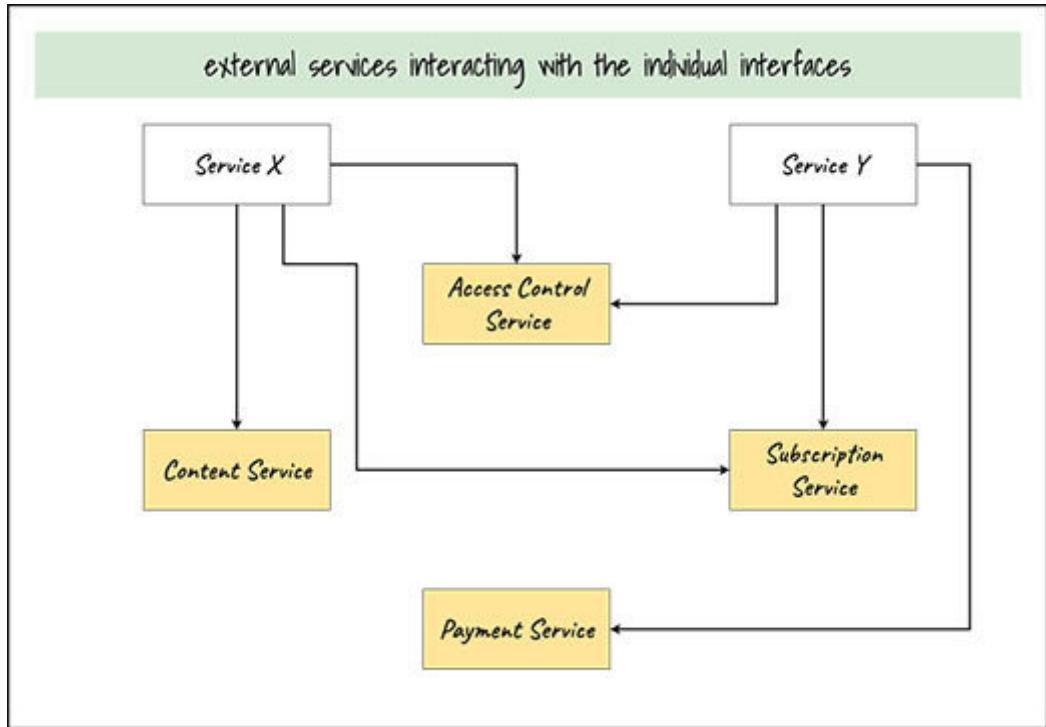
In this chapter, we will learn about the facade design pattern and how it can be utilized to ease the integration between a system and its users.

## Facade

A facade is an abstraction that hides the underlying complexity of a system and presents a much simpler view of it to the outside world. In literal terms, a facade is the front wall of a building that the on-goers see from the outside, unaware of the complex structure housed within the outer walls.

In computing, a facade is an interface that provides a simple and unified view of a system, hides the complexity of the various sub-systems that interact and co-operate to present a holistic view of the overall system. In other words, a facade groups together multiple functionalities that are usually performed in some order and presents them in the form of easy-to-use methods. Those methods can be called by the clients to integrate with the system without any need to understand the underlying complexity.

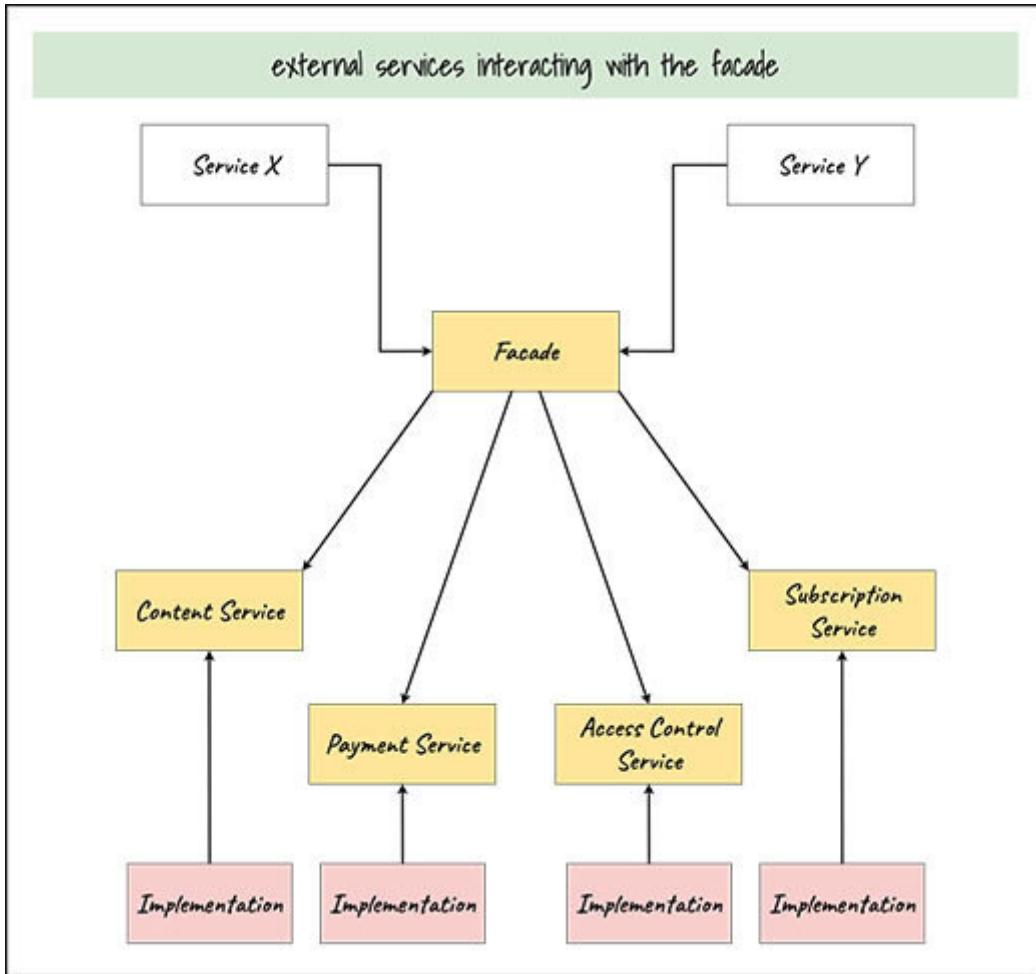
In Figure Service X and Service Y are shown as interacting with the various interfaces available in a system. A facade can help here by providing a simplified interaction point:



*Figure 9.2: Service and Interface Interaction*

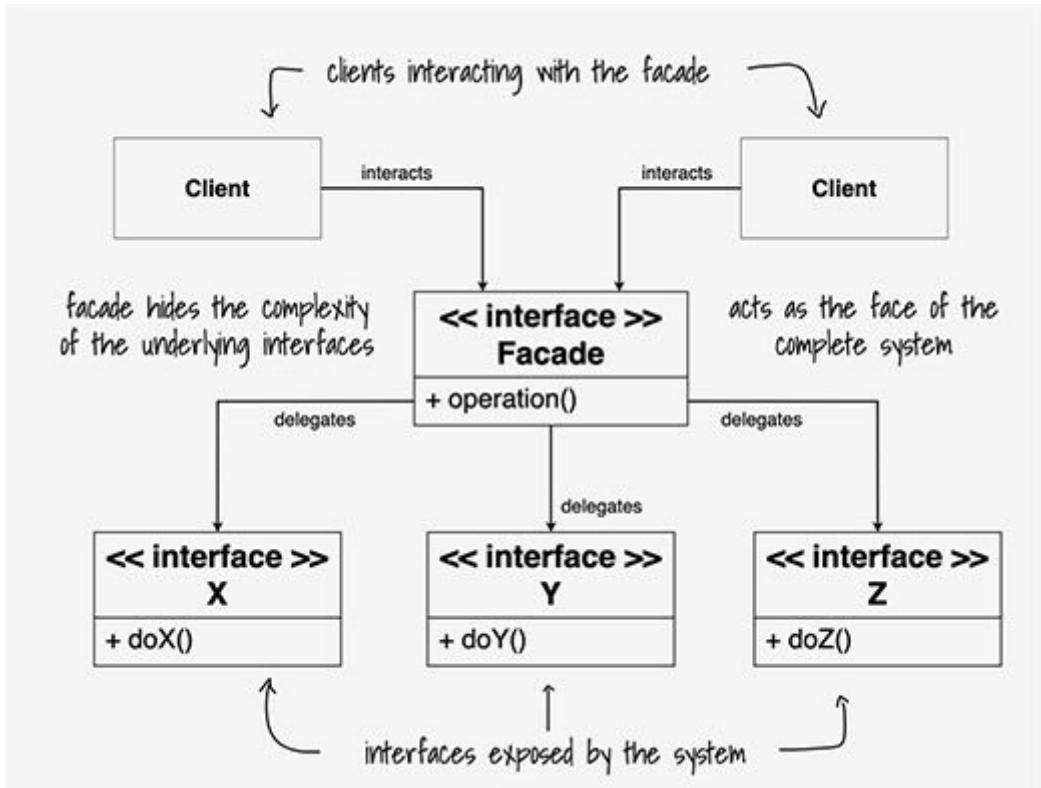
## The Facade design pattern

The facade design pattern is a structural design pattern that allows for the development of a facade. It provides a simple and extensible way to interact with a system that is formed of various sub-systems. Integrating with such a large system would otherwise require multiple integrations, involving numerous interfaces. It simplifies the overall integration process by providing a single interface that internally delegates the calls to the sub-systems and relieves its clients from the complexity of integrating with most of them: [Figure 9.3](#) exhibits the usefulness of a facade design pattern for external services, while interaction with a system that exposes multiple interfaces:



**Figure 9.3: Introducing Facade**

As shown in the [Figure](#) a facade design pattern allows for creation of an interface of interfaces that provides simplified methods to the clients and interacts with the existing sub-systems itself, all while effectively hiding the complexity exhibited by those sub-systems. This reduces the burden on the external systems to integrate with multiple interfaces. It successfully hides internal complexity behind a unified interface that appears simple to the clients:



**Figure 9.4: The Facade Design Pattern**

Here's how it works:

Instantiate an object of the facade type that provides the simplified method definitions.

Facade instance acts as a multiplexer and delegates the calls to multiple interfaces, gathers their output, and processes it if required before finally sending a response to the client.

In the next section, we will demonstrate the working and usage of the facade design pattern.

## [Building a facade](#)

The example that we will walk you through in this chapter makes use of multiple interfaces that provide various methods to access the online services of an over-the-top platform. We will learn about implementing a facade and how it benefits the clients by providing simple and ready-to-use methods. Let's first take a look at the interfaces that provide the various functionalities required to access the over-the-top platform and later we will discuss about the facade implementation.

## All for one

A system usually hosts multiple interfaces that are accessed by its clients to avail the many services provided by it. Many of those interfaces handle only a part of the task that the client intends to perform, requiring the client to make multiple calls to the system. The client must take care of multiple integrations and handle any change that is done to those interfaces:

```
public interface AccessControlService {  
    public String getAccessToken(User user);  
    public AuthResponse authenticate(UserCredentials credentials);  
    public AuthResponse authenticate(User user);  
    public AuthResponse authorize(User user, String accessToken);  
}
```

This interface provides methods for user authentication

*Figure 9.5: An Interface for Access Control*

The system we have chosen to present as a use case for the facade design pattern also exposes multiple interfaces that the client can integrate with to achieve the desired results. Assuming the client only has to integrate with the existing implementations, it still has to ensure the order in which the calls are made to these interfaces and handle the exceptional or error scenarios.

In *Figure* the interface provides multiple methods to control user access. It exposes methods to authenticate and authorize the user

and fetch a unique token that is used to access the content from the platform. In [Figure](#) the interface lists the methods available to access the digital content available on the platform:

```
public interface ContentService {  
    public Content fetchContent(String accessToken, String contentId,  
        boolean advertize);  
    public List<Content> fetchContent(String segment, boolean premium);  
    public List<Content> fetchTrailers();  
    public List<Content> fetchTrailers(String segment);  
    public List<Content> fetchSponsoredContent();  
    public List<Content> fetchSponsoredContent(String segment);  
    public boolean isPremiumContent(String contentId);  
}
```

verifies the user before loading the content and checks if the user is subscribed or not

trailers for movies, motion series, animes etc.

promoted content listed by the content publishers

This interface provides methods to access the content hosted by the platform

**Figure 9.6: An Interface to access content**

It provides multiple methods to fetch the actual content, trailers available for the movies, motion series, anime, basically trailers for everything available on the platform. Not just that, it also has methods to provide promoted or sponsored content.

We will make use of four interfaces for our example, and we have already discussed two of them that are most relevant to the functionality our example demonstrates. The other two interfaces are there to support content monetization, one is the payment interface and the other one is responsible for subscribing and unsubscribing users to the platform.

In [Figure](#) the interface provides methods that can be utilized by the client to initiate and close payment orders related to user subscription. The first method prepares a payment order that is added to the user's cart during checkout and the second method helps to establish connection with payment providers, complete the payment, and subscribe the user with the platform:

```
public interface PaymentService {  
    public PaymentOrder prepareOrder(PaymentMode payMode,  
        double amount, User user);  
    public PaymentResponse pay(PaymentOrder details);  
}
```

This interface provides methods to support payment services

**Figure 9.7:** An interface to support Payment Service

In [Figure](#) the interface provides methods that help with subscription and un-subscription of a user as per the users' consent and the confirmation of payment in case of subscription:

```
public interface SubscriptionService {  
    public double getSubscriptionCharge();  
    public void subscribe(User user);  
    public void unsubscribe(User user);  
}
```

This interface provides methods to help the client with subscribing a user to the platform

subscribe/unsubscribe to the content provided by this platform

**Figure 9.8:** An Interface to support Subscribe and Unsubscribe users

All these interfaces are good enough to be directly utilized by the client, but they are too many to integrate with. Wouldn't it be great if the client has as less interfaces to integrate with if possible, as it hugely decreases the cost of integrating with those interfaces as well as the testing involved to confirm the correct functioning. Not just that, the developers also have to ensure to call these methods in the correct order, something that can be simplified if the ordering of these calls can be cumulated, and an exoskeleton is presented to the client instead of individual interfaces. All this can be achieved by utilizing the facade design pattern and expose methods that relate more to the business flow rather than the technical side of it. Let us now dive into creating another interface that acts as a facade and makes our lives easier.

## One for all

A facade is like an interface of interfaces. The facade design pattern provides a unified interface that is easier to understand and integrate with. It hides the complexity exhibited by the many interfaces that the facade implementation interacts with, all while providing a simpler interface to the clients. In [Figure](#) the interface provides methods that are more in line with the business use cases and make more sense to the clients:

```
public interface OverTheTopService {  
    public Content getContent(String contentId);  
    public List<Content> visitSegment(String segmentId);  
    void subscribe(UserCredentials credentials, PaymentMode payMode);  
}
```

the interface that acts as a facade and provides business specific methods

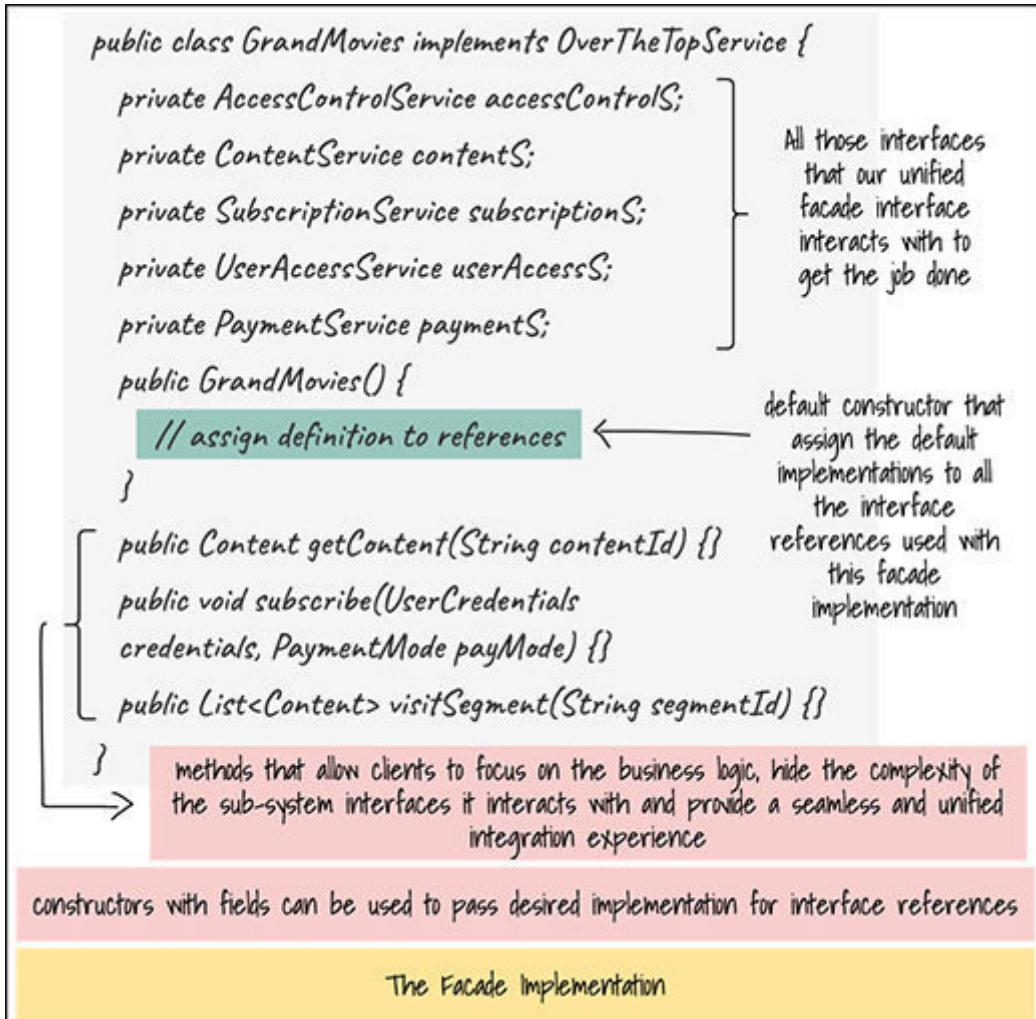
**Figure 9.9: The Facade Interface**

The clients can integrate with one unified interface and achieve the desired output instead of integrating with all the other interfaces provided by the system. This relieves the client from multiple complex integrations and drives the focus towards business logic and its implementation.

Let's now look into one of the many possible implementations of the facade interface. Integrating with interfaces and not the

implementations allow for dynamic behavior, based on the selected implementation, and opens up the possibility to work with multiple approaches without changing the contract.

In [Figure](#) the implementation class presents a way to interact with a set of interfaces to provide a simple and unified integration endpoint for the many clients of the system. If you would have noticed, we have used a default constructor that assigns some pre-selected implementations to the references, limiting the scope of using different implementations of the same interfaces with the facade. To overcome this restriction, we can use constructors with fields and pass the instances of the desired implementation that we want to use with the façade:



**Figure 9.10: A Facade Implementation**

Let's now take a look at the method definitions that form the logic our facade implementation provides in order to simplify the integration process. We will go through the methods one by one in the order they were listed in [Figure](#)

These implementations are only for understanding the concepts and might not be relevant for actual business logic. They are also not suitable for production level code, actually far from it. You are advised to exercise caution and to only use it for learning purposes.

In [Figure](#) the method definition can be seen interacting with multiple interfaces. It is responsible for fetching the digital content based on the unique content ID passed to it. The method does a lot more than just fetching the content though. It takes care of user authorization, content validation, and accessing the right content. It also ensures that the calls to all the interconnecting interfaces are made in the correct order:

```

    → this method fetches the digital content based on the unique contentId passed to it
    public Content getContent(String contentId) {
        Content content = null;
        User user = userAccessS.getCurrentUser();
        String accessToken = accessControlS.getAccessToken(user);
        AuthResponse authR = accessControlS.authorize(user, accessToken);
        if(authR.isAuthorized()) {
            boolean premiumContent = contentS.isPremiumContent(contentId);
            if(premiumContent) {
                if(user.isPremiumMember()) {
                    content = contentS.fetchContent(accessToken, contentId, false);
                } else {
                    content = contentS.fetchContent(accessToken, contentId, true);
                }
            } else {
                content = contentS.fetchContent(accessToken, contentId, false);
            }
        }
        return content;
    }

    gets the current user
    check if the user is authorised to access the requested content, might be a suspended or a guest user
    gets an access token for the user, some basic auth stuff
    check if the content falls in premium category
    defines whether advertisements can be shown

```

the method interacts with 'UserAccess', 'AccessControl' and 'ContentService' interfaces and relieves the client from the complexity of integrating with multiple interfaces

**Figure 9.11: Implementing Facade 1**

[Figure 9.12](#) and [Figure 9.13](#) presents us with the definition of the other two methods of our facade implementation. The method described in [Figure 9.12](#) is responsible for subscribing a user to the platform and promote his profile to a premium user. It takes care of user authentication, creating a payment order, and finally completing it.

The second method, the one described in [Figure](#) fetches all the available content whenever a user visits a particular segment. Both these methods interact with multiple interfaces, like the method definition in [Figure](#)

→ this method subscribes a user to the platform

```
public void subscribe(UserCredentials credentials, PaymentMode payMode) {
    AuthResponse authR = accessControlS.authenticate(credentials);
    User user = userAccessS.getCurrentUser();
    if(authR.isAuthenticated()) { ← check if the user is
        checked in and has his
        session active
        PaymentOrder payOrder = paymentS.prepareOrder(payMode,
            subscriptionS.getSubscriptionCharge(), user);
        PaymentResponse paymentR = paymentS.pay(payOrder);
        if(paymentR.isSuccessful()) {
            subscriptionS.subscribe(user); } } } }
```

} ← complete subscription if the payment is successful

→ this method subscribes a user to the platform and updates his profile to premium

**Figure 9.12:** Implementing Facade II

If you look closely, you will find that these methods are doing the exact same thing a client would do to integrate with the multiple

interfaces exposed by the system. They will end up writing something very similar to a facade. The obvious difference would be that the client will have to work through the complexity of integrating with those interfaces.

By using a facade, the whole integration process can be simplified by leaps and bounds for the clients of the system. The clients no longer have to integrate with multiple interfaces exposed by the system. They no longer need to take care of maintaining the integrations when changes are made to those interfaces. Overall, it greatly reduces the complexity experienced by the clients of the system and helps them focus on the core business logic.

In other words, the facade design pattern provides a higher-level interface that makes the sub-system easier to use and interact with:

```

    > this method fetches the list of content when the user visits a segment
    public List<Content> visitSegment(String segmentId) {
        List<Content> contentL = new LinkedList<>();
        User user = userAccessS.getCurrentUser();
        String accessToken = accessControlS.getAccessToken(user);
        AuthResponse authR = accessControlS.authorize(user, accessToken);

        if(authR.isAuthorized()) {
            check if the
            user is
            authorised to
            access the
            requested
            content, might
            be a suspended
            or a guest
            user
            if(user.isPremiumMember()) {
                contentL.addAll(contentS.fetchContent(segmentId, true));
            } else {
                contentL.addAll(contentS.fetchContent(segmentId, false));
                contentL.addAll(contentS.fetchTrailers(segmentId));
            }
            contentL.addAll(contentS.fetchSponsoredContent(segmentId));
        }
        return contentL;
    }

```

gets an access token for the user, some basic auth stuff

fetching different content based on user status

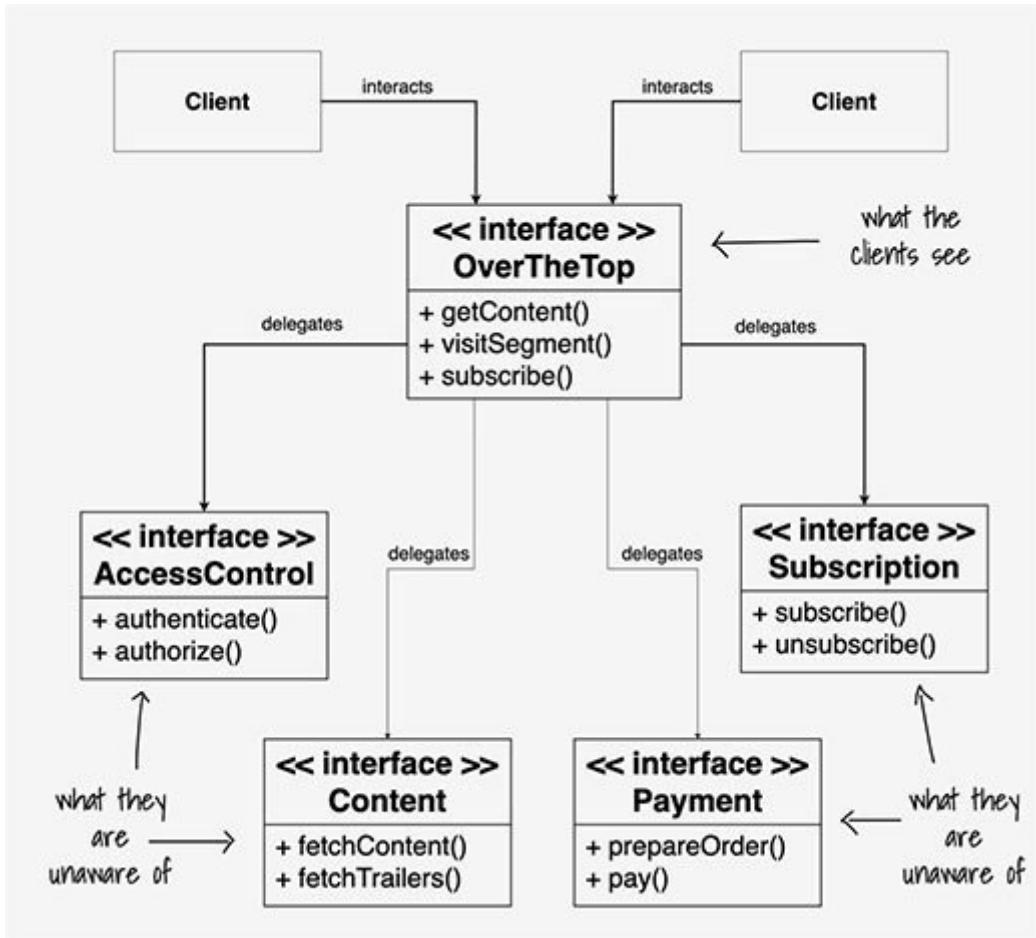
always fetch the sponsored content, after all we are paid for that

**Figure 9.13:** Implementing Facade II

## Facade design pattern - defined

The facade design pattern allows construction of a simplified and unified interface on top of the existing interfaces. It hides the complexity of the sub-systems from the clients and provides a simplified view of the overall system. It ensures that the client is relieved from handling multiple integrations by themselves and exposes a single interface that the clients can rely upon.

The facade design pattern is most useful for large systems that involve multiple interfaces. By limiting the number of integration points with the clients, it also helps to reduce the gaps in how they interact with the system and subsequently decrease the number of integration issues:



*Figure 9.14: Implementing Facade II*

## Benefits and drawbacks

It is time to make note of some of the benefits and drawbacks of the facade design pattern. Let's start with the benefits first before we look into its drawbacks:

It provides a simplified and unified interface to integrate with the system.

It hides the complexity of the system from the outside world.

It helps the clients to focus on the core business logic.

It promotes loose coupling between the clients and the various sub-systems it interacts with.

There aren't any known drawbacks of the facade pattern. Though during its development, it should be made to interact with the interfaces and not directly with any of the implementations. This helps in switching to different implementations if desired rather than sticking to a single implementation.

## Usage

Some of the scenarios relevant for building a facade are as follows:

When there is a need to provide a single, unified interface for a complex, multi-layered system.

When the dependencies between a system and clients are to be reduced.

When there is a need to reduce the impact of future changes in the system, on the clients.

## Conclusion

Facade design pattern is a widely used structural design pattern that can be utilized to create facades that help to improve the integration process by providing methods that are more in line with the client business. It hides the complexity of the system and simplifies the complex integration process.

In the next chapter, we will start our journey towards the behavioral design patterns. We will learn about the template design pattern and how it can be used to alter behavior of a system using some magic of oops.

## Questions

In this section, we will sharpen our brain by indulging in some fun brain activities. Let's get started.

### Pick the odd one out

facade	reduces complexity	payment
simplified	simplifies integration	subscription
unified	decreases performance	refund
complex	provides ready to use methods	content

*Figure 9.15: Pick the odd one out*

### Find the odd one out

Facade design pattern:

Provides a simplified interface

Is a structural design pattern

Allows for creation of multiple small interfaces

Reduces integration complexity

Facade design pattern is useful, when:

Complexity of integration is to be reduced

Clients perform integration

Dependencies are to be reduced

Impact of future changes should be limited

The facade design pattern

Can be used to develop more than one facade

Limits the clients to use only facades

Can interact with any number of internal sub-systems

Relieves the client from major maintenance overload

Select two correct answers

Facade design pattern:

Hides complexity from clients

Allows clients to interact with any number of interfaces

Provides a simplified interface to the clients

Can be private

Facade design pattern can be used to reduce:

Complexity

Interactions

Exceptions

Methods exposed to the client

What goes together?

Integration, system, client

Reduce complexity, facade, simpler integration

Increase complexity, unified interface, facade

Facade, behavioral, multiple integrations

## Answers

### Pick the odd one out

Image [Column Wise]

complex, decreases performance, content

### Find the odd one out

Allows for creation of multiple small interfaces

Dependencies are to be reduced

Limits the clients to use only facades

**Select two correct answers**

Hides complexity from clients, provides a simplified interface to the clients

Complexity, methods exposed to the client

[integration, system, client] & [reduce complexity, facade, simpler integration]

## CHAPTER 10

### Template

---

You can find the editable code files for this chapter on the book's GitHub link.

<https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers/tree/main/Chapter%2010>

---

## Structure

Topics that make this journey worthwhile:

Introduction

What is a Template?

Template method design pattern

Building a template method

Template method design pattern - defined

Benefits and drawbacks

Usage

## Objective

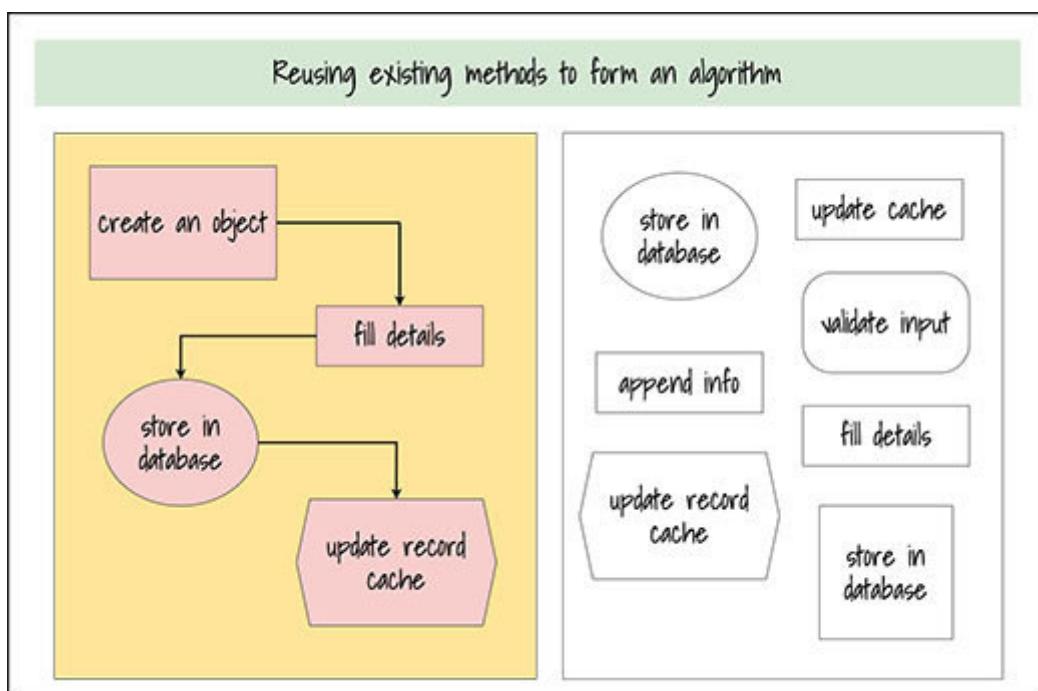
Our objective is to understand dynamic algorithms and learn how to apply polymorphism to bring those algorithms to life. We will learn about the template method design pattern and how it can be utilized to alter the behavior of an algorithm in runtime.

We will also walk you through the implementation details of the template method design pattern, the benefits it provides and a few drawbacks along with some of the places where this design pattern can be used.

## Introduction

As a developer, I have found so many similar algorithms being written time and again, with slightest of changes. These changes often apply only to a part of the complete algorithm and ends up duplicating a lot of code. It requires a good amount of effort to rework and test most of that algorithm, again.

The algorithms that we write today should make way for dynamism and least code duplication. They should be made of multiple smaller chunks of code, the methods, allowing code reuse and should provide placeholders that can be filled in with appropriate method definitions to support varying, dynamic behavior. [Figure 10.1](#) demonstrates an algorithm that is an amalgamation of multiple methods:



### ***Figure 10.1: Reusing Methods***

Left side of the figure describes the flow of algorithm and the right side represents the methods that can be used to give definition to that algorithm.

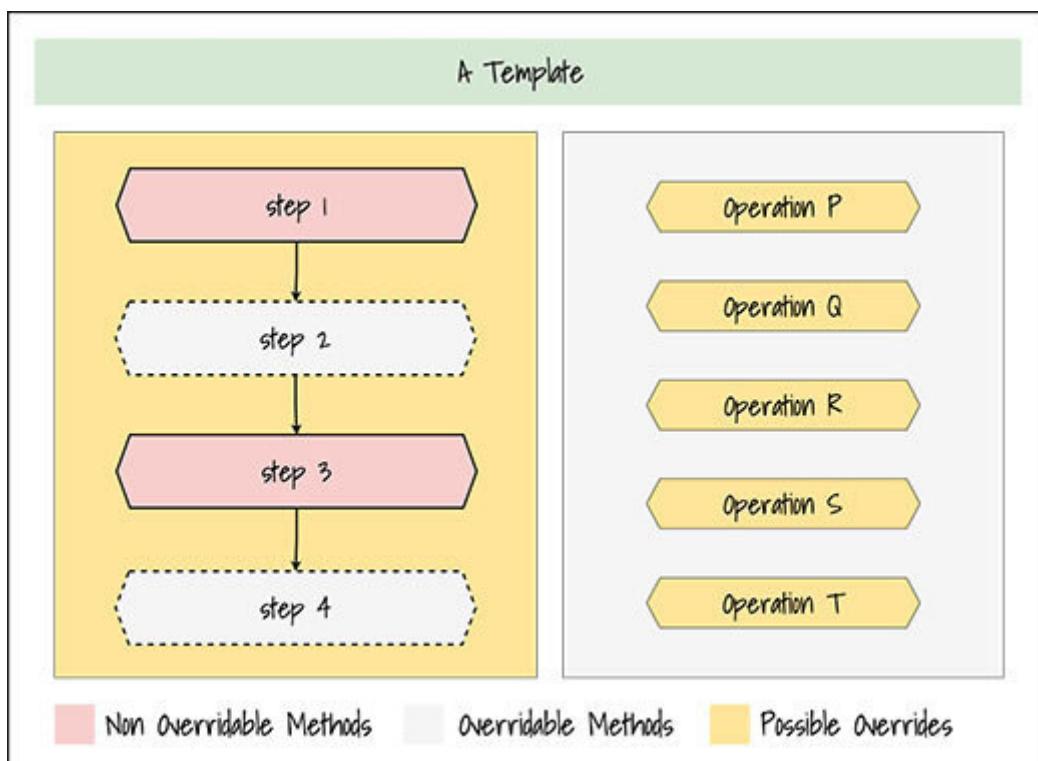
Now, although it is a good practice to break down large methods into multiple small ones that perform specific tasks so that it can be reused with various algorithms, they might not always be suited due to the variation in the method signature. Such methods might require a change in signature or be wrapped within another method.

While all that is good, what about the order and sequence of calling these methods? What if we construct abstract methods in place of concrete definitions and allow them to be overridden or replaced according to the desired behavior?

## Template

Have you ever received an invitation for an Indian marriage? Wonder why they look so similar and often have the same text. The format of the dates, the way the names are mentioned, and the mantras that form an indistinguishable part of the invitation look so much alike. This is because the wedding cards usually follow a set format, that is, in existence since time immemorial.

Figure 10.2 demonstrates an algorithm in the form of a template:



**Figure 10.2: A Template**

The algorithm is divided into four steps, some of those steps are permanent while others could be overridden to change the algorithms behavior.

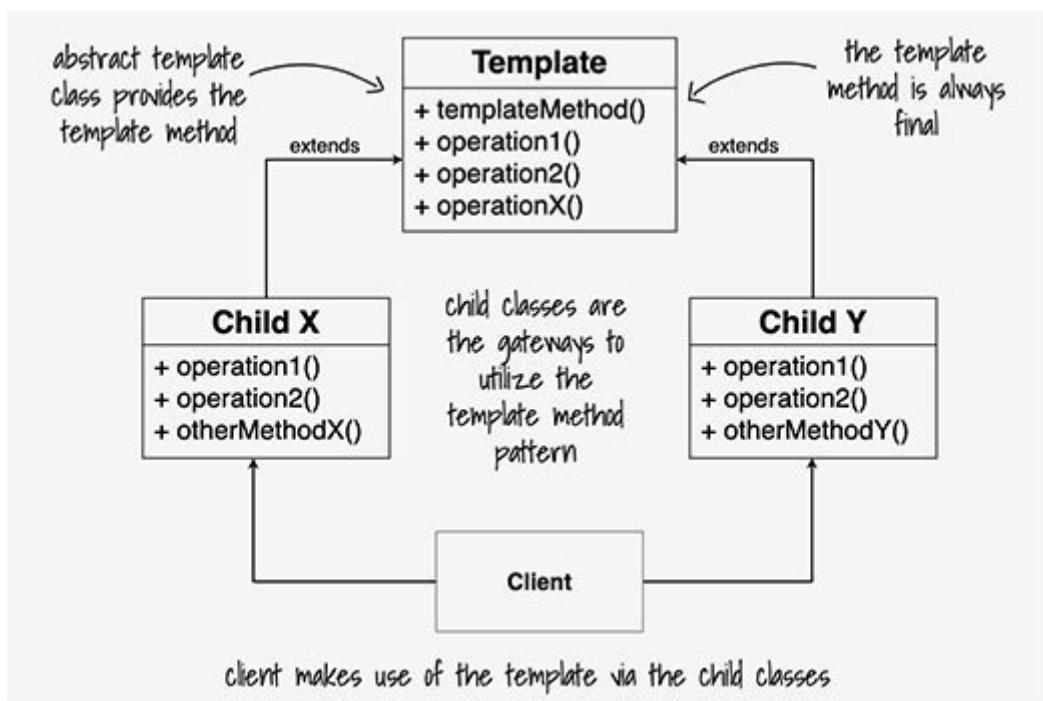
Let's get straight to the point; there is a template that is being followed for preparing the wedding cards. A way of presenting all that information in a pattern we are accustomed to. It also gives less headache to those who prepare and print these wedding cards. So, our next big question is how can we define a template?

In computing, a template is a skeleton, a kind of a model. It defines a sequence of operations that is organized in an order. It could also be understood as an arrangement of steps that are taken to process an algorithm. These steps could be abstract or concrete, depending upon the arrangement of steps in a template.

## Template method design pattern

The Template Method design pattern is a behavioral design pattern that allows for creation of a skeleton of operations. It provides the high-level steps of an algorithm. The steps are in the form of methods that can be overridden to support dynamic behavior without making any change in the overall structure.

[Figure 10.3](#) demonstrates the class diagram of template method design pattern:



**Figure 10.3:** Template Method Design Pattern

The class that exposes the template method is called the parent class. The template method preserves the sequence of the algorithm and is therefore marked final to restrict it from being overridden. It does not have any definition of its own; it merely acts as an organizer that operates the flow of the algorithm by delegating responsibilities to other methods.

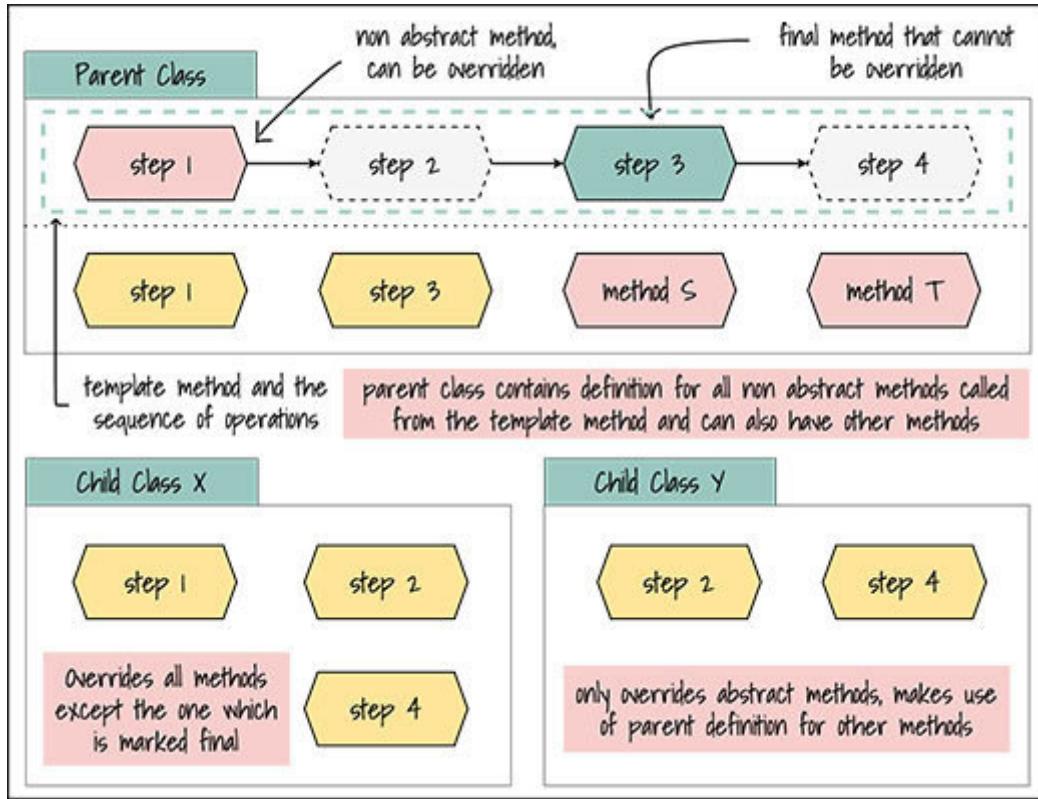
The parent class has all sort of methods, such as:

The final ones that cannot be overridden

The abstract ones that must mandatorily be overridden

The usual overridable methods that may or may not be overridden by the child is based on the requirement. The overridable ones are often referred to as default implementations or the helper methods. The abstract ones are also called **placeholders** or

Figure 10.4 demonstrates the structure of a template method and how child classes can override it to alter the definition of an algorithm:



**Figure 10.4: Template Method and Parent Class**

The parent class usually provide helper methods for the common, generic kind of implementation that is required in most of the variations of the algorithm. The child class is responsible for implementing rest of the skeleton by providing definition to the abstract methods defined by the parent.

The dynamic behavior that is at the core of the template method pattern is made possible due to inheritance and polymorphism, one of the four fundamental legs of object-oriented programming. It is only through a child object that the template method can be utilized. The template method calls all the other methods that form the sequence of steps of the algorithm it exposes. While some of the method definitions are provided by the parent itself, definitions for all the abstract ones are provided by the child. It is

at this moment when polymorphism comes into action and the child implementation is called for all the hook or placeholder entries in the template method pattern.

Here's how it works:

Instantiate an object of the child class

Call the template method on the child class object

The template method calls all those methods that form the algorithm in the natural order

Each time a call is made, and if the called method is overridden by the child, the child implementation is called; otherwise, the parent implementation is used

In the next section, we will demonstrate the working and usage of the template method design pattern.

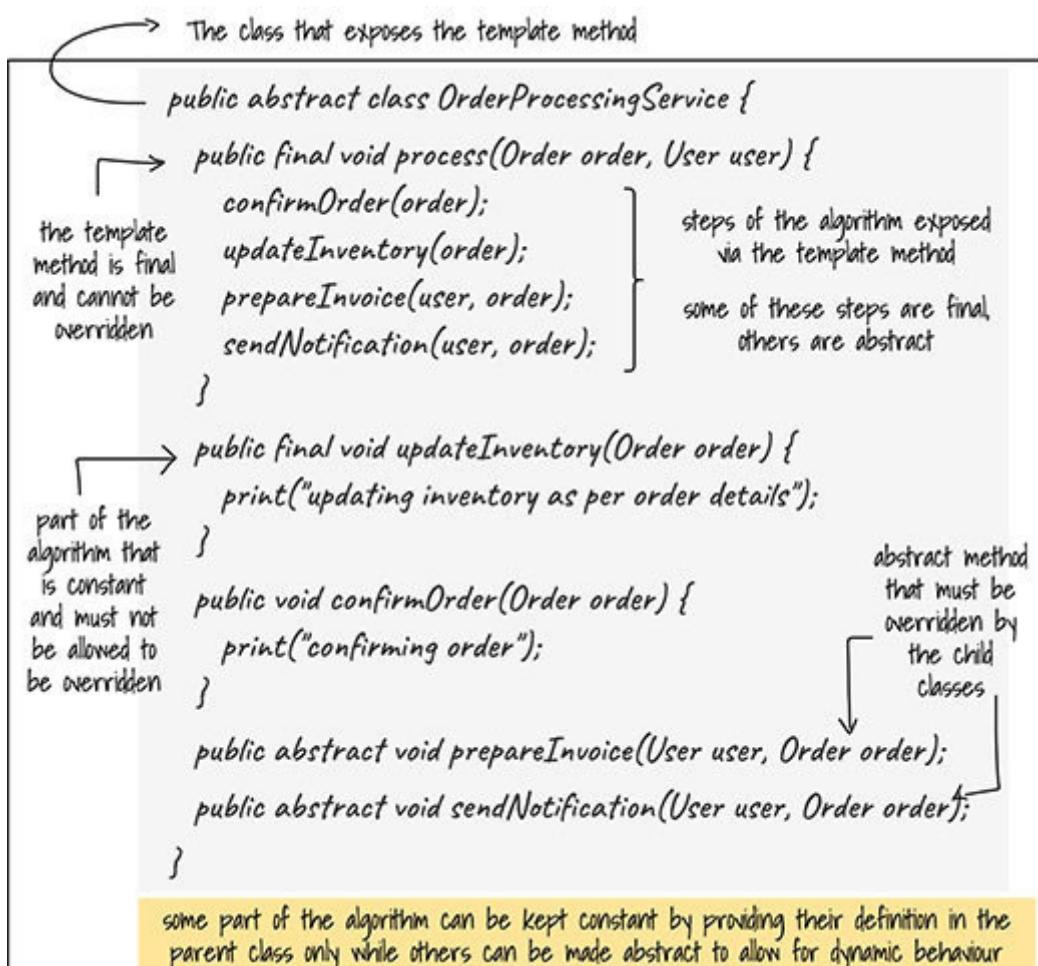
## **Building a template method**

The example that we will walk you through, in this chapter, makes use of an abstract class that exposes the template method and a few of its subclasses. The subclasses override the abstract methods to complete the algorithm exposed as a template method. We will learn about implementing the template method and how it helps to shape the behavior of an algorithm while keeping its structure intact. Let us first look at the abstract class that provides the template method and forms the skeleton of our algorithm. Later, we will discuss the implementations that enable dynamic behavior.

## The template method

The template method provides the skeleton of an algorithm. It lays out a sequence of steps that are executed one after another. The provided steps are in the form of methods that are defined by the parent class or its subclasses.

In [Figure](#) the **process** method serves as the template method:



**Figure 10.5: The Template Method**

It contains calls to four different methods that complete the order processing after a successful payment. The methods that form a part of the template method can be final as well as abstract depending on what task they perform and whether that task is invariable or variable.

In every algorithm that can be made generic to work for many different use cases, some part of it can be common enough to not need any variation or is a must to perform in only one way. Such parts of the algorithm are defined in the template using final methods. It ensures that they are not accidentally overridden. For other parts of the algorithm, the parent class could either provide overridable method implementations or abstract methods. The non-final methods facilitate the dynamic behavior of the algorithm wherein the methods that are abstract in the parent class must be overridden by the subclasses.

## Varying implementations

The abstract methods that we talked about earlier need a definition so that the algorithm exposed using the template method can be provided a proper, overall definition. The child classes control the behavior of the algorithm by providing definitions to those methods.

In *Figure* the child class has overridden both the abstract methods:

a child class that defines the abstract methods used in the template method

```
public class BulkOrderProcessingService extends OrderProcessingService {  
    public void prepareInvoice(User user, Order order) {  
        print("preparing invoice for bulk order");  
    }  
    public void sendNotification(User user, Order order) {  
        print("sending email to merchants");  
        print("sending email to user");  
    }  
}
```

hooks or placeholders

these methods bring the much talked about dynamic behaviour to the template method and the algorithm it wraps within itself

**Figure 10.6: Implementation 1**

The definitions provided by these implementations shape the behavior of the algorithm. Polymorphism plays a big role in ascertaining which particular behavior will be selected during runtime based on the type of object the template method is

called. For example, when the template method is called on an object of the class type described in [Figure](#) the notification is sent to the merchants as well as the user.

These abstract methods are placed in the template as hooks or placeholders and are supposed to be overridden by the implementations. While the abstract methods should mandatorily be overridden, the methods with default implementation can also be overridden if required.

In [Figure](#) the child class not only overrides the abstract methods but also overrides a method that already has a definition in the parent class:

```
public class RetailOrderProcessingService extends OrderProcessingService {  
    public void prepareInvoice(User user, Order order) {  
        print("preparing invoice for the user");  
    }  
    public void sendNotification(User user, Order order) {  
        print("notifying user via sms");  
    }  
    public void confirmOrder(Order order) {  
        print("verifying payment status");  
        super.confirmOrder(order);  
    }  
}
```

hooks or placeholders

a non-abstract method  
overridden by this extension  
of the parent class

polymorphism ensures that correct  
methods are called during runtime

to experience polymorphism in a compound way, the classes can be extended in a hierarchy  
to keep some part of the algorithm unchanged while changing others with every definition

**Figure 10.7: Implementation II**

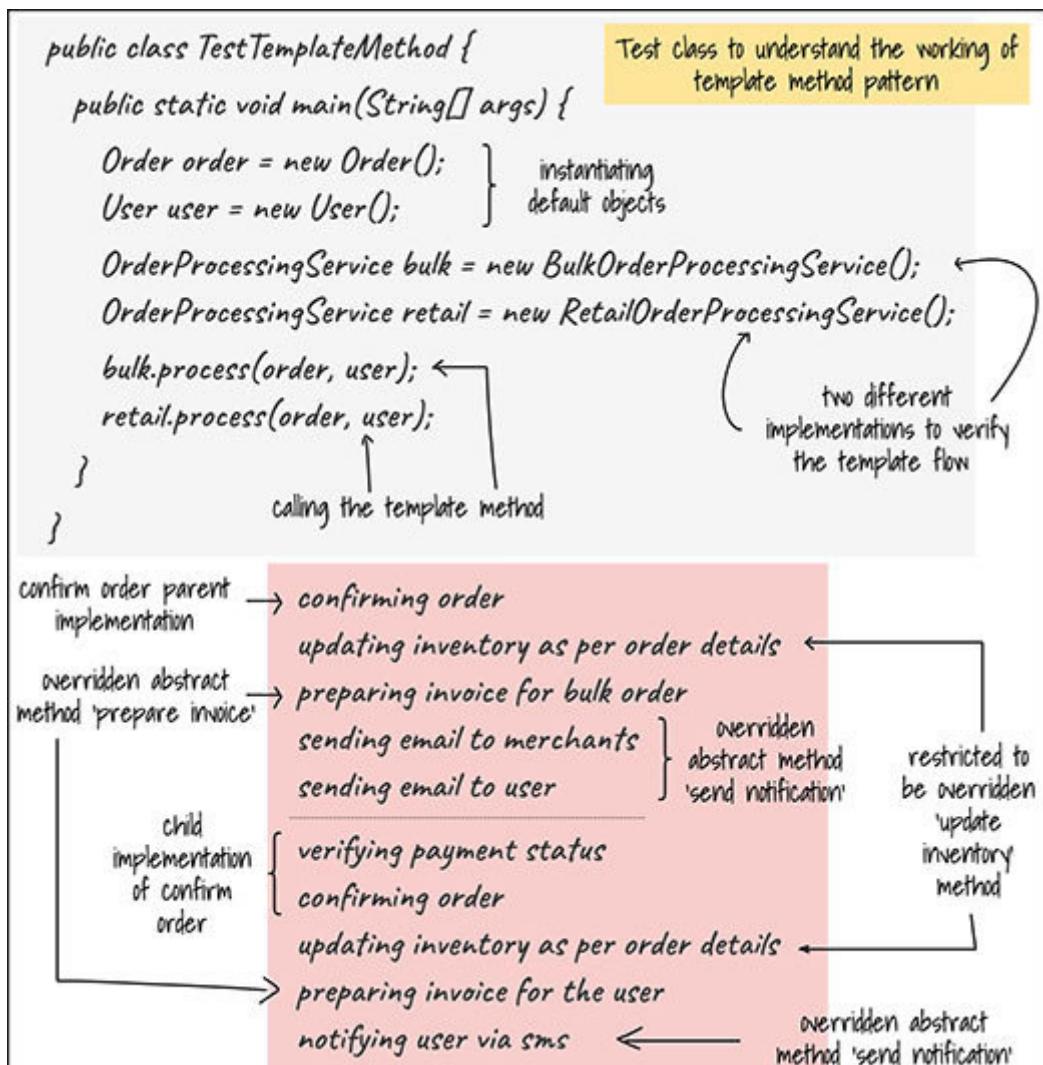
This scenario demonstrates that some methods can have a default implementation that may or may not be overridden based on the use case. While some child classes may end up providing a different implementation to those methods, others can just make use of the default one.

These implementations are only for understanding the concepts and might not be relevant for actual business logic. You are advised to exercise caution and only use it for learning purposes.

## Testing the template method pattern

We have written a test class to verify that the pattern we worked on works. Let's execute the same.

In Figure we first instantiate objects of both the implementations of the parent class and then call the template method on both:



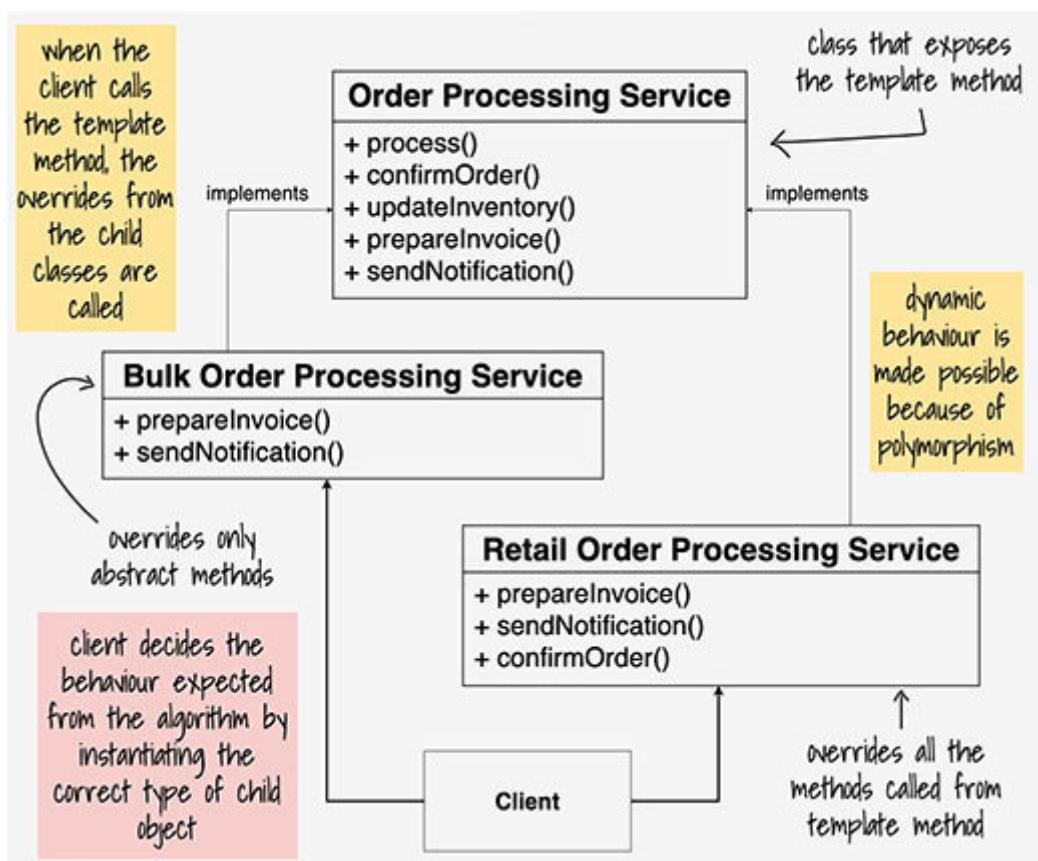
***Figure 10.8: Test Template Method***

The difference in the output confirms two different behaviors achieved using the template method pattern.

## Template method design pattern - defined

The template method design pattern allows for defining a generic algorithm, which can provide different behaviors by overriding the methods that are a part of the template method. The methods can be abstract or final, depending upon what part of the algorithm must be kept uniform and what must be variable.

[Figure 10.9](#) demonstrates the class diagram of the template method design pattern in the context of the example used in this chapter:



***Figure 10.9: Template Method Pattern - Defined***

The client is responsible for deciding the behavior the algorithm demonstrates, by selecting the correct child implementation.

Polymorphism then ensures that the right set of methods are called when the template method is invoked on the object.

## Benefits and drawbacks

It is time to make note of some of the benefits and drawbacks of the template method design pattern. Let's start with the benefits first before we look into the drawbacks of it:

Easy to introduce new behavior to an existing algorithm

Promotes code reusability

Avoids code duplication by implementing common steps, which are followed by each variation of the algorithm in the parent class

The drawbacks of the template method design pattern are as follows:

The template has a predefined sequence of steps; any change in the ordering will require a new template

Having more abstract steps could be hard to manage

The sequence of steps should be known before the pattern is applied

## Usage

Some of the scenarios relevant for creating a template method are:

When multiple outcomes are possible with slight change in the algorithm

For writing frameworks where only necessary steps are exposed to the implementing classes

## Conclusion

The template method design pattern is a widely used behavioral design pattern that can be utilized to expose a sequence of steps to form an algorithm. These steps can be defined in multiple ways by various implementations to provide a wide range of behavior.

In the next chapter, we will learn about the observer design pattern, and how it can be used to notify state change of an object to its observers.

## Questions

In this section, we will sharpen our brain by indulging in some fun brain activities. Let's get started.

## Pick the odd one out

template method	child implementation	parent provides generic implementation
final	abstract	child provides custom implementation
abstract	non overridable	final template method
sequence of steps	overridable	abstract template method

**Figure 10.10:** Pick the odd one out

### Find the odd one out

The template method design pattern

Promotes code reusability

Provides a sequence of steps to perform some task

Reduces readability

Relies on polymorphism

The template method design pattern

Can be used to reduce code duplication

Allows multiple behaviors with minor changes

Can have final methods

Is a structural design pattern

The template method

Is final

Provides a skeleton

Is abstract

Can call abstract and non-abstract methods

Select two correct answers

The template method design pattern

Promotes code reusability

Can be used without polymorphism

Cannot be used with inheritance

Reduces code duplication

The template method

Allows for multiple behaviors

Can be used to drive behavior using conditional statements

Is exposed by the child classes

Contains only abstract methods

What goes together?

Polymorphism, template method, abstract, final

Dynamic behavior, behavioral, template method

Structural, template method pattern, sequence of steps

Polymorphism, inheritance, template, skeleton

## Answers

### Pick the odd one out

final, non-overridable, abstract template method

## Find the odd one out

Reduces readability

Is a structural design pattern

Is abstract

**Select two correct answers**

Promotes code reusability, reduces code duplication

Allows for multiple behaviors, can be used to drive behavior using conditional statements

[dynamic behavior, behavioral, template method] and [polymorphism, inheritance, template, skeleton]

## CHAPTER 11

### Keep a Close Eye

---

You can find the editable code files for this chapter on the book's GitHub link.

<https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers/tree/main/Chapter%2011>

---

## **Structure**

Topics that make this journey worthwhile are as follows:

Introduction

Observer design pattern

Building observer design pattern

Observer Design Pattern: Defined

Benefits and drawbacks

Usage

## Objective

Our objective is to understand the mechanism of subscription-based information transfer. When some data or change in data is relevant to multiple systems, transferring it to all of them is a challenge. The observer design pattern helps us to address that challenge in a way that is both organized and asynchronous in nature.

We will walk you through the implementation details of the observer design pattern, the benefits it provides and a few drawbacks along with some of the places where this design pattern can be used.

## Introduction

Transferring information in a controlled manner is a common use case among systems that coexist and cooperate. The information that is being transferred could be an integer, a text value or maybe an object that stores complex information. It largely depends on the use case and the implementation of it. If there are only two systems involved, then it becomes a straightforward solution where one system is the sender of the information and the other is the receiver. But as soon as the number of receivers increase the same implementation is not enough, it requires changes to accommodate multiple receivers that are interested in the same information.

Systems that rely on each other for updates often prefer some sort of asynchronous communication. This helps to keep the resources in check and not being used up for unnecessary waiting periods. Subscribe and notify mechanism is one such way that can be used to transfer information from the sender to the receiver. The sender often keeps a list of all the receivers who are interested in the information and updates them whenever there is a change.

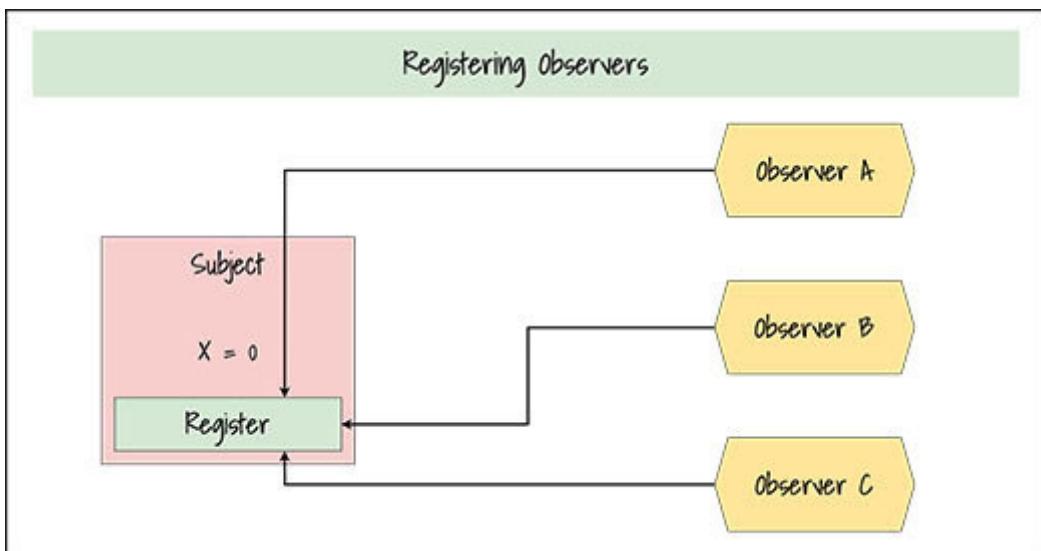
In this chapter, we will discuss the observer design pattern and how it can be utilized to broadcast information to all the subscribers that are interested in it.

## Observer design pattern

The observer design pattern is a behavioral design pattern that allows to define a subscribe and notify based mechanism for transfer of information. The two kinds of object that form the basis of the observer design pattern are called the subject and the observer. The subject is the point of interest and the observers are those multiple objects that take interest in the subject. In short, the subject is what is being observed by the observers.

The observer design pattern describes a one-to-many relationship between objects where a single subject is observed by multiple observers. These observers subscribe to the subject and are notified about any change in it.

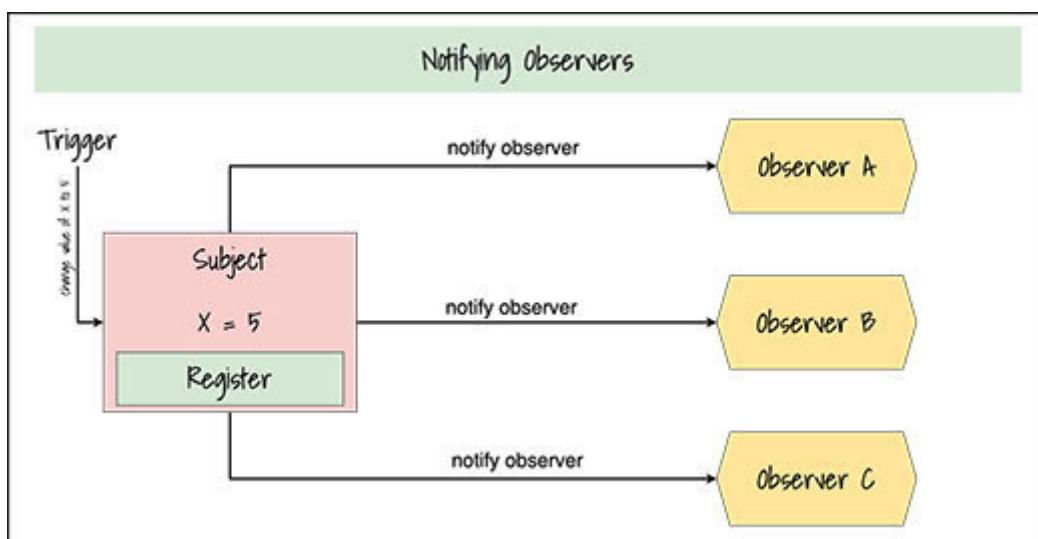
In [Figure](#) multiple observers are depicted registering themselves with a subject:



**Figure 11.1: Registering Observers**

A request is sent to the subject with a reference to the observer that should be registered to receive updates. The same process must be followed every time an observer has to connect to an observable. Value of X in the subject is the point of interest for all the registered observers.

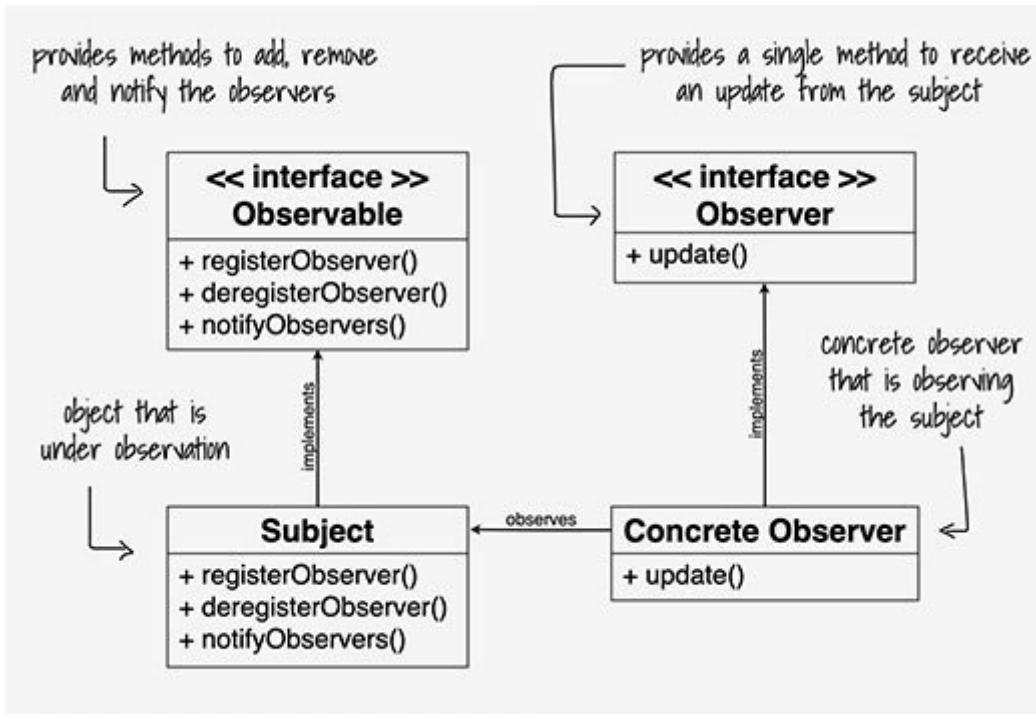
In [Figure](#) the subject is depicted notifying all the active observers about the change in the value of X, that is now 5:



**Figure 11.2: Notifying Observers**

The observers are ensured to receive an update whenever there is a change in the value of the concerned variable or object. The subject notifies all the active observers by calling the update method of the observer interface.

Figure 11.3 demonstrates the architecture of the observable design pattern. It describes the methods provided by the observable and the observer interfaces and their concrete implementations:



**Figure 11.3: The Observer Design Pattern**

Here's how it works:

Instantiate an object of type observer

Register that object with the observable

The observable notifies all the observers including the one we registered in the step above

When the observer receives an update, it can process that information as required

The observer design pattern could be made to work in two different ways, that is, push or pull. The information can be directly transferred to the observers whenever there is any change to the subject using the push mechanism. The pull mechanism, on the other hand, can be used to allow the observers to fetch the information in the form and shape they expect. The subject still notifies the observers about any change in the state. We will talk more about the push and pull mechanism later in this chapter.

The traffic light on a cross section is a great example of the observers and the subject. The motorists and the riders all observe the change in the traffic light when they are at the cross section. A change in the signal from red to green or green to yellow or from yellow to red is visible to all the motorists and the riders that helps them decide whether they should wait at the cross section or drive ahead.

In the next section, we will demonstrate the working and usage of the observer design pattern.

## **Building observer design pattern**

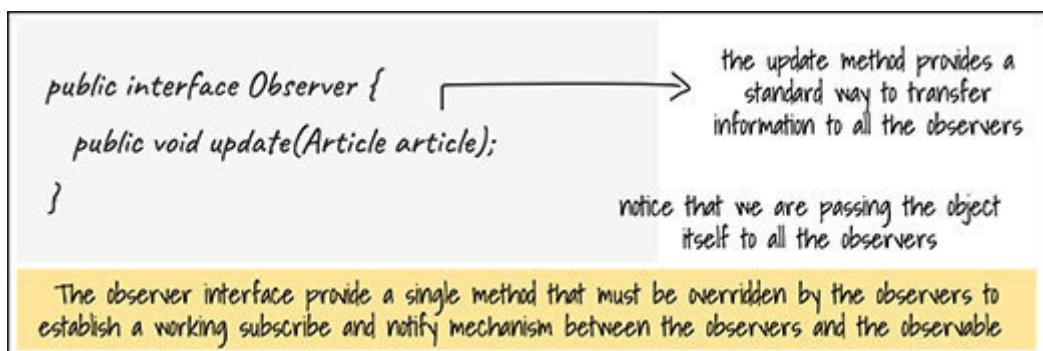
The example that we will walk you through in this chapter is of a virtual auction event. The auction event is supposed to host multiple buyers for the articles that will be up for sale during the event. An auctioneer will oversee the proceedings, virtually, to introduce the article to all the buyers and to sell them to the highest bidder. The publish and subscribe mechanism that we will develop through this example should take care of notifying the auctioneer and the registered buyers about the articles that are currently up for sale, one article at a time.

Our example makes use of interfaces and their concrete implementations, that together form the publish and subscribe mechanism to support this event. The interfaces form the basis for setting up the observers and the observable. We will learn about implementing the observer design pattern and how it can be used to build a publish subscribe model to broadcast changes in a controlled manner. Let's first look at the interfaces that form the backbone of the observer design pattern and later we will discuss the concrete implementations of them to set up a working subscription mechanism for the virtual auction.

## The interfaces

The observer design pattern provides two interfaces:

The **Observer** interface provides a single method that is used by the subject to update the observers whenever a notification is made as shown in [Figure](#)



**Figure 11.4: The Observer Interface**

The **Observable** interface provides methods that are used to register and deregister observers to the subject and to notify the observers about a change as shown in [Figure](#)

The observable interface provides methods that allow observers to register and deregister themselves and to receive notifications from the subject

```
public interface Observable {  
    public void registerObserver(Observer observer);  
    public void deregisterObserver(Observer observer);  
    public void notifyObservers(Article article);  
}
```

→ Adds an observer to the registration list. Once added, the observer will be notified of the changes that happen to the subject.

→ Removes an observer from the registration list

↓ Notify observers when there is a change. The change triggers the notification event.

Figure 11.5: The Observer Interface

## The subject

The auction must allow buyers to register and deregister and it must notify the registered buyers about the change in the article that is currently being auctioned. To support all this, our auction class implements the observable interface.

In [Figure](#) we have showcased the implementation of our auction class. It maintains a list of all the active observers. Whenever an observer is registered with or deregistered from the auction, this list is updated:

```

    → a class that allows to be observed
public class Auction implements Observable, Event {
    private List<Observer> observers; → maintains a list of
    private List<Article> articles;
    private int articleCount; → these articles are on sale in this auction
    private int totalArticles;
    private Optional<Article> currentArticle; → current article on sale
    public Auction(List<Article> articles) {
        this.observers = new ArrayList<Observer>();
        this.articles = articles;
        this.articleCount = 0;
        this.totalArticles = articles.size();
    }
    → adds an observer to the list,
    public void registerObserver(Observer observer) {
        observers.add(observer);
    } → these observers will receive
    → removes an observer from
    public void deregisterObserver(Observer observer) {
        Iterator<Observer> iterator = observers.iterator();
        while(iterator.hasNext()) {
            if(iterator.next().equals(observer)) {
                iterator.remove();
            }
        }
    } → the observers can register and deregister
    → we still have some
    } → themselves dynamically in runtime methods left to discuss
}

the observers are loosely coupled to the subject, the observer keeps track of all the active
observers using a collection

```

**Figure 11.6: The Subject | Register and Deregister**

It also maintains a list of articles that will be up for sale during the event. The figure also demonstrates a working definition of the register and deregister methods that are overridden by the auction implementation.

[Figure 11.7](#) demonstrates a working definition of the **notify** method, the third method from the **Observable** interface. It traverses through every active observer and calls the update method on it:

```
public void notifyObservers(Article article) {  
    observers.forEach((o) -> {  
        try {  
            o.update((Article) article.clone());  
        } catch (CloneNotSupportedException e) {  
            log(e);  
        }  
    });  
}
```

notify all observes about  
the change in the  
current article that is up  
for sale in this auction

we are passing a clone of the original object just to ensure  
that the original one remains untouched and any change to  
the passed object is not propagated to all the other  
receivers

**Figure 11.7: The Subject | Notify**

Notice that we have created a clone of the original article object. This is to ensure that any change done by an observer to this object does not affect other observers.

Since the auction is also an event that must start and come to an end, it therefore inherits the event interface that allows for setting up and managing an event. The event interface is not related to the observer design pattern; it is used to demonstrate the working of notifications and the event that triggers them. The event interface provides the trigger points that allow for selecting the article that is next in line for auction and to subsequently send that information to the registered buyers. The buyers are updated about the next article through notifications that are sent by calling the update method on each buyer instance.

Figure 11.8 demonstrates the definitions of the methods inherited from the event interface. The **next** method does most of the work for us; it is this method that is called whenever the next article is to be made available for auction. It fetches the next available article from the list and notifies each buyer about that article by calling the **notify** method on all the buyer instances.

The event is concluded if there are no more articles available to auction:

```
public class Auction implements Observable, Event {
    // continuing where we left ...
    public void start() {
        next(); → calling the next method to start
    } → with the first object in the
    articles collection
    public void next() {
        nextArticle();
        if(!currentArticle.isEmpty()) {
            notifyObservers(currentArticle.get());
        } else {
            end(); → end the auction if all the
            articles are auctioned
        }
    }
    private void nextArticle() {
        if(totalArticles > articleCount) {
            currentArticle = Optional.ofNullable(articles.get(articleCount++));
        } else {
            currentArticle = Optional.empty()
        }
    }
}
```

The event interface provides methods that help to manage and progress through an event. In our case it is the auction event.

notifies all the observers registered with the auction event about the current article in display for auction

scrolls through the articles and provides the next one from the collection until all the articles are traversed

provides an empty optional value when the end of the collection is reached

**Figure 11.8: The Subject | Event**

A call to the **next** method triggers the event that updates the next article up for auction and that triggers the notifications to the registered buyers.

## The observers

The buyers who register for the auction expect to be notified about the articles when they are made available for sale. The buyers are observing the auction event. Apart from the buyers we also have another observer, the auctioneer, who drives and supervises the virtual auction event.

In [Figure](#) both the auctioneer and the buyer are demonstrated implementing the **Observer** interface and overriding the update method. The **update** method, as we also mentioned in the previous sections, is called by the subject to send updates to the observers:

```
public class Auctioneer implements Observer {  
    public void update(Article article) {  
        AuctioneerViewController.refresh(article);  
    }  
}
```

the received update triggers a call to the respective view controllers that refresh the user interface with the latest article up for auction

```
public class Buyer implements Observer {  
    public void update(Article article) {  
        BuyerViewController.refresh(article);  
    }  
}
```

we have kept the observer implementation simple, just to elaborate the purpose  
in real use cases the observers can do much more than refreshing the views

**Figure 11.9: The Observers**

The observer implementations are kept simple to emphasize on the design aspect. The observers are not just limited to receiving the data and sending it forward as in [\*Figure\*](#) instead they can process it, store it, do complex calculations on it, and much more.

## Testing the observer design pattern

We have written a test class to verify that the pattern we worked on works. Let's execute the same.

In [Figure](#) we first instantiate our observers, that is, an instance of auctioneer and a few buyer instances. Then we fetch the articles that will be up for sale from the article repository. Once we have the list of articles, we instantiate an auction object passing in the list of articles to set up the auction event. The next thing we have done is to register the auctioneer and all the buyers as observers to the auction event.

And at the very end, we commence the auction event calling its start method. To continue with the auction, we are traversing through all the available articles one by one, each time calling the **next** method on the auction. At last, when all the articles will be traversed, the auction will be concluded:

```

public class TestObserver {
    public static void main(String[] args) {
        Observer auctioner = new Auctioner();           → instantiate an
        List<Observer> buyers = new ArrayList<>();     → auctioneer object to
        IntStream.range(0, 4).forEach(x -> buyers.add(new Buyer())); → conduct the auction
        List<Article> articles = ArticleRepository.getArticles();
        Auction auction = new Auction(articles);          → instantiate four buyer instances
        auction.registerObserver(auctioner);             → register observers with
        buyers.stream().forEach(x -> auction.registerObserver(x)); → the auction event
        auction.start();                                → start the auction
        articles.stream().forEach(x -> auction.next());   ↓
    }
}

```

**Figure 11.10:** Test Observer Design Pattern

The buyers will be updated about the new article from time to time whenever the next method on the auction is called.

## Variations

The observer design pattern has two variations, the push variation, and the pull variation. Both use the same interfaces and have similar implementations apart from a small difference that defines them.

To demonstrate the difference in their implementation, the previous example requires few changes and some additional interfaces. Most of these changes are relevant for implementing the push mechanism and that is the one we will take up first.

## Push mechanism

This variation of the observer design pattern states that the subject should be responsible for sending the desired information to its observers. The subject should know the type and form of information required by its observers and should implement the logic to construct it in that form. It should be the responsibility of the subject to ensure that the observers receive the right information and do not have to implement any additional logic to convert it to the desired form.

[Figure 11.11](#) describes the changes and additions required for our previous example to support push mechanism:

```
public interface Observer {  
    public void update(Object article);  
}  
  
public interface Seller extends Observer {}  
public interface Customer extends Observer {}
```

notice that the update method has a parameter of type Object, this allows to pass any type of object to the observers

we have created two new interfaces to distinguish among the type of observer

**Figure 11.11: Push Mechanism | Observers I**

We have added two new interfaces to distinguish between two kinds of observers and our existing observer implementations now implement one of them each instead of the original observer interface as shown in [Figure](#)

```

public class Buyer implements Customer {
    public void update(Object article) {
        BuyerViewController.refresh(article);
    }
}

public class Auctioneer implements Seller {
    public void update(Object article) {
        AuctioneerViewController.refresh(article);
    }
}

```

the controllers will still receive the kind of object they expect, regardless of the object parameter type

our concrete observers now implement the two new interfaces that allow us to create a distinction among them

keeping Object as a parameter type increases the scope of the type of objects that can be passed to this method

**Figure 11.12:** Push Mechanism | Observers II

[Figure 11.13](#) demonstrates the updated definition of the **notify** method. The **notify** method now has an additional check to ascertain the type of observer and updates them with the information, in the form expected by each of them:

```

public void notifyObservers(Article article) {
    observers.forEach((o) -> {
        try {
            if(o instanceof Customer) {
                o.update(new ArticleOnSale(article));
            } else if(o instanceof Trader) {
                o.update((Article) article.clone());
            }
        } catch (CloneNotSupportedException e) {
            log(e);
        }
    });
}

```

the auction event, our subject, now notifies each observer with information expected by them

the subject must know the type and form of information to send when calling the update method

in push mechanism the observers and the observable become tightly coupled if each observer expects a different type of object in the call to update method

**Figure 11.13:** Push Mechanism | Notification

The requirement for the subject to know about the different types of observers and their expected form of information bounds them in a tightly coupled relationship.

### Pull mechanism

This variation of the observer design pattern states that the observers should pull the required information from what is passed to them via notification. The subject should pass on the entire information when they update the observers, and the observers should decide and fetch the relevant information as per their requirements.

This variation does not require changes in the subject but in the observers. The responsibility of processing the information received from the subject now rests with the observers. This keeps the relationship between the subject and observers loosely coupled and the subject is no more required to process the information for every different kind of observer. It does not even need to know the types of observers. All that the subject must ensure is the delivery of information whenever a change is triggered.

Figure 11.14 demonstrates an updated implementation of one of the observers observing the auction instance:

```
public class Buyer implements Customer {  
    public void update(Object article) {  
        Article artcl = (Article) article;  
        ArticleOnSale artosl = new ArticleOnSale(artcl);  
        BuyerViewController.refresh(artosl);  
    }  
}
```

instantiating an object of type ArticleOnSale from the received object of type Article

in pull mechanism the observer is responsible for extracting the information received in the update call

both the observers and the subject remain loosely coupled in this approach since the subject need not to be aware of the type of observers

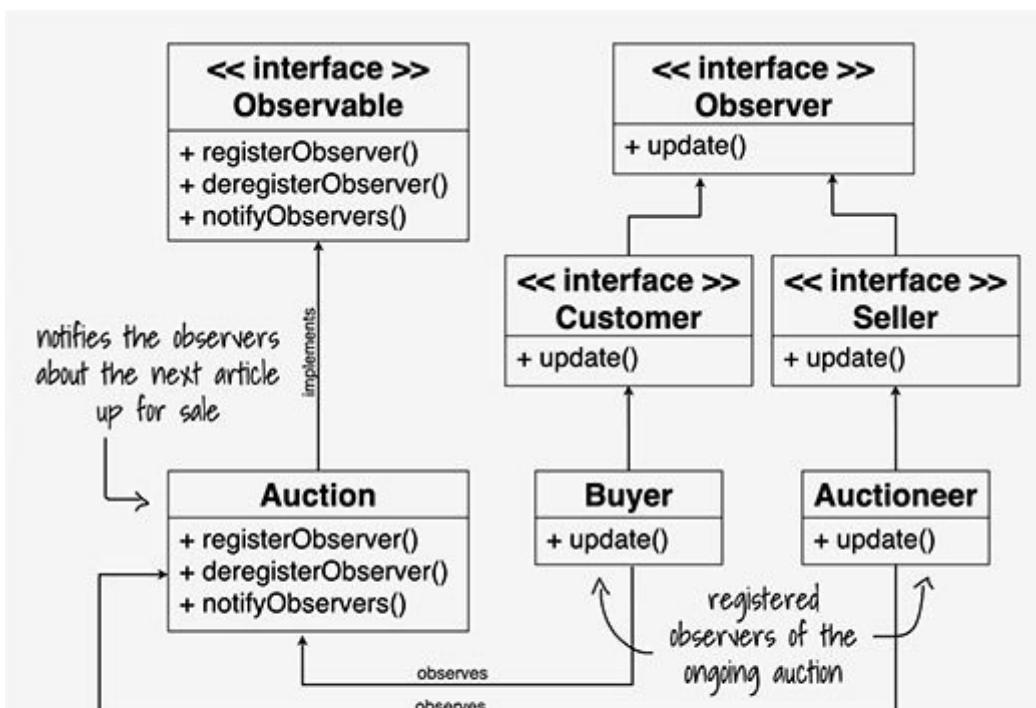
**Figure 11.14: Pull Mechanism | Observer**

It converts received information into the desired form before using it for further processing.

## Observer design pattern - defined

The observer design pattern allows for setting up a subscription model for information sharing where multiple observers observe a subject. The subject maintains a list of registered observers and notifies them of any change in state.

[Figure 11.15](#) presents a high-level view of the observer design pattern using the constructs we used to demonstrate the same in this chapter. It presents a consolidated view of those constructs and the relation between them. It also mentions the necessary methods required for the pattern to function correctly:



the observer design pattern can be implemented via the push or pull mechanism, in push mechanism the transferring information in the expected form is the responsibility of the observable whereas in pull mechanism the observers are responsible for fetching information as expected by them

**Figure 11.15:** Observer Design Pattern - Defined

## Benefits and drawbacks

It is time to make note of some of the benefits and drawbacks of the observe design pattern. Let's start with the benefits first before we look at the drawbacks of it. Some of the benefits are as follows:

The interacting objects, that is, the subject and the observers are loosely coupled in push mechanism.

Observers can be added and removed in runtime.

Broadcasts the information to multiple observers.

The drawbacks of the observer method design pattern are as follows:

The observers are notified in a random order.

The registration and deregistration process can cause memory leaks.

## Usage

Some of the scenarios relevant for the observer design pattern are as follows:

When the list of observers is dynamic in nature, can shrink and grow in runtime.

When the information is to be broadcasted to multiple receivers.

## Conclusion

Observer design pattern is a widely used behavioral design pattern that can be utilized to prepare a subscription-based mechanism for transferring information. There are two ways to implement this pattern, push and pull. Push mechanism states that the subject should be responsible for transferring correct form of information to the observers whereas the pull mechanism states that the observers should be responsible for fetching the correct form of information from what is received in the notification.

In the next chapter, we will learn about the state design pattern and how it can be utilized to allow an object to alter its behavior when its state changes.

## Questions

In this section, we will sharpen our brain by indulging in some fun brain activities. Let's get started.

## Pick the odd one out

push	register	update
pull	deregister	observer
tightly coupled	update	observable
subject is responsible for transferring correct information	notify	pull

*Figure 11.16: Pick the odd one out*

### Find the odd one out

Observer design pattern

Broadcasts information

Supports multiple subject

Restricts notification to a single observer

Cannot be used with multiple observers

Observer design pattern

Can be implemented in two ways

Supports data transfer from observer to subject

Holds on to the information if not queried for

Is a structural design pattern

The notification from the subject

Reaches all the observers at the same time

Is only sent to registered observers

Reached the observers in a specific order

Is only sent when the observer queries for it

Select two correct answers

Observer design pattern

Push and pull mechanism

Broadcasts information in a controlled manner

Cannot be used with inheritance

Reduces network latency

The observers

Can register and deregister in runtime

Can only connect to a single subject at a time

Can choose to not to receive notification when subscribed

Inherit a standard interface

What goes together?

Push, pull, observer design pattern, broadcast

Structural, observer design pattern, loose coupling

Observer, observable, notification, subscription

Push, observer, broadcast, structural

## Answers

Pick the odd one out

pull, update, observable

### Find the odd one out

Restricts notification to a single observer

Is a structural design pattern

Is only sent to registered observers

**Select two correct answers**

push and pull mechanism, broadcasts information in a controlled manner can register and deregister in runtime, inherit a standard interface

[push, pull, observer design pattern, broadcast] and [observer, observable, notification, subscription]

## CHAPTER 12

### State and Behaviors

---

You can find the editable code files for this chapter on the book's GitHub link.

<https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers/tree/main/Chapter%2012>

---

## Structure

Topics that make this journey worthwhile are as follows:

Introduction

State design pattern

Implementing state pattern

State design pattern: defined

Benefits and drawbacks

Usage

## Objective

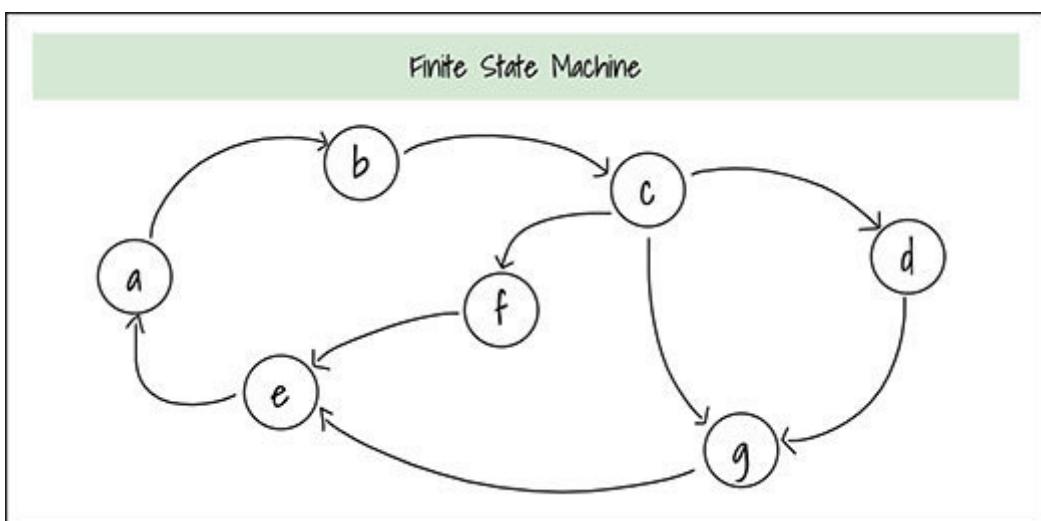
Our objective is to understand the complexity involved in maintaining different behaviors of an object under different conditions and how to utilize the state design pattern to streamline it.

We will walk you through the implementation details of the state design pattern, the benefits it provides and a few drawbacks along with some of the places where this design pattern can be used.

## Introduction

At any moment, there is a finite number of states an object can be in. These states govern the behavior that the object exhibits when an operation is performed on it. This concept is close to that of a finite state machine. A finite state machine can only be in one of a finite number of states at any moment in time. It behaves differently in different states and can be made to switch between those states instantaneously. The transition from one state to another is predetermined and hence the machine may or may not switch to certain states depending on the current state of the machine.

As described in [Figure](#) a state machine is often implemented using conditional operators to allow it to exhibit correct behavior and to switch between the states:

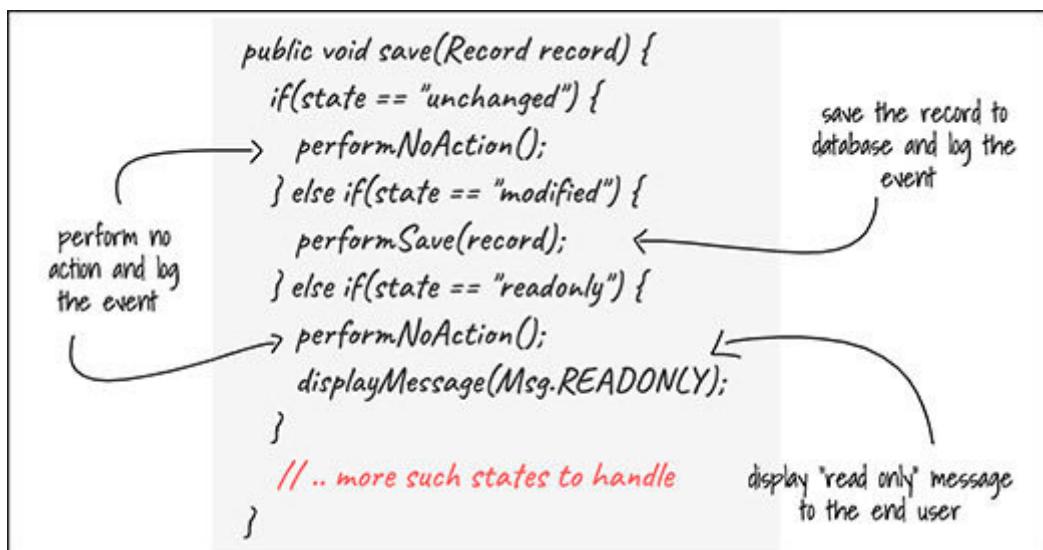


**Figure 12.1: Finite State Machine**

The class definition of a state machine uses multiple if else or switch statements to describe the flow of algorithm. A limited set of conditions can be easily handled, but the addition of new states and dependent behaviors, from time to time, increases the complexity of class definition. After a while, it becomes difficult to manage such an implementation as any change to the existing transition logic may require changes to the already complex and large method definitions that encapsulate the distinct behaviors of that object.

Frequent changes to the class definition keep it open for modification, an opportunity for bugs to creep in unnoticed.

Figure 12.2 demonstrates a method that uses multiple if else statements to perform correct operation based on the current state of the object:



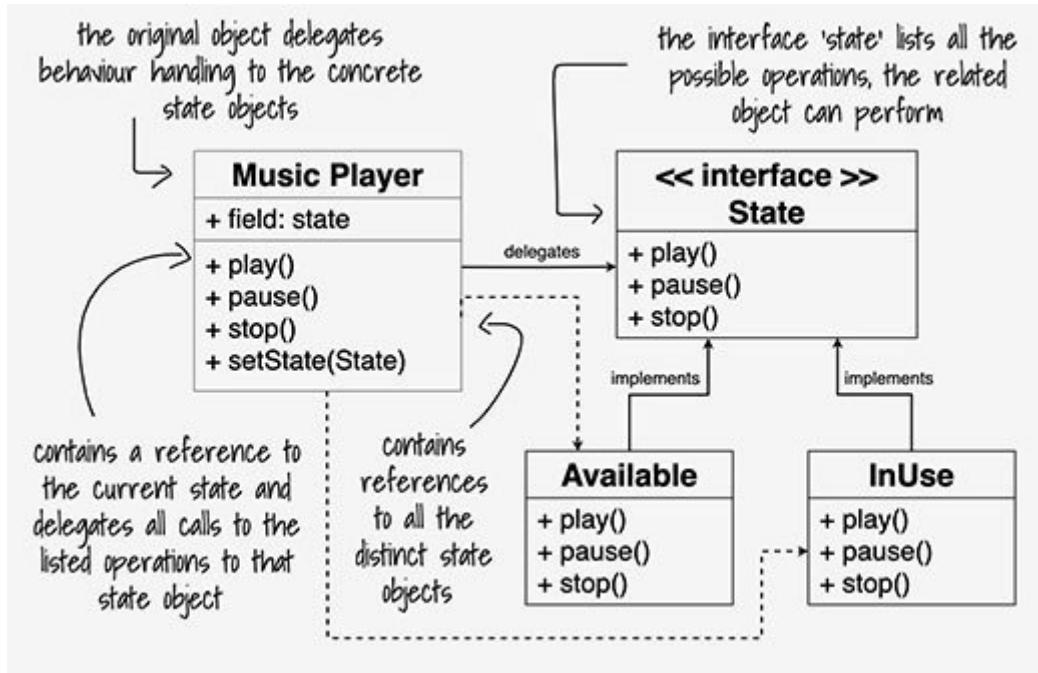
**Figure 12.2:** Managing object states using conditional statements

In this chapter, we will discuss the state design pattern and how it can be utilized to express states in the form of objects that encapsulate the distinct behaviors of the object they are bound to. We will learn about the power of delegation and how composition plays a big role in the state design pattern.

## State design pattern

The state design pattern is a behavioral design pattern that allows an object to alter its behavior whenever its internal state changes. The state design pattern promotes the single responsibility principle. It advocates segregating the conditional behavior of an object into distinct classes, that is, create classes for each state of that object and encapsulate the state specific behavior in them. This approach provides a clean and simplified way to work with multiple states of an object and makes it easier to transition between them, subsequently changing the object's behavior. In short, it eliminates the need of conditional statements to decide the correct behavior of an object.

Figure 12.3 demonstrates the architecture of the state design pattern:



**Figure 12.3: The State Design Pattern**

The **state** interface acts as a pivot and supports the conversion of a stateful object from a complex, condition driven, single class architecture to a streamlined, and composition-based architecture. Each implementation of the interface is responsible for a specific state and the respective object behavior.

The stateful object maintains a reference to its current state and uses it to perform operations. Instead of implementing the behaviors by itself, the stateful object delegates calls to the listed operations to the current state. The current state is managed with a polymorphic reference that can point to multiple concrete implementations at different points in time.

Having a distinct implementation for each state of an object makes it easier to introduce new states as it does not affect any

existing behavior that is encapsulated in other state specific implementations. The only required change is made to the definition of the stateful object itself. For the new state to be operational, it is added to the class definition of the stateful object. The state design pattern also promotes the open close principle by encapsulating distinct behaviors in state-specific classes.

The boarding pass kiosk on airports is a great example of the state design pattern. The kiosk has a set of predetermined states that are observed by the user when the kiosk is used. These states govern the behavior of the kiosk when various operations are performed on it.

## Implementing state pattern

The example that we will walk you through in this chapter is an imitation of the boarding pass kiosk that we discussed in the previous section. Our example makes use of an interface and its concrete implementations that form the distinct states of the kiosk. The concrete implementations need not override operations that are not relevant to them; the interface provides the default definitions for those.

We will learn about implementing the state design pattern and how it can be used to manage distinct behaviors of an object. Let's first look at the interface that forms the backbone of the state design pattern and later we will discuss the concrete implementations that encapsulate the behavior of the kiosk object.

## The interface

The state design pattern offers a single interface called state. The **state** interface provides exact replicas of the operations as the stateful object, that is, they have the same signatures. These methods are overridden by the concrete implementations to provide state-specific behaviors.

Figure 12.4 demonstrates the state interface that we have created for our example. It contains five different operations that can be performed on the kiosk object. The interface provides a default implementation for each of the methods to ensure that the concrete implementations need to override only those operations that are valid for the object state they represent. All the other operations are deemed invalid for that state:

```

    ↗ public interface State {
the interface lists
the operations that
are exhibited by the
object
    public default void selectAirline(Scanner scanner) {
        System.out.printf("Invalid Operation");
    }

    public default void inputPNR(Scanner scanner) {
        System.out.printf("Invalid Operation");
    }

    public default void pickSeats(Scanner scanner) {
        System.out.printf("Invalid Operation");
    }

    public default void reviewSelection(Scanner scanner) {
        System.out.printf("Invalid Operation");
    }

    public default void printBoardingPass() {
        System.out.printf("Invalid Operation");
    }
}

```

the default methods in our interface allow each state implementation to override only relevant operations

the methods encapsulate the varying behaviours of an object based on its state

not all operations are relevant in each state, such operations can be left unattended by the state implementations

an object can behave differently in different states

**Figure 12.4:** The State Interface

### The concrete implementations

The concrete implementations of the state interface encapsulate the behavior of the stateful object in accordance with the object state that they express. Each of these implementations has a reference to the stateful object that is used to change the current state of that object.

Our example, which demonstrates the use of state design pattern, uses five different states to illustrate the working of a boarding pass kiosk. These five states are called available, waiting, verified, review, and confirm. All of them implement the **state** interface and override the operations that exhibit a state-specific behavior.

Figure 12.5 demonstrates the implementation of the initial state of the kiosk, that is,

```

public class Available implements State {
    private BoardingPassKiosk kiosk; → the object the
    states are bound to
    public Available(BoardingPassKiosk kiosk) {
        this.kiosk = kiosk;
    }

    public void selectAirline(Scanner scanner) {
        Map<Integer, String> airlines = this.kiosk.getAirlines();
        System.out.println("Select the Airlines");
        airlines.forEach((key, value) -> {
            System.out.printf("%d -> %s\n", key, value);
        });
        System.out.print(": ");
        int airline = scanner.nextInt(); → display airlines for selection
        scanner.nextLine(); → scan user input
        kiosk.setAirline(airline);
        kiosk.setState(kiosk.getWaiting()); → Available: the boarding pass
    }                                     kiosk is available for use
}

```

this state can perform only one operation, all the other operations are invalid in this state

**Figure 12.5: State: Available**

This state indicates that the kiosk is available for use, to generate and print the boarding pass. The only valid operation in this state allows to select the airline to prepare a boarding pass. It presents the user with multiple options to select from, scans the user selection, and then switches the current state of the kiosk object to

All the other operations are invalid and are not defined in this state implementation. If those operations would be called from this state of the kiosk object, they will return with an invalid operation message. This holds true for all the other state

implementations as well, for the operations that are not overridden by them.

Figure 12.6 demonstrates the implementation of the next state of the kiosk, that is,

```
public class Waiting implements State {  
    private BoardingPassKiosk kiosk;  
  
    public Waiting(BoardingPassKiosk kiosk) {  
        this.kiosk = kiosk;  
    }  
  
    private static boolean validate(String pnr) {  
        return Pattern.matches("[a-zA-Z0-9]{6}", pnr);  
    }  
  
    public void inputPNR(Scanner scanner) {  
        System.out.printf("Airline: %s / Enter PNR: ",  
            kiosk.getAirlines().get(kiosk.getAirline()));  
        String pnr = scanner.nextLine();  
        kiosk.setPnr(pnr);  
        if (validate(pnr)) {  
            System.out.println("-- PNR Verified -- ");  
            kiosk.setState(kiosk.getVerified());  
        } else {  
            System.out.println("-- PNR Invalid -- ");  
            kiosk.setState(kiosk.getAvailable());  
        }  
    }  
}
```

Waiting: the boarding pass kiosk is waiting for user input

the input pnr operation allows to take pnr as input and validate it for further processing

validates input, the validation should include some data checks and more, but we have kept it simple for understanding

if the pnr is valid, switch to verified state

if the pnr is invalid, switch to the initial state: available

this state can perform only one operation, all the other operations are invalid in this state

**Figure 12.6:** State - Waiting

After successful selection of the airline in the previous state, the kiosk switches to this state. This state indicates that the kiosk is

waiting for the user to enter the PNR information of the airline ticket, to prepare the boarding pass.

This state has only one valid operation. This operation takes the PNR as input, verifies it for correctness and then makes a state transition based on the result of the verification process. If the PNR is validated to be correct, the current state of the kiosk is switched to ‘verified’. If the verification process fails because of an incorrect PNR, the kiosk is switched back to the initial state, that is, **Available** and then the user must reselect the airline.

To keep the example simple, the validation process only checks for the correctness of the entered PNR information using some preset regular expression. This is possibly the weakest form of verification. For a production-ready code, it must be ensured that the entered PNR does not only follow the format but also a valid PNR of the selected airline. Plenty of other validations can be made depending on the implementation.

[Figure 12.7](#) demonstrates the implementation of the next state of the kiosk, that is,

```

public class Verified implements State {
    private BoardingPassKiosk kiosk;
    public Verified(BoardingPassKiosk kiosk) {
        this.kiosk = kiosk;
    }
    public void pickSeats(Scanner scanner) {
        System.out.println("\n Seat Matrix ");
        displaySeatMatrix(); → displays the seating
        System.out.print("Enter Seat: "); → capacity and availability to
        String seat = scanner.next(); → the user
        kiosk.setSeat(seat); → scan user input and switch
        kiosk.setState(kiosk.getReview()); → to the review state
    }
}

```

Verified: the pnr received as input received is valid

the pick seats operation allows the user to select a seat from the available ones, displayed on the kiosk screen

this state can perform only one operation, all the other operations are invalid in this state

**Figure 12.7: State - Verified**

After successful verification of the PNR in the previous state, the kiosk switches to this state. This state indicates that the PNR has been verified and the kiosk is now waiting for the user to select a seat for travel.

This state too, has only one valid operation. This operation displays the seat matrix to the user and takes her input to reserve that seat. To keep the example simple, we are assuming that the user will not select an already occupied seat. We are also not validating the selected seat's availability, though this must be ensured in a production-ready application.

[Figure 12.8](#) demonstrates the implementation of the next state of the kiosk, that is,

```

public class Review implements State {
    private String CONFIRM_INPUT = "x";
    private BoardingPassKiosk kiosk;
    public Review(BoardingPassKiosk kiosk) {
        this.kiosk = kiosk;
    }
    public void reviewSelection(Scanner scanner) {
        System.out.println("-- Review Your Selection --");
        System.out.printf("Airline: %s\n", kiosk.getAirline());
        System.out.printf("PNR: %s\n", kiosk.getPNR());
        System.out.printf("Seat: %s\n", kiosk.getSeat());
        System.out.printf("Press 'x' to Confirm: ");
        String input = scanner.next();
        if (CONFIRM_INPUT.equals(input)) {
            kiosk.setState(kiosk.getConfirm());
        } else {
            kiosk.setState(kiosk.getAvailable());
        }
    }
}

```

expected input from the user to confirm his selection after review

Review the kiosk is ready to prepare the boarding pass, post user review

the review selection operation allows the user to review his seat selection before preparing a boarding pass

if user confirms his selection switch to confirm state to generate the boarding pass

otherwise take the user to the first screen

this state can perform only one operation, all the other operations are invalid in this state

**Figure 12.8: State - Review**

After successful selection of a seat in the previous state, the kiosk switches to this state. This state indicates that the kiosk is ready to prepare the boarding pass for the user and is waiting for her to review the selections done in the previous states.

This state too, has only one valid operation. This operation displays the prepared information for the boarding pass to the user and takes her confirmation to finally print that boarding pass. If the user confirms that the details of the boarding pass are

correct, the current state of the kiosk is switched to otherwise, the kiosk is switched back to the initial state, that is, **Available** and then the user must reselect the airline.

Figure 12.9 demonstrates the final state of the boarding pass kiosk,

```
public class Confirm implements State {  
    private BoardingPassKiosk kiosk;  
  
    public Confirm(BoardingPassKiosk kiosk) {  
        this.kiosk = kiosk;  
    }  
  
    public void printBoardingPass() {  
        System.out.println("Boarding Pass Details");  
        System.out.printf("Airline: %s\n",  
            kiosk.getAirlineName());  
        System.out.printf("PNR: %s\n", kiosk.getPNR());  
        System.out.printf("Seat: %s\n", kiosk.getSeat());  
        kiosk.setState(kiosk.getAvailable());  
    }  
}
```

Confirm: the kiosk is ready to generate the boarding pass

the print boarding pass operation prints the generated boarding pass for the user

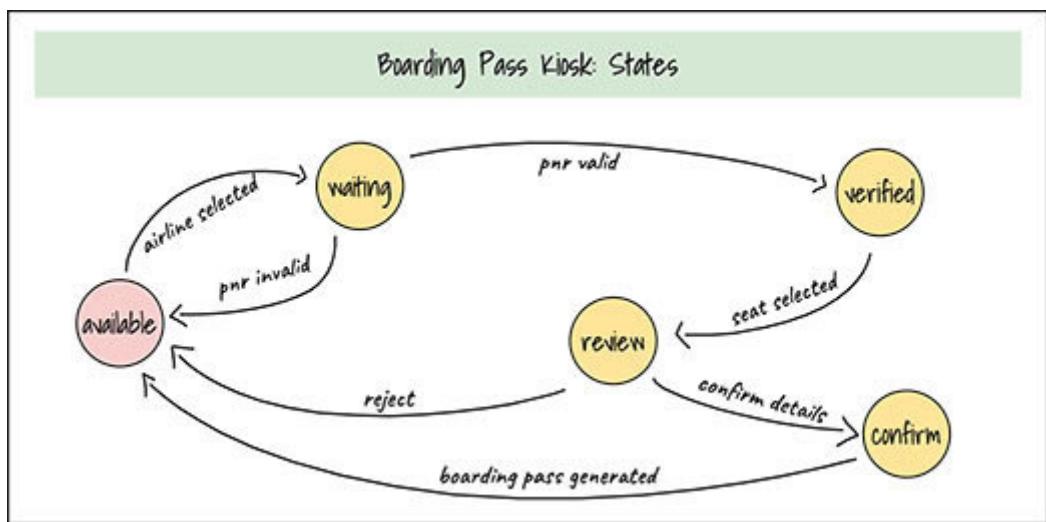
print the boarding pass and set the state to available, to make it available for reuse

this state can perform only one operation, all the other operations are invalid in this state

**Figure 12.9: State - Confirm**

This state is an auto-transition state and does not require user intervention. This state provides only one valid operation that prints the prepared boarding pass and switches the current state of the kiosk to **Available** so that it could be used by another user to generate and print a boarding pass.

Figure 12.10 demonstrates the state diagram for our boarding pass kiosk:

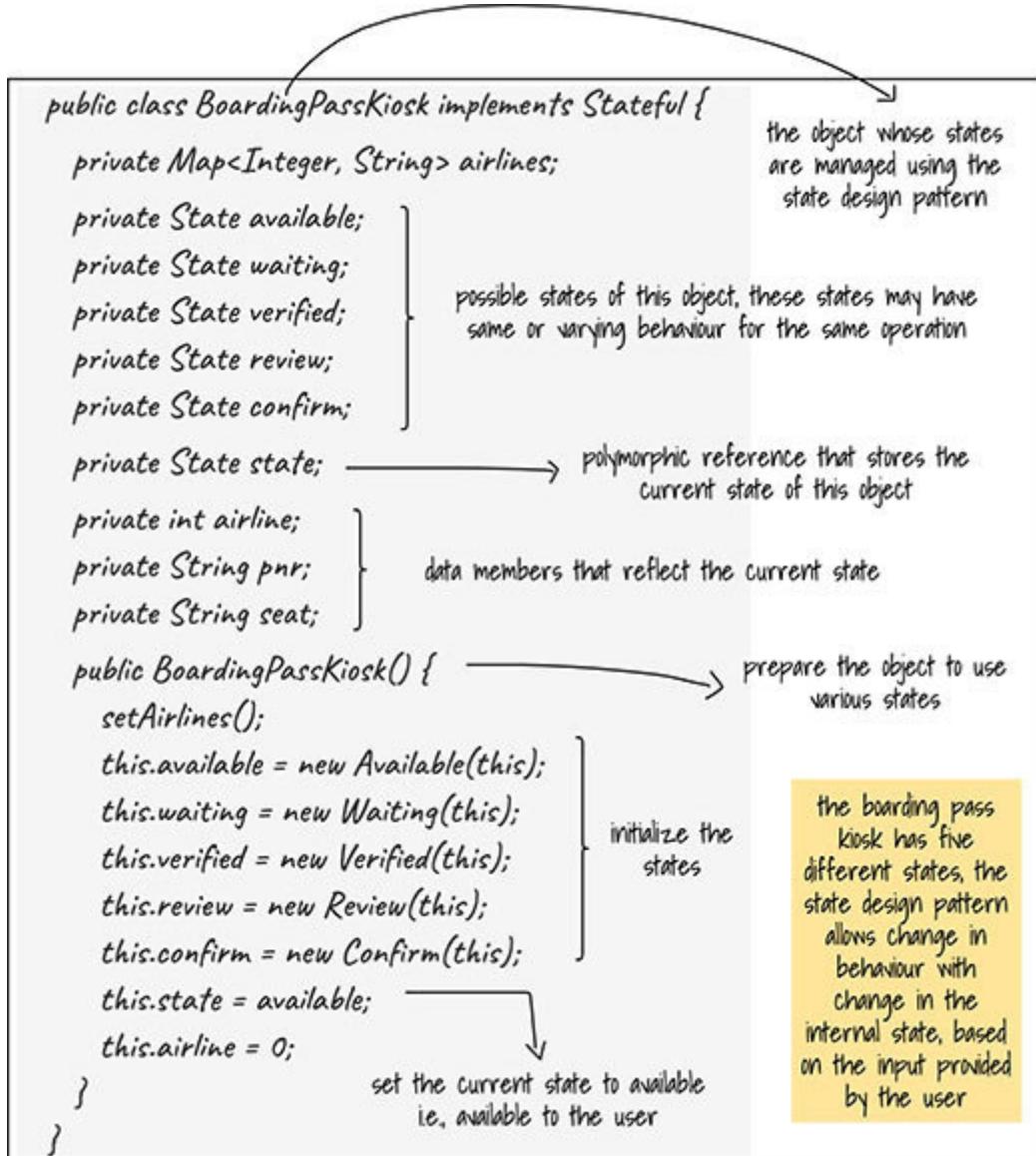


**Figure 12.10: State Diagram**

### The stateful object

A stateful object could be any object that has different states and that exhibits distinct, state-specific behaviors for the operations that can be performed on it. It keeps a reference to its current state and delegates all the incoming requests to it.

[Figure 12.11](#) demonstrates the data members and the constructor of the boarding pass kiosk:



**Figure 12.11: The Stateful Object I**

It stores the value of its current state and has a few data members that reflect it. It also keeps references to all the possible states of itself and provides getter methods to access them. They are used within the concrete state implementations to switch to the next appropriate state. Keeping these references in the kiosk object makes it easier for the state implementations to drive the transition while being unaware of other states existence. This also keeps these implementations loosely coupled.

Notice that the constructor is responsible for instantiating all the states and passes a reference of itself to them, to bind the state instances and kiosk object together. After calling the constructor, there is no need to initialize any of the object's state explicitly; it has already been taken care of.

[Figure 12.12](#) demonstrates the operations that can be performed on the kiosk object:

```
public class BoardingPassKiosk implements Stateful {
    public State getState() {
        return state;
    }
    public void setState(State state) {
        this.state = state;
    }
    public void selectAirline(Scanner scanner) {
        this.state.selectAirline(scanner);
    }
    public void inputPNR(Scanner scanner) {
        this.state.inputPNR(scanner);
    }
    public void pickSeats(Scanner scanner) {
        this.state.pickSeats(scanner);
    }
    public void reviewSelection(Scanner scanner) {
        this.state.reviewSelection(scanner);
    }
    public void printBoardingPass() {
        this.state.printBoardingPass();
    }
}
```

The diagram shows the `BoardingPassKiosk` class with several methods. Annotations explain the behavior of each:

- `getState()` and `setState(State state)`: A callout points to these methods with the text "get and set the current state of the object".
- `selectAirline(Scanner scanner)`, `inputPNR(Scanner scanner)`, `pickSeats(Scanner scanner)`, and `reviewSelection(Scanner scanner)`: A callout points to these methods with the text "the request to perform an operation on this object is delegated to the object that represents the current state of the object".
- `printBoardingPass()`: A callout points to this method with the text "for all the operations performed on this object, its behaviour is determined by the current state of the object".
- A yellow box contains the text "every state implementation must ensure to change the current state of the object after processing the input and/or its current state".
- A red box contains the text "polymorphism takes care of calling the correct behaviour based on state implementation".
- A pink box contains the text "an objects behaviour is encapsulated inside the different states it exhibits".

***Figure 12.12: The Stateful Object II***

It stores the value of its current state and has a few data members that reflect it. Notice that the operation definitions delegate the responsibility of performing that operation on the current state. This is much more readable and cleaner than having multiple conditional statements decide the current behavior of this object.

## Testing the state design pattern

We have written a test class to verify that the pattern we worked on works. Let's execute the same.

In [Figure](#) we first instantiate our stateful object, that is, an instance of boarding pass kiosk and later call different operations on it in the order, that with the help of correct input values, guarantees the generation of a boarding pass kiosk:

```
public class TestState {  
    public static void main(String[] args) {  
        BoardingPassKiosk kiosk = new BoardingPassKiosk();  
        try (Scanner scanner = new Scanner(System.in)) {  
            kiosk.selectAirline(scanner);  
            kiosk.inputPNR(scanner);  
            kiosk.pickSeats(scanner);  
            kiosk.reviewSelection(scanner);  
            kiosk.printBoardingPass();  
        }  
    }  
}
```

create an object of the kiosk

the calls made to the kiosk object are delegated to the implementation that represents the current state of the object

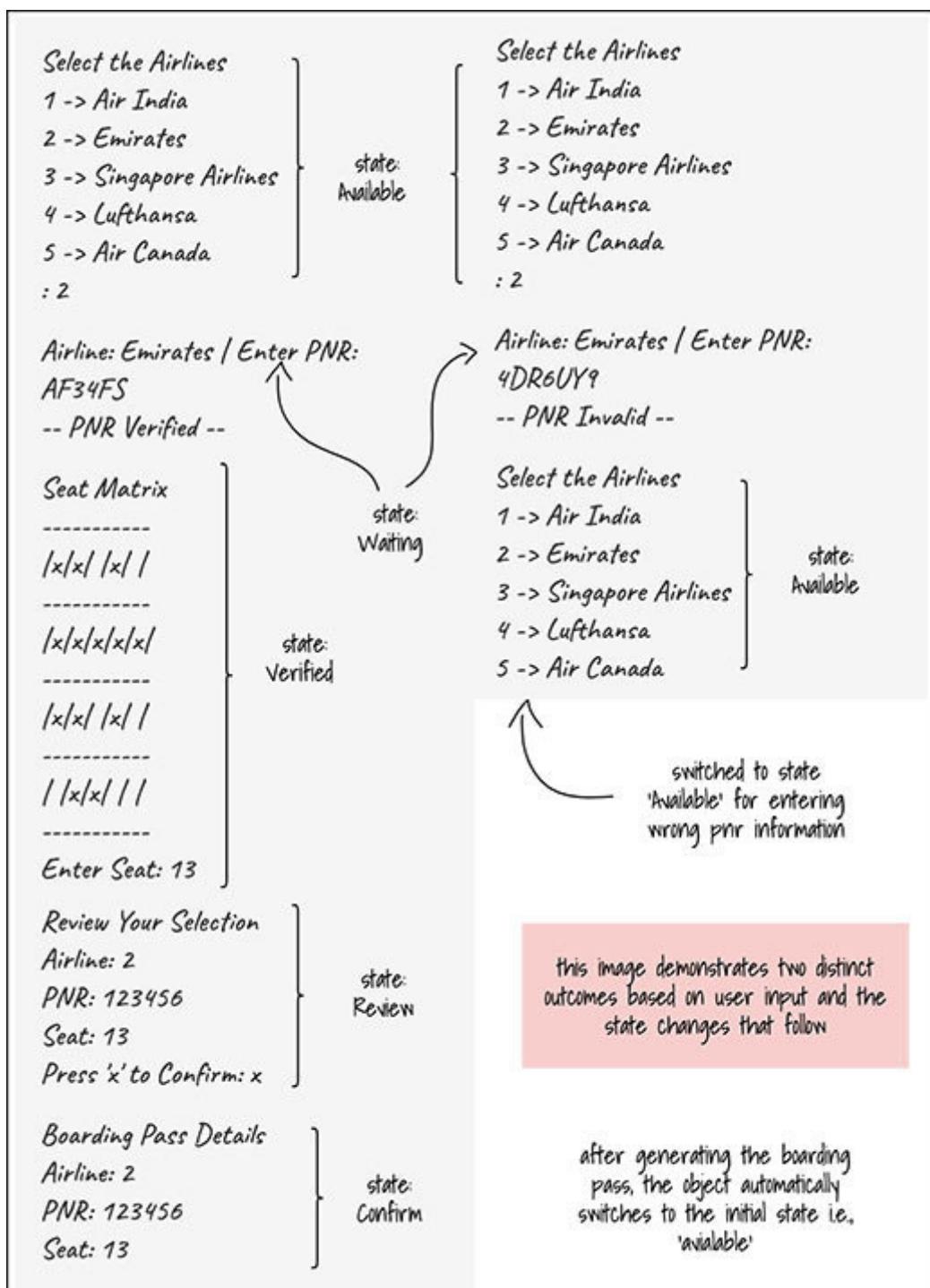
the test class performs all the operations on the kiosk object in the order a user should perform them to generate a boarding pass

**Figure 12.13:** Test State Design Pattern

[Figure 12.14](#) demonstrates two different outcomes based on user inputs and the state changes that follow. The left section of the figure demonstrates that by following correct steps and entering

the correct input the user was able to generate a boarding pass from the kiosk.

The right section demonstrates the outcome of entering wrong PNR information; the object's state switched back to



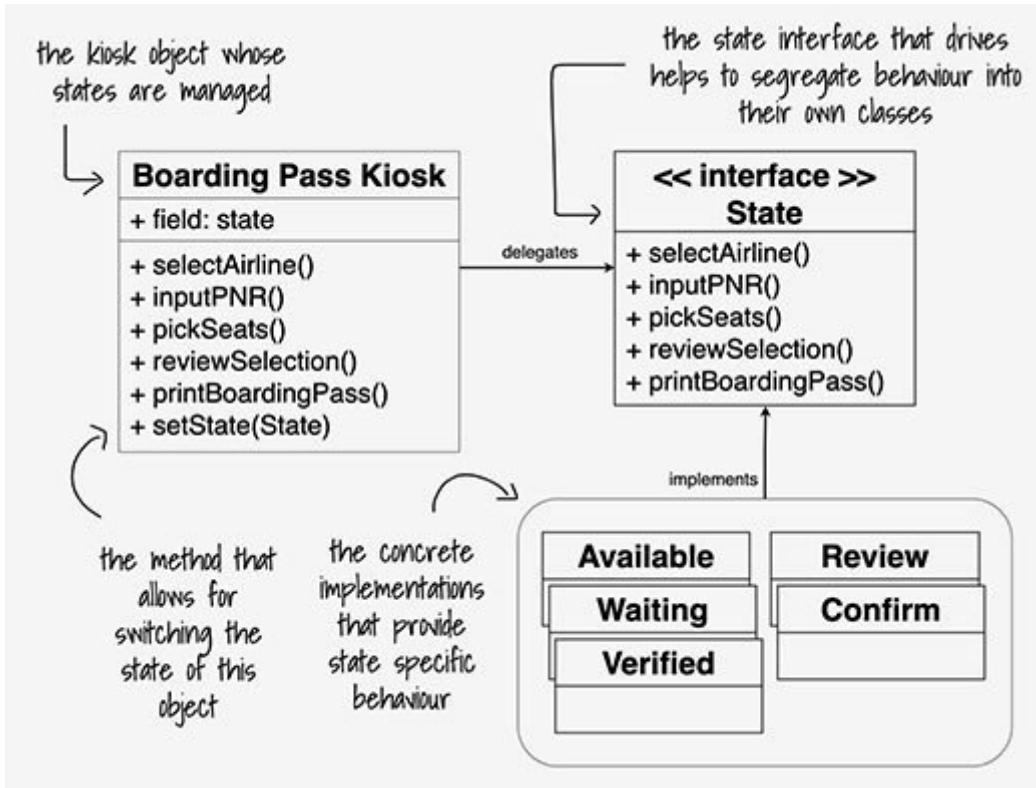
**Figure 12.14:** Test State Design Pattern: Outcome

## State design pattern - defined

The state design pattern allows for managing the state-specific behaviors of an object by separating the behavior from the object definition. It promotes separation of concern and the single responsibility principle. It suggests defining state-specific behaviors in their own classes while keeping those definitions loosely coupled to each other.

The state design pattern can be used to manage multiple states and switch between them effortlessly, in runtime.

Figure 12.15 demonstrates a high-level view of the state design pattern:



**Figure 12.15: State Design Pattern - Defined**

It presents a consolidated view of the relationship between the stateful object and the concrete state implementations.

## Benefits and drawbacks

It is time to make note of some of the benefits and drawbacks of the state design pattern. Let's start with the benefits first:

Makes it easier to introduce new states that support additional behavior.

Minimizes the complexity of the stateful object by separating the behavior from the object definition and reducing the use of condition statements.

Uses composition and promotes open close principle that reduces the need for regression testing.

The drawbacks of the state method design pattern are as follows:

A lot of code must be written and managed for state design pattern

Increased number of states can increase memory footprint if the state objects are initialized in abundance

## Usage

Some of the scenarios relevant for the state design pattern are as follows:

When an object's behavior is dependent on its internal state, the state design pattern can help to segregate the behavior into specific classes

When an object's definition is polluted with conditional statements, the state design pattern can help to reduce the clutter and ease code maintenance

## Conclusion

State design pattern is an outstanding mechanism to reduce avoidable conditional checks and move the state-specific behavior into its own class. The behavior of the object can easily be changed by switching states, and new behaviors can be introduced without effecting existing state or behavior.

In the next chapter, we will learn about the command design pattern and how it can be utilized to encapsulate information needed to perform an action at a later stage.

## Questions

In this section, we will sharpen our brain by indulging in some fun brain activities. Let's get started.

## Pick the odd one out

composition	single responsibility	multiple states
extension	separation of concern	conditional execution
loosely coupled	interface segregation	distinct behaviour
polymorphic	open close	transition

*Figure 12.16: Pick the odd one out*

### Find the odd one out

State design pattern

Allows managing state-specific behavior

Supports multiple states

Requires states to be directly aware of each other

Provides for switching object state in runtime

State design pattern

Depends on inheritance and polymorphism

Works on the principle of interface segregation

Segregates object behavior into state-specific classes

Is a behavioral design pattern

State transition using the state design pattern

Can only be done using the base state implementation

Is a simple and clean approach

Can only be done in runtime

Requires the stateful object to expose switch mechanism

Select two correct answers

State design pattern

Cannot work without polymorphism

Increases object complexity

Promotes separation of concern

Increases cohesion

The stateful object

Allows for switching states in runtime

Encapsulate its behavior

Delegates responsibility to current state

Inherits a standard interface

What goes together?

Encapsulation, concrete states, object behavior

Structural, state design pattern, loose coupling

State design pattern, behavioral, loose coupling

Inheritance, extension, single state implementation

## Answers

### Pick the odd one out

extension, interface segregation, conditional execution

### Find the odd one out

Requires states to be directly aware of each other

Works on the principle of interface segregation

Requires the stateful object to expose switch mechanism

Select two correct answers

Cannot work without polymorphism, promotes separation of concern

Delegates responsibility to current state, inherits a standard interface

[encapsulation, concrete states, object behavior] and [state design pattern, behavioral, loose coupling]

## CHAPTER 13

### Executing Commands

---

You can find the editable code files for this chapter on the book's GitHub link.

<https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers/tree/main/Chapter%2013>

---

## Structure

Topics that make this journey worthwhile:

Introduction

Command design pattern

Implementing command pattern

Command design pattern - defined

Benefits and drawbacks

Usage

## Objective

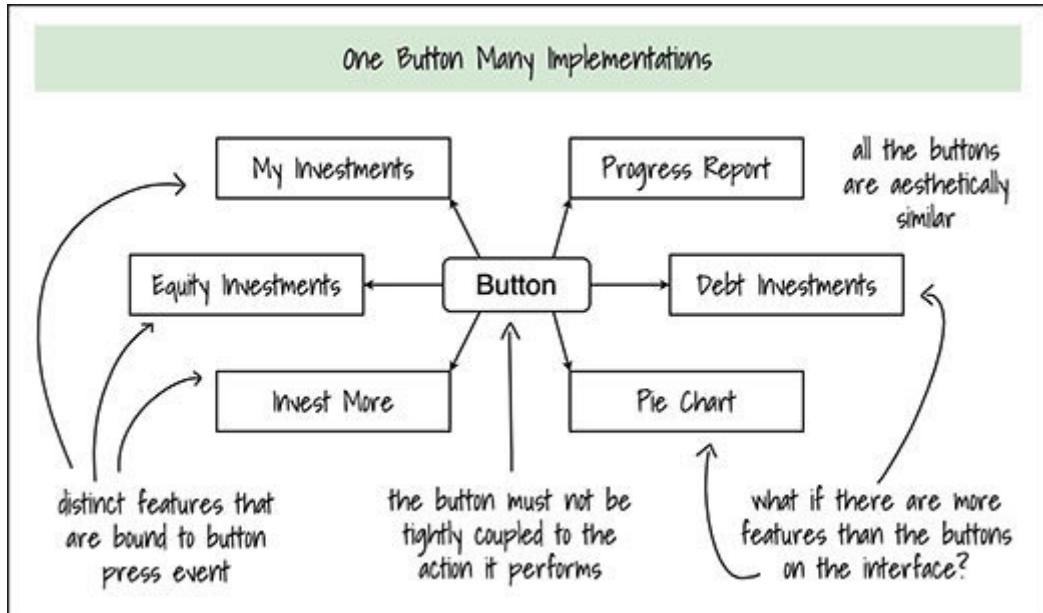
Our objective is to learn about the command design pattern that allows to create standalone objects for processing specific actions as and when required by the user. We will explore the benefits of decoupling the class that performs some operation from the one that invokes it.

We will walk you through the implementation details of the command design pattern, the benefits it provides, and a few drawbacks along with some of the places where this design pattern can be used.

## Introduction

Software applications are increasingly providing a host of new options to their users. While it is good to have new features in the application that add to the product catalog, their implementation isn't always straightforward. Imagine a user interface that allows the user to select from multiple options. The user interface is most likely to have buttons or menu items to provide access to these features, where each feature has a button dedicated to it.

One way to support all these features is to have multiple implementations, each dedicated to performing one single task, and a new button on the user interface for each of these features. Moreover, every API that is accessed through these buttons may have a different interface requiring specific integrations. When the buttons shouldn't really be aware of what they do, they remain coupled to their directives. [Figure 13.1](#) demonstrates a link between a button and the various services that can be triggered as a result of an event performed on that button:



**Figure 13.1: Multiple Feature Implementations**

It demonstrates that while all the buttons are aesthetically same, bounding them to specific tasks will take away the advantage of reusing them.

The interface and program logic are two different aspects of an application. An interface should not be tightly coupled to the functionality it serves; otherwise, it would be difficult to reuse any of it. Imagine duplicating the button code every time there is a new feature requirement. The code will become highly redundant and the benefit of reusability will be lost.

What we need is a way to decouple the button from the action it performs. The button does not need to know about the action that is performed and who performs it. It should only be responsible for providing a way to access some functionality whenever it is pressed, or some other event is performed on it.

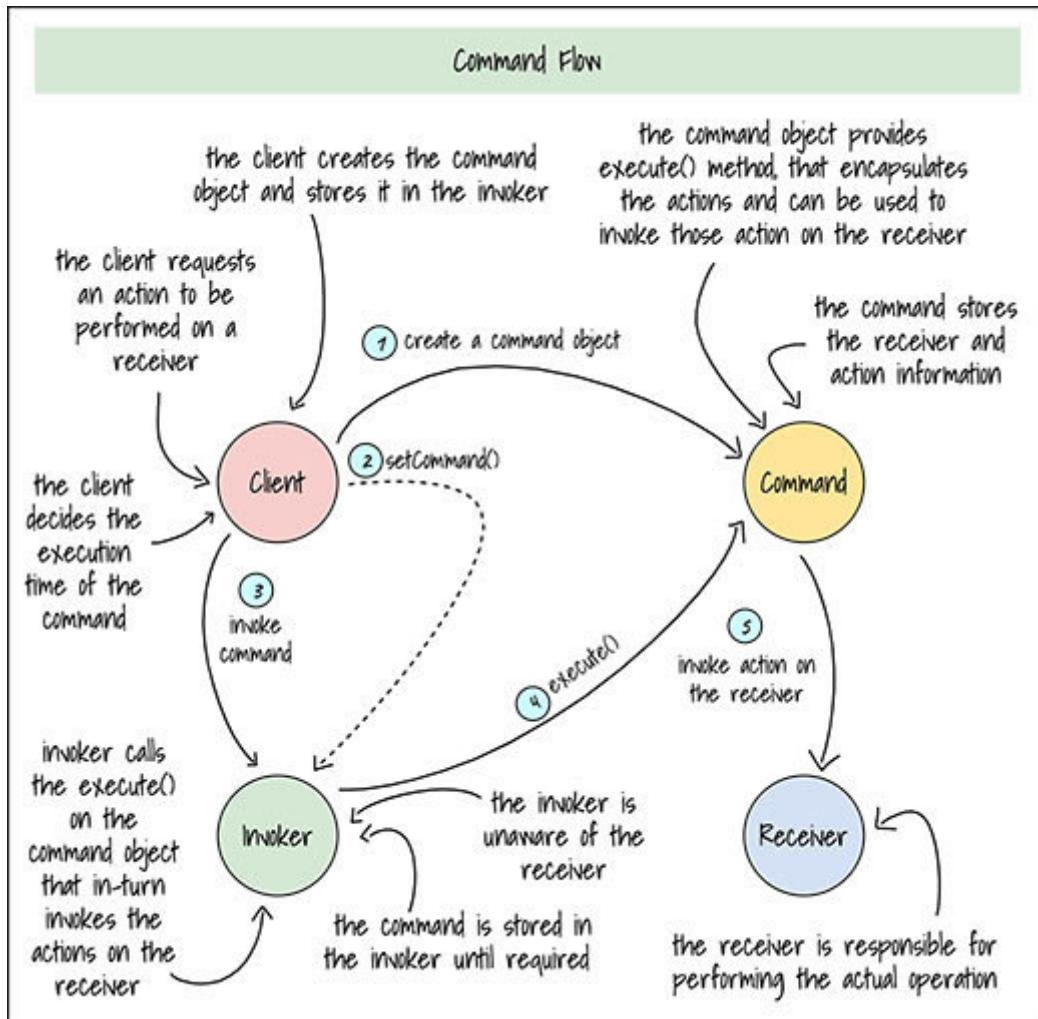
There should be a total separation of concern between these two different layers of an application.

In this chapter, we will discuss the command design pattern and how it can be utilized to encapsulate a request in the form of an object to perform it later or multiple times. We will learn about separating the business logic from the interface and how it helps to reuse each of them.

## Command design pattern

The command design pattern is a behavioral design pattern that allows to encapsulate a request as an object, parameterize other objects with different requests, delay or queue requests, and support undoable operations. It decouples the request invoker from the request operator and allows for the formation of a common interface that can be used across applications to perform actions on the operator.

Using the command design pattern, a class can delegate a request to a command object instead of implementing it on its own. A hard-wired request on the other hand couples the implementing class to a specific request and its operator, disallowing reusability and effortless integration. [Figure 13.2](#) demonstrates the flow of operations in the command design pattern:

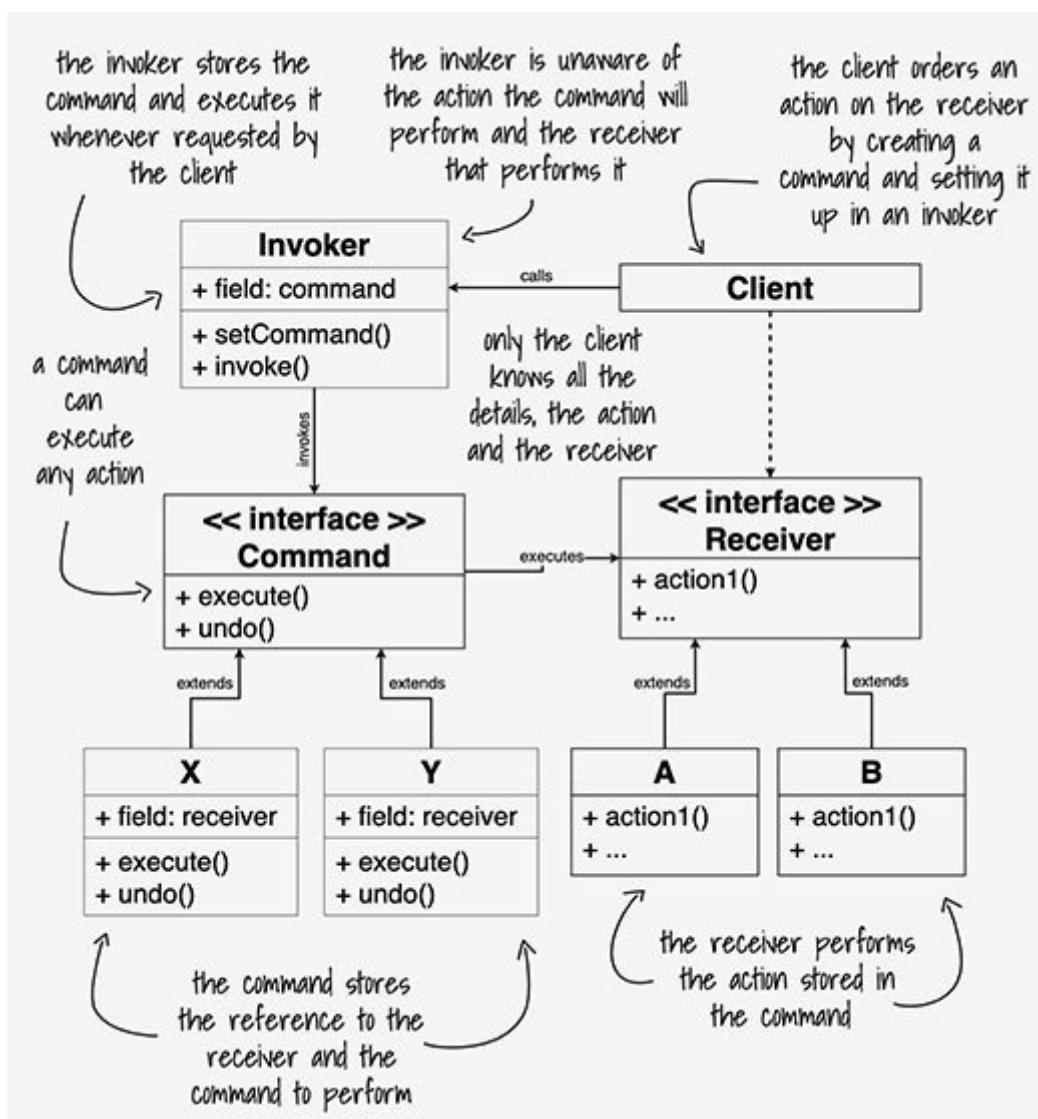


**Figure 13.2: Command Flow**

The client instantiates a command object, stores it in an invoker and executes it when required to request an action on the receiver.

The command object stores a reference to the receiver and the action that must be performed on that receiver. The receiver performs the action when the command is executed by the invoker on the direction of the client. By using the command object, both the invoker and receiver remain unaware of each

other's existence. The invoker can execute different types of commands without the need for separate integrations and the receiver can perform the actions mentioned in the command without any knowledge of its invoker or client. [Figure 13.3](#) demonstrates the architecture of the command design pattern:



**Figure 13.3: The Command Design Pattern**

The **Invoker** provides a setter method to set the command that the invoker instance should execute when the invoke method is called on it. The command interface provides two methods - **execute** and **undo**. The first is used to perform some action(s) on the

receiver and the second can be used to reverse those action(s). The receiver interface provides the set of operations that can be performed when the command is executed.

## Implementing command pattern

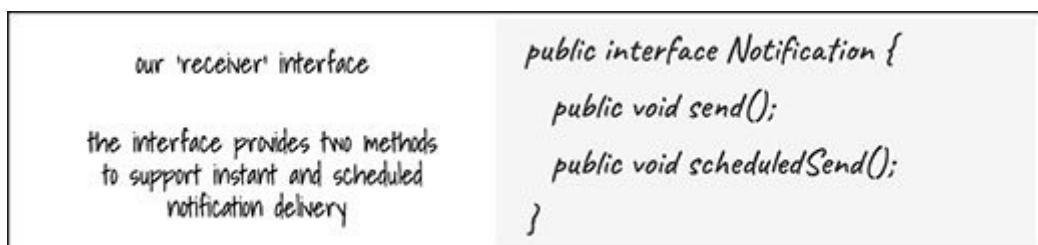
The example that we will walk you through in this chapter is about a notification service that can be used to send notifications to a user through e-mail or messages. The example sheds light on the use of command pattern to generalize architecture and streamline how various layers interact with each other.

We will learn about implementing the command design pattern and how it can be used to perform actions on a receiver by keeping our design loosely coupled. Let us start with the **Receiver** interface that lists the operations that can be performed, then shift to the **Command** interface and the **Invoker** and finally focus on the **Client** that creates and invokes a command.

## The receiver

Receivers in the command design pattern are responsible for performing the actual operations. They are also known as To which receiver does a command interact with is decided by the client. A receiver can be implemented to perform multiple actions and commands can execute one or many of those. Different commands can execute different actions on the same receiver.

[Figure 13.4](#) demonstrates a **Receiver** interface that provides two methods to support instant and scheduled delivery of notifications to the users:



**Figure 13.4: Receiver: Interface Notification**

The receivers do not necessarily need to be implemented in the form of interfaces; a command could store a reference to a class as well, though it is advisable to interact with interfaces and not directly to classes to keep the interaction loosely coupled. An interface allows interaction with multiple implementations of itself via polymorphism, something that cannot be done if the interaction is done directly with a class. [Figure 13.5](#) demonstrates one of the concrete 'receiver' implementations:

```

    public class EmailNotification implements Notification {
        private Email email;
        public EmailNotification(Email email) {
            this.email = email;
        }
        public void send() {
            print("Preparing Email");
            print("Connecting to Email Server");
            print("Sending Email with subject " +
                email.getSubject() + " to " + email.getTo());
        }
        public void scheduledSend() {
            print("Preparing Email");
            print("Connecting to Email Server");
            print("Scheduling Email with subject " +
                email.getSubject() + " to " + email.getTo());
            print("Email Scheduled");
        }
    }

```

a concrete implementation of our 'receiver' interface

the send method instantly sends an email to the designated recipients

the scheduled send method can be used to schedule delivery of an email for a later time

to keep our example simple, interaction with actual delivery systems is kept out of scope

this is where the information about the email is kept, the likes of the recipient, email subject etc.

**Figure 13.5:** Receiver: E-mail Notification

It stores the e-mail-related information in another object and overrides the two interface methods to support instant and scheduled delivery of notifications in the form of e-mails. To keep our example simple and focus on the overall design, interaction with actual notification delivery systems is left to the user.

[Figure 13.6](#) demonstrates another concrete **Receiver** implementation, that delivers notifications in the form of short messages over a cellular network:

```

    ↗ public class SmsNotification implements Notification {
    ↗     private ShortMessage message; ↙
    ↗     public SmsNotification(ShortMessage message) {
    ↗         this.message = message;
    ↗     }
    ↗     public void send() {
    ↗         System.out.println("Preparing Message");
    ↗         System.out.println("Connecting to SMS Server");
    ↗         System.out.println("Sending Message to " +
    ↗             message.getTo());
    ↗     }
    ↗     public void scheduledSend() {
    ↗         System.out.println("Preparing Message");
    ↗         System.out.println("Connecting to SMS Server");
    ↗         System.out.println("Sending Message to " +
    ↗             message.getTo());
    ↗         System.out.println("SMS Scheduled");
    ↗     }
    ↗ }
  
```

another concrete implementation of our 'receiver' interface

the send method instantly sends a message to the designated recipients over cellular network

the scheduled send method can be used to schedule delivery of a message

to keep our example simple, interaction with actual delivery systems is kept out of scope

this is where the information about the message is kept, the likes of the recipient number

**Figure 13.6:** Receiver: SMS Notification

## The command

A command is a representation of a stand-alone request. It stores all the information necessary to execute a command, that is, the action that is to be performed and the receiver that should perform it and some other information, if required. [Figure 13.7](#) demonstrates the command interface that is at the center of the command design pattern:

```
public interface Command {  
    public void execute();  
}
```

the command interface that provides the important execute method and is largely responsible for the loosely coupled, polymorphic architecture of the command design pattern

**Figure 13.7: The Command Interface**

Execution of a command has its own lifespan, independent of the original request. Commands can be grouped together to be processed in a batch or could be scheduled for delayed operation. They can be joined in a sequence to perform operations that depend on each other or can be stacked to perform undo operations. [Figure 13.8](#) demonstrates a concrete implementation of our command interface:

```

public class QuickNotify implements Command {
    private Notification nService;
    public QuickNotify(Notification nService) {
        this.nService = nService;
    }
    public void execute() {
        nService.send();
    }
}

```

reference to the receiver, that performs the actual operation

client passes an instantiated receiver object to the command

when the command is executed it delegates the request to the receiver

this implementation of our command interface executes the instant delivery of notification.

**Figure 13.8: Command - Quick Notify**

The implementation has a reference to a receiver, that delegates the request for performing an action when the command is executed. [Figure 13.9](#) demonstrates another concrete implementation of our **Command** interface:

```

public class ScheduledNotify implements Command {
    private Notification nService;
    public ScheduledNotify(Notification nService) {
        this.nService = nService;
    }
    public void execute() {
        nService.scheduledSend();
    }
}

```

interaction with an interface allows the client to select among the type of notifications currently available

it also makes it easier to interact with future notification types without making any change to the command implementation

this implementation of our command interface schedules delivery of a notification.

**Figure 13.9: Command: Scheduled Notify**

This implementation schedules delivery of a notification, be it e-mail notification or message over cellular network.

Command objects allow for the construction of a common interface that can be utilized to delegate, sequence, or execute operations. The command can be executed at any time as required by the client. The same command can be executed multiple times by the client once it is stored in an invoker. A command object can be constructed anonymously to perform actions that are supposed to be performed only once. It is this object that generalizes the solution and binds the receiver and the invoker without them being aware of each other.

## The invoker

The invoker invokes the command by calling the execute method. It can be used to log access-related information, perform audit trail or control execution of the command. The invoker can execute any command at any time chosen by the client. The command can be executed at-once, setup for a delayed execution, or may never be executed. The command set in the invoker can be changed at any time without any behavioral change in the invoker. [Figure 13.10](#) demonstrates an example of the invoker class that has a reference to the command object and provides a method to execute it:

```
public class Notifier { ←  
    private Command command;  
  
    public void setCommand(Command command) {  
        this.command = command;  
    } } ← the setCommand() is  
          used to store a  
          command in an  
          invoker instance  
  
    public void invoke() {  
        command.execute();  
    } } ← the invoke() executes the  
          stored command at a time of  
          clients choosing
```

the invoker class, that is used to store a command and execute it when required

the stored command can be executed instantaneously, setup for delayed execution or maybe never executed

[Figure 13.10: Command Invoker](#)

### The client

The client is responsible for creation of the command object by selecting a receiver and an action that the receiver can perform.

[Figure 13.11](#) demonstrates the implementation of a client process in the form a test class:

```

public class TestCommand {
    public static void main(String[] args) {
        String emailTo = "office@creativehumans.com";
        String emailSubject = "Order Confirmation";
        String emailContent = "We are delighted to inform you that your order  
has been confirmed. Thank you for shopping with us.";
        String smsTo = "9000090000";
        String smsContent = "Your order has been confirmed. Thank you for  
shopping with us.";
        Email email = new Email(emailTo, emailSubject, emailContent);
        ShortMessage shortMessage = new ShortMessage(smsTo, smsContent);
        EmailNotification emailService = new EmailNotification(email);
        SmsNotification smsService = new SmsNotification(shortMessage);
        Command quickNotify = new QuickNotify(smsService);
        Command scheduledNotify1 = new ScheduledNotify(emailService);
        Command scheduledNotify2 = new ScheduledNotify(smsService);
        Notifier notifier1 = new Notifier(); ←
        notifier1.setCommand(quickNotify); ←
        notifier1.invoke(); ←
        Notifier notifier2 = new Notifier(); ←
        notifier2.setCommand(scheduledNotify1); ←
        notifier2.invoke(); ←
        notifier2.setCommand(scheduledNotify2); ←
        notifier2.invoke(); ←
    }
}

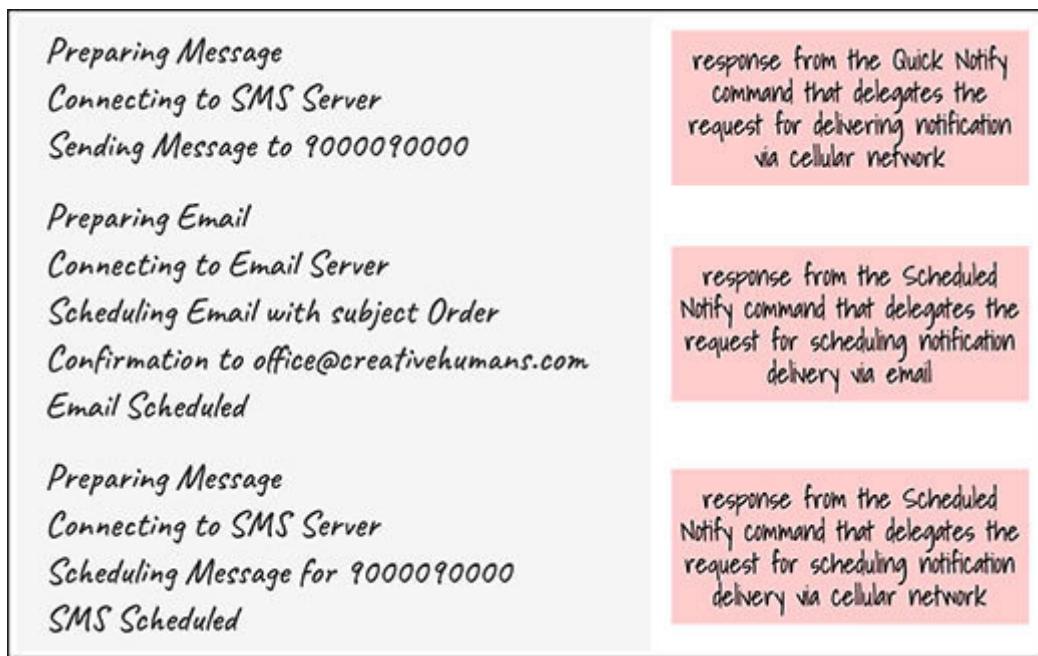
```

our client implementation

**Figure 13.11:** The Client

The client constructs the command objects and stores the necessary information in it, then passes this command to a newly constructed invoker object before finally making a call to that invoker object to execute the command. [Figure 13.12](#) demonstrates

the result of executing the commands using the command design pattern:



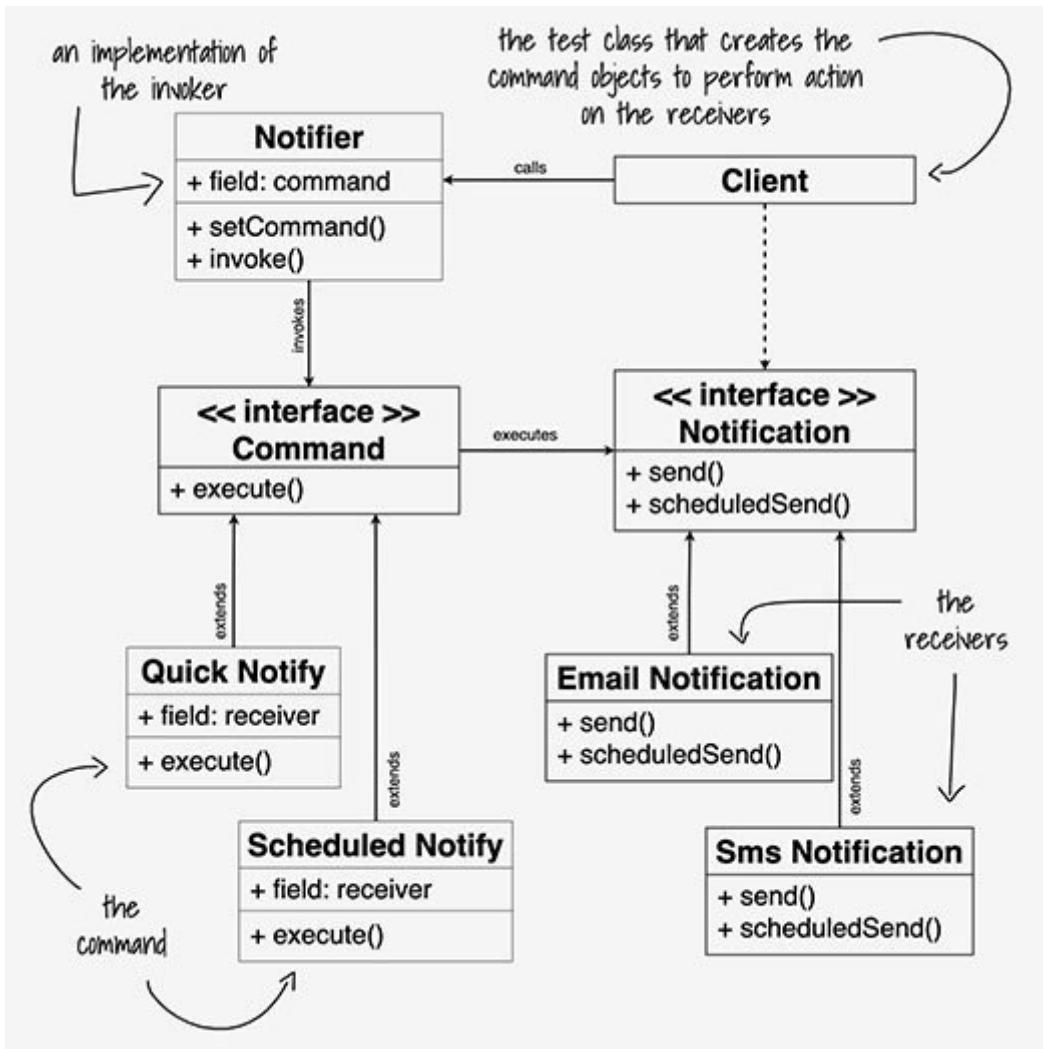
**Figure 13.12: Testing the Command Pattern**

## Command design pattern - defined

The command design pattern allows for constructing standalone commands that can be triggered as and when required. These commands contain information about the action to perform and the receiver that executes it. The command pattern works with three main objects - the receiver, command, and invoker. The command design pattern promotes separation of concern of and a loosely coupled architecture.

The command pattern can also expose another method called **undo**. The **undo** method can be called from an invoker to reverse the previous operation performed on some command. The undo operation is very useful when we are dealing with actions that are reversible, for example, maintaining binary switches to control information flow or traffic on a system. Together, the **execute** and **undo** methods can be used to perform operations that rely on a sequence as in a stack, for example, controlling log behavior by increasing and decreasing the severity level of the information allowed for logging from within the execute and undo methods.

[Figure 13.13](#) demonstrates a high-level view of the command design pattern:



**Figure 13.13: Command Design Pattern: Defined**

It presents the classes that we have used in our representation of the command design pattern. In our example, we have concrete implementations of the ‘notification’ interface that resembles two command objects that call the methods exposed by the **notification** interface, and an invoker that executes the command object.

## Benefits and drawbacks

It is time to make a note of some of the benefits and drawbacks of the command design pattern. Let's start with the benefits first before we look at its drawbacks:

It allows to encapsulate the request in an object.

It promotes separation of concern by utilizing a layered approach of software development where no two layers are tightly coupled with each other.

It is easy to introduce new commands without changing the existing logic of application.

It can also be used to define reversal operations.

The drawbacks of the command method design pattern are:

Multiple command implementations can become difficult to manage, especially when they are not reusable.

A lot of code must be written and managed for command design pattern.

The execute method in the command interface doesn't allow returning a value in its original definition.

## Usage

Some of the scenarios relevant for the command design pattern are:

When application restructuring is required to reduce coupling between objects without any major changes in the underlying logic.

When the dynamic behavior of the application allows it to perform distinct actions using the same set of operations.

## Conclusion

Command design pattern presents a reliable approach for defining commands in runtime; commands that can perform all sorts of actions while reducing object coupling and streamlining the integration mechanism. The three types of classes that are necessary for proper functioning of the command design pattern are command, receiver, and invoker. A command stores the information about the action and its executor, and the invoker executes it.

In the next chapter, we will learn to use design patterns together and how it can benefit the overall design of an application. We will also explore further on their application and how existing frameworks can be utilized to improve our implementation of these designs.

## Questions

In this section, we will sharpen our brain by indulging in some fun brain activities. Let's get started.

## Pick the odd one out

invoker	separation of concern	series of operations
command	loose coupling	redo operation
executor	layered architecture	undo operation
receiver	predefined commands	operation stack

*Figure 13.14: Pick the odd one out*

### Find the odd one out

Command design pattern:

Allows creation of standalone commands

Is a behavioral design pattern

Reduces the need for object interaction

Reduces coupling between objects

Types of classes used in command design pattern are:

Exporter

Caller

Invoker

Converter

Operations performed via command design pattern are:

Queueing of operations in a predefined order

Reversal of an operation using undo

Callback operations

Execution of same operation multiple times

Select two correct options

State design pattern:

Employs polymorphism

Increases application complexity

Promotes separation of concern

Increases coupling

The command object provides these methods:

Execute

Perform

Redo

Undo

What goes together:

Encapsulation, command object, action

Structural, command design pattern, tight coupling

Command design pattern, behavioral, loose coupling

Invoker, commander, receiver

## Answers

### Pick the odd one out

Executor, predefined commands, operation stack

## Find the odd one out

Reduces the need for object interaction

Invoker

Callback operations

Select two correct options

Employs polymorphism, promotes separation of concern

Execute, undo

[encapsulation, command object, action] and [command design pattern, behavioral, loose coupling]

## CHAPTER 14

### Beyond Design Patterns

## Structure

Topics that make this journey worthwhile:

Introduction

Quick recap

Patterns – a summary

Design principles

Anti-patterns

## Objective

Our objective is to summarize the knowledge we gathered in the past chapters, understand some of the design principles that are an integral part of software design and learn about some of the anti-patterns that should be avoided while designing software applications.

## Introduction

In the previous chapters, we discussed multiple design patterns, mastered their implementation and learned about their advantages and disadvantages. In this chapter, we will follow up and expand our understanding of a good system design by exploring some of the design principles that are derived from the previous architecture and design experiences, which form the basis of many of the design patterns. We will explore some of the anti-patterns that must be avoided when finalizing a system design, as they are usually ineffective and risky of being counterproductive.

To conclude our journey towards improving the system design and providing effective solutions, we would like to reiterate that the design patterns that we explored in this book aren't the only ones and that there are multiple others that are equally important to be explored and learned.

## Quick recap

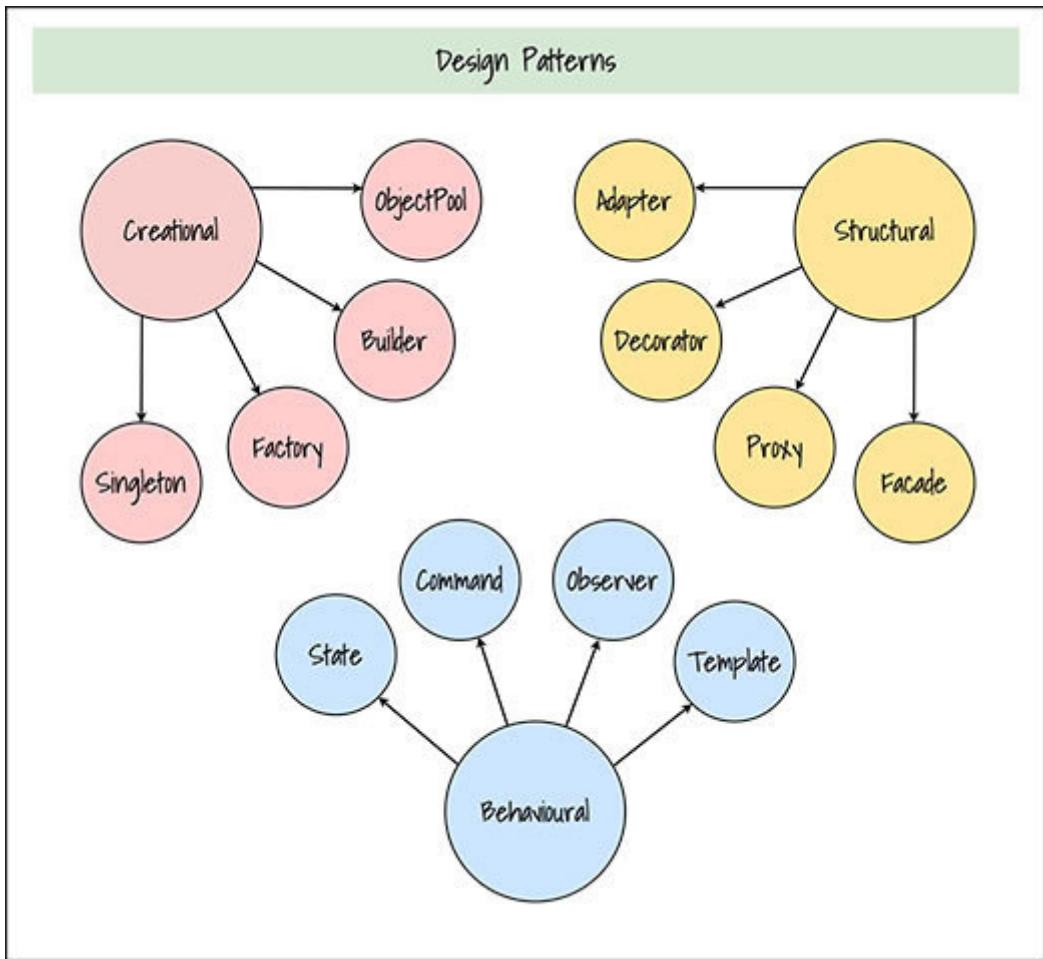
We learned about three different categories of software design patterns, that is, creational, structural, and behavioral. We also explored some of the design patterns that fall under each of these categories and learned to implement them with close to actual application examples. Here's a quick recap:

Provides mechanism for object creation in an efficient and controlled manner while improving code reusability. Promotes the idea of hiding object composition from the outside world.

Relates to the idea of forming larger structures by assembling various objects or their classes together and provides new functionalities. They do not modify existing classes or objects, but instead add new features on top of them.

Defines the relationship between objects and how they interact with each other. Aims to reduce the communication complexity by using abstraction to decouple the classes and promotes interaction with interfaces.

[Figure 14.1](#) demonstrates the design patterns that we explored in the previous chapters along with the categories they belong to:



**Figure 14.1: Design Patterns**

The design patterns serve the objective of streamlining and simplifying the architecture of a system. They make use of the four pillars of object-oriented programming, that is, abstraction, encapsulation, polymorphism, and inheritance and provide solutions for recurring and general software design problems.

## Patterns - a summary

Here's a quick and precise summary of the design patterns we explored in this book. We have listed down the design patterns and the objective they serve. [Figure 14.2](#) describes the design patterns we learned throughout the book, in simple one-liners:

Design Pattern	Description
State	Encapsulates statewise behaviour and switches between states using object delegation
Singleton	Ensures creation of only one object
Decorator	Wraps an object to provide new behaviour
Factory	Lets the subclasses decide which concrete classes to create
Builder	Separates the construction of an object from its representation
Command	Encapsulates a request as an object
Template Method	Encapsulates an algorithm and lets subclasses decide the steps of that algorithm
Proxy	Wraps an object to control access
Object Pool	Pre initializes ready to use objects
Observer	Notifies objects when state changes
Facade	Simplifies object interface
Adapter	Provides a different interface to a wrapped object

***Figure 14.2: Patterns - Summary***

It points out the intent and usefulness of these design patterns in system design that is both imperative and purposeful.

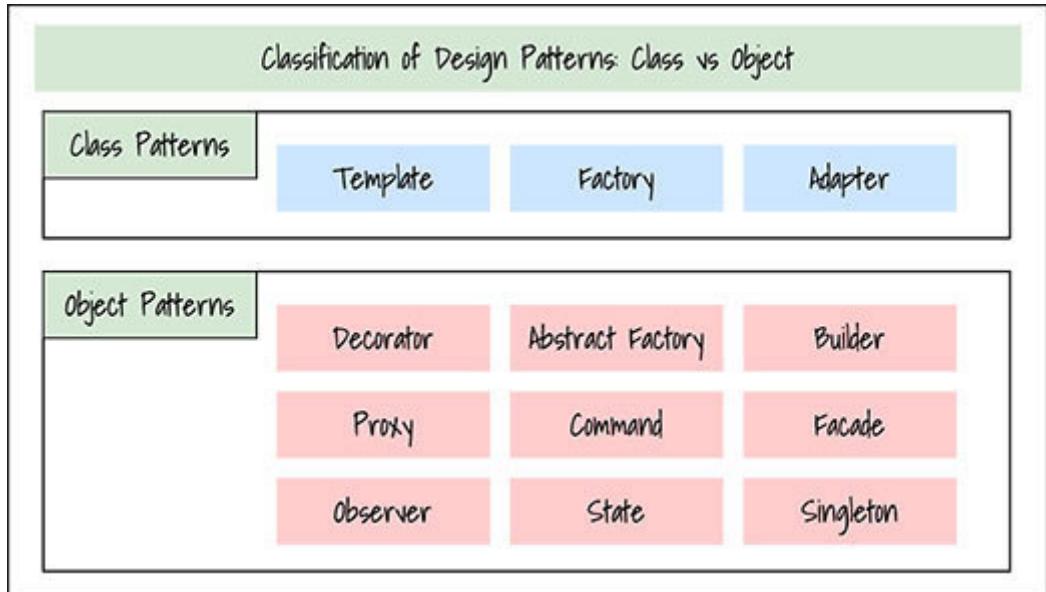
## Classification - class versus object patterns

The design patterns can be further classified by another attribute - whether the pattern deals with classes or objects. The classification is best known as class patterns and object patterns.

Class patterns describe the relationship between classes via inheritance. These relationships are established at compile time.

Object patterns describe the relationship between objects, primarily using composition. These relationships are established at runtime and are more flexible and dynamic compared to those established using the class patterns.

Figure 14.3 demonstrates the classification of various design patterns based on their interaction with either of classes or objects:



**Figure 14.3:** Classification of Design Patterns - Class versus Object

## Design principles

Design principles are the guidelines or recommendations that streamline the design and reduce the complexity of an application. These principles are devised following years of software development best practices. They bring a kind of maturity to software development and show us how the code should be written to avoid a bad design. In this section, we will explore some of these design principles and will learn about the benefits that come with them.

## *Don't repeat yourself*

Famously known with the acronym DRY, this principle asserts that the same code should not be repeated in multiple places. Instead, the common logic should be extracted into methods.

[Figure 14.4](#) demonstrates two methods that contain duplicate code of writing to a file:

```

    stores xml content
    to a file
    → public void saveXmlToFile(String content) {
        try {
            validateXml(content);
            String name = fetchFileName(content, "XML");
            File file = new File(name);
            FileWriter writer = new FileWriter(file);
            writer.write(content);
            writer.close();
        } catch(ValidationException / IOException ex) {
            // handle exception
        }
    }

both these methods
duplicate the file I/O
code, this code can be
moved to a separate
method and called from
within these methods
to perform the same
operation

    opens a file writes
    and writes the content
    to a file
    → public void saveJsonToFile(String content) {
        try {
            validateJson(content);
            String name = fetchFileName(content, "JSON");
            File file = new File(name);
            FileWriter writer = new FileWriter(file);
            writer.write(content);
            writer.close();
        } catch(ValidationException / IOException ex) {
            // handle exception
        }
    }
    stores json
    content to a file

```

**Figure 14.4:** Design Principle - DRY I

[Figure 14.5](#) demonstrates the application of the DRY principle, by extracting the common code into a separate method:

```

public void saveToFile(String filename, String content) throws IOException {
    File file = new File(filename);
    FileWriter writer = new FileWriter(file);
    writer.write(content);
    writer.close();
}

public void saveXmlToFile(String content) {
    try {
        validateXml(content);
        String name = fetchFileName(content, "XML");
        saveToFile(name, content); ←
    } catch(ValidationException / IOException ex) {
        // handle exception
    }
}

public void saveJsonToFile(String content) {
    try {
        validateJson(content);
        String name = fetchFileName(content, "JSON");
        saveToFile(name, content); ←
    } catch(ValidationException / IOException ex) {
        // handle exception
    }
}

```

duplicate code is moved to a separate method in the new implementation

the existing method now calls the newly implemented method to perform the file I/O operation

creating a new method to support the file I/O operation ensures that any future change will only have to be made in one method

changed here too!

for any future modification the duplicate code has to be mandatorily updated everywhere to avoid erroneous behaviour

**Figure 14.5:** Design Principle - DRY II

An example of a common logic is reading a file or using the same literal in multiple places. A method could be constructed for reading a file and the literal could be stored in a final variable. Extracting the common logic into methods or the use of final variables to store repeatedly used values reduces the number of places where a change would have to be replicated if the same

code is used at various places. It also reduces the number of tests that need to be performed for testing the same code at multiple places. This principle promotes code reusability, reduces testing effort and lowers the risk of bringing changes to the system.

## Single responsibility

The *single responsibility principle* asserts that each construct should only perform one single task; a construct could either be a class or a method. In other words, there should only be one reason for the constructs to change. [Figure 14.6](#) demonstrates the application of the *single responsibility* principle:

```
public class PressArticle {  
    private String title;  
    private String author;  
    private String content;  
  
    public void replace(String from, String to) {  
        content.replaceAll(from, to);  
    }  
    public boolean contains(String text) {  
        return content.contains(text);  
    }  
}  
  
public class PressArticle {  
    ...  
    public void publish() {  
        // publish the article on web  
    }  
}  
  
public class ArticlePublisher {  
    public void publish(PressArticle article) {  
        // publish the press article on web  
    }  
}
```

the PressArticle class stores information related to an online article and provides some basic methods to operate on the content

the single responsibility principle states that a class should have only one responsibility

an updated definition of the PressArticle now supports a publish method, making it responsible for the generic publish operation as well

the principle asserts that instead of adding responsibility to the PressArticle class we should create another class and make it responsible for publishing an article

***Figure 14.6: Design Principle - Single Responsibility***

It does so by creating another class for publishing press articles instead of the existing class.

This principle states that if the construct is responsible for more than one functionality, then it should be split into as many constructs as the functionalities themselves. It helps to regulate the effect of change on a class. By ensuring that a class performs only one task, it limits the scope of the change and the domino effect it could have if the same class would have served multiple purposes. In short, by clearly defining the responsibility that a class has, a change in that class is only required when there is a change in how that responsibility is performed. It promotes development of small, simple classes that have fewer dependencies. It reduces the effort of testing by reducing the complexity of classes.

## Dependency inversion

This depends upon abstractions, not on concrete classes. The *dependency inversion* principle sounds very similar to the *favor interface over implementation* principle, but there is a slight difference. This principle asserts that the high-level classes should be decoupled from the low-level classes by introducing an abstraction layer. It also inverts the dependency. With this change, the details or the concrete implementations are written following the abstraction instead of the other way around.

By decoupling the two layers, they are no more dependent and can be modified in isolation without affecting each other. Due to the abstraction, it becomes straightforward to replace an existing implementation with another, without the other side of abstraction being aware of it.

It promotes interaction with interfaces and reduces coupling. The classes are no longer dependent and can be developed in isolation.

### Encapsulate what varies

Encapsulation is the process of binding the state and behavior of a class together. It hides the internal complexity of a class and prevents it from being exposed. By using access modifiers and methods, an object's state can be guarded from direct access.

Figure 14.7 demonstrates a class that is open for changes and will remain so in future as well. Any addition to this class requires retesting the existing behavior as well.

```

public class Validator {
    private String content;
    private ContentType cType;
    public boolean validate() {
        result = false;
        switch(cType) {
            case JSON:
                result = validateJson(content);
                break;
            case XML:
                result = validateXml(content);
                break;
            default:
                break;
        }
        return result;
    }
    private boolean validateJson(String content) {
        // validate if the content format is json
    }
    private boolean validateXml(String content) {
        // validate if the content format is xml
    }
}

```

the Validator class validates the format of the content it stores based on the content type parameter

calling the private methods to do actual validation based on the content type

the methods that do the validation

**Figure 14.7:** Design Principle - Encapsulate What Varies |

The principle, *encapsulate what* asserts that those aspects of an application that frequently change should be separated from what stays the same. This way, the change can be isolated and tested separately without affecting the unchanged behavior. [Figure 14.8](#) demonstrates the application of the *encapsulate what varies* principle:

```

public class Validator {
    ...
    public boolean validate() {
        result = false;
        switch(cType) {
            ...
            case YAML:
                result = validateYAML(content);
                break;
            ...
        }
        return result;
    }
    private boolean validateYAML(String content) {
        // validate if the content format is yaml
    }
}

public interface Validator {
    public abstract boolean validate();
}

public class JsonValidator implements Validator {
    public boolean validate(String content) {
        // validate if the content format is json
    }
}

```

the Validator class must be modified for any new type of validation required in the future, what can be done?

calling the newly added method to do yaml validation

the validate yaml method

Instead of modifying the validator class, the code could be separated out into specific classes that all implement the same interface

validator is now an interface and provides the abstract validate method

we now have a separate class that validates only json content, notice that the content is not a data member any more

**Figure 14.8:** Design Principle - Encapsulate What Varies II

The principle mentions of creating an interface and its concrete implementations to support different forms of validation, an improvement over the previous implementation.

## Open closed

The *open close* principle asserts that the classes should be open for extension but closed for modification. It advocates the use of inheritance to add additional behavior to a class, instead of modifying it directly.

[Figure 14.9](#) demonstrates the principle by extending the Report class to provide additional feature, instead of modifying it:

```

public class Report {
    private List<Record> record; ← list of records that act as input for the report
    private List<Filter> filters; ← filters to apply on the records for report generation
    private List<Field> fields; ← list of fields to show in the report
    public void generate() {
        // generate report
    }
    public void publish() {
        // publish report
    }
}

public class Report {
    ...
    private Map<Field, Color> legends;
    public void draw() { ← the new draw method for colourful pie charts
        // draw a pie chart
    }
    ...
}

public class PieChart extends Report { ← instead of modifying the existing class, a new PieChart class can be created that extends the Report class, remember to change the variable access from private to protected
    private Map<Field, Color> legends;
    public void draw() {
        // draw a pie chart
    }
}

```

the Report class generates and publishes a report

to allow for drawing pie charts as well, we have modified the Report class

the open/close principle asserts that the class should be open for extension but closed for modification, and this change breaks that principle

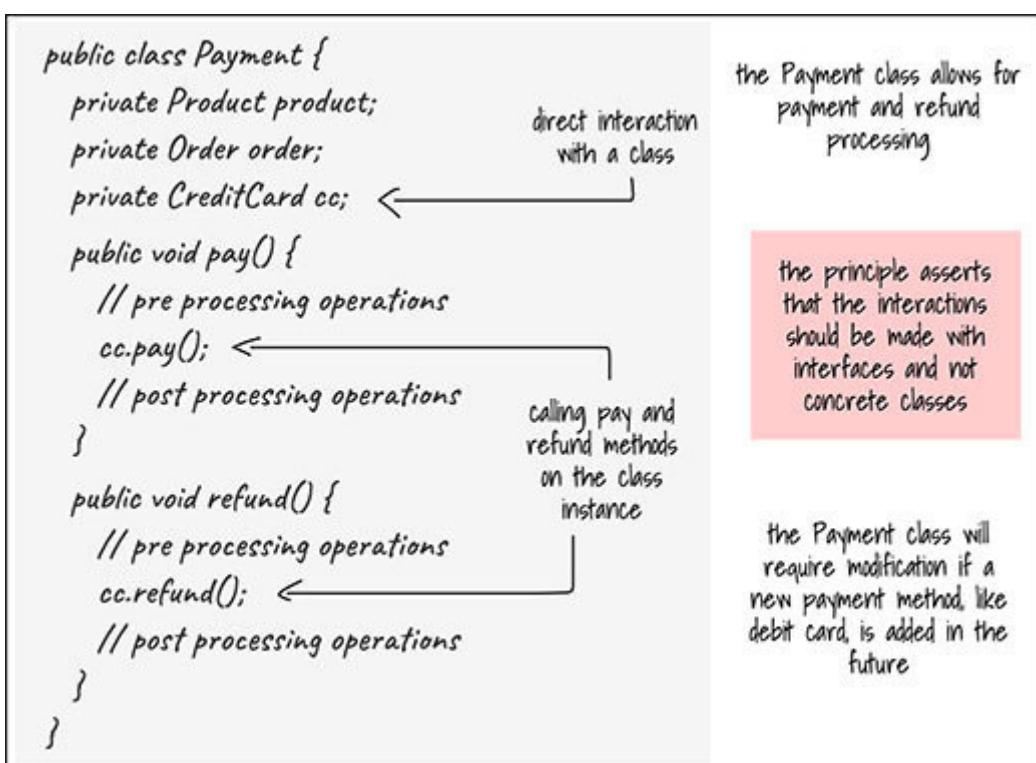
the new PieChart class

**Figure 14.9:** Design Principle: Open Closed

This principle demotes modifying an existing class to ensure better future maintenance, limits the testing effort to the newly added class or method, and reduces risk.

## Favor interface over implementation

More commonly known as **program to an interface, not an** this design principle asserts that interaction with interfaces should be chosen over interaction with concrete implementations. What this means is that whenever a class type interacts with or is used within another class type, the reference type should be an interface. [Figure 14.10](#) demonstrates a class that maintains has-a relationship with concrete classes instead of interfaces:

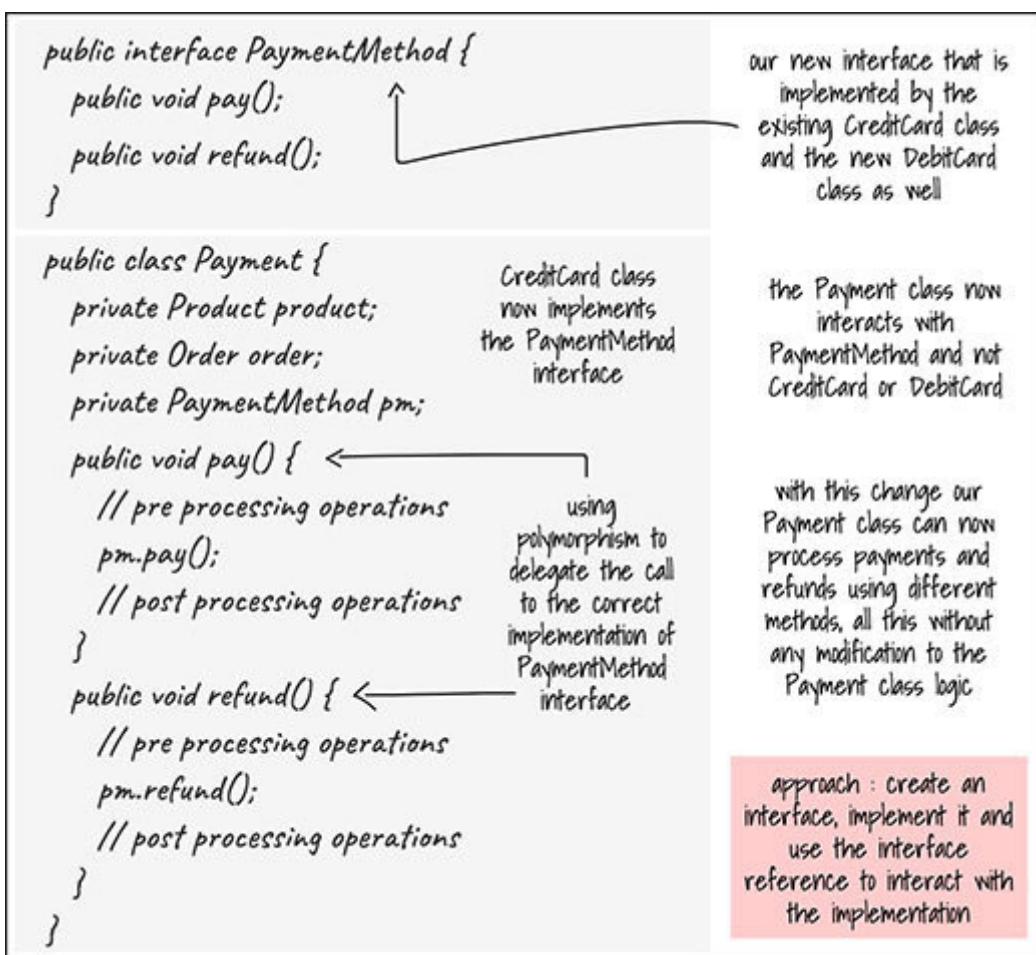


**Figure 14.10:** Design Principle - Favor Interface Over Implementation

In the preceding figure, **CreditCard** is a class and not an interface. This class is a potential candidate for behavior change if a new

payment method is added to it in the future.

Figure 14.11 demonstrates interaction with interfaces instead of concrete classes. This one-time change to the existing implementation of the **Payment** class will allow it to be used with many different payment methods, old and new alike, without any additional modification:



**Figure 14.11: Design Principle - Favor Interface Over Implementation**

This principle is very useful for building solutions that support dynamic behavior without modifying the existing code. Programming to interfaces leads to a flexible, reusable, and

incremental approach to software development where new behaviors can be added to the existing system by adding new implementations to the existing code.

## Interface segregation

The **interface segregation** principle asserts that the specialized interfaces should be favored over generic ones. An interface that exposes many behaviors is hard to manage and leads to unnecessary implementations. A lean interface that provides specialized behavior is both, easy to implement and manage.

Figure 14.12 demonstrates a generic interface for media players:

```
public interface Player {  
    public void play();  
    public void pause();  
    public void stop();  
    public void insert();  
    public void eject();  
    public void next();  
    public void previous();  
    public void changeBrightness(int change);  
    public void changeContrast();  
    public void changeVolume(int change);  
}
```

one generic interface that exposes large number of methods  
such a generic interface, when implemented, mandatorily requires all the methods to be overridden even if they are not relevant to the implementing class  
only relevant for video players  
some of these methods might not be relevant to a music player, like eject is irrelevant for a player with no option to play from a disc

**Figure 14.12: Design Principle - Interface Segregation I**

Figure 14.13 demonstrates how the generic **Player** interface can be broken into multiple smaller interfaces to support specialized behaviors:

```

public interface Player {
    public void play();
    public void pause();
    public void stop();
    public void next();
    public void previous();
    public void changeVolume(int change);
}

public interface VideoPlayer {
    public void changeBrightness(int change);
    public void changeContrast();
}

public interface PhysicalDiskPlayer {
    public void insert();
    public void eject();
}

```

interface Player provides methods to support the default functionality, that are relevant for most media players

with interfaces, providing specific functionality, classes can implement only those interfaces that are relevant to them

provides methods to support functionality specific to a video player

provides methods to support functionality specific to a disc player

interface segregation principle states that interfaces should deal with specific functionality and not be generalized to support multiple features

**Figure 14.13:** Design Principle - Interface Segregation II

An interface with multiple behaviors is often termed as polluted. Such an interface should be broken down into multiple specialized interfaces, each responsible for behaviors that are highly cohesive and taken from the generic interface. A class can implement multiple specialized interfaces to expose relevant behavior instead of implementing a larger interface and override all the methods.

## Least knowledge

The principle of *least knowledge* asserts that the classes should not be highly coupled to each other. It guides us to reduce the interaction between classes to just a few necessary ones.

[Figure 14.14](#) demonstrates the use of the principle of *least knowledge* to reduce coupling and dependency among class implementations:

```
public Record fetchRecord(String recordId) {  
    RecordAccessor ra = recordUtil.getRecordAccessor();  
    return ra.fetchRecord(recordId);  
}  
  
public Record fetchRecord(String recordId) {  
    return recordUtil.getRecord(recordId);  
}
```

the RecordAccessor is unnecessarily coupled to this class

the RecordAccessor fetch record logic should be moved to the Recordutil class so that the dependency can be broken

**Figure 14.14:** Design Principle - Least Knowledge

It prevents us from designing highly coupled class structures and saves us from a cascading effect triggered due to change in a class that is coupled with multiple other classes. A single change could trigger a batch of changes in all the related classes, and those changes can lead to more changes.

The principle states that we should only invoke method that belong to:

The object itself

An object that is referenced by this object

An object that is passed as an argument to a method

An object that the method creates or instantiates

Building lots of dependencies leads to high maintenance costs and a weak, delicate architecture that can be broken because of its own internals.

There are many more design principles that are highly effective and can be used to improve everyday design solutions. The design principles are so exhaustive that they deserve a book of their own, not just a chapter.

## [Anti-patterns](#)

An anti-pattern is a formulated response to solve a common, recurring problem but is often ineffective and counterproductive. While the design patterns are generally considered a good development practice, the anti-patterns are undesirable and should be avoided. Anti-patterns are the opposite of design patterns and are often mistaken as good design practices.

Initially, an anti-pattern may seem to be the right approach for problem solving, but they often lead to trouble in the long run. They are often considered bad design that leads to messed up code and faulty architecture. There are almost as many or may be more anti-patterns as there are design patterns and deserve a book of their own. In this chapter, we will only discuss a few of them that are often used in regular practice.

## An interface for constants

Using interfaces just to define constants and having classes implement them is one practice that is commonly followed to gain access to constant values. Such an interface not only deviates from its core objective, that is, to define the actions that an implementing class can perform, but it also exposes the implementation details in the form of those constants.

It could be argued that keeping constants in an interface increases the convenience with which they can be accessed but they also end up being publicly exposed to all the classes that implement the interface and probably do not need them.

Constants should ideally be kept inside:

The class definition if they are only relevant to that class.

The parent class definition if it has multiple child classes where the constant is relevant.

The interface if the constants are relevant to all or most of its implementations.

Another class if the constants are relevant to a wide range of classes not necessarily in the same hierarchy.

One should try to avoid using interfaces for keeping a constant value because it could be difficult to remove it later for the sake of backward compatibility.

## The god object

An object that provides multiple functionalities and handles too many responsibilities is often termed as a god object.

At times, we end up writing so much logic in a single object that it becomes a know-all, do-all construct. Such an object is in sharp contrast with the single responsibility principle and is usually related to multiple functionalities. While the responsibilities should ideally be divided among multiple objects that interact with each other to solve a computational or business scenario, one large, heavily dependent object ends up doing everything.

Maintaining such an object is not only difficult, but a nightmare at the least when it comes to testing. Even a small change might end up requiring a rigorous amount of testing. A large object defeats the purpose of having a low maintenance, replaceable, and scalable architecture.

Classes should be made smaller and less responsible. They should be cohesive and reflect the purpose of their existence.

## Caching irregularities

Caching data for future reuse is a common approach to speed up performance and reduce the load on the databases. The caching mechanism is often written as a separate layer that exposes certain APIs to interact with the clients. While caching has become an integral part of the system design, it is usually let down by incomplete or bad implementations.

Having multiple caches and a few wrong implementations can reverse all the good things that caching provides. The problem is not with the use of multiple caches, but with implementations that write a value to one cache and read it from another. It could end up with multiple unnecessary calls to the backend and could be highly difficult to trace and correct.

Another wrong practice is not updating the cache after a certain database operation. It is highly significant when a deletion takes place and the cache is not properly updated, resulting in stale or wrong data being shown or sent in response.

## Hard code

Embedding changeable piece of information directly in the implementation is called **hard**. Do not confuse it with constants; they are more aligned with the implementation. Examples of hard coding range from links to services, login credentials, paths to internal resources. Hard coding mostly requires recompilation and server restarts for the changes to take effect. They cannot be changed in runtime or by any user. They require code changes that can only be addressed by a developer.

Ideally, login credentials should not be stored at all, and link to services should be behind a gateway that the application can connect to. Yet, if there is a need to store such information, they can be stored in files, outside the code base and read during system initialization or when there is an alteration to that file.

## **Boat anchor**

Obsolete or not-in-use code that is still a part of the code base is called a **boat**. Retaining outdated functionality in the system is an invitation to an invisible risk. Although the unused modules, classes or any other structs seem highly unlikely to cause a problem, they still can. Here's a list of possible risks that arise due to disused code:

It could be unknowingly referred to and used in the active code leading to some unwarranted behavior.

It could impact the performance of a system by starting up some scheduled or queueing tasks when they are no longer needed.

It increases the compilation time.

Any active singletons or spring beans may use valuable system resources.

Developers may end up debugging the wrong code before they eventually get to know that it is not related to the problem they are solving.

Obsolete code, even if it may be required in future, should be kept in a separate repository from where it can be referred to or

reinstituted when required.

## Copy and paste

Code redundancy and duplication is what is termed as copy and paste programming. Although it may be useful in some scenarios like boilerplate code, loop unrolling or code segments that are short lived and that do not require maintenance, but mostly it is better to refactor and extract the duplicate code into methods or classes.

Extracting redundant code into methods helps with easy maintenance and fast development as one does not need to reinvent and maintain all copies of the code. Another way to approach the problem is to use existing libraries that are well tested and provide efficient solutions rather than programming the entire code base afresh.

### No silver bullets

The simple act of believing that one solution is fit for many, if not all the technical problems is termed as a silver bullet assumption. Every problem is unique, and although there may be valid reasons to use a tested approach to solve many similar problems, it is not always the best one.

There may be other approaches that are more performant and better designed to solve a problem. But we choose the one we prefer the most usually due to a cognitive bias.

## Conclusion

In this concluding chapter, we discussed some of the important design principles and anti-patterns. The design principles guide us on how to improve our design and the anti-patterns warn us about the bad practices.

## Questions

In this section, we will sharpen our brain by indulging in some fun brain activities. Let's get started.

### Find the odd one out

Which one of these is not an anti-pattern?

No silver bullets

Hard code

Least knowledge

God object

Which one of these is an anti-pattern?

Boat anchor

Least knowledge

Open closed

Interface segregation

## Answers

## Find the odd one out

Least knowledge

Boat anchor

**A**

abstract factory [67](#)  
implementation [68](#)  
abstraction [44](#)  
abstract response class [51](#)  
updating [59](#)  
adapter  
building [118](#)  
concrete implementations [121](#)  
implementations  
interfaces  
testing [124](#)  
adapter design pattern [117](#)  
benefits [125](#)  
defining [125](#)  
drawbacks [126](#)  
usage [126](#)  
working [118](#)  
anti-patterns [251](#)  
boat anchor [253](#)  
caching irregularities [253](#)  
copy and paste [254](#)  
god object [252](#)  
hard code [253](#)  
interface for constants [252](#)  
no silver bullets [254](#)

**B**

base decorator [134](#)  
behavioral design patterns [10](#)  
command design pattern [12](#)  
objective [11](#)  
observer design pattern [12](#)  
state design pattern [12](#)  
strategy design pattern [12](#)  
use cases [13](#)  
boat anchor [253](#)  
builder design pattern [6](#)  
advantages [88](#)  
defining [88](#)  
drawbacks [89](#)  
usage [89](#)  
builders  
concrete builders  
deployment manager [86](#)  
implementing  
interface [82](#)  
testing [87](#)

**C**

class patterns [241](#)  
command design pattern [223](#)  
benefits [232](#)  
client [230](#)

command [228](#)  
command flow [225](#)  
definition [232](#)  
drawbacks [233](#)

implementing [225](#)  
invoker [229](#)  
receiver [227](#)  
usage [233](#)  
complex objects  
constructing [75](#)  
constructors [75](#)  
concrete builders  
concrete decorator [136](#)  
concrete response success [52](#)  
creational design patterns  
builder design pattern [6](#)  
factory design pattern [6](#)  
objective [5](#)  
object pool design pattern [6](#)  
singleton design pattern [6](#)  
use cases [7](#)

## D

decorator  
base decorator [134](#)  
building [133](#)  
concrete decorator [136](#)  
concrete implementation [134](#)  
interface [133](#)

map decorator [135](#)  
testing [138](#)  
decorator design pattern [132](#)  
benefits [139](#)  
definition [139](#)

drawbacks [140](#)  
usage [140](#)  
working [133](#)  
Dependency Injection [66](#)  
dependency inversion principle [244](#)  
deployment builder interface [82](#)  
deserialization [38](#)  
design patterns [3](#)  
application examples [238](#)  
behavioral design patterns [10](#)  
classification [4](#)  
class patterns, versus object patterns [241](#)  
core attributes [3](#)  
creational design patterns [4](#)  
structural design patterns [7](#)  
summary [240](#)  
design principles [241](#)  
dependency inversion principle [244](#)  
DRY  
encapsulate what varies  
favor interface over implementation  
interface segregation principle [250](#)  
least knowledge principle [251](#)  
open close principle [247](#)  
single responsibility principle [244](#)  
DRY principle

## E

eager instantiation [31](#)

drawback [32](#)

encapsulate what varies principle

## F

facade [159](#)

building [161](#)

implementing

interface [164](#)

interface, for access control [162](#)

interface, for payment service [163](#)

facade design pattern [160](#)

benefits [168](#)

definition [168](#)

drawbacks [168](#)

usage [169](#)

working [161](#)

factories

building

factory design pattern [6](#)

factory method pattern

favor interface over implementation

french response generator [58](#)

## G

good design  
benefits [14](#)  
centralization [14](#)  
performance [15](#)  
readability [14](#)  
streamlines codebase [15](#)

## H

hard coding [253](#)

## I

incompatible interfaces [116](#)  
interface  
implementing [63](#)  
interface methods, object pool  
acquire [101](#)  
destroy [101](#)  
init [101](#)  
release [101](#)  
interfaces  
interacting [45](#)  
interface segregation principle [250](#)

## L

least knowledge principle [251](#)

logger [23](#)  
using [26](#)  
log level  
adding, to logger [25](#)  
using [24](#)

## M

multiple factories  
multiple feature implementations [223](#)  
multiple instances, of singleton [31](#)

## O

object construction  
delegating  
object patterns [241](#)

object pool [96](#)  
characteristics [99](#)  
destroying [104](#)  
implementing  
initializing [103](#)  
interface [100](#)  
interface methods [101](#)  
life cycle [98](#)  
objects, acquiring [105](#)  
objects, releasing [106](#)  
objects, releasing to [108](#)  
partial initialization [107](#)  
using [100](#)

variations, in implementations [106](#)

object pool design pattern

benefits [109](#)

drawbacks [109](#)

object reusability [96](#)

objects

creating [22](#)

observer design pattern [188](#)

benefits [200](#)

building [191](#)

definition [199](#)

drawbacks [200](#)

interfaces [192](#)

observers [195](#)

observers, notifying [189](#)

observers, registering [189](#)

pull mechanism [199](#)

push mechanism

subject

testing [196](#)

usage [200](#)

variations [196](#)

working [190](#)

open close principle [247](#)

## P

protection proxy [148](#)

proxy [144](#)

building [148](#)

inserting, in call stack [145](#)

interface [149](#).

protection proxy [148](#)

proxy class [151](#)

remote proxy [147](#).

subject [149](#).

testing [152](#)

types [146](#)

virtual proxy [147](#).

proxy design pattern [145](#)

benefits [153](#)

definition [152](#)

drawbacks [153](#)

usage [153](#)

working [146](#)

## R

remote proxy [147](#).

representational state [78](#)

response dialect factory interface

implementing [64](#)

response factory

testing [67](#)

## S

serialization [36](#)

serialized singleton [37](#).

simple factory [55](#)

single responsibility principle [244](#)

singleton [27](#)

enum, using [36](#)

lazy initialization [30](#)

multiple instances [31](#)

reflection, using

singleton design pattern

software design [13](#)

software design patterns [2](#)

state design pattern [205](#)

architecture [206](#)

benefits [216](#)

concrete implementations

definition [216](#)

drawbacks [217](#)

implementing [207](#)

interface [207](#)

stateful object

testing [215](#)

usage [217](#)

state machine [205](#)

strategy design pattern [12](#)

structural design patterns

adapter design pattern [9](#)

decorator design pattern [9](#)

facade design pattern [9](#)

objective [8](#)

proxy design pattern [9](#)

use cases [10](#)

template [176](#)  
template method  
building [179](#).  
varying implementations  
template method design pattern [177](#).  
benefits [183](#)  
definition [183](#)  
drawbacks [183](#)  
testing [182](#)  
usage [184](#)  
working [178](#)  
thread safe singleton [34](#).  
drawback [33](#)

## v

variations [29](#).  
virtual proxy [147](#).

## w

wrappers [131](#)