Java Design Patterns

A Hands-On Experience with Real-World Examples

Second Edition

Vaskaran Sarcar

Foreword by Sunil Sati

Apress®

Java Design Patterns

A Hands-On Experience with Real-World Examples

Second Edition

Vaskaran Sarcar
Foreword by Sunil Sati

Apress®

Java Design Patterns: A Hands-On Experience with Real-World Examples

Vaskaran Sarcar
Bangalore, Karnataka, India

ISBN-13 (pbk): 978-1-4842-4077-9 ISBN-13 (electronic): 978-1-4842-4078-6
https://doi.org/10.1007/978-1-4842-4078-6

Library of Congress Control Number: 2018964945

Copyright © 2019 by Vaskaran Sarcar

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: Laura Berendson
Coordinating Editor: Aditee Mirashi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4077-9. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to
Almighty God, my family, and the Gang of Four.
You are my inspiration.

Table of Contents

About the Author

Vaskaran Sarcar obtained his Master of Engineering degree from Jadavpur University, Kolkata. Currently, he is senior software engineer and team lead in the R&D Hub at HP Inc. India. He was a national Gate Scholar and has more than 12 years of experience in education and the IT industry. He is an alumnus of prestigious institutions in India, such as Jadavpur University, Vidyasagar University, and Presidency University (formerly Presidency College). Reading and learning new things are his passions. You can connect with him at `vaskaran@rediffmail.com` or find him on LinkedIn at `www.linkedin.com/in/vaskaransarcar`.

Other books by Vaskaran include the following:

- *Design Patterns in C#* (Apress, 2018)
- *Interactive C#* (Apress, 2017)
- *Interactive Object-Oriented Programming in Java* (Apress, 2016)
- *Java Design Patterns (First Edition)* (Apress, 2016)
- *C# Basics: Test Your Skill* (CreateSpace, 2015)
- *Operating System: Computer Science Interview Series* (CreateSpace, 2014)

About the Technical Reviewers

Shekhar Kumar Maravi is a system software engineer whose main interests are programming languages, algorithms, and data structures. He obtained his master's degree in computer science and engineering from the Indian Institute of Technology, Bombay. After graduation, he joined Hewlett-Packard's R&D Hub in India to work on printer firmware. Currently, he is a technical lead for automated lab diagnostic device firmware and software at Siemens Healthcare India. He can be reached by email at `shekhar.maravi@gmail.com` or via LinkedIn at `www.linkedin.com/in/shekharmaravi`.

Ritesh Jha is passionate about large-scale distributed systems. Currently, he is working as a senior development engineer for the Supply Chain Technology Group at Walmart Labs. Before Walmart, he worked at eBay and Hewlett-Packard. He has a BE in computer science from Jadavpur University, Kolkata. When he is not exploring new technologies, he can be found exploring new places on his bike. He can be reached by email at `ritesh.jha@hotmail.com` or via LinkedIn at `www.linkedin.com/in/riteshjha9/`.

Ankit Khare is a senior software engineer with expertise in software architecture and designing, programming languages, algorithms, and data structure. After obtaining a BE in computer science, he joined Hewlett-Packard's R&D Center in India in 2010, where he worked with various laser-jet firmware teams. He is currently involved in future machine vision development for print image diagnostic tools involving ink-jet, large-format, and laser-jet printers. He can be reached by email at `akikhare@gmail.com` or via LinkedIn `https://www.linkedin.com/in/khareankit/`.

Acknowledgments

At first, I thank the Almighty. I sincerely believe that with His blessings only, I could complete this book. I extend my deepest gratitude and thanks to

Ratanlal Sarkar and Manikuntala Sarkar. My dear parents, with your blessings only, I could complete the work.

Indrani, my wife, and Ambika, my daughter. Sweethearts, once again, without your love, I could not proceed at all. I know that we needed to limit many social gatherings and invitations to complete this work on time and each time I promise you that I'll take a long break and spend more time with you.

Sambaran, my brother. Thank you for your constant encouragement toward me.

Shekhar, Ritesh, and Ankit. You are my friends and technical advisors. I know that whenever I was in need, your supports were there. Thank you one more time.

Anupam. My friend and another technical advisor. Though this time, you were not involve but still I acknowledge your support and help toward me in the development of Java Design Patterns first edition.

Sunil Sati. My ex-colleague cum senior. A special thanks to you for investing your time to write a foreword for my book. From the moment when experts like you agreed to write for me, I got some additional motivation to enhance the quality of my work.

Celestin, thanks for giving me another opportunity to work with you and Apress.

Laura, Amrita, Nagarajan, Sivachandran, Pradapsankar and Vinoth thank you for your exceptional support to beautify my work.

Lastly, I extend my deepest gratitude to my publisher, the editorial board members, and everyone who directly or indirectly supported this book.

Foreword

"A problem well stated is a problem half solved."

—Charles Kettering, inventor and engineer

To build on this concept, I must say that thinking about all possible scenarios and coming out with a best possible option is the key to a robust and lasting solution. This new second edition of *Java Design Patterns* will serve as a mentor and guide to engineers and designers who are regularly challenged to come up with the best possible solution in resource-constrained environments. This book explains in very clear terms the design patterns, the alternatives, and the concept of antipatterns. The complete section on antipatterns is very thought-provoking and helps us appreciate the utility of design in the first place.

As in his previous books, Vaskaran has provided hands-on experience in implementing design patterns. His innate way of getting engineers to think of an alternative solution is very insightful. This book will serve as mentor and task master as one traverses the chapters.

When I reflect on my interactions with Vaskaran while facing some complex engineering issues in HPI, I find him to be a keen listener, deep thinker, and a person who doesn't rush for a solution. After analyzing all possible alternatives, he picks the best one among the lot. In summary, this is what this book all about.

Sunil Sati

Senior Project Manager, BU Automotive Division, NXP Semiconductors

About Sunil Sati

Sunil Sati is an engineer with a major in electronics and communication from NIIT Surathkal and EGMP from IIM-Bangalore. He has 23 years of experience in various roles and capacities in process automation and semiconductor industries. Currently, he is working as senior manager with NXP Semiconductors in the automotive division. He was the brand ambassador in HPI for the Print Renaissance program. He loves to work and build teams across geographic locations.

Introduction

Welcome to your journey through *Design Patterns in C#*.

This is an introductory guide to the design patterns that you want to use in Java. You probably know that the concept of design patterns became extremely popular with the Gang of Four's famous book *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994). Most important, these concepts still apply in today's programming world. The book came out at the end of 1994, and it primarily focused on C++.

But Sun Microsystems released its first public implementation Java 1.0 in 1995. So, in 1995, Java was totally new to the programming world. But it grew rapidly, becoming rich with features. It has now secured its rank in world's top programming languages. In today's programming world, it is always in high demand. On the other hand, the concepts of design patterns are universal. So, when you exercise these fundamental concepts of design patterns with Java, you will be a better programmer and you'll open new opportunities for yourself.

In 2015, I wrote *Design Patterns in C#: Computer Science Interview Series*, and in 2016, I wrote *Java Design Pattern : A tour with 23 Gang of Four Design Patterns in Java*. They are basically the companions to this book.

In those books, my core intention was to implement each of the 23 Gang of Four (GoF) design patterns with C# and Java implementations. I wanted to present each pattern with simple examples. One thing was always on my mind when writing *Java Design Patterns* (First Edition): I wanted to use the most basic constructs of Java, so that the code would be compatible with both the upcoming version and the legacy version of Java. I have found this method helpful in the world of programming.

In the last two years, I got a lot of constructive feedback from my readers. This fully revised and updated version is created keeping those feedback in mind. I also took the opportunity to update the formatting and correct some typos in the previous version of the book and add new content to this new edition.

This time, I wanted to focus on another important area; I call it the "doubt-clearing sessions." I knew that if I could add some more information such as alternative ways to write these implementations, the pros and cons of these patterns, when to choose one

approach over another, and so on, readers would find this book even more helpful. So, in this enhanced version of the original, I have added a "Q&A Session" section to each chapter that can help you learn about each pattern in more depth.

In the world of programming, there is no shortage of patterns, and each has its own significance. So, in addition to the 23 GoF design patterns covered in Part I, I discuss three design patterns that are equally important in today's world of programming in Part II. Finally, in Part III, I discuss the criticism of design patterns and give you an overview of antipatterns, which are also important when you implement the concepts of design patterns in your applications.

Before jumping into these topics, I want to highlight few more points.

- You are an intelligent person. You have chosen a subject that can assist you throughout your career. If you are a developer/programmer, you need these concepts. If you are an architect at a software organization, you need these concepts. If you are a college student, you need these concepts, not only to score high on exams but to enter the corporate world. Even if you are a tester who needs to take care of white-box testing or needs to know about the code paths of a product, these concepts will help you a lot.
- I already mentioned that this book was written using the most basic features of Java so that you do not need to be familiar with advanced Java topics. These examples are simple and straightforward. I believe that these examples are written in such a way that even if you are familiar with another popular language, such as C#, C++, and so on, you can still easily grasp the concepts in this book.
- There are many books about design patterns and related topics. You may be wondering why I would want to write a new one about the same topics. The simple answer is that I have found other reference material to be scattered. Second, in most cases, many of those examples are unnecessarily large and complex. I like simple examples. I believe that anyone can grasp a new idea with simple examples, and if the core concept is clear, you can easily move into more advanced areas. I believe that this book scores high in this context. The illustrated examples are simple. I wanted to keep this book concise so that it motivates you to continue your journey of learning.

- Each chapter is divided into six parts: a definition (which is basically called *intent* in *Design Patterns: Elements of Reusable Object-Oriented Software*), a core concept, a real-world example, a computer/coding–world example, a sample program with various output, and the Q&A Session section. These Q&A Session sections help you learn about each pattern in more depth.
- Please remember that you have just started on this journey. As you learn about these concepts, try to write your own code; only then will you master the area.
- You will be able to download all the source code in the book from the publisher's website. I plan to maintain the "Errata," and if required, I can also make update/announcements there. So, I suggest that you visit those pages to receive any corrections or updates.

Guidelines for Using This Book

Here are some suggestions for you to use the book more effectively.

- You should have a basic understanding of Java and you should know how to create classes, interfaces, and so forth. It is helpful to be familiar with common terms in object-oriented programming; for example, encapsulation, abstraction, polymorphism, and inheritance.
- I assume that you have some idea about the GoF design patterns. If you are absolutely new to design patterns, I suggest you quickly go through Appendix A. This appendix will help you become familiar with the basic concepts of design patterns.
- If you are confident about the content in Appendix A, you can start with any part of the book. But I suggest you go through the chapters sequentially. The reason is that some fundamental design techniques may be discussed in the "Q&A Session" section of a previous chapter, and I did not repeat those techniques in the later chapters.

- There is only one exception to the previous suggestion. There are three factory patterns: simple factory, factory method, and abstract factory. These three patterns are closely related, but the simple factory pattern does not directly fall into the GoF design catalog, so it appears in Part II of the book. So, I suggest that when you start learning about these three factory patterns, you begin with the simple factory pattern.
- These programs are tested with Java 8 (update 172). I used the Eclipse editor in a Windows 10 environment. So, in the Eclipse Package Explorer view, you may notice the string jdk1.8.0_172. At the time of this writing, Photon is the latest edition of Eclipse (released in June 27, 2018), Java 8 is the long-term support (LTS) version, and Java 10 is the rapid release version. Java 11 is the next LTS version after Java 8 and planned for September 2018. But all of this version information should not matter much because I used the most basic constructs of Java. So, I believe that the code should execute smoothly in the upcoming versions of Java/Eclipse as well.
- One of my reviewers tested the code in a Linux environment. So, I believe that the results should not vary in other environments, but you know the nature of software—it is naughty. So, I recommend that if you want to see the same output, it is best if you can mimic the same environment.
- To draw class diagrams, ObjectAid Uml Explorer is used in the Eclipse editor. It is a lightweight tool for Eclipse. At the time of this writing, it is free if you want to draw the class diagrams, but to draw the sequence diagrams, you need to purchase a license. The website at `www.objectaid.com/home` gives more information about licenses, terms, and conditions.

Lastly, I hope that this enhanced edition provides more help to you.

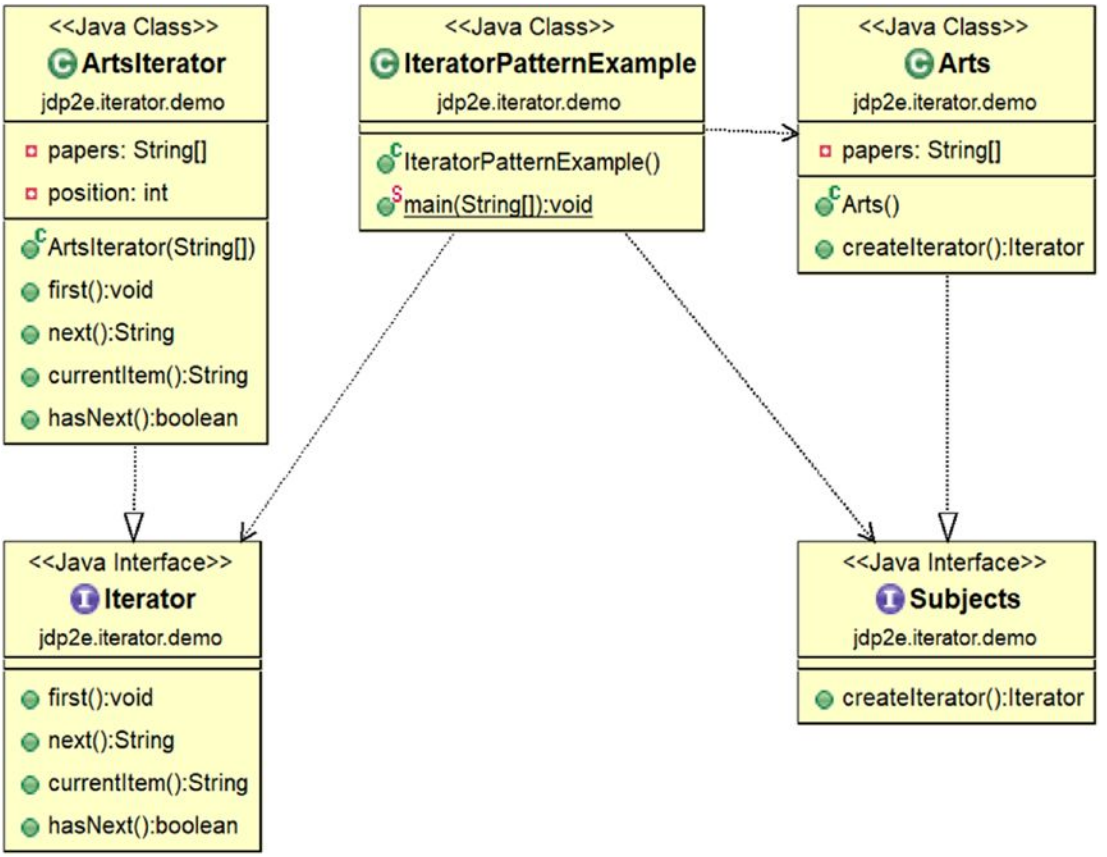
Conventions Used in This Book

- All the output and code in the book follow the same font and structure. To draw your attention, in some places, I made it bold, like the following.

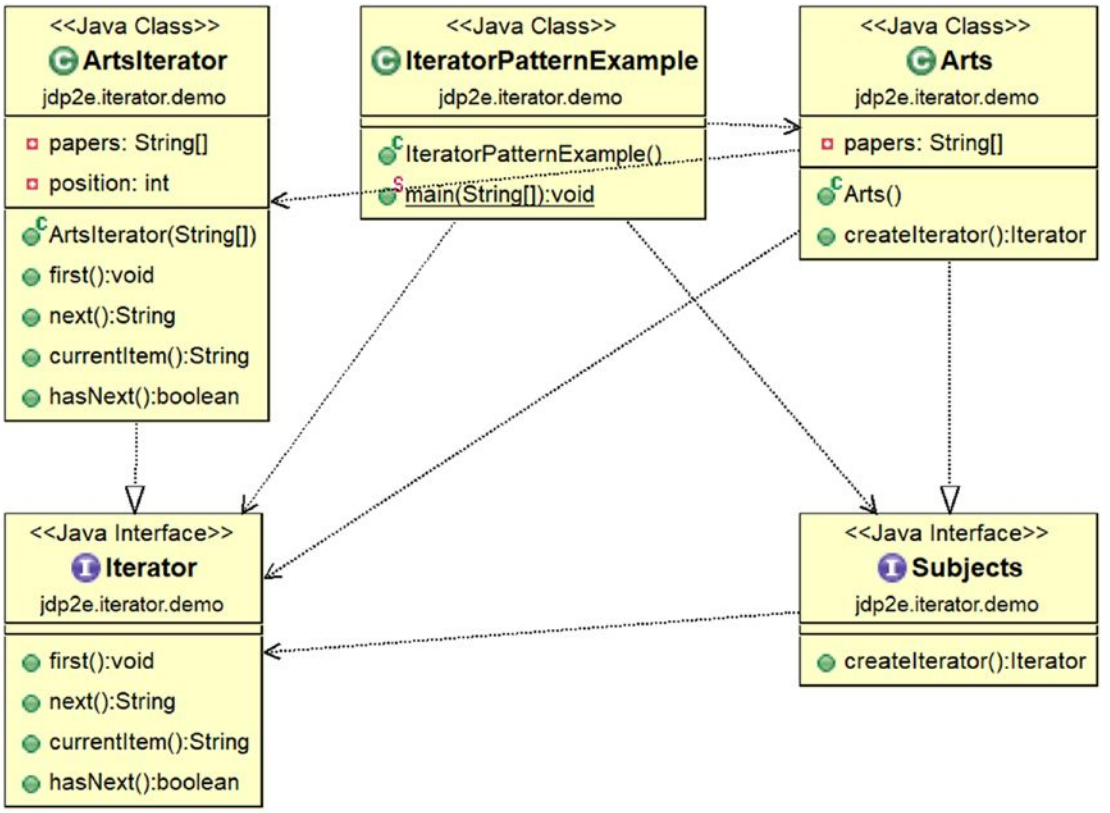
```
***Mediator Pattern Demo***
At present, registered employees are:
Amit
Sohel
Raghu
Communication starts among participants...
Amit posts: Hi Sohel, can we discuss the mediator pattern? Last
message posted at 2018-09-09T17:41:21.868
Sohel posts: Hi Amit, yup, we can discuss now. Last message posted
at 2018-09-09T17:41:23.369
Raghu posts: Please get back to work quickly. Last message posted
at 2018-09-09T17:41:24.870
```

  An outsider named Jack trying to send some messages.

- In some cases, to present a cleaner class diagram and focus on the important parts, the less important dependencies are not shown. For example, consider the diagram presented in Chapter 18.


<<Java Class>>
ArtsIterator
jdp2e.iterator.demo
papers: String[]
position: int
ArtsIterator(String[])
first():void
next():String
currentItem():String
hasNext():boolean
<<Java Class>>
IteratorPatternExample
jdp2e.iterator.demo
IteratorPatternExample()
main(String[]):void
<<Java Class>>
Arts
jdp2e.iterator.demo
papers: String[]
Arts()
createIterator():Iterator
<<Java Interface>>
Iterator
jdp2e.iterator.demo
first():void
next():String
currentItem():String
hasNext():boolean
<<Java Interface>>
Subjects
jdp2e.iterator.demo
createIterator():Iterator

But if I need to show all the dependencies, it may look like the following.



You can see that the later one is much more complex and difficult to understand. For the ObjectAid class diagrams in Eclipse, you can always show these dependencies by selecting an element in the diagram, right-clicking and selecting Add ➤ Dependencies.

- I like to put curly braces on a new line. and I love to see a method body like the following.

```
public void myFunction()
{
//Some code
}
Instead of the following:
public void myFunction(){
//Some code
}
```

I used the same format for most of the methods, except main() methods.

PART I

Gang of Four Patterns

CHAPTER 1

Singleton Pattern

This chapter covers the singleton pattern.

GoF Definition

Ensure a class only has one instance, and provide a global point of access to it.

Concept

A class cannot have multiple instances. Once created, the next time onward, you use only the existing instance. This approach helps you restrict unnecessary object creations in a centralized system. The approach also promotes easy maintenance.

Real-World Example

Let's assume that you are a member of a sports team, and your team is participating in a tournament. Your team needs to play against multiple opponents throughout the tournament. Before each of these matches, as per the rules of the game, the captains of the two sides must do a coin toss. If your team does not have a captain, you need to elect someone as a captain. Prior to each game and each coin toss, you may not repeat the process of electing a captain if you already nominated a person as a captain for the team. Basically, from every team member's perspective, there should be only one captain of the team.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_1

Computer-World Example

In some specific software systems, you may prefer to use only one file system for the centralized management of resources. Also, this pattern can implement a caching mechanism.

Note You notice a similar pattern when you consider the getRuntime() method of the java.lang.Runtime class. It is implemented as an eager initialization of a Singleton class. You'll learn about eager initialization shortly.

Illustration

These are the key characteristics in the following implementation.

- The constructor is private to prevent the use of a "new" operator.
- You'll create an instance of the class, if you did not create any such instance earlier; otherwise, you'll simply reuse the existing one.
- To take care of thread safety, I use the "synchronized" keyword.

Class Diagram

Figure 1-1 shows the class diagram for the illustration of the singleton pattern.

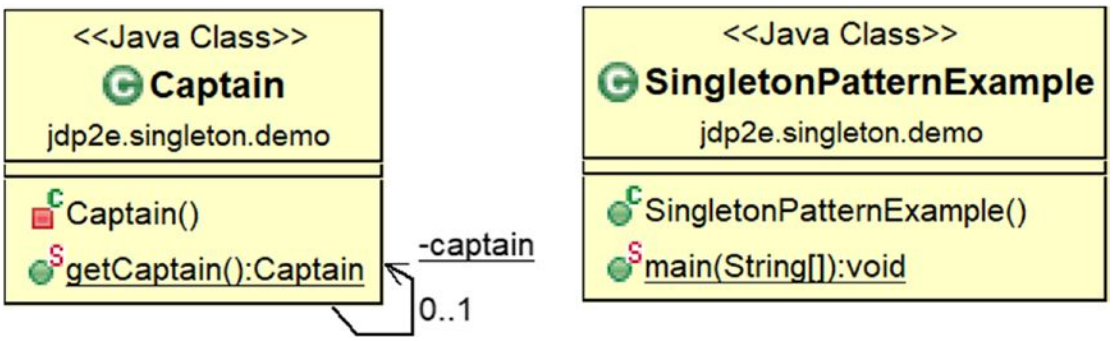


Figure 1-1. *Class diagram*

Package Explorer View

Figure 1-2 shows the high-level structure of the program.

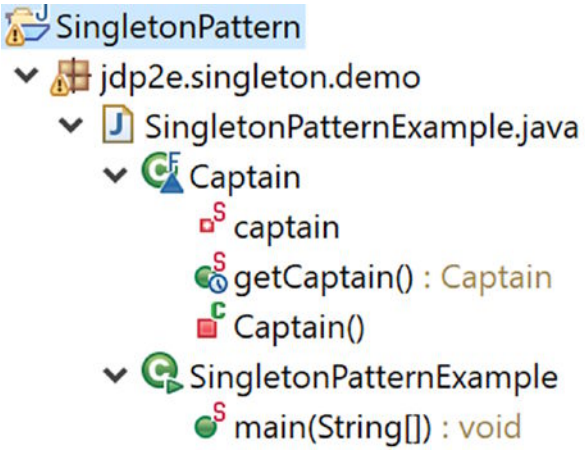


Figure 1-2. *Package Explorer view*

Discussion

I have shown you a simple example to illustrate the concept of the singleton pattern.

Let's review the notable characteristics with the following approach.

- The constructor is private, so you cannot instantiate the Singleton class(Captain) outside. It helps us to refer the only instance that can exist in the system, and at the same time, you restrict the additional object creation of the Captain class.
- The private constructor also ensures that the Captain class cannot be extended. So, subclasses cannot misuse the concept.
- I used the "synchronized" keyword. So, multiple threads cannot involve in the instantiation process at the same time. I am forcing each thread to wait its turn to get the method, so thread- safety is ensured. But synchronization is a costly operation and once the instance is created, it is an additional overhead. (I'll discuss some alternative methods in the upcoming sections, but each of them has its own pros and cons).

Implementation

Here's the implementation.

```
package jdp2e.singleton.demo;

final class Captain
{
    private static Captain captain;
    //We make the constructor private to prevent the use of "new"
    private Captain() {    }

    public static synchronized Captain getCaptain()
    {

        // Lazy initialization
        if (captain == null)
        {
            captain = new Captain();
            System.out.println("New captain is elected for your team.");
        }
        else
        {
            System.out.print("You already have a captain for your team.");
            System.out.println("Send him for the toss.");
        }
        return captain;
    }
}
// We cannot extend Captain class.The constructor is private in this case.
//class B extends Captain{}// error
public class SingletonPatternExample {
    public static void main(String[] args) {
        System.out.println("***Singleton Pattern Demo***\n");
        System.out.println("Trying to make a captain for your team:");
        //Constructor is private.We cannot use "new" here.
        //Captain c3 = new Captain();//error
```

```
        Captain captain1 = Captain.getCaptain();
        System.out.println("Trying to make another captain for your
        team:");
        Captain captain2 = Captain.getCaptain();
        if (captain1 == captain2)
        {
            System.out.println("captain1 and captain2 are same instance.");
        }

    }
}
```

Output

Here's the output.

```
***Singleton Pattern Demo***

Trying to make a captain for your team:
New captain is elected for your team.
Trying to make another captain for your team:
You already have a captain for your team.Send him for the toss.
captain1 and captain2 are same instance.
```

Q&A Session

1. **Why are you complicating the program? You could simply write the Singleton class as follows.**

```
class Captain
{
   private static Captain captain;
    //We make the constructor private to prevent the use of "new"
   private Captain() { }
```

```
    public static Captain getCaptain()
    {
              // Lazy initialization
              if (captain == null)
              {
                 captain = new Captain();
                 System.out.println("New captain is elected for
                 your team.");
              }
              else
              {
                  System.out.print("You already have a captain for
                  your team.");
                  System.out.println("Send him for the toss.");
              }
              return captain;
      }
}
```

Is this understanding correct?

It can work in a single threaded environment only but it cannot be considered a thread-safe implementation in a multithreaded environment. Consider the following case. Suppose, in a multithreaded environment, two (or more) threads try to evaluate this:

```
if (captain == null)
```

And if they see that the instance is not created yet, each of them will try to create a new instance. As a result, you may end up with multiple instances of the class.

2. **Why did you use the term *lazy initialization* in the code?**

   Because the singleton instance will not be created until the `getCaptain()` method is called here.

3. **What do you mean by *lazy initialization*?**

   In simple terms, lazy initialization is a technique through which you delay the object creation process. It says that you should create an object only when it is required. This approach can be helpful when you deal with expensive processes to create an object.

4. **Why are you making the class final? You have a private constructor that could prevent the inheritance. Is this correct?**

   Subclassing can be prevented in various ways. Yes, in this example, since the constructor is already marked with the "private" keyword, it was not needed. But if you make the Captain class final, as shown in the example, that approach is considered a better practice. It is effective when you consider a nested class. For example, let's modify the private constructor body to examine the number of instances (of the Captain class) created. Let's further assume that in the preceding example, you have a *non-static nested class* (called *inner class* in Java) like the following. (All changes are shown in bold.)

```
//final class Captain
class Captain
{
    private static Captain captain;
    //We make the constructor private to prevent the use of "new"
    static int numberOfInstance=0;
    private Captain()
    {
        numberOfInstance++;
        System.out.println("Number of instances at this moment="+
        numberOfInstance);
    }

    public static synchronized Captain getCaptain()
    {
```

```
        // Lazy initialization
        if (captain == null)
        {
            captain = new Captain();
            System.out.println("New captain is elected for your
            team.");
        }
        else
        {
            System.out.print("You already have a captain for your
            team.");
            System.out.println("Send him for the toss.");
        }
        return captain;
    }
     //A non-static nested class (inner class)
   public class CaptainDerived extends Captain
   {
     //Some code
   }
}
```

Now add an another line of code (shown in bold) inside the `main()` method, as follows.

```
public class SingletonPatternExample {
    public static void main(String[] args) {
        System.out.println("***Singleton Pattern Demo***\n");
        System.out.println("Trying to make a captain for your
        team:");
        //Constructor is private.We cannot use "new" here.
        //Captain c3 = new Captain();//error
        Captain captain1 = Captain.getCaptain();
        System.out.println("Trying to make another captain for your
        team:");
        Captain captain2 = Captain.getCaptain();
```

```
            if (captain1 == captain2)
            {
                System.out.println("captain1 and captain2 are same
                instance.");
            }
            Captain.CaptainDerived derived=captain1.new
            CaptainDerived();
        }
    }
```

Now notice the output.

Output

Now, you can see that the program has violated the key objective, because I never intended to create more than one instance.

```
***Singleton Pattern Demo***

Trying to make a captain for your team:
Number of instances at this moment=1
New captain is elected for your team.
Trying to make another captain for your team:
You already have a captain for your team.Send him for the toss.
captain1 and captain2 are same instance.
Number of instances at this moment=2
```

5. **Are there any alternative approaches for modelling singleton design patterns?**

   There are many approaches. Each has its own pros and cons. You have already have seen two of them. Let's discuss some alternative approaches.

Eager Initialization

Here is a sample implementation of the eager initialization.

```
class Captain
{
    //Early initialization
    private static final Captain captain = new Captain();
    //We make the constructor private to prevent the use of "new"
    private Captain()
    {
        System.out.println("A captain is elected for your team.");
    }
    /* Global point of access.The method getCaptain() is a public static
    method*/
    public static Captain getCaptain()
    {
        System.out.println("You have a captain for your team.");
        return captain;
    }
}
```

Discussion

An eager initialization approach has the following pros and cons.

Pros

- It is straightforward and cleaner.
- It is the opposite of lazy initialization but still thread safe.
- It has a small lag time when the application is in execution mode because everything is already loaded in memory.

Cons

- The application takes longer to start (compared to lazy initialization) because everything needs to be loaded first. To examine the penalty, let's add a dummy method (shown in bold) in the Singleton class. Notice that in the main method, I am invoking only this dummy method. Now examine the output.

```
package jdp2e.singleton.questions_answers;

class Captain
{
    //Early initialization
    private static final Captain captain = new Captain();
    //We make the constructor private to prevent the use of "new"
    private Captain()
    {
        System.out.println("A captain is elected for your team.");
    }
    /* Global point of access.The method getCaptain() is a public static
    method*/
    public static Captain getCaptain()
    {
        System.out.println("You have a captain for your team.");
        return captain;
    }
    public static void dummyMethod()
    {
        System.out.println("It is a dummy method");
    }
}
public class EagerInitializationExample {
    public static void main(String[] args) {
        System.out.println("***Singleton Pattern Demo With Eager
        Initialization***\n");
        Captain.dummyMethod();
        /*System.out.println("Trying to make a captain for your team:");
        Captain captain1 = Captain.getCaptain();
        System.out.println("Trying to make another captain for your
        team:");
        Captain captain2 = Captain.getCaptain();
```

```
            if (captain1 == captain2)
            {
                System.out.println("captain1 and captain2 are same
                instance.");
            }*/
    }
}
```

Output

```
***Singleton Pattern Demo With Eager Initialization***
```

A captain is elected for your team.
It is a dummy method

Analysis

Notice that *A captain is elected for your team* still appears in the output, though you may have no intention to deal with that.

So, in the preceding situation, an object of the Singleton class is always instantiated. Also, prior to Java 5, there were many issues that dealt with Singleton classes.

Bill Pugh's Solution

Bill Pugh came up with a different approach using a static nested helper class.

```
package jdp2e.singleton.questions_answers;

class Captain1
{
    private Captain1() {
        System.out.println("A captain is elected for your team.");
    }
    //Bill Pugh solution
    private static class SingletonHelper{
        /*Nested class is referenced after getCaptain() is called*/

        private static final Captain1 captain = new Captain1();
    }
```

```
    public static Captain1 getCaptain()
    {
        return SingletonHelper.captain;
    }
    /*public static void dummyMethod()
    {
        System.out.println("It is a dummy method");
    }  */
}
```

This method does not use a synchronization technique and eager initialization. Notice that the SingletonHelper class comes into consideration only when someone invokes the `getCaptain()` method. And this approach will not create any unwanted output if you just call any `dummyMethod()` from `main()`, as with the previous case (to examine the result, you need to uncomment the `dummyMethod()` body). It is also treated one of the common and standard methods for implementing singletons in Java.

Double-Checked Locking

There is another popular approach, which is called *double-checked locking*. If you notice our synchronized implementation of the singleton pattern, you may find that synchronization operations are costly in general and the approach is useful for some initial threads that might break the singleton implementation. But in later phases, the synchronization operations might create additional overhead. To avoid that problem, you can use a synchronized block inside an `if` condition, as shown in the following code, to ensure that no unwanted instance is created.

```
package jdp2e.singleton.questions_answers;

final class Captain2
{
    private static Captain2 captain;
    //We make the constructor private to prevent the use of "new"
    static int numberOfInstance=0;
    private Captain2() {
        numberOfInstance++;
```

```
        System.out.println("Number of instances at this moment="+
        numberOfInstance);
    }

    public static  Captain2 getCaptain(){
        if (captain == null) {
            synchronized (Captain2.class) {
                // Lazy initialization
                if (captain == null){
                    captain = new Captain2();
                    System.out.println("New captain is elected for your
                    team.");
                }
                else
                {
                    System.out.print("You already have a captain for your
                    team.");
                    System.out.println("Send him for the toss.");
                }
            }
        }
        return captain;
    }
}
```

If you are further interested in singleton patterns, read the article at www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples.

6. **In short, if I need to create synchronized code, I can use the synchronized keyword in Java. Is this correct?**

   Yes, JVM ensures this. Internally, it uses locks on a class or an object to ensure that only one thread is accessing the data. In Java, you can apply this keyword to a method or statements(or, block of code). In this chapter, I have exercised it in both ways. (In the initial implementation, you used the synchronized method, and in double-checked locking, you saw the use of the other version).

7. **Why are multiple object creations a big concern?**

   - In real-world scenarios, object creations are treated as costly operations.
   - Sometimes you need to implement a centralized system for easy maintenance, because it can help you provide a global access mechanism.

8. **When should I consider singleton patterns?**

   Use of a pattern depends on particular use cases. But in general, you can consider singleton patterns to implement a centralized management system, to maintain a common log file, to maintain thread pools in a multithreaded environment, to implement caching mechanism or device drivers, and so forth.

9. **I have some concern about the eager initialization example. Following the definition, it appears that it is not exactly eager initialization. This class is loaded by the JVM only when it is referenced by code during execution of the application. That means this is also lazy initialization. Is this correct?**

   Yes, to some extent your observation is correct. There is a debate on this discussion. In short, it is eager compared to the previous approaches. You saw that when you called only the `dummyMethod()`; still, you instantiated the singleton, though you did not need it. So, in a context like this, it is eager but it is lazy in the sense that the singleton instantiation will not occur until the class is initialized. So, the degree of laziness is the key concern here.

CHAPTER 2

Prototype Pattern

This chapter covers the prototype pattern.

GoF Definition

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Concept

In general, creating a new instance from scratch is a costly operation. Using the prototype pattern, you can create new instances by copying or cloning an instance of an existing one. This approach saves both time and money in creating a new instance from scratch.

Real-World Example

Suppose we have a master copy of a valuable document. We need to incorporate some changes to it to see the effect of the change. In such a case, we can make a photocopy of the original document and edit the changes.

Consider another example. Suppose a group of people decide to celebrate the birthday of their friend Ron. They go to a bakery and buy a cake. To make it special, they request the seller to write, "Happy Birthday Ron" on the cake. From the seller's point of view, he is not making any new model. He already defined the model and produces many cakes (which all look the same) every day by following the same process, and finally makes each special with some small changes.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_2

Computer-World Example

Let's assume that you have an application that is very stable. In the future, you may want to update the application with some small modifications. So, you start with a copy of your original application, make changes, and analyze further. Surely, to save your time and money, you do not want to start from scratch.

Note Consider the Object.clone() method as an example of a prototype.

Illustration

Figure 2-1 illustrates a simple prototype structure.

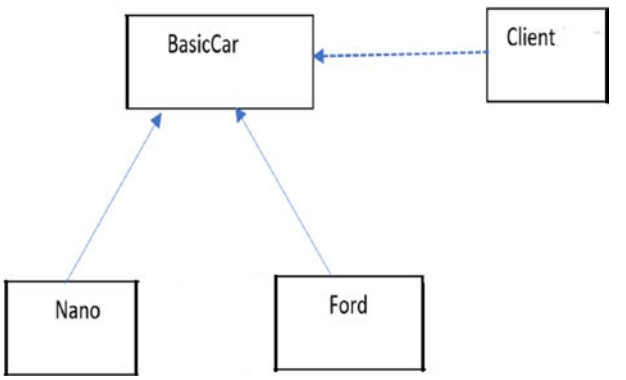


Figure 2-1. *A sample prototype structure*

Here, BasicCar is a basic prototype. Nano and Ford are the concrete prototypes that have implemented the `clone()` method defined in BasicCar. In this example, we have created a BasicCar class with a default price (in Indian currency). Later, we modify the price per model. PrototypePatternExample.java is the client in this implementation.

Class Diagram

Figure 2-2 shows a class diagram of the prototype pattern.

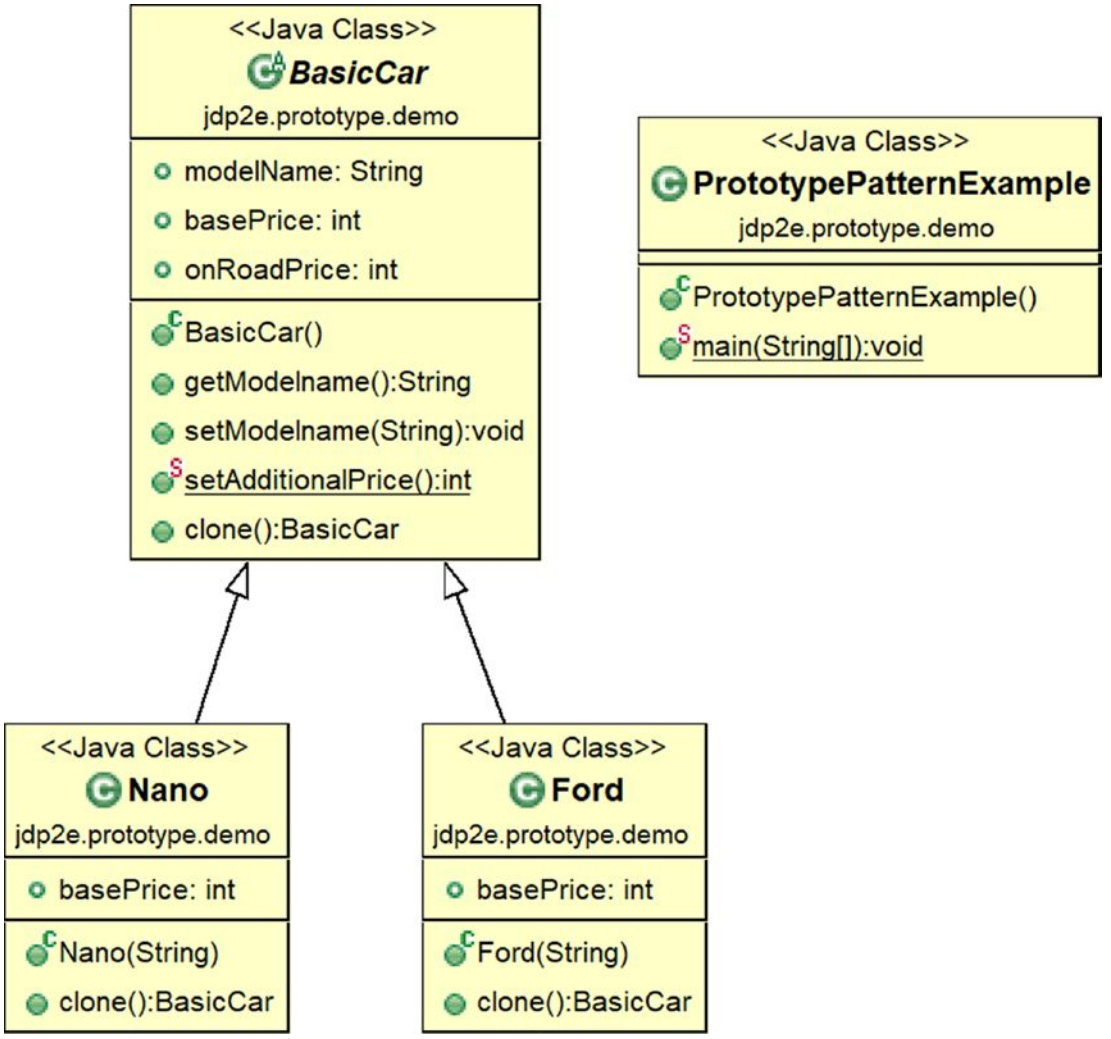


Figure 2-2. *Class diagram*

Package Explorer View

Figure 2-3 shows the high-level structure of the program.

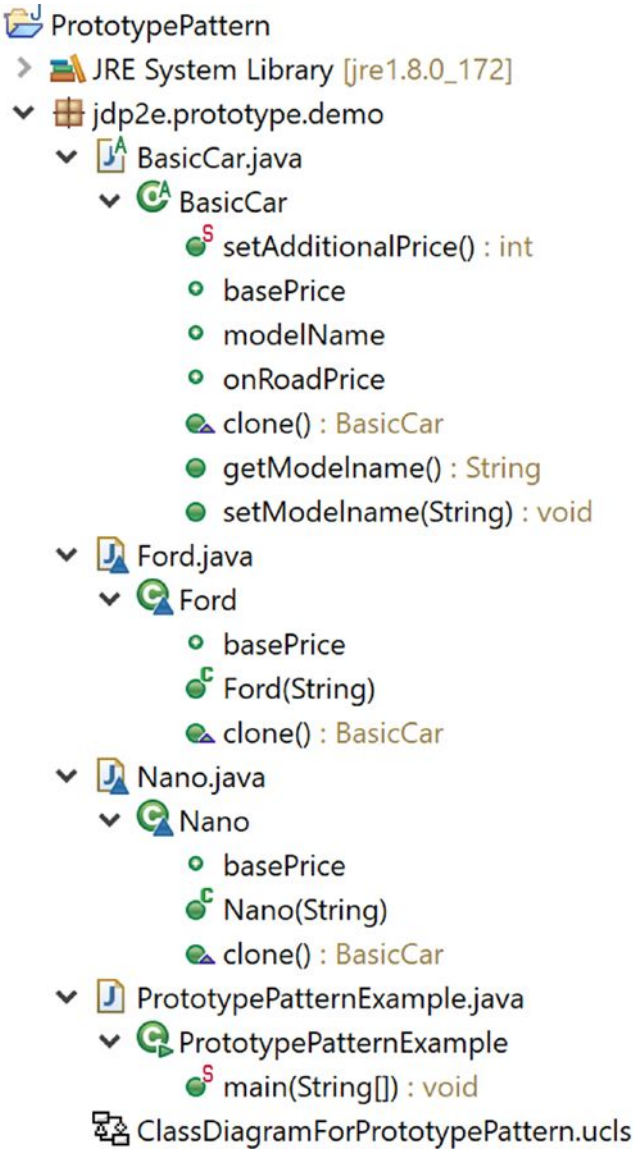


Figure 2-3. *Package Explorer view*

Implementation

Here's the implementation.

```
//BasicCar.java
package jdp2e.prototype.demo;
import java.util.Random;

public abstract class BasicCar implements Cloneable
{
    public String modelName;
    public int basePrice,onRoadPrice;

    public String getModelname() {
        return modelName;
    }
    public void setModelname(String modelname) {
        this.modelName = modelname;
    }

    public static int setAdditionalPrice()
    {
        int price = 0;
        Random r = new Random();
        //We will get an integer value in the range 0 to 100000
        int p = r.nextInt(100000);
        price = p;
        return price;
    }
    public BasicCar clone() throws CloneNotSupportedException
    {
        return  (BasicCar)super.clone();
    }
}
//Nano.java
package jdp2e.prototype.demo;
```

```
class Nano extends BasicCar
{
    //A base price for Nano
    public int basePrice=100000;
    public Nano(String m)
    {
        modelName = m;
    }
    @Override
    public BasicCar clone() throws CloneNotSupportedException
    {
         return (Nano)super.clone();
         //return this.clone();
    }
}
//Ford.java

package jdp2e.prototype.demo;

class Ford extends BasicCar
{
    //A base price for Ford
    public int basePrice=100000;
    public Ford(String m)
    {
        modelName = m;
    }

    @Override
    public BasicCar clone() throws CloneNotSupportedException
    {
        return (Ford)super.clone();
    }
}
//Client
// PrototypePatternExample.java
```

```
package jdp2e.prototype.demo;

public class PrototypePatternExample
{
    public static void main(String[] args) throws
    CloneNotSupportedException
    {
        System.out.println("***Prototype Pattern Demo***\n");
        BasicCar nano = new Nano("Green Nano") ;
        nano.basePrice=100000;

        BasicCar ford = new Ford("Ford Yellow");
        ford.basePrice=500000;

        BasicCar bc1;
        //Nano
        bc1 =nano.clone();
        //Price will be more than 100000 for sure
        bc1.onRoadPrice = nano.basePrice+BasicCar.setAdditionalPrice();
        System.out.println("Car is: "+ bc1.modelName+" and it's price is
        Rs."+bc1.onRoadPrice);

        //Ford
        bc1 =ford.clone();
        //Price will be more than 500000 for sure
        bc1.onRoadPrice = ford.basePrice+BasicCar.setAdditionalPrice();
        System.out.println("Car is: "+ bc1.modelName+" and it's price is
        Rs."+bc1.onRoadPrice);
    }
}
```

Output

Here's the output.

```
***Prototype Pattern Demo***

Car is: Green Nano and it's price is Rs.123806
Car is: Ford Yellow and it's price is Rs.595460
```

Note You can see a different price in your system because we are generating a random price in the setAdditionalPrice() method inside the BasicCar class. But I have assured that the price of the Ford will be greater than the Nano.

Q&A Session

1. **What are the advantages of using prototype design patterns?**
   - It is useful when creating an instance of a class is a complicated (or boring) process. Instead, you can focus on other key activities.
   - You can include or discard products at runtime.
   - You can create new instances at a cheaper cost.
2. **What are the challenges associated with using prototype design patterns?**
   - Each subclass needs to implement the cloning or copying mechanism.
   - Sometimes creating a copy from an existing instance is not simple. For example, implementing a cloning mechanism can be challenging if the objects under consideration do not support copying/cloning or if there are circular references. For example, in Java, a class with the `clone()` method needs to implement the Cloneable marker interface; otherwise, it will throw a CloneNotSupportedException.
   - In this example, I have used the `clone()` method that performs a shallow copy in Java. Following the convention, I obtained the returned object by calling `super.clone()`.(If you want to learn more about this, put your cursor on the eclipse editor and go through the instructions). If you need a deep copy for your application, that can be expensive.

3. **Can you please elaborate the difference between a shallow copy and a deep copy?**

   A *shallow copy* creates a new object and then copies various field values from the original object to the new object. So, it is also known as a *field-by-field copy*. If the original object contains any references to other objects as fields, then the references of those objects are copied into the new object, (i.e., you do not create the copies of those objects).

   Let's try to understand the mechanism with a simple diagram. Suppose we have an object, `X1`, and it has a reference to another object, `Y1`. Further assume that object `Y1` has a reference to object `Z1`.

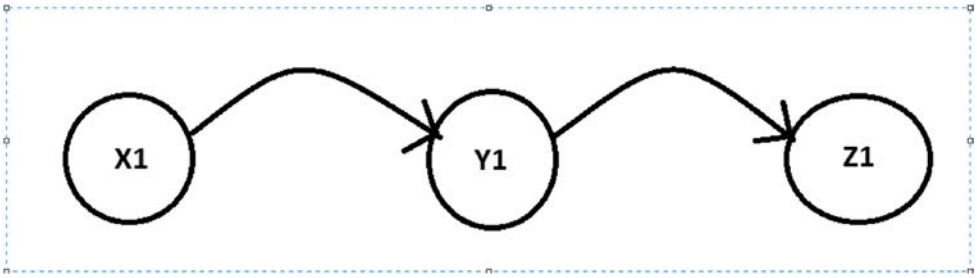


Figure 2-4. *Before shallow copy of the reference/s*

Now, with the shallow copy of `X1`, a new object, `X2`, is created; it also has a reference to `Y1`.

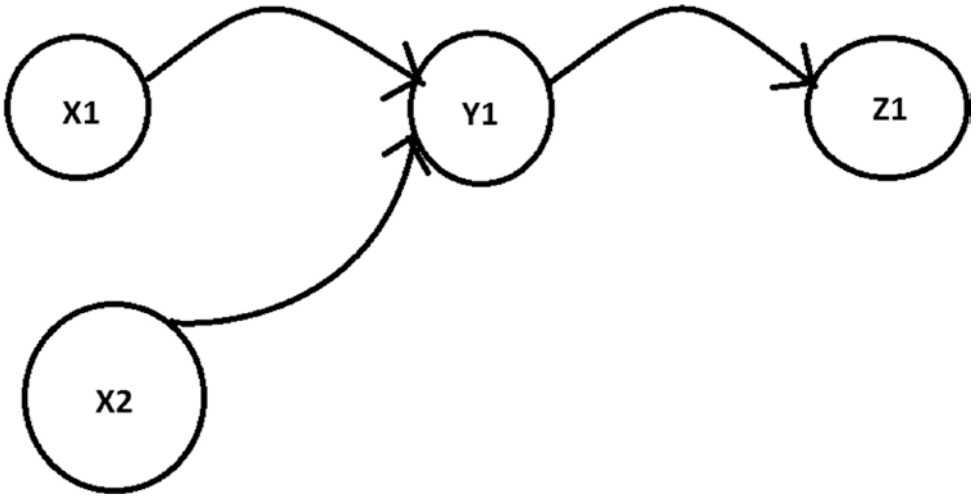


Figure 2-5. *After the shallow copy of the reference*

You have already seen the use of the `clone()` method in our implementation. It performs a shallow copy.

For a *deep copy* of `X1`, a new object, `X3`, is created. `X3` has a reference to new object `Y3`, which is actually a copy of `Y1`. Also, `Y3`, in turn, has a reference to another new object, `Z3`, which is a copy of `Z1`.

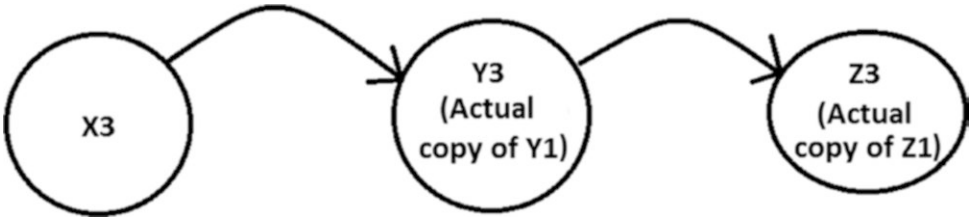


Figure 2-6. *After the deep copy of the reference*

In a deep copy, the new object is totally separated from the original one. Any changes made in one object should not be reflected on the other one. To create a deep copy in Java, you may need to override the `clone()` method and then proceed. Also, a deep copy is expensive because you need to create

additional objects. A complete implementation of deep copy is presented in the "Q&A Session" of Memento Pattern (Chapter 19) in this book.

4. **When do you choose a shallow copy over a deep copy (and vice versa)?**

   A shallow copy is faster and less expensive. It is always better if your target object has the primitive fields only.

   A deep copy is expensive and slow. But it is useful if your target object contains many fields that have references to other objects.

5. **When I copy an object in Java, I need to use the clone() method. Is this understanding correct?**

   No. There are alternatives available, and one of them is to use the serialization mechanism. But you can always define your own copy constructor and use it.

6. **Can you give a simple example that demonstrates a user-defined copy constructor?**

   Java does not support a default copy constructor. You may need to write your own. Consider the following program, which demonstrates such a usage.

Demonstration

Here's the demonstration.

```
package jdp2e.prototype.questions_answers;

class Student
{
    int rollNo;
    String name;
    //Instance Constructor
```

```
    public Student(int rollNo, String name)
    {
        this.rollNo = rollNo;
        this.name = name;
    }
    //Copy Constructor
    public Student( Student student)
    {
        this.name = student.name;
        this.rollNo = student.rollNo;
    }
    public void displayDetails()
    {
        System.out.println(" Student name: " + name + ",Roll no: "+rollNo);
    }
}

class UserDefinedCopyConstructorExample {
    public static void main(String[] args) {
        System.out.println("***User defined copy constructor example in
        Java***\n");
        Student student1 = new Student(1, "John");
        System.out.println(" The details of Student1 is as follows:");
        student1.displayDetails();
        System.out.println("\n Copying student1 to student2 now");
        //Invoking the user-defined copy constructor
        Student student2 = new Student (student1);
        System.out.println(" The details of Student2 is as follows:");
        student2.displayDetails();
    }
}
```

Output

Here's the output.

```
***User defined copy constructor example in Java***

 The details of Student1 is as follows:
 Student name: John,Roll no: 1

 Copying student1 to student2 now
 The details of Student2 is as follows:
 Student name: John,Roll no: 1
```

CHAPTER 3

Builder Pattern

This chapter covers the builder pattern.

GoF Definition

Separate the construction of a complex object from its representation so that the same construction processes can create different representations.

Concept

The builder pattern is useful for creating complex objects that have multiple parts. The creational mechanism of an object should be independent of these parts. The construction process does not care about how these parts are assembled. The same construction process must allow us to create different representations of the objects.

The structure in Figure 3-1 is an example of the builder pattern. The structure is adopted from the Gang of Four book, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

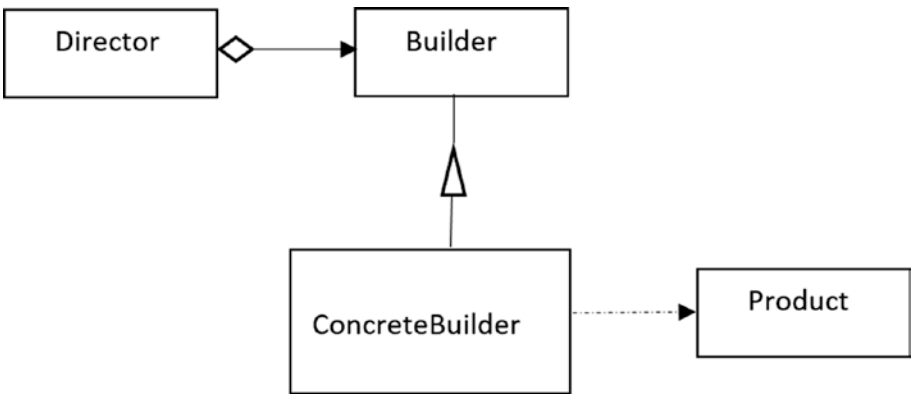


Figure 3-1. *An example of the builder pattern*

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_3

Product is the complex object that you want to create. ConcreteBuilder constructs and assembles the parts of a product by implementing an abstract interface, Builder. The ConcreteBuilder objects build the product's internal representations and define the creational process/assembling mechanisms. Builders can also provide methods to get an object that is created and available for use (notice the `getVehicle()` method in the Builder interface in the following implementation). Director is responsible for creating the final object using the Builder interface. In other words, Director uses Builder and controls the steps/sequence to build the final Product. Builders can also keep reference of the products that they built, so that they can be used again.

Real-World Example

To complete an order for a computer, different parts are assembled based on customer preferences (e.g., one customer can opt for a 500 GB hard disk with an Intel processor, and another customer can choose a 250 GB hard disk with an AMD processor). Consider another example. Suppose that you intend to go on a tour with a travel company that provides various packages for the same tour (for example, they can provide special arrangements, a different kind of vehicle for the sightseeing, etc.). You can choose your package based on your budget.

Computer-World Example

The builder pattern can be used when we want to convert one text format to another text format (e.g., RTF to ASCII text).

Note The Java.util.Calendar.Builder class is an example in this category, but it is available in Java 8 and onward only. The java.lang.StringBuilder class is a close example in this context. But you need to remember that the GoF definition says that this pattern allows you to use the same construction process to make different representations. In this context, this example does not fully qualify for this pattern.

Illustration

In this example, we have the following participants: Builder, Car, MotorCycle, Product, and Director. The first three are very straightforward; Car and MotorCycle are concrete classes and they implement the Builder interface. Builder is used to create parts of the Product object, where Product represents the complex object under construction.

Since Car and MotorCycle are the concrete implementations of the Builder interface, they need to implement the methods that are declared in the Builder interface. That's why they needed to supply the body for the `startUpOperations()`, `buildBody()`, `insertWheels()`, `addHeadlights()`, `endOperations()`, and `getVehicle()`methods. The first five methods are straightforward; they are used to perform an operation at the beginning (or end), build the body of the vehicle, insert the wheels, and add headlights. `getVehicle()` returns the final product. In this case, `Director` is responsible for constructing the final representation of these products using the `Builder` interface. (See the structure defined by the GoF). Notice that `Director` is calling the same `construct()` method to create different types of vehicles.

Now go through the code to see how different parts are assembled for this pattern.

Class Diagram

Figure 3-2 shows the class diagram of the builder pattern.

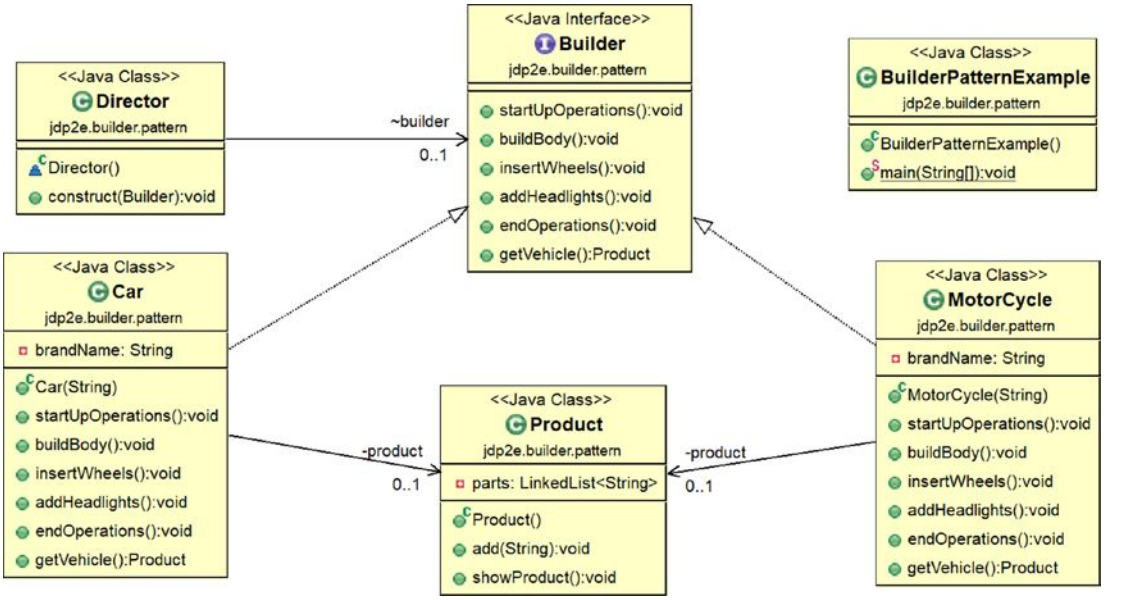


Figure 3-2. *Class diagram*

Package Explorer View

Figure 3-3 shows the high-level structure of the program.

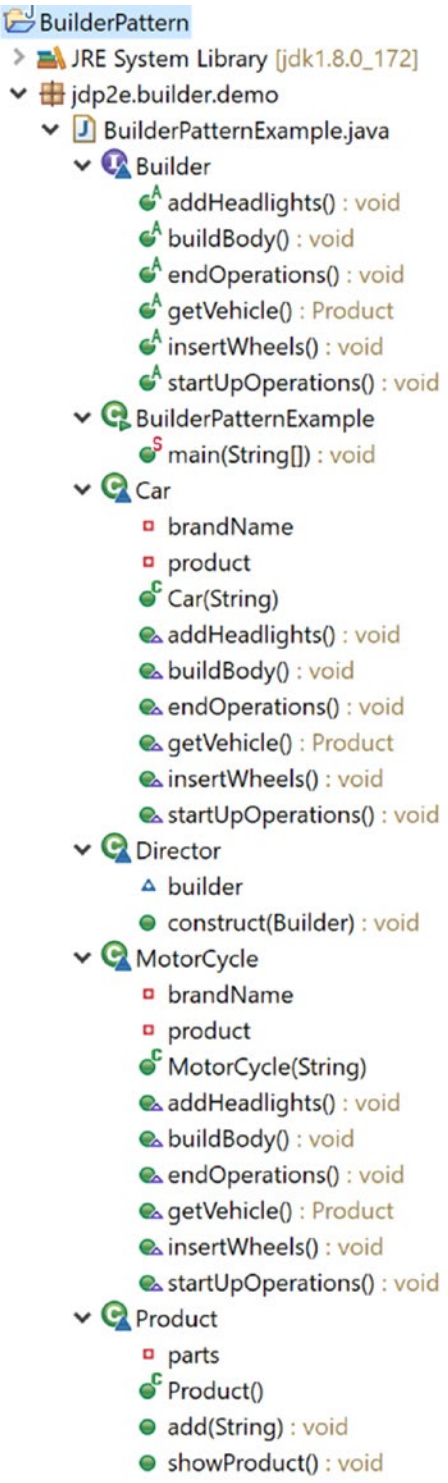


Figure 3-3. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.builder.demo;

import java.util.LinkedList;

//The common interface
interface Builder
{
     void startUpOperations();
     void buildBody();
     void insertWheels();
     void addHeadlights();
     void  endOperations();
     /*The following method is used to retrieve the object that is
     constructed.*/
     Product getVehicle();
}
//Car class
class Car implements Builder
{
    private String brandName;
    private Product product;
    public Car(String brand)
    {
        product = new Product();
        this.brandName = brand;
    }
    public void startUpOperations()
    {
      //Starting with brand name
        product.add(String.format("Car model is :%s",this.brandName));
    }
```

```
    public void buildBody()
    {
        product.add("This is a body of a Car");
    }
    public void insertWheels()
    {
        product.add("4 wheels are added");
    }

    public void addHeadlights()
    {
        product.add("2 Headlights are added");
    }
    public void endOperations()
    {   //Nothing in this case
    }

    public Product getVehicle()
    {
        return product;
    }
}
//Motorcycle class
class MotorCycle implements Builder
{
    private String brandName;
    private Product product;
    public MotorCycle(String brand)
    {
        product = new Product();
        this.brandName = brand;
    }
    public void startUpOperations()
    {   //Nothing in this case
    }
```

```
    public  void buildBody()
    {
        product.add("This is a body of a Motorcycle");
    }

    public void insertWheels()
    {
        product.add("2 wheels are added");
    }

    public void addHeadlights()
    {
        product.add("1 Headlights are added");
    }
    public void endOperations()
    {
        //Finishing up with brand name
        product.add(String.format("Motorcycle model is :%s", this.
        brandName));
    }
    public Product getVehicle()
    {
        return product;
    }
}

// Product class
class Product
{
   /* You can use any data structure that you prefer.
       I have used LinkedList<String> in this case.*/
    private LinkedList<String> parts;
    public Product()
    {
        parts = new LinkedList<String>();
    }
```

```
    public void add(String part)
    {
       //Adding parts
        parts.addLast(part);
    }

    public void showProduct()
    {
      System.out.println("\nProduct completed as below :");
        for (String part: parts)
        System.out.println(part);
    }
}
// Director class
class Director
{
    Builder builder;
    // Director knows how to use the builder and the sequence of steps.
    public void construct(Builder builder)
    {
        this.builder = builder;
        builder.startUpOperations();
        builder.buildBody();
        builder.insertWheels();
        builder.addHeadlights();
        builder.endOperations();
    }
}
public class BuilderPatternExample {

      public static void main(String[] args) {
            System.out.println("***Builder Pattern Demo***");
        Director director = new Director();

        Builder fordCar = new Car("Ford");
        Builder hondaMotorycle = new MotorCycle("Honda");
```

```
        // Making Car
        director.construct(fordCar);
        Product p1 = fordCar.getVehicle();
        p1.showProduct();

        //Making MotorCycle
        director.construct(hondaMotorycle );
        Product p2 = hondaMotorycle.getVehicle();
        p2.showProduct();
      }
}
```

Output

Here's the output.

```
***Builder Pattern Demo***

Product completed as below :
Car model is :Ford
This is a body of a Car
4 wheels are added
2 Headlights are added

Product completed as below :
This is a body of a Motorcycle
2 wheels are added
1 Headlights are added
Motorcycle model is :Honda
```

Q&A Session

1. **What are the advantages of using a builder pattern?**

    - You can create a complex object, step by step, and vary the steps. You promote encapsulation by hiding the details of the complex construction process. The director can retrieve the final product from the builder when the whole construction is over. In general,

at a high level, you seem to have only one method that makes the complete product. Other internal methods only involve partial creation. So, you have finer control over the construction process.

- Using this pattern, the same construction process can produce different products.
- Since you can vary the construction steps, you can vary product's internal representation.

2. **What are the drawbacks/pitfalls associated with the builder pattern?**

   - It is not suitable if you want to deal with mutable objects (which can be modified later).
   - You may need to duplicate some portion of the code. These duplications may have significant impact in some context and turn into an antipattern.
   - A concrete builder is dedicated to a particular type of product. So, to create different type of products, you may need to come up with different concrete builders.
   - The approach makes more sense if the structure is very complex.

3. **Can I use an abstract class instead of the interface in the illustration of this pattern?**

   Yes. You can use an abstract class instead of an interface in this example.

4. **How do I decide whether I should use an abstract class or an interface in an application?**

   I believe that if you want to have some centralized or default behavior, the abstract class is a better choice. In those cases, you can provide some default implementation. On the other hand, interface implementation starts from scratch. They indicate some kind of rules/contracts on *what* is to be done (e.g., you must implement the method) but they will not enforce the *how* part of it. Also, interfaces are preferred when you are trying to implement the concept of multiple inheritance.

But at the same time, if you need to add a new method in an interface, then you need to track down all the implementations of that interface and you need to put the concrete implementation for that method in all of those places. You can add a new method in an abstract class with a default implementation and the existing code can run smoothly.

Java has taken special care with this last point. Java 8 introduced the use of 'default' keyword in the interface. You can prefix the default word before the intended method signature and provide a default implementation. Interface methods are public by default, so you do not need to mark it with the keyword public.

These summarized suggestions are from the Oracle Java documentation at https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html.

You should prefer the abstract class in the following scenarios:

- You want to share code among multiple closely related classes.
- The classes that extend the abstract class can have many common methods or fields, or they require non-public access modifiers inside them.
- You want to use non-static or/and non-final fields, which enables us to define methods that can access and modify the state of the object to which they belong.
- On the other hand, you should prefer interfaces for these scenarios:
- You expect that several unrelated classes are going to implement your interface (e.g., comparable interface can be implemented by many unrelated classes).
- You specify the behavior of a particular data type, but it does not matter how the implementer implements that.
- You want to use the concept of multiple inheritance in your application.

Note In my book *Interactive Object-Oriented Programming in Java* (Apress, 2016), I discussed abstract classes, interfaces, and the use of the "default" keyword with various examples and outputs. Refer to that book for a detailed discussion and analysis.

5. **I am seeing that in cars, model names are added in the beginning, but for motorcycles, model names are added at the end. Is it intentional?**

   Yes. It was used to demonstrate the fact that each of the concrete builders can decide how they want to produce the final products. They have this freedom.

6. **Why are you using a separate class for director? You could use the client code to play the role of the director.**

   No one restricts you to do that. In the preceding implementation, I wanted to separate this role from the client code in the implementation. But in the upcoming/modified implementation, I have used the client as a director.

7. **What do you mean by *client code*?**

   The class that contains the `main()` method is the client code. In most parts of the book, client code means the same.

8. **You mentioned varying steps several times. Can you demonstrate an implementation where the final product is created with different variations and steps?**

   Good catch. You asked for a demonstration of the real power of the builder pattern. So, let us consider another example.

Modified Illustration

Here are the key characteristics of the modified implementation.

- In this modified implementation, I consider only cars as the final products.
- I create custom cars that have the following attributes: a start-up message (`startUpMessage`), a process completion message (`endOperationsMessage`), the body material of the car (`bodyType`), the number of wheels on the car (`noOfWheels`), and the number of headlights (`noOfHeadLights`) on the car.
- The client code is playing the role of a director in this implementation.
- I have renamed the builder interface as ModifiedBuilder. Apart from the `constructCar()` and `getConstructedCar()` methods, each of the methods in the interface has the ModifiedBuilder return type, which helps us to apply method chaining mechanism in the client code.

Modified Package Explorer View

Figure 3-4 shows the modified Package Explorer view.

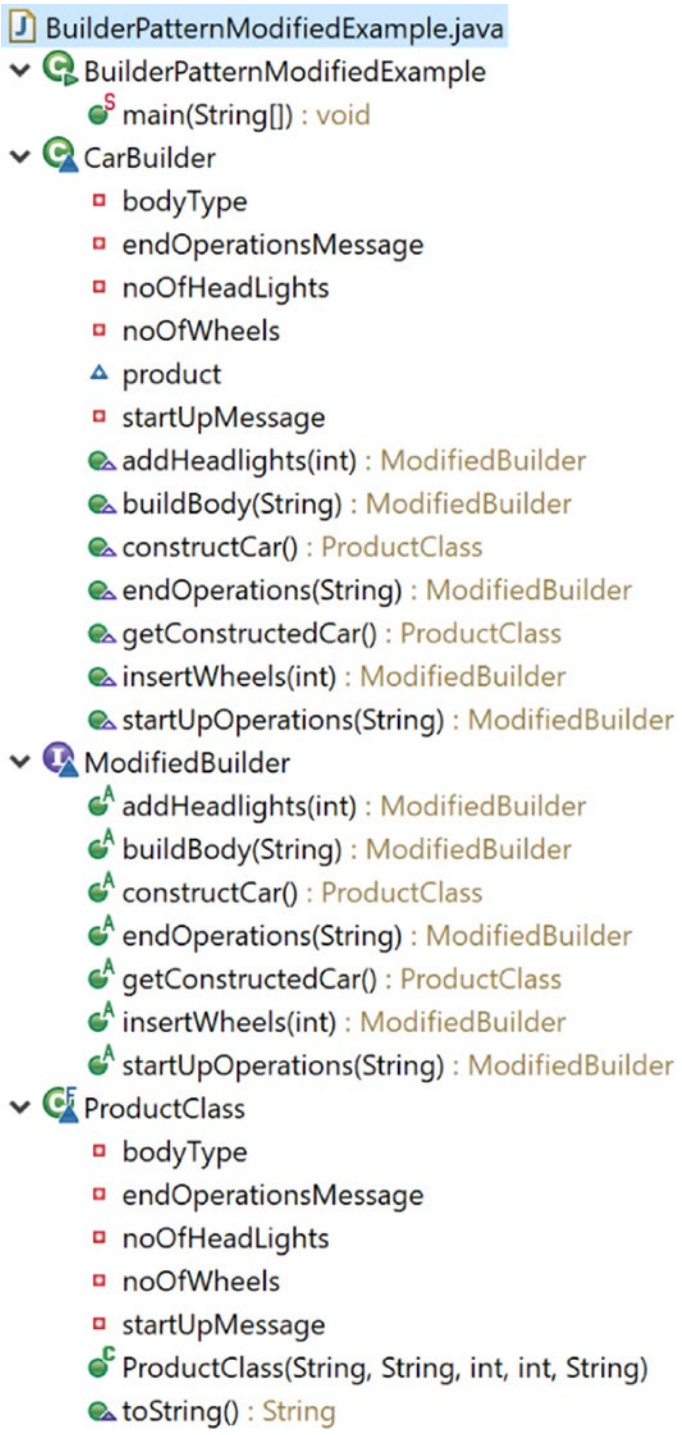


Figure 3-4. *Modified Package Explorer view*

Modified Implementation

Here is the modified implementation.

```
package jdp2e.builder.pattern;

//The common interface
interface ModifiedBuilder
{
	/*All these methods return type is ModifiedBuilder.
	 * This will help us to apply method chaining*/
	ModifiedBuilder startUpOperations(String startUpMessage);
	ModifiedBuilder buildBody(String bodyType);
	ModifiedBuilder insertWheels(int noOfWheels);
	ModifiedBuilder addHeadlights(int noOfHeadLights);
	ModifiedBuilder endOperations(String endOperationsMessage);
	//Combine the parts and make the final product.
	ProductClass constructCar();
	//Optional method:To get the already constructed object
	ProductClass getConstructedCar();
}
//Car class
class CarBuilder implements ModifiedBuilder
{
	private String startUpMessage="Start building the product";//Default
	//start-up message
	private String  bodyType="Steel";//Default body
	private int noOfWheels=4;//Default number of wheels
	private int  noOfHeadLights=2;//Default number of head lights
	//Default finish up message
	private String  endOperationsMessage="Product creation completed";
	ProductClass product;
	@Override
```

```
public ModifiedBuilder startUpOperations(String startUpMessage)
{
      this.startUpMessage=startUpMessage;
      return this;
}

@Override
public ModifiedBuilder buildBody(String bodyType)
{
      this.bodyType=bodyType;
      return this;
}

@Override
public ModifiedBuilder insertWheels(int noOfWheels)
{
      this.noOfWheels=noOfWheels;
      return this;
}

@Override
public ModifiedBuilder addHeadlights(int noOfHeadLights)
{
      this.noOfHeadLights=noOfHeadLights;
      return this;
}
@Override
public ModifiedBuilder endOperations(String endOperationsMessage)
{       this.endOperationsMessage=endOperationsMessage;
return this;
}

@Override
public ProductClass constructCar() {

      product= new ProductClass(this.startUpMessage,this.
      bodyType,this.noOfWheels,this.noOfHeadLights,this.
      endOperationsMessage);
```

```
            return product;
        }

        @Override
        public ProductClass   getConstructedCar()
 {
        return product;
 }
}

//Product class
final class ProductClass
{
        private String startUpMessage;
        private String  bodyType;
        private int noOfWheels;
        private int  noOfHeadLights;
        private String  endOperationsMessage;
        public ProductClass(final String startUpMessage, String bodyType,
        int noOfWheels, int noOfHeadLights,
                    String endOperationsMessage) {
              this.startUpMessage = startUpMessage;
              this.bodyType = bodyType;
              this.noOfWheels = noOfWheels;
              this.noOfHeadLights = noOfHeadLights;
              this.endOperationsMessage = endOperationsMessage;
        }
        /*There is no setter methods used here to promote immutability.
        Since the attributes are private and there is no setter methods, the
        keyword "final" is not needed to attach with the attributes.
         */
        @Override
        public String toString() {
              return "Product Completed as:\n startUpMessage=" +
              startUpMessage + "\n bodyType=" + bodyType + "\n noOfWheels="
```

```
                        + noOfWheels + "\n noOfHeadLights=" +
                        noOfHeadLights + "\n endOperationsMessage=" +
                        endOperationsMessage
                        ;
        }
}
//Director class
public class BuilderPatternModifiedExample {

        public static void main(String[] args) {
              System.out.println("***Modified Builder Pattern Demo***");
              /*Making a custom car (through builder)
                Note the steps:
                Step1:Get a builder object with required parameters
                Step2:Setter like methods are used.They will set the
                optional fields also.
                Step3:Invoke the constructCar() method to get the final car.
               */
              final ProductClass customCar1 = new CarBuilder().
              addHeadlights(5)
                        .insertWheels(5)
                        .buildBody("Plastic")
                        .constructCar();
              System.out.println(customCar1);
              System.out.println("--------------");
              /* Making another custom car (through builder) with a
              different
               * sequence and steps.
               */
              ModifiedBuilder carBuilder2=new CarBuilder();
              final ProductClass customCar2 = carBuilder2.insertWheels(7)
                        .addHeadlights(6)
                        .startUpOperations("I am making my own car")
                        .constructCar();
              System.out.println(customCar2);
```

```
            System.out.println("--------------");

            //Verifying the getConstructedCar() method
            final ProductClass customCar3=carBuilder2.getConstructedCar();
            System.out.println(customCar3);

        }
}
```

Modified Output

Here's the modified output. (Some of the lines are bold to draw your attention to notice the differences in the output).

```
***Modified Builder Pattern Demo***
Product Completed as:
 startUpMessage=Start building the product
 bodyType=Plastic
 noOfWheels=5
 noOfHeadLights=5
 endOperationsMessage=Product creation completed
--------------
Product Completed as:
 startUpMessage=I am making my own car
 bodyType=Steel
 noOfWheels=7
 noOfHeadLights=6
 endOperationsMessage=Product creation completed
--------------
Product Completed as:
 startUpMessage=I am making my own car
 bodyType=Steel
 noOfWheels=7
 noOfHeadLights=6
 endOperationsMessage=Product creation completed
```

Analysis

Note the following lines of code (from the preceding implementation) for the custom cars creation in the client code.

```
System.out.println("***Modified Builder Pattern Demo***");
            /*Making a custom car (through builder)
              Note the steps:
              Step1:Get a builder object with required parameters
              Step2:Setter like methods are used.They will set the
              optional fields also.
              Step3:Invoke the constructCar() method to get the final car.
             */
            final ProductClass customCar1 = new CarBuilder().
            addHeadlights(5)
                   .insertWheels(5)
                   .buildBody("Plastic")
                   .constructCar();
            System.out.println(customCar1);
      System.out.println("--------------");
            /* Making another custom car (through builder) with a
            different
             * sequence and steps.
             */
            ModifiedBuilder carBuilder2=new CarBuilder();
            final ProductClass customCar2 = carBuilder2.insertWheels(7)
             .addHeadlights(6)
             .startUpOperations("I am making my own car")
             .constructCar();
      System.out.println(customCar2);
```

You are using a builder to create multiple objects by varying the builder attributes between calls to the "build" methods; for example, in the first case, you are invoking the `addHeadLights()`, `insertWheels()`, `buildBody()` methods, one by one, through a builder object, and then you are getting the final car (`customCar1`). But in the

second case, when you create another car object (`customCar2`), you are invoking the methods in a different sequence. When you are not invoking any method, the default implementation is provided for you.

9. **I am seeing the use of final keywords in client codes. But you have not used those for ProductClass attributes. What is the reason for that?**

   In the client code, I used the final keywords to promote immutability. But in the ProductClass class, the attributes are already marked with private keywords and there are no setter methods, so these are already immutable.

10. **What is the key benefit of immutable objects?**

    Once constructed, they can be safely shared, and most importantly, they are thread safe, so you save lots in synchronization costs in a multithreaded environment.

11. **When should I consider using a builder pattern?**

    If you need to make a complex object that involves various steps in the construction process, and at the same time, the products need to be immutable, the builder pattern is a good choice.

CHAPTER 4

Factory Method Pattern

This chapter covers the factory method pattern.

GoF Definition

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

POINTS TO REMEMBER

To understand this pattern, I suggest you go to Chapter 24, which covers the simple factory pattern. The simple factory pattern does not fall directly into the Gang of Four design patterns, so I put the discussion of that pattern in Part II of this book. The factory method pattern will make more sense if you start with the simple factory pattern.

Concept

Here you start your development with an abstract creator class (creator) that defines the basic structure of the application. The subclasses that derive from this abstract class perform the actual instantiation process. The concept will make sense to you when you start thinking about the pattern using the following examples.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_4

Real-World Example

Consider a car manufacturing company that produces different models of a car and runs its business well. Based on the model of the car, different parts are manufactured and assembled.

The company should be prepared for changes where customers can opt for better models in the near future. If the company needs to do a whole new setup for a new model, which demands only a few new features, it can hugely impact its profit margin. So, the company should set up the factory in such a way that it can produce parts for the upcoming models also.

Computer-World Example

Suppose that you are building an application that needs to support two different databases, let's say Oracle and SQL Server. So, whenever you insert a data into a database, you create a SQL Server–specific connection (SqlServerConnection) or an Oracle server–specific connection (OracleConnection) and then you can proceed. If you put these codes into `if-else` (or switch) statements, you may need to repeat a lot of code. This kind of code is not easily maintainable because whenever you need to support a new type of connection, you need to reopen your code and place the modifications. A factory method pattern focuses on solving similar problems in application development.

Note Since the simple factory pattern is the simplest form of the factory method pattern, you can consider the same examples here. So, the static getInstance() method of the java.text.NumberFormat class is an example of this category. The createURLStreamHandler(String protocol) of the java.net.URLStreamHandlerFactory interface is another example in this category. You can pass ftp, http, and so forth as different protocols and the method will return a URLStreamHandler for the specific protocol.

Illustration

I am continuing the discussion the simple factory pattern that is covered in Chapter 24. So, I'll try to improve the implementation. For simplicity, I have placed all classes in this implementation in a single file. So, you do not need to create any separate folders for the individual classes. I suggest that you refer to the associated comments for a better understanding.

Class Diagram

Figure 4-1 shows the class diagram of the factory method pattern.

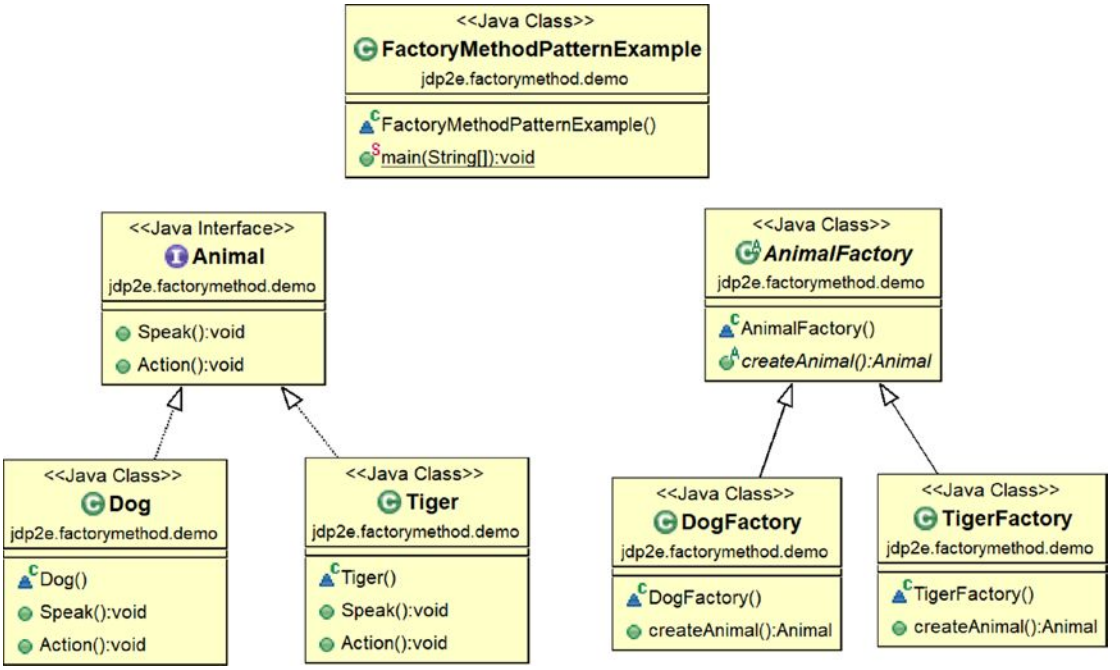


Figure 4-1. *Class diagram*

Package Explorer View

Figure 4-2 shows the high-level structure of the program.

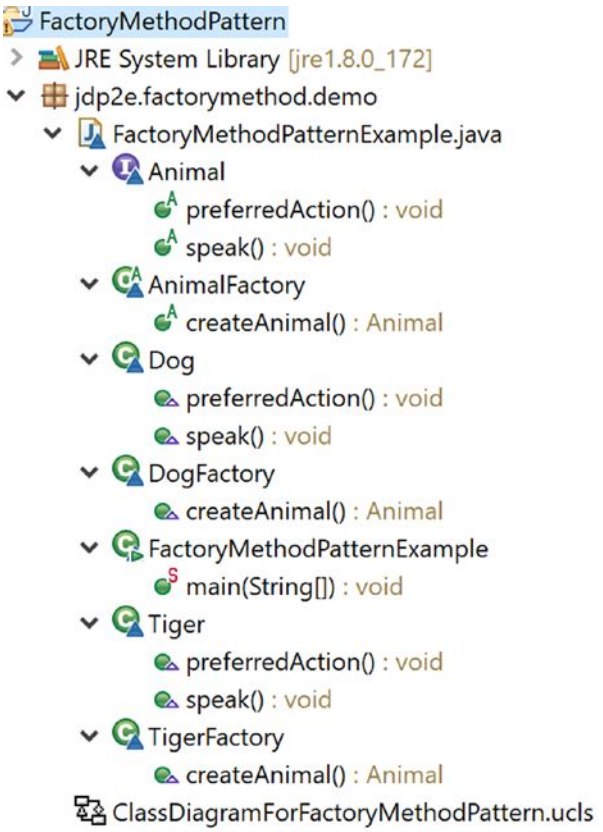


Figure 4-2. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.factorymethod.demo;
interface Animal
{
        void speak();
        void preferredAction();
}
```

```
class Dog implements Animal
{
        public void speak()
        {
                System.out.println("Dog says: Bow-Wow.");
        }
        public void preferredAction()
        {
                System.out.println("Dogs prefer barking...\n");
        }
}
class Tiger implements Animal
{
        public void speak()
        {
                System.out.println("Tiger says: Halum.");
        }
        public void preferredAction()
        {
                System.out.println("Tigers prefer hunting...\n");
        }
}
abstract class AnimalFactory
{
/*Remember that the GoF definition says "....Factory method lets a class
defer instantiation to subclasses."
In our case, the following method will create a Tiger or Dog but at this
point it does not know whether it will get a Dog or a Tiger. This decision
will be taken by the subclasses i.e. DogFactory or TigerFactory. So,in this
implementation, the following method is playing the role of a factory (of
creation)*/
        public abstract Animal createAnimal();
}
```

```
class DogFactory extends AnimalFactory
{
        public Animal createAnimal()
        {
                //Creating a Dog
                return new Dog();
        }
}
class TigerFactory extends AnimalFactory
{
        public Animal createAnimal()
        {
                //Creating a Tiger
                return new Tiger();
        }
}

class FactoryMethodPatternExample {
        public static void main(String[] args) {
                System.out.println("***Factory Pattern Demo***\n");
                // Creating a Tiger Factory
                AnimalFactory tigerFactory =new TigerFactory();
                // Creating a tiger using the Factory Method
                Animal aTiger = tigerFactory.createAnimal();
                aTiger.speak();
                aTiger.preferredAction();

                // Creating a DogFactory
                AnimalFactory dogFactory = new DogFactory();
                // Creating a dog using the Factory Method
                Animal aDog = dogFactory.createAnimal();
                aDog.speak();
                aDog.preferredAction();
        }
}
```

Output

Here's the output.

```
***Factory Pattern Demo***

Tiger says: Halum.
Tigers prefer hunting...

Dog says: Bow-Wow.
Dogs prefer barking...
```

Modified Implementation

In this implementation, the AnimalFactory class is an abstract class. So, let us take advantage of using an abstract class. Suppose that you want a subclass to follow a rule that can be imposed from its parent (or base) class. So, I am testing such a scenario in the following design.

The following are the key characteristics of the design.

- Only AnimalFactory is modified as follows (i.e., I am introducing a new `makeAnimal()` method).

```
//Modifying the AnimalFactory class.
abstract class AnimalFactory
{
    public Animal makeAnimal()
    {
      System.out.println("I am inside makeAnimal() of AnimalFactory.You
      cannot ignore my rules.");
        /*
        At this point, it doesn't know whether it will get a Dog or a
        Tiger. It will be decided by the subclasses i.e.DogFactory or
        TigerFactory.But it knows that it will Speak and it will have a
        preferred way of Action.
        */
        Animal animal = createAnimal();
        animal.speak();
```

```
        animal.preferredAction();
        return animal;
    }

/*Remember that the GoF definition says "....Factory method lets a class
defer instantiation to subclasses."
In our case, the following method will create a Tiger or Dog but at this
point it does not know whether it will get a Dog or a Tiger.
This decision will be taken by the subclasses i.e. DogFactory or
TigerFactory. So,in this implementation, the following method is playing
the role of a factory (of creation)*/
     public abstract Animal createAnimal();
}
```

- Client code has adapted these changes:

```
class ModifiedFactoryMethodPatternExample {
       public static void main(String[] args) {
              System.out.println("***Modified Factory Pattern Demo***\n");
              // Creating a Tiger Factory
              AnimalFactory tigerFactory =new TigerFactory();
              // Creating a tiger using the Factory Method
              Animal aTiger = tigerFactory.makeAnimal();
              //aTiger.speak();
              //aTiger.preferredAction();

              // Creating a DogFactory
              AnimalFactory dogFactory = new DogFactory();
              // Creating a dog using the Factory Method
              Animal aDog = dogFactory.makeAnimal();
              //aDog.speak();
              //aDog.preferredAction();
       }
}
```

Modified Output

Here's the modified output.

```
***Modified Factory Pattern Demo***

I am inside makeAnimal() of AnimalFactory.You cannot ignore my rules.
Tiger says: Halum.
Tigers prefer hunting...

I am inside makeAnimal() of AnimalFactory.You cannot ignore my rules.
Dog says: Bow-Wow.
Dogs prefer barking...
```

Analysis

In each case, you see the message (or warning) *"…You cannot ignore my rules."*

Q&A Session

1. **Why have you separated the CreateAnimal() method from client code?**

   It is my true intention. I want the subclasses to create specialized objects. If you look carefully, you will find that only this "creational part" is varying across the products. I discuss this in detail in the Q&A session on the simple factory pattern (see Chapter 24).

2. **What are the advantages of using a factory like this?**

   - You are separating code that can vary from the code that does not vary (i.e., the advantages of using a simple factory pattern is still present). This technique helps you easily maintain code.
   - Your code is not tightly coupled; so, you can add new classes like Lion, Beer, and so forth, at any time in the system without modifying the existing architecture. So, you have followed the "closed for modification but open for extension" principle.

3. **What are the challenges of using a factory like this?**

   If you need to deal with a large number of classes, then you may encounter maintenance overhead.

4. **I see that the factory pattern is supporting two parallel hierarchies. Is this correct?**

   Good catch. Yes, from the class diagram (see Figure 4-3), it is evident that this pattern supports parallel class hierarchies.

   So, in this example, AnimalFactory, DogFactory, and TigerFactory are placed in one hierarchy, and Animal, Dog, and Tiger are placed in another hierarchy. So, the creators and their creations/products are two hierarchies running in parallel.

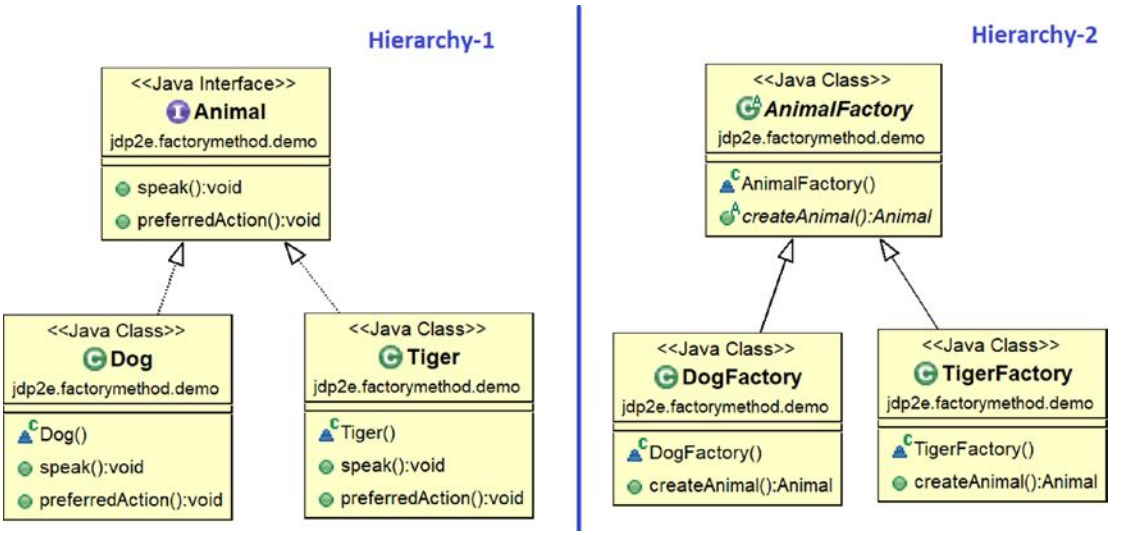


Figure 4-3. *The two class hierarchies in our example*

5. **I should always mark the factory method with an abstract keyword so that subclasses can complete them. Is this correct?**

   No. You may be interested in a default factory method if the creator has no subclasses. And in that case, you cannot mark the factory method with the abstract keyword.

   To show the real power of the factory method pattern, you may need to follow a similar design, which is implemented here.

6. **It still appears to me that the factory method pattern is not much different from simple factory. Is this correct?**

   If you look at the subclasses in the examples in both chapters, you may find some similarities. But you should not forget the key aim of the factory method pattern is that it is supplying the framework through which different subclasses can make different products. But in a simple factory, you cannot vary the products like you can with the factory method pattern. Think of simple factory as a *one-time deal* but most importantly, your creational part will not be closed for modification. Whenever you want to add a new stuff, you need to add an `if/else` block or a switch statement in the factory class of your simple factory pattern.

   In this context, remember the GoF definition: *the factory method lets a class defer instantiation to subclasses.* So, in our simple factory pattern demonstration, you used a concrete class only (`SimpleFactory`). You did not need to override the `createAnimal()` method and there was no subclass that participated in the final decision/product making process. But if you try to *code to an abstract class (or interface)*, that is always considered a good practice, and this mechanism provides you the flexibility to put some common behaviors in the abstract class.

Note In the simple factory pattern, you simply segregate the instantiation logic from client code. In this case, it knows about all the classes whose objects it can create. On the other hand, when using a factory method pattern, you delegate the object creation to subclasses. Also, the factory method is not absolutely sure about the product subclasses in advance.

---

7. **In the factory method pattern, I can simply use a subclassing mechanism (i.e., using inheritance) and then implement the factory method (that is defined in the parent class). Is this correct?**

   The answer to this question is yes if you want to strictly follow the GoF definitions. But it is important to note that in many applications/implementations, there is no use of an abstract class or interface; for example, in Java, an XML reader object is used like this:

```
//Some code before…
XMLReader xmlReader1 = XMLReaderFactory.createXMLReader();
//Some code after
```

   XMLReaderFactory is a final class in Java. So, you cannot inherit from it.

   But when you use SAXPaserFactory, as follows, you are using an abstract class SAXParserFactory.

```
//some code before….
SAXParserFactory factory = SAXParserFactory.newInstance();
              SAXParser parser = factory.newSAXParser();
              XMLReader xmlReader2 = parser.getXMLReader();
//Some code after
```

CHAPTER 5

Abstract Factory Pattern

This chapter covers the abstract factory pattern.

GoF Definition

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Note To better understand this pattern, I suggest that you start at Chapter 24 (simple factory pattern) and then cover Chapter 4 (factory method pattern). The simple factory pattern does not fall directly in the Gang of Four design patterns, so the discussion on that pattern is placed in Part II of this book.

Concept

This is basically a factory of factories that provides one level of higher abstraction than the factory method pattern. This pattern helps us to interchange specific implementations without changing the code that uses them, even at runtime.

This pattern provides a way to encapsulate a group of individual factories that have a common theme. Here a class does not create the objects directly; instead, it delegates the task to a factory object.

The simple factory method pattern creates a set of related objects. In a similar way, since an abstract factory is basically a factory of factories, it returns factories that create a set of related objects. (I discuss the differences in detail in the "Q&A Session" section.)

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_5

Real-World Example

Suppose that we are decorating our room with two different tables: one made of wood and one made of steel. For the wooden table, we need to visit to a carpenter, and for the other table, we need to go to a metal shop. Both are table factories, so based on our demand, we decide what kind of factory we need.

In this context, you may consider two different car manufacturing companies: Honda and Ford. Honda makes models, such as CR-V, Jazz, Brio, and so forth. Ford makes different models, such as Mustang, Figo, Aspire, and so forth. So, if you want to purchase a Jazz, you must visit a Honda showroom, but if you prefer a Figo, you go to a Ford showroom.

Computer-World Example

To understand this pattern, I'll extend the requirement in the factory method pattern. In factory method pattern, we had two factories: one created dogs and the other created tigers. But now, you want to categorize dogs and tigers further. You may choose a domestic animal (dog or tiger) or a wild animal (dog or tiger) through these factories. To fulfil that demand, I introduce two concrete factories: WildAnimalFactory and PetAnimalFactory. The WildAnimalFactory is responsible for creating wild animals and the PetAnimalFactory is responsible for creating domestic animals, or pets.

Note The newInstance() method of javax.xml.parsers.DocumentBuilderFactory is an example of the abstract factory pattern in JDK. The newInstance() method of javax.xml.transform.TransformerFactory is another such example in this context. If you are familiar with C#, you may notice that ADO.NET has already implemented similar concepts to establish a connection to a database.

Illustration

Wikipedia describes a typical structure of the abstract factory pattern (https://en.wikipedia.org/wiki/Abstract_factory_pattern), which is similar to what's shown in Figure 5-1.

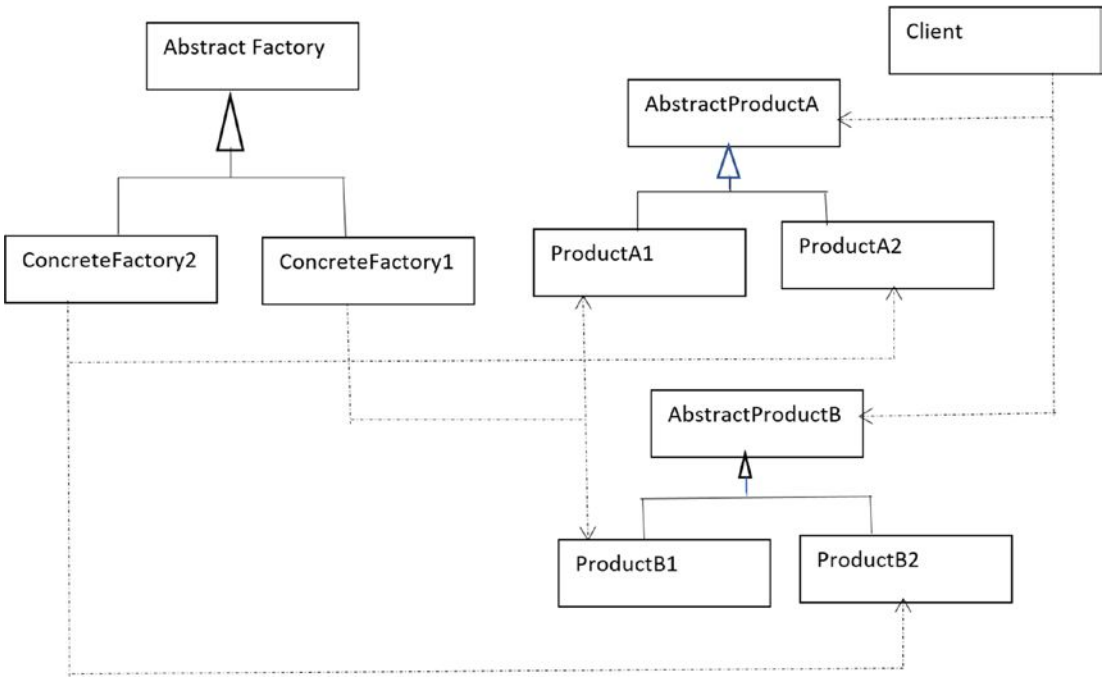


Figure 5-1. *A typical example of an abstract factory pattern*

I am going to follow a similar structure in our implementation. So, in the following implementation, I used two concrete factories: WildAnimalFactory and PetAnimalFactory. They are responsible for the creations of the concrete products, dogs and tigers. WildAnimalFactory creates wild animals (wild dogs and wild tigers) and PetAnimalFactory creates domesticated pet animals (pet dogs and pet tigers). For your ready reference, the participants with their roles are summarized as follows.

- *AnimalFactory*: An interface that is treated as the abstract factory in the following implementation.
- *WildAnimalFactory*: A concrete factory that implements AnimalFactory interface. It creates wild dogs and wild tigers.
- *PetAnimalFactory*: Another concrete factory that implements the AnimalFactory interface. It creates pet dogs and pet tigers.
- *Tiger* and *Dog*: Two interfaces that are treated as abstract products in this example.
- *PetTiger*, *PetDog*, *WildTiger*, and *WildDog*: The concrete products in the following implementation.

Here the client code is looking for animals (dogs and tigers). A common usage of this pattern is seen when we compose classes using the concrete instances of the abstract factory. I have followed the same. Notice that the client class contains the composed implementation of AnimalFactory. You can explore the construction process of both pet and wild animals in the following implementation.

Class Diagram

Figure 5-2 shows the class diagram.

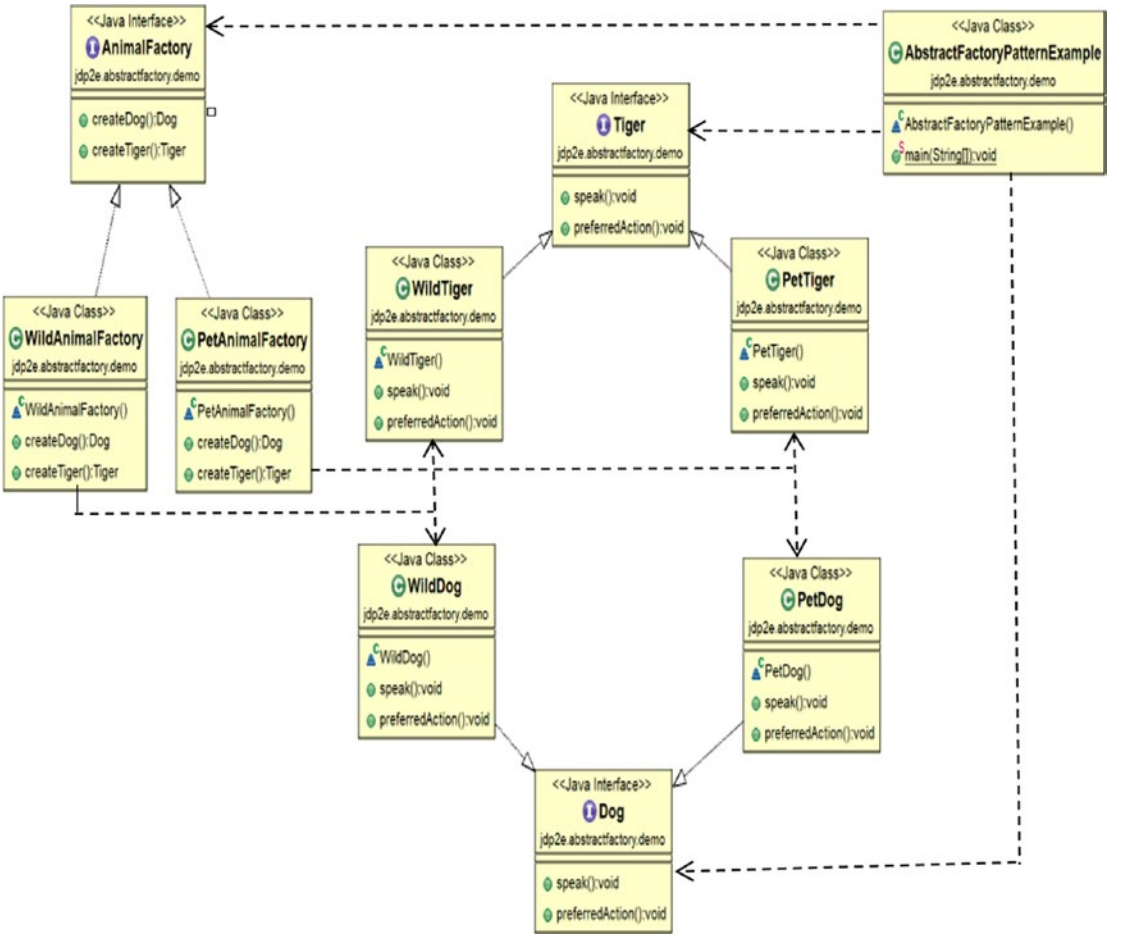


Figure 5-2. *Class diagram*

Package Explorer View

Figure 5-3 shows the high-level structure of the program.

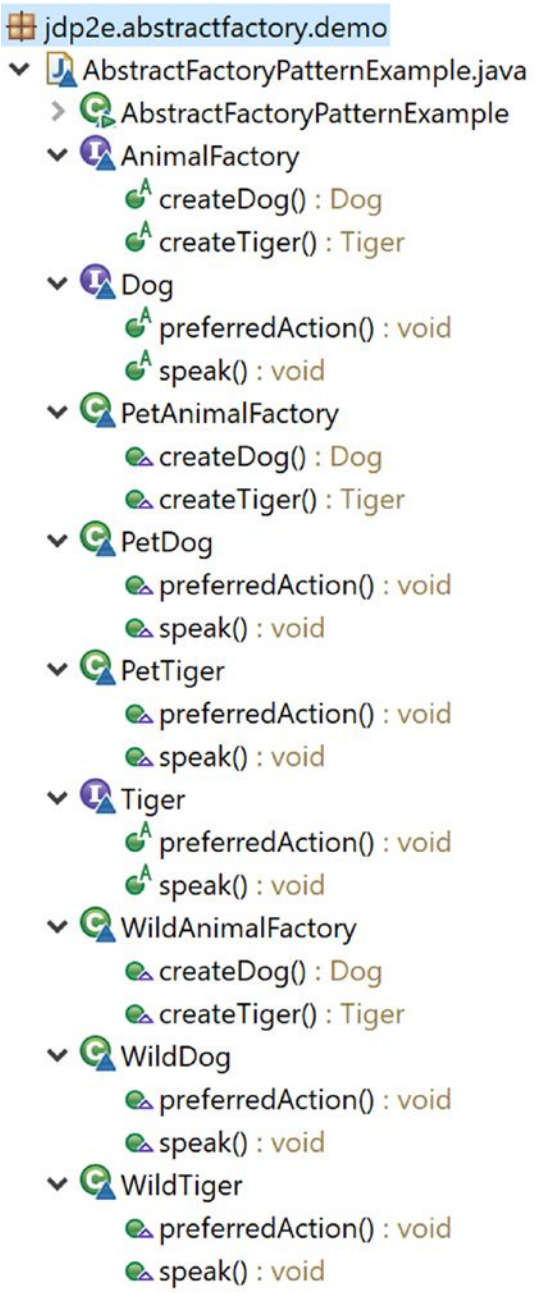


Figure 5-3. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.abstractfactory.demo;

interface Dog
{
      void speak();
      void preferredAction();
}

interface Tiger
{
      void speak();
      void preferredAction();
}

//Types of Dogs-wild dogs and pet dogs
class WildDog implements Dog
{
      @Override
      public void speak()
      {
            System.out.println("Wild Dog says loudly: Bow-Wow.");

      }
      @Override
      public void preferredAction()
      {
            System.out.println("Wild Dogs prefer to roam freely in
            jungles.\n");

      }
}
class PetDog implements Dog
{
      @Override
      public void speak()
```

```
	{
		System.out.println("Pet Dog says softly: Bow-Wow.");
	}
	@Override
	public void preferredAction()
	{
		System.out.println("Pet Dogs prefer to stay at home.\n");
	}
}
//Types of Tigers-wild tigers and pet tigers
class WildTiger implements Tiger
{
	@Override
	public void speak()
	{
		System.out.println("Wild Tiger says loudly: Halum.");
	}
	@Override
	public void preferredAction()
	{
		System.out.println("Wild Tigers prefer hunting in jungles.\n");
	}
}
class PetTiger implements Tiger
{
	@Override
	public void speak()
	{
		System.out.println("Pet Tiger says softly: Halum.");
	}
```

```
	@Override
	public void preferredAction()
	{
		System.out.println("Pet Tigers play in the animal circus.\n");
	}
}

//Abstract Factory
interface AnimalFactory
{
	Dog createDog();
	Tiger createTiger();
}
//Concrete Factory-Wild Animal Factory
class WildAnimalFactory implements AnimalFactory
{
	@Override
	public Dog createDog()
	{
		return new WildDog();
	}
	@Override
	public Tiger createTiger()
	{
		return new WildTiger();
	}

}
//Concrete Factory-Pet Animal Factory
class PetAnimalFactory implements AnimalFactory
{
	@Override
	public Dog createDog()
	{
		return new PetDog();
	}
```

```
	@Override
	public Tiger createTiger()
	{
		return new PetTiger();
	}
}
//Client
class AbstractFactoryPatternExample {

	public static void main(String[] args) {
		AnimalFactory myAnimalFactory;
		Dog myDog;
		Tiger myTiger;
		System.out.println("***Abstract Factory Pattern Demo***\n");
		//Making a wild dog through WildAnimalFactory
		myAnimalFactory = new WildAnimalFactory();
		myDog = myAnimalFactory.createDog();
		myDog.speak();
		myDog.preferredAction();
		//Making a wild tiger through WildAnimalFactory
		myTiger = myAnimalFactory.createTiger();
		myTiger.speak();
		myTiger.preferredAction();

		System.out.println("******************");

		//Making a pet dog through PetAnimalFactory
		myAnimalFactory = new PetAnimalFactory();
		myDog = myAnimalFactory.createDog();
		myDog.speak();
		myDog.preferredAction();
		//Making a pet tiger through PetAnimalFactory
		myTiger = myAnimalFactory.createTiger();
		myTiger.speak();
		myTiger.preferredAction();

	}

}
```

Output

Here's the output.

```
***Abstract Factory Pattern Demo***

Wild Dog says loudly: Bow-Wow.
Wild Dogs prefer to roam freely in jungles.

Wild Tiger says loudly: Halum.
Wild Tigers prefer hunting in jungles.

******************
Pet Dog says softly: Bow-Wow.
Pet Dogs prefer to stay at home.

Pet Tiger says softly: Halum.
Pet Tigers play in the animal circus.
```

Q&A Session

1. **I am seeing that both the dog and the tiger interfaces contain methods that have the same names (both interfaces contain the speak() and the preferredAction() methods. Is it mandatory?**

   No. You can use different names for your methods. Also, the number of methods can be different in these interfaces. But I covered a simple factory pattern and factory method pattern in this book. You may be interested in the similarities or the differences between them. So, I started with an example and keep modifying it. This is why I kept both the `speak()` and `preferredAction()` methods in this example. Notice that these methods are used in both the simple factory pattern (see Chapter 24) and the factory method pattern (see Chapter 4).

2. **What are the challenges of using an abstract factory like this?**

   - Any change in the abstract factory forces us to propagate the modification of the concrete factories. If you follow the design philosophy that says *program to an interface, not to an implementation*, you need to prepare for this. This is one of the key principles that developers always keep in mind. In most scenarios, developers do not want to change their abstract factories.
   - The overall architecture may look complex. Also, debugging becomes tricky in some scenarios.

3. **How can you distinguish a simple factory pattern from a factory method pattern or an abstract factory pattern?**

   I discussed the differences between a simple factory pattern and factory method pattern in the "Q&A Session" section of Chapter 4.

   Let's revise all three factories with the following diagrams.

Simple Factory Pattern Code Snippet

Here's the code snippet.

```
Animal preferredType=null;
SimpleFactory simpleFactory = new SimpleFactory();
// The code that will vary based on users preference.
preferredType = simpleFactory.createAnimal();
```

Figure 5-4 shows how to get animal objects in the Simple Factory pattern.

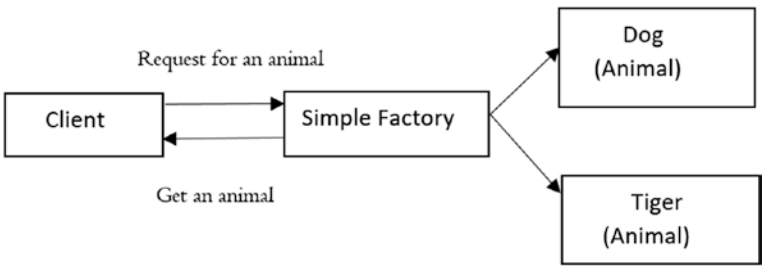


Figure 5-4. *Simple factory pattern*

Factory Method Pattern Code Snippet

Here's the code snippet.

```
// Creating a Tiger Factory
AnimalFactory tigerFactory =new TigerFactory();
// Creating a tiger using the Factory Method
Animal aTiger = tigerFactory.createAnimal();

//...Some code in between...


// Creating a DogFactory
AnimalFactory dogFactory = new DogFactory();
// Creating a dog using the Factory Method
Animal aDog = dogFactory.createAnimal();
```

Figure 5-5 shows how to get animal objects in the factory method pattern.

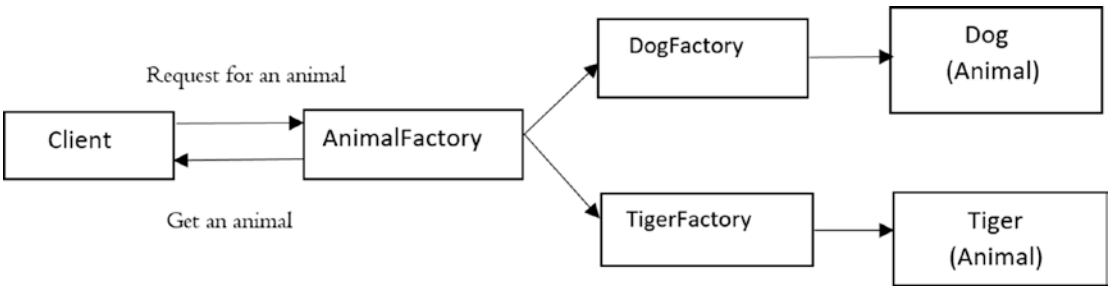


Figure 5-5. *Factory method pattern*

Abstract Factory Pattern Code Snippet

Here's the code snippet.

```
AnimalFactory myAnimalFactory;
Dog myDog;
Tiger myTiger;
System.out.println("***Abstract Factory Pattern Demo***\n");
//Making a wild dog through WildAnimalFactory
myAnimalFactory = new WildAnimalFactory();
myDog = myAnimalFactory.createDog();
```

```
//Making a wild tiger through WildAnimalFactory
myTiger = myAnimalFactory.createTiger();

//Making a pet dog through PetAnimalFactory
myAnimalFactory = new PetAnimalFactory();
myDog = myAnimalFactory.createDog();
//Making a pet tiger through PetAnimalFactory
myTiger = myAnimalFactory.createTiger();
myTiger.speak();
myTiger.preferredAction();
```

Figure 5-6 shows how to get animal objects in the abstract factory method pattern.

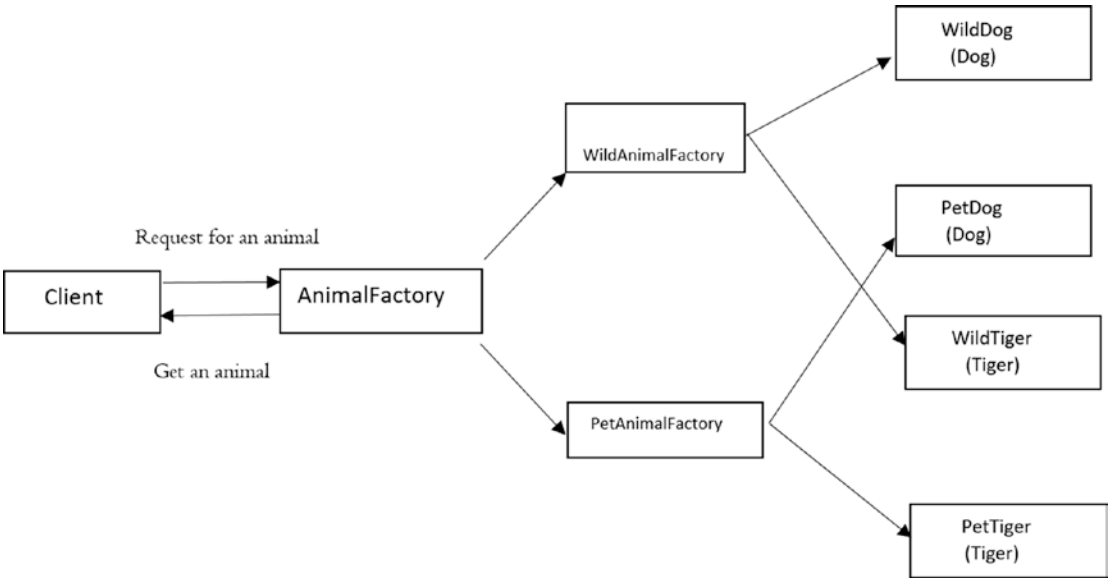


Figure 5-6. *Abstract factory method pattern*

Conclusion

With simple factory, you can separate the code that varies from the rest of the code (basically, you decouple the client codes). This approach helps you easily manage your code. Another key advantage of this approach is that the client is unaware of how the objects are created. So, it promotes both security and abstraction. But it can violate the open-close principle.

You can overcome this drawback using the factory method pattern that allows subclasses to decide how the instantiation process is completed. In other words, you delegate the objects creation to the subclasses that implement the factory method to create objects.
The abstract factory is basically a factory of factories. It creates the family of related objects but it does not depend on the concrete classes.
I tried to maintain simple examples that were close to each other. The factory method promotes inheritance; their subclasses need to implement the factory method to create objects. The abstract factory pattern promotes object composition, where you compose classes using the concrete instances of an abstract factory.
Each of these factories promote loose coupling by reducing the dependencies on concrete classes.

4. **In all of these factory examples, you avoid the use of parameterized constructors. Was this intentional?**

   In many applications, you see the use of parameterised constructors; many experts prefer this approach. But my focus is purely on design, and so, I ignored the use of parameterised constructors. But if you are a fan of parameterized constructors, let's modify the implementation slightly so that you can do the same for the remaining parts.

Modified Illustration

Let's assume that you want your factories to initialize tigers with specified colors, and the client can choose these colors.

Let's modify the following pieces of code (changes are shown in bold).

Modified Implementation

Here's the modified implementation.

```
package jdp2e.abstractfactory.questions_answers;

interface Dog
{
      void speak();
      void preferredAction();
}
```

```
interface Tiger
{
      void speak();
      void preferredAction();
}

//Types of Dogs-wild dogs and pet dogs
class WildDog implements Dog
{
      @Override
      public void speak()
      {
            System.out.println("Wild Dog says loudly: Bow-Wow.");

      }
      @Override
      public void preferredAction()
      {
            System.out.println("Wild Dogs prefer to roam freely in
            jungles.\n");

      }
}
class PetDog implements Dog
{
      @Override
      public void speak()
      {
            System.out.println("Pet Dog says softly: Bow-Wow.");
      }
      @Override
      public void preferredAction()
      {
            System.out.println("Pet Dogs prefer to stay at home.\n");

      }
}
```

```
//Types of Tigers-wild tigers and pet tigers
class WildTiger implements Tiger
{
      public WildTiger(String color)
      {
            System.out.println("A wild tiger with "+ color+ " is
            created.");
      }
      @Override
      public void speak()
      {
            System.out.println("Wild Tiger says loudly: Halum.");
      }
      @Override
      public void preferredAction()
      {
            System.out.println("Wild Tigers prefer hunting in jungles.\n");
      }
}
class PetTiger implements Tiger
{
      public PetTiger(String color)
      {
            System.out.println("A pet tiger with "+ color+ " is created.");
      }
      @Override
      public void speak()
      {
            System.out.println("Pet Tiger says softly: Halum.");
      }
      @Override
      public void preferredAction()
```

```
	{
		System.out.println("Pet Tigers play in the animal circus.\n");
	}
}

//Abstract Factory
interface AnimalFactory
{
	Dog createDog();
	Tiger createTiger(String color);
}
//Concrete Factory-Wild Animal Factory
class WildAnimalFactory implements AnimalFactory
{
	@Override
	public Dog createDog()
	{
		return new WildDog();
	}
	@Override
	public Tiger createTiger(String color)
	{
		return new WildTiger(color);
	}

}
//Concrete Factory-Pet Animal Factory
class PetAnimalFactory implements AnimalFactory
{
	@Override
	public Dog createDog()
	{
		return new PetDog();
	}
	@Override
	public Tiger createTiger(String color)
```

```
    {
        return new PetTiger(color);
    }
}
//Client

class AbstractFactoryPatternModifiedExample {
        public static void main(String[] args) {
          AnimalFactory myAnimalFactory;
          Dog myDog;
          Tiger myTiger;
          System.out.println("***Abstract Factory Pattern Demo***\n");
          //Making a wild dog through WildAnimalFactory
          myAnimalFactory = new WildAnimalFactory();
          myDog = myAnimalFactory.createDog();
          myDog.speak();
          myDog.preferredAction();
          //Making a wild tiger through WildAnimalFactory
          //myTiger = myAnimalFactory.createTiger();
          myTiger = myAnimalFactory.createTiger("white and black stripes");
          myTiger.speak();
          myTiger.preferredAction();

          System.out.println("******************");

          //Making a pet dog through PetAnimalFactory
          myAnimalFactory = new PetAnimalFactory();
          myDog = myAnimalFactory.createDog();
          myDog.speak();
          myDog.preferredAction();
          //Making a pet tiger through PetAnimalFactory
          myTiger = myAnimalFactory.createTiger("golden and cinnamon
          stripes");
          myTiger.speak();
          myTiger.preferredAction();

    }

}
```

Modified Output

Here's the modified output.

```
***Abstract Factory Pattern Demo***

Wild Dog says loudly: Bow-Wow.
Wild Dogs prefer to roam freely in jungles.

A wild tiger with white and black stripes is created.
Wild Tiger says loudly: Halum.
Wild Tigers prefer hunting in jungles.

******************

Pet Dog says softly: Bow-Wow.
Pet Dogs prefer to stay at home.

A pet tiger with golden and cinnamon stripes is created.
Pet Tiger says softly: Halum.
Pet Tigers play in the animal circus.
```

CHAPTER 6

Proxy Pattern

This chapter covers the proxy pattern.

GoF Definition

Provide a surrogate or placeholder for another object to control access to it.

Concept

A proxy is basically a substitute for an intended object. Access to the original object is not always possible due to many factors. For example, it is expensive to create, it is in need of being secured, it resides in a remote location, and so forth. The proxy design pattern helps us in similar contexts. When a client deals with a proxy object, it assumes that it is talking to the actual object. So, in this pattern, you may want to use a class that can perform as an interface to something else.

Real-World Example

In a classroom, when a student is absent, his best friend may try to mimic his voice during roll call to try to get attendance for his friend.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_6

Computer-World Example

In the programming world, to create multiple instances of a complex object (heavy object) is costly . So, whenever you are in need, you can create multiple proxy objects that point to the original object. This mechanism can also help save your system/application memory. An ATM can implement this pattern to hold proxy objects for bank information that may exist on a remote server.

Note In the java.lang.reflect package, you can have a Proxy class and an InvocationHandler interface that supports a similar concept. The *java.rmi.** package also provides methods through which an object on one Java virtual machine can invoke methods on an object that resides in a different Java virtual machine.

Illustration

In the following program, I am calling the `doSomework()` method of the proxy object, which in turn, calls the `doSomework()` method of an object of ConcreteSubject. When clients see the output, they do not know that the proxy object does the trick.

Class Diagram

Figure 6-1 shows the class diagram.

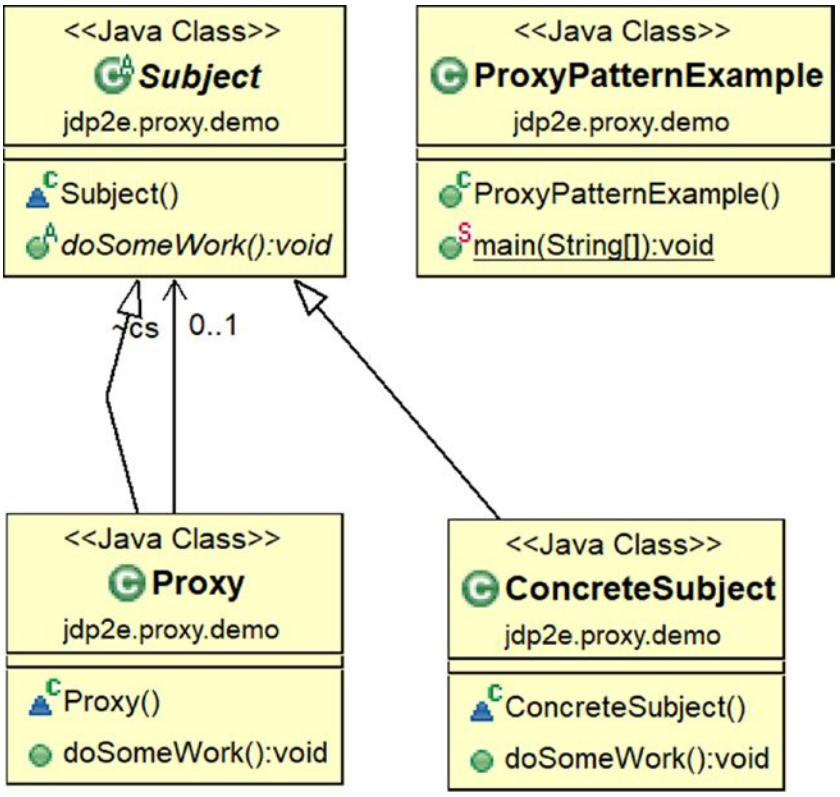


Figure 6-1. *Class diagram*

Package Explorer View

Figure 6-2 shows the high-level structure of the program.

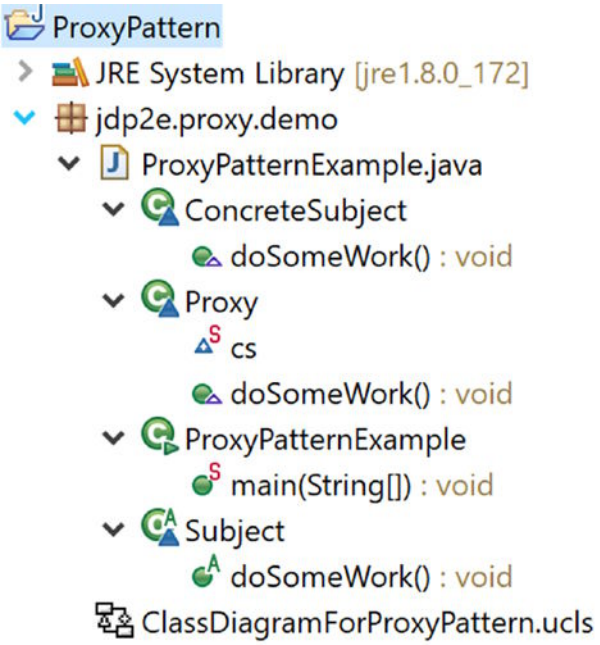


Figure 6-2. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.proxy.demo;

// Abstract class Subject
abstract class Subject
{
      public abstract void doSomeWork();
}
// ConcreteSubject class
class ConcreteSubject extends Subject
{
      @Override
```

```
	public void doSomeWork()
	{
		System.out.println("doSomeWork() inside ConcreteSubject is
		invoked.");
	}
}

/**
 * Proxy Class: It will try to invoke the doSomeWork()
 * of a ConcreteSubject instance
 */
Class Proxy extends  Subject
{
	static Subject cs;
	@Override
	public void doSomeWork()
	{
		System.out.println("Proxy call happening now...");
		//Lazy initialization:We'll not instantiate until the method is
		//called
		if (cs == null)
		{
			cs = new ConcreteSubject();
		}
		cs.doSomeWork();

	}
}

/**
 * The client is talking to a ConcreteSubject instance
 * through a proxy method.
 */
```

```
public class ProxyPatternExample {
      public static void main(String[] args) {
            System.out.println("***Proxy Pattern Demo***\n");
            Proxy px = new Proxy();
            px.doSomeWork();

      }
}
```

Output

Here's the output.

```
***Proxy Pattern Demo***

Proxy call happening now...
doSomeWork() inside ConcreteSubject is invoked.
```

Q&A Session

1. **What are the different types of proxies?**

   These are the common types:

   - *Remote proxies*. Hide the actual object that stays in a different address space.
   - *Virtual proxies*. Perform optimization techniques, such as the creation of a heavy object on a demand basis.
   - *Protection proxies*. Deal with different access rights.
   - *Smart reference*. Performs additional housekeeping work when an object is accessed by a client. A typical operation is counting the number of references to the actual object at a particular moment.

2. **You could create the ConcreteSubject instance in the proxy class constructor, as follows.**

```
class Proxy extends  Subject
{
      static Subject cs;
      public Proxy()
      {
            //Instantiating inside the constructor
            cs = new ConcreteSubject();
      }

      @Override
      public void doSomeWork()
      {
            System.out.println("Proxy call happening now...");
            cs.doSomeWork();
      }
}
```

Is this correct?

Yes, you could do that. But if you follow this design, whenever you instantiate a proxy object, you need to instantiate an object of the ConcreteSubject class also. So, this process may end up creating unnecessary objects. You can simply test this with the following piece of code and the corresponding outputs.

Alternate Implementation

Here's the alternative implementation.

```
package jdp2e.proxy.questions_answers;

//Abstract class Subject
abstract class Subject
{
      public abstract void doSomeWork();
}
```

```
//ConcreteSubject class
class ConcreteSubject extends Subject
{
      @Override
      public void doSomeWork()
      {
            System.out.println("doSomeWork() inside ConcreteSubject is
            invoked");
      }
}

/**
 * Proxy Class
 * It will try to invoke the doSomeWork() of a ConcreteSubject instance *
 */
class Proxy extends  Subject
{
      static Subject cs;
      static int count=0;//A counter to track the number of instances
      public Proxy()
      {
            //Instantiating inside the constructor
            cs = new ConcreteSubject();
            count ++;
      }

      @Override
      public void doSomeWork()
      {
            System.out.println("Proxy call happening now...");
            //Lazy initialization:We'll not instantiate until the method is
            //called
            /*if (cs == null)
```

```
            {
                  cs = new ConcreteSubject();
                  count ++;
             }*/
             cs.doSomeWork();
      }
}

/**
  * The client is talking to a ConcreteSubject instance
 * through a proxy method.
 */
public class ProxyPatternQuestionsAndAnswers {
      public static void main(String[] args) {
             System.out.println("***Proxy Pattern Demo without lazy
             instantiation***\n");
             //System.out.println("***Proxy Pattern Demo with lazy
             instantiation***\n");
             Proxy px = new Proxy();
             px.doSomeWork();
             //2nd proxy instance
             Proxy px2 = new Proxy();
             px2.doSomeWork();

             System.out.println("Instance Count="+Proxy.count);
      }
}
```

Output Without Lazy Instantiation

Here's the output.

```
***Proxy Pattern Demo without lazy instantiation***

Proxy call happening now...
doSomeWork() inside ConcreteSubject is invoked
```

```
Proxy call happening now...
doSomeWork() inside ConcreteSubject is invoked
Instance Count=2
```

Analysis

Notice that you have created two proxy instances.
Now, try our earlier approach with lazy instantiation. (Remove the proxy constructor and uncomment the lazy instantiation stuffs).

Output with Lazy Instantiation

Here's the output.

```
***Proxy Pattern Demo with lazy instantiation***

Proxy call happening now...
doSomeWork() inside ConcreteSubject is invoked
Proxy call happening now...
doSomeWork() inside ConcreteSubject is invoked
Instance Count=1
```

Analysis

Notice that you have created only one proxy instance this time.

3. **But in this lazy instantiation technique, you may create unnecessary objects in a multithreaded application. Is this correct?**

   Yes. In this book, I am presenting simple illustrations only, so I have ignored that part. In the discussions on the singleton pattern, I analyzed some alternative approaches to deal with a multithreaded environment. You can always refer to those discussions in situations like this. (For example, in this particular scenario, you can implement a synchronization technique, or a locking mechanism, or a smart proxy, and so forth to ensure that a particular object is locked before you grant access to the object.)

4. **Can you give an example of a remote proxy?**

   Suppose, you want to call a method of an object but the object is running in a different address space (e.g., different locations or different computers, etc.). How do you proceed? With the help of remote proxies, you can call the method on the proxy object, which in turn forwards the call to the actual object that is running on the remote machine. This type of need can be realized through well-known mechanisms like ASP.NET, CORBA, C#'s WCF (version 3.0 onward), or Java's RMI (Remote Method Invocation).

   Figure 6-3 demonstrates a simple remote proxy structure.

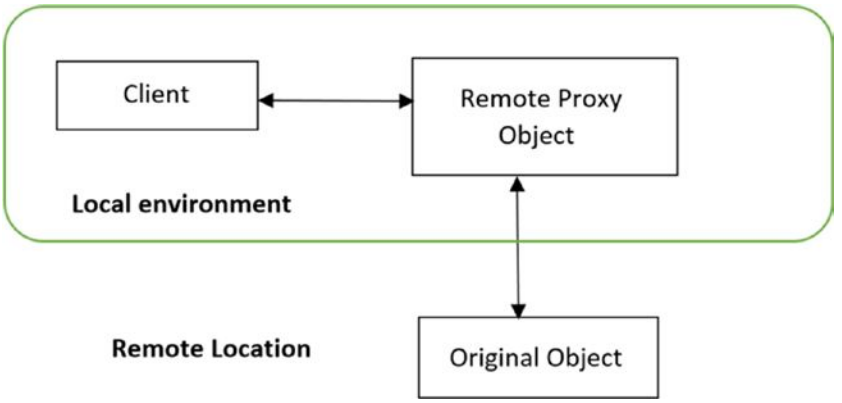


Figure 6-3. *A simple remote proxy diagram*

5. **When can you use a virtual proxy?**

   It can be used to avoid multiple loadings of an extremely large image.

6. **When can you use a protection proxy?**

   The security team in an organization can implement a protection proxy to block Internet access to specific websites.

   Consider the following example, which is basically a modified version of the proxy pattern implementation described earlier. For simplicity, let's assume that at present, we have only three registered users who can exercise the `doSomeWork()` proxy method. Apart from them, if any other user (say, Robin) tries to

invoke the method, the system will reject those attempts. You must agree, when the system will reject this kind of unwanted access; there is no point in making a proxy object. So, if you avoid instantiating an object of ConcreteSubject in the Proxy class constructor, you can easily avoid these kinds of additional objects creation.

Now go through the modified implementation.

Modified Package Explorer View

Figure 6-4 shows the modified high-level structure of the program.

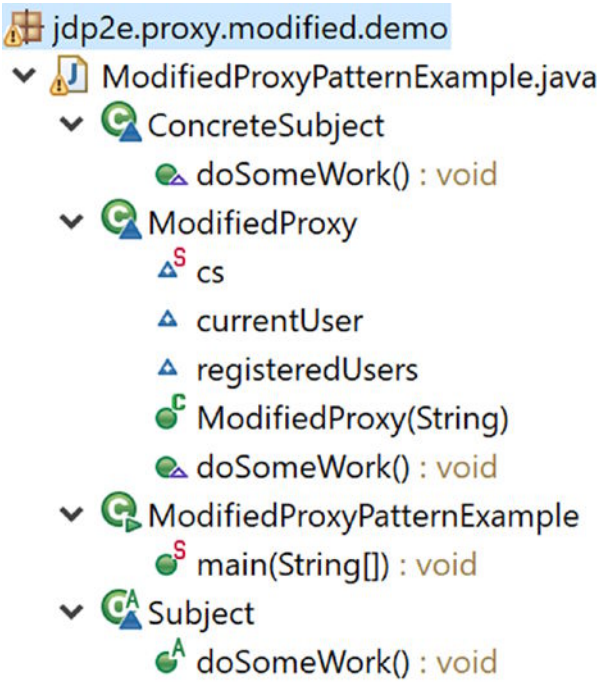


Figure 6-4. *Modified Package Explorer view*

Modified Implementation

Here's the modified implementation.

```
package jdp2e.proxy.modified.demo;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

//Abstract class Subject
abstract class Subject
{
	public abstract void doSomeWork();
}
//ConcreteSubject class
class ConcreteSubject extends Subject
{
	@Override
	public void doSomeWork()
	{
		System.out.println("doSomeWork() inside ConcreteSubject is
		invoked.");
	}
}

/**
 * Proxy Class:It will try to invoke the doSomeWork()
 * of a ConcreteSubject instance
 */
class ModifiedProxy extends  Subject
{
	static Subject cs;
	String currentUser;
	List<String> registeredUsers;
	//Or, simply create this mutable list in one step
```

```
        /*List<String> registeredUsers=new ArrayList<String>(Arrays.asList(
        "Admin","Rohit","Sam"));*/
        public ModifiedProxy(String currentUser)
        {
                //Registered users are Admin, Rohit and Sam only.
                registeredUsers = new ArrayList<String>();
                registeredUsers.add("Admin");
                registeredUsers.add("Rohit");
                registeredUsers.add("Sam");
                this.currentUser = currentUser;
        }
        @Override
        public void doSomeWork()
        {
                System.out.println("\n Proxy call happening now...");
                System.out.println(currentUser+" wants to invoke a proxy
                method.");
                if (registeredUsers.contains(currentUser))
                {
                        //Lazy initialization:We'll not instantiate until the
                        //method is called
                        if (cs == null)
                        {
                                cs = new ConcreteSubject();
                        }
                        //Allow the registered user to invoke the method
                        cs.doSomeWork();
                }
                else
                {
                        System.out.println("Sorry "+ currentUser+ " , you do
                        not have access rights.");
                }
        }
}
```

```
/**
 * The client is talking to a ConcreteSubject instance
 * through a proxy method.
 */
public class ModifiedProxyPatternExample {

	public static void main(String[] args) {
		System.out.println("***Modified Proxy Pattern Demo***\n");
		//Admin is an authorized user
		ModifiedProxy px1 = new ModifiedProxy("Admin");
		px1.doSomeWork();
		//Robin is an unauthorized user
		ModifiedProxy px2 = new ModifiedProxy("Robin");
		px2.doSomeWork();

	}

}
```

Modified Output

Here's the modified output.

```
***Modified Proxy Pattern Demo***

Proxy call happening now...
Admin wants to invoke a proxy method.
doSomeWork() inside ConcreteSubject is invoked.

Proxy call happening now...
Robin wants to invoke a proxy method.
Sorry Robin, you do not have access rights.
```

7. **Proxies act like decorators. Is this correct?**

   You can implement a protection proxy similar to decorators but you should not forget the intent. Decorators focus on adding responsibilities, but proxies focus on controlling the access to an object. Proxies differ from each other with their types and implementations. *Also, in general, proxies work on the same interface but decorators can work on extended interfaces*. So, if you can remember their purposes, in most cases, you can clearly distinguish them from decorators.

8. **What are the cons associated with proxies?**

   If you are careful enough in your implementation, the pros are much greater than the cons, but

   - You can raise your concern about the response time. Since you are not directly talking to the actual object, it is possible that the response time through these proxies is longer.
   - You need to maintain additional code for the proxies.
   - A proxy can hide the actual responses from objects, which may create confusion in special scenarios.

CHAPTER 7

Decorator Pattern

This chapter covers the decorator pattern.

GoF Definition

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Concept

This pattern says that the class must be closed for modification but open for extension; that is, a new functionality can be added without disturbing existing functionalities. The concept is very useful when we want to add special functionalities to a specific object instead of the whole class. In this pattern, we try to use the concept of object composition instead of inheritance. So, when we master this technique, we can add new responsibilities to an object without affecting the underlying classes.

Real-World Example

Suppose you already own a house. Now you have decided to build an additional floor on top of it. You may not want to change the architecture of the ground floor (or existing floors), but you may want to change the design of the architecture for the newly added floor without affecting the existing architecture.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_7

Figure 7-1, Figure 7-2, and Figure 7-3 illustrate this concept.

Figure 7-1. *Original house*

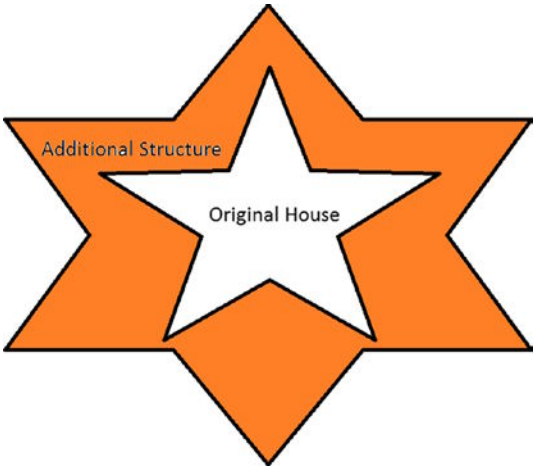


Figure 7-2. *Original house with a decorator (new structure is built on top of original structure)*

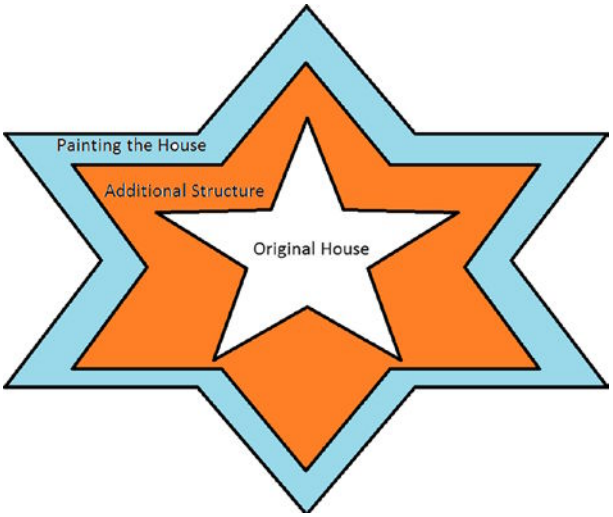


Figure 7-3. *Creating an additional decorator from an existing one (and painting the house)*

Note Case 3 is optional. You can use already a decorated object to enhance the behavior in this way or you can create a new decorator object and put all the new behavior in it.

Computer-World Example

Suppose that in a GUI-based toolkit, we want to add some border properties. We can do this with inheritance. But it cannot be treated as an ultimate solution because the user cannot have absolute control over this creation from the beginning. So, the core choice is static in this case.

Decorators comes into picture with a flexible approach. They promote the concept of dynamic choices, for example, we can surround the component in another object. The enclosing object is called a *decorator*. It must conform to the interface of the component that it decorates. It forwards the requests to the component. It can perform additional operations before or after the forwardings. An unlimited number of responsibilities can be added with this concept.

Note You can notice the use of the decorator pattern in the I/O streams implementations in both .NET Framework and Java. For example, the java.io.BufferedOutputStream class can decorate any java.io.OutputStream object.

Illustration

Go through the following example. Here we never tried to modify the core `makeHouse()` method. We have created two additional decorators: ConcreteDecoratorEx1 and ConcreteDecoratorEx2 to serve our needs but we kept the original structure intact.

Class Diagram

Figure 7-4 shows the class diagram for the illustration of the decorator pattern.

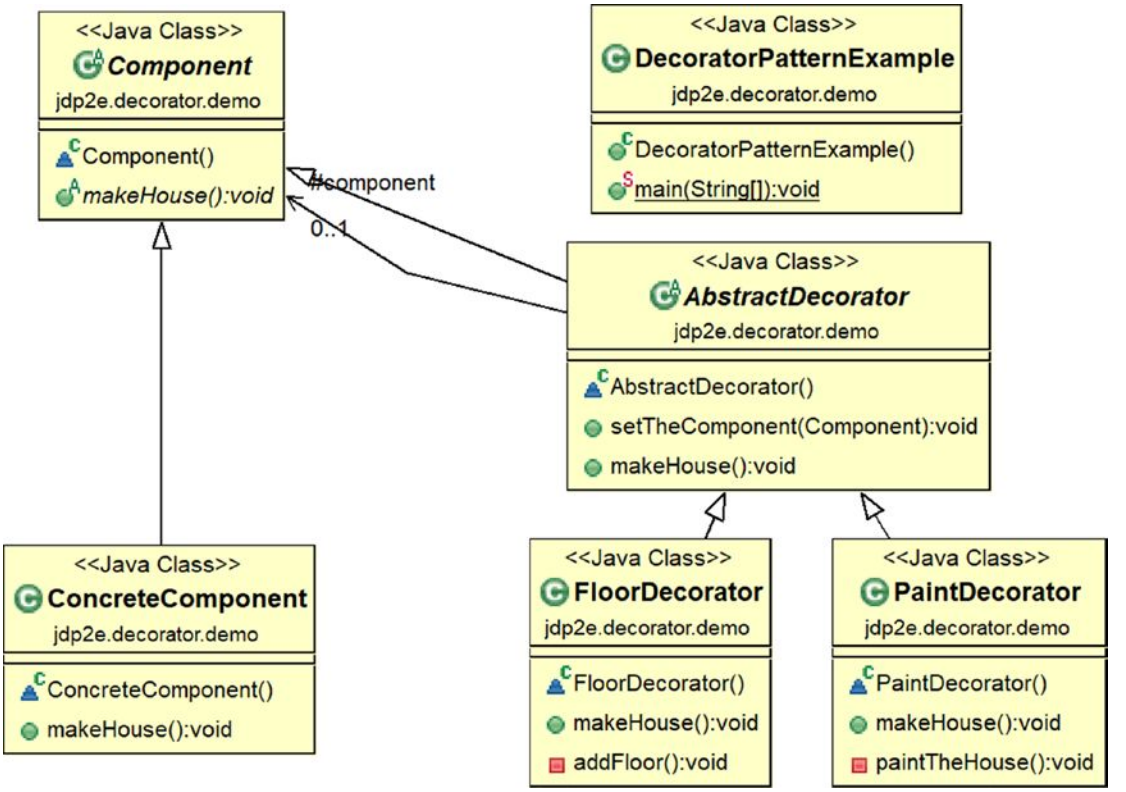


Figure 7-4. *Class diagram*

Package Explorer View

Figure 7-5 shows the high-level structure of the program.

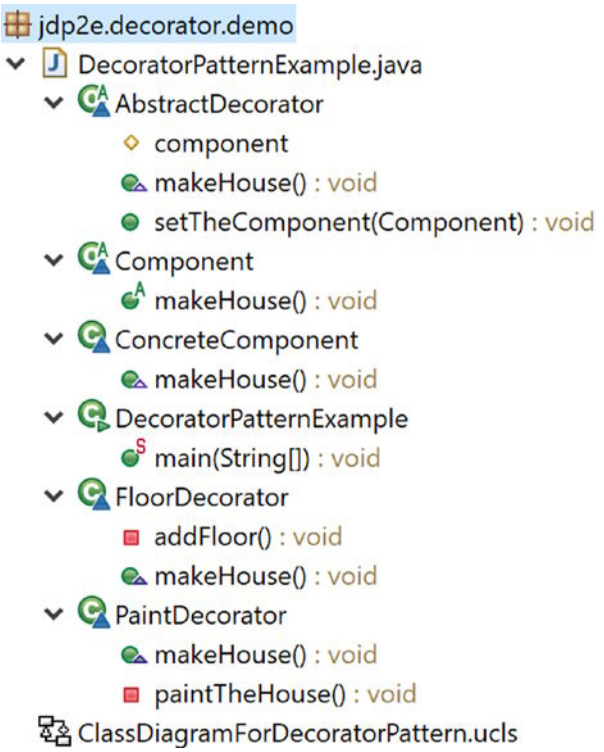


Figure 7-5. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.decorator.demo;
abstract class Component
{
    public abstract void makeHouse();

}
class ConcreteComponent extends Component
{
```

```
    public void makeHouse()
    {
        System.out.println("Original House is complete. It is closed for
        modification.");
    }
}

abstract class AbstractDecorator extends Component
{
    protected Component component ;
    public void setTheComponent(Component c)
    {
        component = c;
    }
    public void makeHouse()
    {
        if (component != null)
        {
            component.makeHouse();//Delegating the task
        }
    }
}
//A floor decorator
class FloorDecorator extends AbstractDecorator
{
    public  void makeHouse()
    {
        super.makeHouse();
        //Decorating now.
        System.out.println("***Floor decorator is in action***");
        addFloor();
        /*You can put additional stuffs as per your need*/
    }
```

```
    private void addFloor()
    {
        System.out.println("I am making an additional floor on top
        of it.");
    }
}
//A paint decorator
class PaintDecorator extends AbstractDecorator
{
    public void makeHouse()
    {
        super.makeHouse();
        //Decorating now.
        System.out.println("***Paint decorator is in action now***");
        paintTheHouse();
        //You can add additional stuffs as per your need
    }
    private void paintTheHouse()
    {
        System.out.println("Now I am painting the house.");
    }
}

public class DecoratorPatternExample {

    public static void main(String[] args) {
        System.out.println("***Decorator pattern Demo***\n");
        ConcreteComponent withoutDecorator = new ConcreteComponent();
        withoutDecorator.makeHouse();
        System.out.println("_____________________");

        //Using a decorator to add floor
        System.out.println("Using a Floor decorator now.");
        FloorDecorator floorDecorator = new FloorDecorator();
        floorDecorator.setTheComponent(withoutDecorator);
```

```
        floorDecorator.makeHouse();
        System.out.println("_____________________");

        //Using a decorator to add floor to original house and then
        //paint it.
        System.out.println("Using a Paint decorator now.");
        PaintDecorator paintDecorator = new PaintDecorator();
        //Adding results from floor decorator
        paintDecorator.setTheComponent(floorDecorator);
        paintDecorator.makeHouse();
        System.out.println("_____________________");
    }
}
```

Output

Here's the output.

```
***Decorator pattern Demo***

Original House is complete. It is closed for modification.

_____________________
Using a Floor decorator now.
Original House is complete. It is closed for modification.
***Floor decorator is in action***
I am making an additional floor on top of it.

_____________________
Using a Paint decorator now.
Original House is complete. It is closed for modification.
***Floor decorator is in action***
I am making an additional floor on top of it.
***Paint decorator is in action now***
Now I am painting the house.

_____________________
```

Q&A Session

1. **Can you explain how composition is promoting a dynamic behavior that inheritance cannot?**

   We know that when a derived class inherits from a parent class, it inherits the behavior of the base class at that time only. Though different subclasses can extend the base/parent class in different ways, this type of binding is known in compile-time, so the choice is static in nature. But the way that you used the concept of composition in the example lets you experiment with dynamic behavior.

   When we design a parent class, we may not have enough visibility about *what kind of additional responsibilities our clients may want in later phases*. And our constraint is that we should not modify the existing code frequently. In such a case, object composition not only outclasses inheritances, it also ensures that we are not introducing bugs to the existing architecture.

   Lastly, in this context, you must remember one of the key design principles: *Classes should be open for extension but closed for modification.*

2. **What are the key advantages of using a decorator?**

   - The existing structure is untouched, so that you are not introducing bugs there.
   - New functionalities can be easily added to an existing object.
   - You do not need to predict/implement all the supported functionalities at the initial design phase. You can develop incrementally (e.g., add decorator objects one by one to support incremental needs). You must acknowledge the fact that if you make a complex class first, and then you try to extend the functionalities, it will be a tedious process.

3. **How is the overall design pattern different from inheritance?**

   You can add or remove responsibilities by simply attaching or detaching decorators. But with a simple inheritance mechanism, you need to create a new class for the new responsibilities. So, it is possible that you may end up with a complex system.

   Consider the example again. Suppose that you want to add a new floor, paint the house, and do *some extra work*. To fulfill this need, you start with decorator2 because it is already providing the support to add a floor to the existing architecture, and then you can paint it. So, you can add a simple wrapper to complete those additional responsibilities.

   But if you start with inheritance from the beginning, then you may have multiple subclasses (e.g., one for adding a floor, one for painting the house). Figure 7-6 shows hierarchical inheritance.

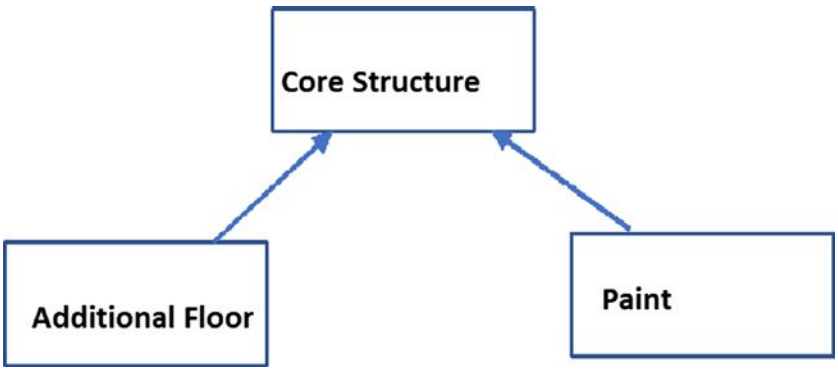


Figure 7-6. *A hierarchical inheritance*

   If you need an additional painted floor with extra features, you may end up with a design like the one shown in Figure 7-7.

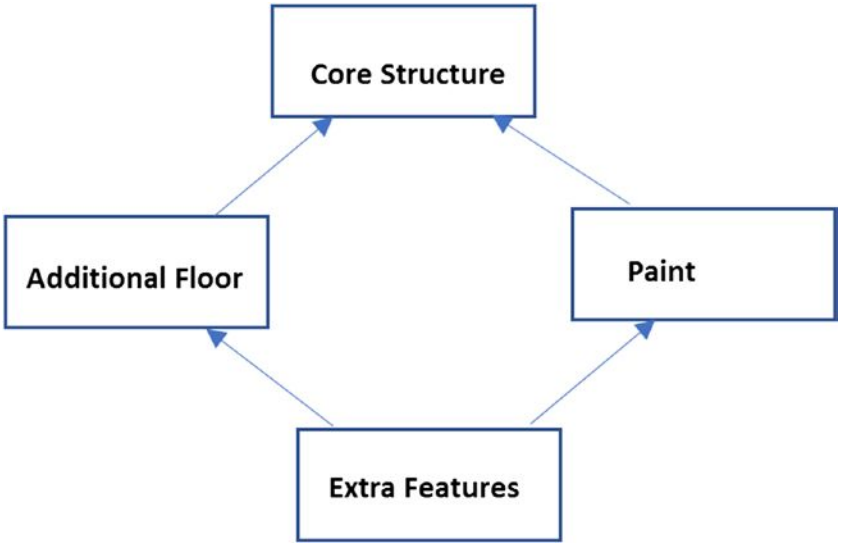


Figure 7-7. *A class (Extra Features) needs to inherit from multiple base classes*

Now you feel the heat of the diamond effect because in many programming languages including Java, multiple parent classes are not allowed.

In this context, even if you consider multilevel inheritance, you discover that overall the inheritance mechanism is much more challenging and time-consuming than the decorator pattern, and it may promote duplicate code in your application. Lastly, you must remember that inheritance mechanism is promoting only compile-time binding (not the dynamic binding).

4. **Why can't multilevel inheritance score higher in the previous context?**

   Let's assume that the *Paint* class is derived from *Additional Floor*, which in turn is derived from the *Core Architecture*. Now if your client wants to paint the house without creating an additional floor, the decorator pattern surely outclasses the inheritance mechanism because you can simply add a decorator to the existing system that supports the paint only.

5. **Why are you creating a class with a single responsibility? You could make a subclass that can simply add a floor and then paint. In that case, you end up with fewer subclasses. Is this understanding correct?**

   If you are familiar with SOLID principles, you know that there is a principle called *single responsibility*. The idea behind this principle is that each class should have a responsibility over a single part of the functionality in the software. The decorator pattern is very much effective when you use the single responsibility principle because you can simply add/remove responsibilities dynamically.

6. **What are the disadvantages associated with this pattern?**

   I believe that if you are careful enough, there is no significant disadvantage. But you must be aware of the fact that if you create too many decorators in the system, it will be hard to maintain and debug. So, in that case, it can create unnecessary confusion.

7. **In the example, there is no abstract method in the AbstractDecorator class. How is this possible?**

   In Java, you can have an abstract class without any abstract method in it, but the reverse is not true; that is, if a class contains at least one abstract method, then the class itself is incomplete and you are forced to mark it with the abstract keyword.

   Let's revisit the AbstractDecorator class in the comment shown in bold.

```
abstract class AbstractDecorator extends Component
{
    protected Component component ;
    public void setTheComponent(Component c)
    {
        component = c;
    }
```

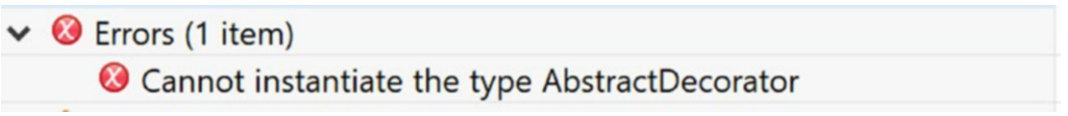
```
    public void makeHouse()
    {
      if (component != null)
        {
         component.makeHouse();//Delegating the task
        }
    }
}
```

You can see that I am delegating the task to a concrete decorator because I want to use and instantiate the concrete decorators only.

Also, in this example, you cannot simply instantiate an `AbstractDecorator` instance because it is marked with the abstract keyword.

The following line creates the *Cannot instantiate the type AbstractDecorator* compilation error.

AbstractDecorator abstractDecorator = new AbstractDecorator();



8. **In your example, instead of using concrete decorators, you could use the concept of polymorphism in the following way to generate the same output.**

```
System.out.println("Using a Floor decorator now.");
//FloorDecorator floorDecorator = new FloorDecorator();
AbstractDecorator floorDecorator = new FloorDecorator();
floorDecorator.setTheComponent(withoutDecorator);
floorDecorator.makeHouse();

//Using a decorator to add floor to original house and then paint
//it.
System.out.println("Using a Paint decorator now.");
```

```
//PaintDecorator paintDecorator = new PaintDecorator();
AbstractDecorator paintDecorator = new PaintDecorator();
//Adding results from decorator1
paintDecorator.setTheComponent(floorDecorator);
paintDecorator.makeHouse();
System.out.println("_____________________");
```

Is this correct?

Yes.

9. **Is it mandatory to use decorators for dynamic binding only?**

   No. You can use both static and dynamic binding. But dynamic binding is its strength, so I concentrated on it. You may notice that the GoF definition also focused on dynamic binding only.

10. **You are using decorators to wrap your core architecture. Is this correct?**

    Yes. The decorators are wrapper code to extend the core functionalities of the application. But the core architecture is untouched when you use them.

CHAPTER 8

Adapter Pattern

This chapter covers the adapter pattern.

GoF Definition

Convert the interface of a class into another interface that clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

Concept

The core concept is best described by the following examples.

Real-World Example

A very common use of this pattern can be seen in an electrical outlet adapter/AC power adapter in international travels. These adapters act as a middleman when an electronic device (let's say, a laptop) that accepts a US power supply can be plugged into a European power outlet. Consider another example. Suppose that you need to charge your mobile phone, but you see that the switchboard is not compatible with your charger. In this case, you may need to use an adapter. Or, a translator who is translating language for someone can be considered following this pattern in real life.

Now you can imagine a situation where you need to plug in an application into an adapter (which is X-shaped in this example) to use the intended interface. Without using this adapter, you cannot properly join the application and the interface.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_8

Figure 8-1 shows the case before using an adapter.

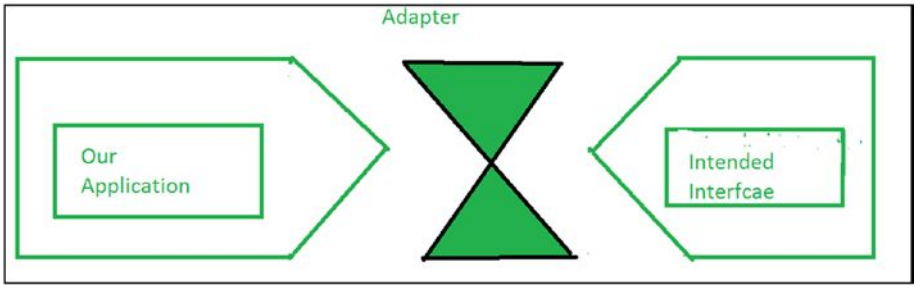


Figure 8-1. *Before using an adapter*

Figure 8-2 shows the case after using an adapter.

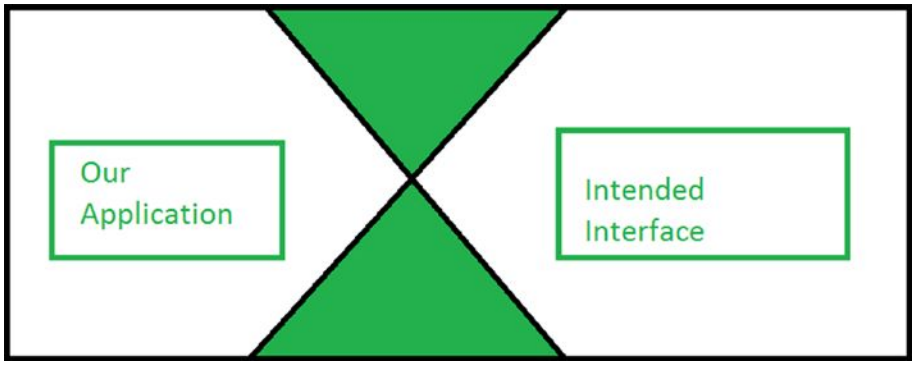


Figure 8-2. *After using an adapter*

Computer-World Example

Suppose that you have an application that can be broadly classified into two parts: user interface (UI or front end) and database (back end). Through the user interface, clients can pass a specific type of data or objects. Your database is compatible with those objects and can store them smoothly. Over a period of time, you may feel that you need to upgrade your software to make your clients happy. So, you may want to allow new type of objects to pass through the UI. But in this case, the first resistance comes from your

database because it cannot store these new types of objects. In such a situation, you can use an adapter that takes care of the conversion of the new objects to a compatible form that your old database can accept.

Note In Java, you can consider the java.io.InputStreamReader class and the java.io.OutputStreamWriter class as examples of object adapters. They adapt an existing InputStream/OutputStream object to a Reader/Writer interface. You will learn about class adapters and object adapters shortly.

Illustration

A simple use of this pattern is described in the following example.

In this example, you can easily calculate the area of a rectangle. If you notice the Calculator class and its `getArea()` method, you understand that you need to supply a rectangle object in the `getArea()` method to calculate the area of the rectangle. Now suppose that you want to calculate the area of a triangle, but your constraint is that you want to get the area of it through the `getArea()`method of the Calculator class. So how can you achieve that?

To deal with this type of problem, I made CalculatorAdapter for the Triangle class and passed a triangle in its `getArea()` method. In turn, the method treats the triangle like a rectangle and calculates the area from the `getArea()` method of the Calculator class.

Class Diagram

Figure 8-3 shows the class diagram.

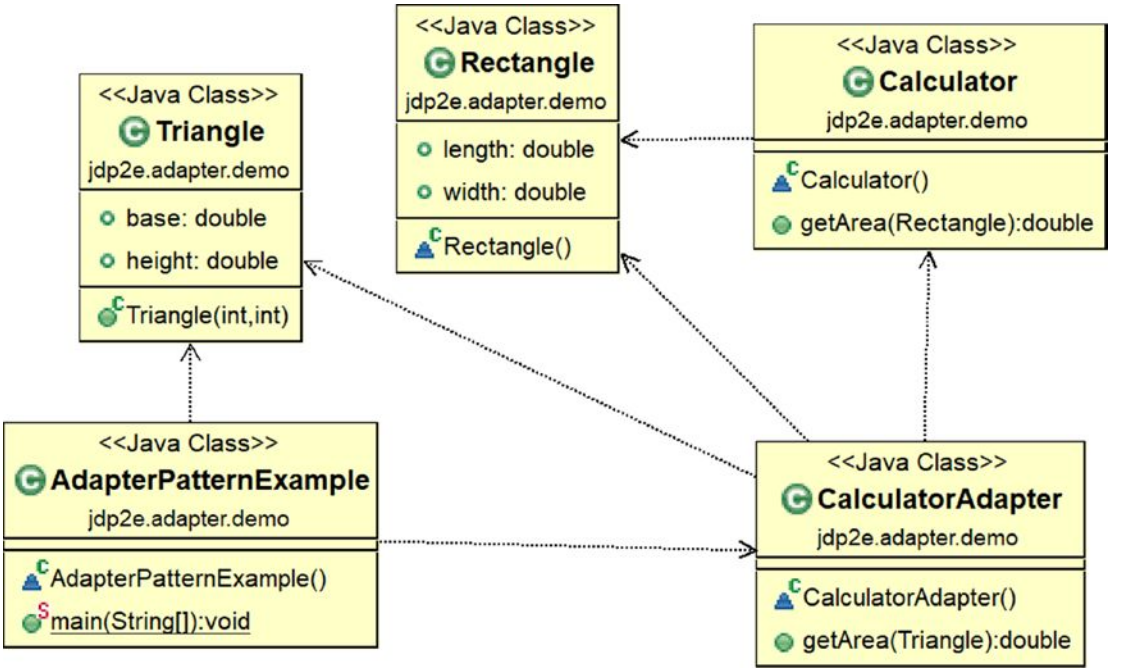


Figure 8-3. *Class diagram*

Package Explorer View

Figure 8-4 shows the high-level structure of the program.

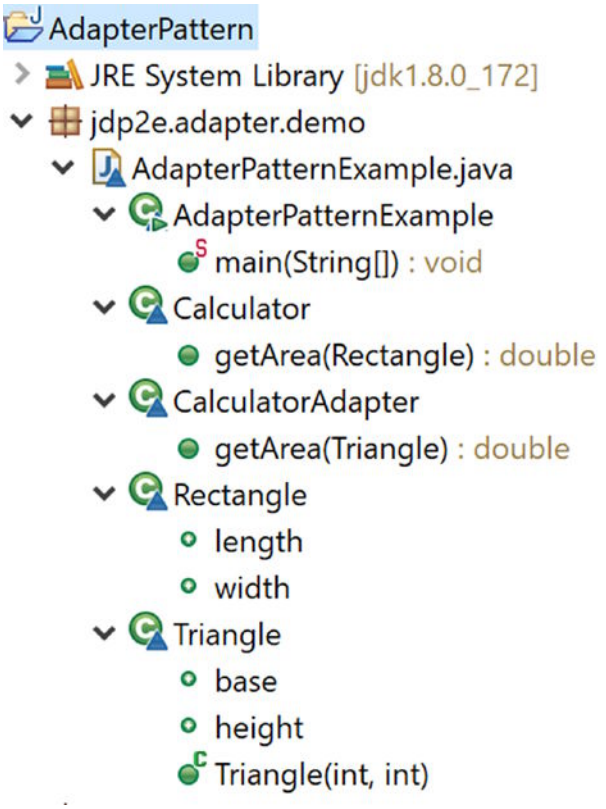


Figure 8-4. *Package Explorer view*

Implementation

```
package jdp2e.adapter.demo;
class Rectangle
{
    public double length;
    public double width;
}
class Calculator
{
    public double getArea(Rectangle rect)
    {
        return rect.length * rect.width;
    }
}
```

```
class Triangle
{
    public double base;//base
    public double height;//height
    public Triangle(int b, int h)
    {
        this.base = b;
        this.height = h;
    }
}
class CalculatorAdapter
{
    public double getArea(Triangle triangle)
    {
        Calculator c = new Calculator();
        Rectangle rect = new Rectangle();
        //Area of Triangle=0.5*base*height
        rect.length = triangle.base;
        rect.width = 0.5 * triangle.height;
        return c.getArea(rect);
    }
}

class AdapterPatternExample {
    public static void main(String[] args) {
        System.out.println("***Adapter Pattern Demo***\n");
        CalculatorAdapter calculatorAdapter = new CalculatorAdapter();
        Triangle t = new Triangle(20,10);
        System.out.println("Area of Triangle is " + calculatorAdapter.
        getArea(t) + " Square unit");
    }
}
```

Output

Here's the output.

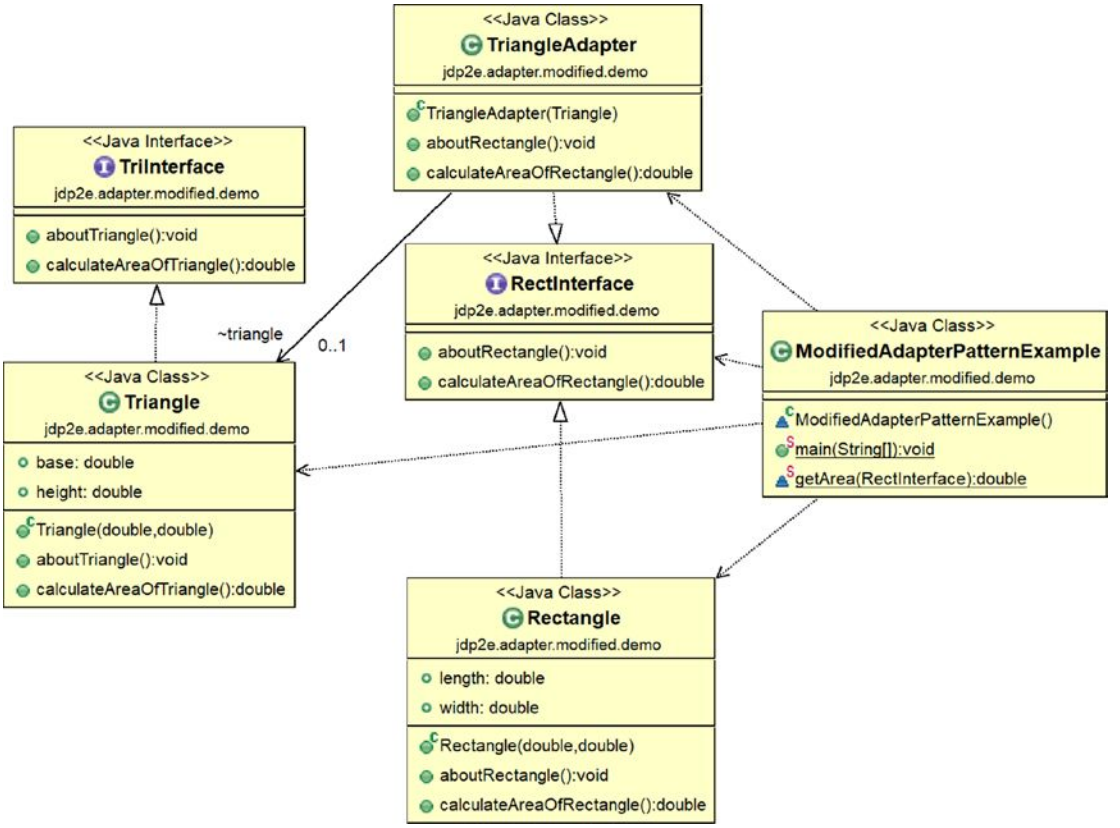
```
***Adapter Pattern Demo***

Area of Triangle is 100.0 Square unit
```

Modified Illustration

You have just seen a very simple example of the adapter design pattern. But if you want to strictly follow object-oriented design principles, you may want to modify the implementation because you have learned that instead of using concrete classes, you should always prefer to use interfaces. So, keeping this key principle in mind, let's modify the implementation.

Modified Class Diagram



Key Characteristics of the Modified Implementation

The following are the key characteristics of the modified implementation.

- The Rectangle class is implementing RectInterface and the `calculateAreaOfRectangle()` method helps calculate the area of a rectangle object.
- The Triangle class implements TriInterface and the `calculateAreaOfTriangle()` method helps calculate the area of a triangle object.
- But the constraint is that you need to calculate the area of the triangle using the RectInterface (or, you can simply say that your existing system needs to adapt the triangle objects). To serve this purpose, I introduced an `adapter(TriangleAdapter)`, which interacts with the RectInterface interface.
- Neither the rectangle nor the triangle code needs to change. You are simply using the adapter because it implements the RectInterface interface, and using a RectInterface method, you can easily compute the area of a triangle. This is because I am overriding the interface method to delegate to the corresponding method of the class (Triangle) that I am adapting from.
- Notice that `getArea(RectInterface)` method does not know that through TriangleAdapter, it is actually processing a Triangle object instead of a Rectangle object.
- Notice another important fact and usage. Suppose that in a specific case, you need to play with some rectangle objects that have an area of 200 square units, but you do not have a sufficient number of such objects. But you notice that you have

triangle objects whose area are 100 square units. So, using this pattern, you can adapt some of those triangle objects. How? Well, if you look carefully, you find that when using the adapter's `calculateAreaOfRectangle()` method, you are actually invoking `calculateAreaOfTriangle()` of a Triangle object (i.e., you are delegating the corresponding method of the class you are adapting from). So, you can modify (override) the method body as you need (e.g., in this case, you could multiply the triangle area by 2.0 to get an area of 200 square units (just like a rectangle object with length 20 units and breadth 10 units).

This technique can help you in a scenario where you may need to deal with objects that are not exactly same but are very similar. In the last part of the client code, I have shown such a usage where the application displays current objects in the system using an enhanced `for` loop (which was introduced in Java 5.0).

Note In the context of the last point, you must agree that you should not make an attempt to convert a circle to a rectangle (or similar type of conversion) to get an area because they are totally different. But in this example, I am talking about triangles and rectangles because they have some similarities and the areas can be computed easily with minor changes.

Modified Package Explorer View

Figure 8-5 shows the structure of the modified program.

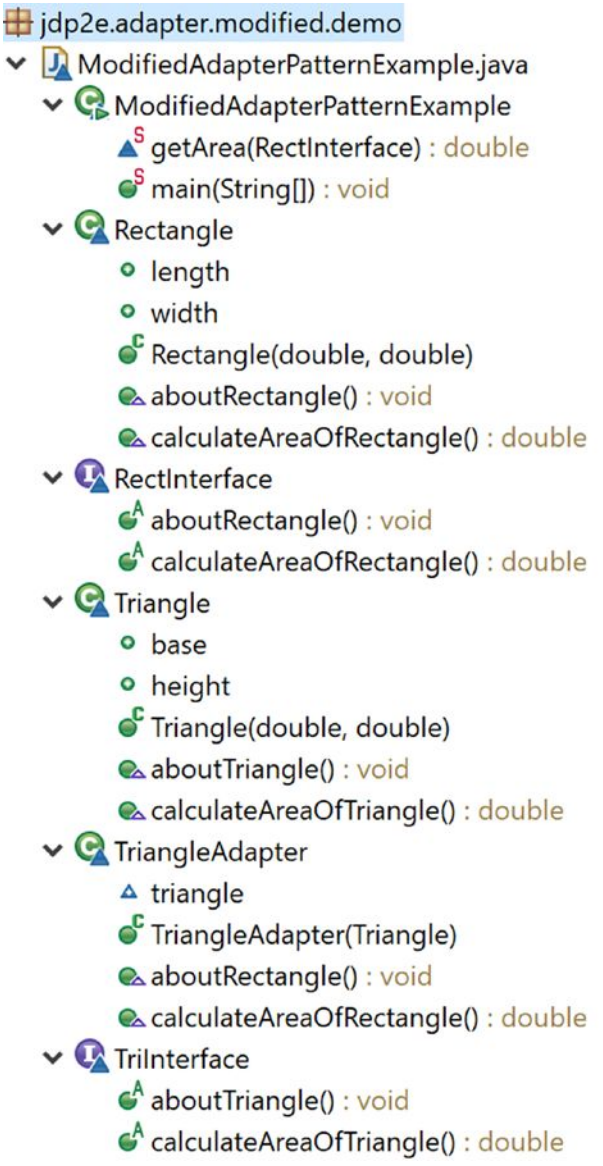


Figure 8-5. *Modified Package Explorer view*

Modified Implementation

This is the modified implementation.

```
package jdp2e.adapter.modified.demo;

import java.util.ArrayList;
import java.util.List;

interface RectInterface
{
    void aboutRectangle();
    double calculateAreaOfRectangle();
}
class Rectangle implements RectInterface
{
    public double length;
    public double width;
    public Rectangle(double length, double width)
    {
        this.length = length;
        this.width = width;
    }

    @Override
    public void aboutRectangle()
    {
        System.out.println("Rectangle object with length: "+ this.length +"
        unit and width :" +this.width+ " unit.");

    }

    @Override
    public double calculateAreaOfRectangle()
    {
        return length * width;
    }
}
```

```
interface TriInterface
{
    void aboutTriangle();
    double calculateAreaOfTriangle();
}
class Triangle implements TriInterface
{
    public double base;//base
    public double height;//height
    public Triangle(double base, double height)
    {
        this.base = base;
        this.height = height;
    }

    @Override
    public void aboutTriangle() {
        System.out.println("Triangle object with base: "+ this.base +" unit
        and height :" +this.height+ " unit.");

    }

    @Override
    public double calculateAreaOfTriangle() {
        return 0.5 * base * height;
    }
}

/*TriangleAdapter is implementing RectInterface.
 So, it needs to implement all the methods defined
in the target interface.*/
class TriangleAdapter implements RectInterface
{
    Triangle triangle;
    public TriangleAdapter(Triangle t)
    {
        this.triangle = t;
    }
```

```
    @Override
    public void aboutRectangle() {
        triangle.aboutTriangle();
    }
    @Override
    public double calculateAreaOfRectangle() {
        return triangle.calculateAreaOfTriangle();
    }
}

class ModifiedAdapterPatternExample {
    public static void main(String[] args) {
        System.out.println("***Adapter Pattern Modified Demo***\n");
        Rectangle rectangle = new Rectangle(20, 10);
        System.out.println("Area of Rectangle is :  "+ rectangle.
        calculateAreaOfRectangle()+" Square unit.");
        Triangle triangle = new Triangle(10,5);
        System.out.println("Area of Triangle is : "+triangle.
        calculateAreaOfTriangle()+ " Square unit.");
        RectInterface adapter = new TriangleAdapter(triangle);
        //Passing a Triangle instead of a Rectangle
        System.out.println("Area of Triangle using the triangle adapter is
        : "+getArea(adapter)+" Square unit.");

        //Some Additional code (Optional) to show the power of adapter
        //pattern
        List<RectInterface> rectangleObjects=new ArrayList<RectInterfa
        ce>();
        rectangleObjects.add(rectangle);
        //rectangleObjects.add(triangle);//Error
        rectangleObjects.add(adapter);//Ok
        System.out.println("");
        System.out.println("*****Current objects in the system
        are:******");
```

```
        for(RectInterface rectObjects:rectangleObjects)
        {
            rectObjects.aboutRectangle();
        }
    }
    /*getArea(RectInterface r) method  does not know that through
    TriangleAdapter, it is getting a Triangle  object instead of a
    Rectangle object*/
    static double getArea(RectInterface r)
    {
        return r.calculateAreaOfRectangle();
    }
}
```

Modified Output

This is the modified output.

```
***Adapter Pattern Modified Demo***

Area of Rectangle is :  200.0 Square unit.
Area of Triangle is : 25.0 Square unit.
Area of Triangle using the triangle adapter is : 25.0 Square unit.

*****Current objects in the system are:******
Rectangle object with length: 20.0 unit and width :10.0 unit.
Triangle object with base: 10.0 unit and height :5.0 unit.
```

Types of Adapters

GoF explains two types of adapters: class adapters and object adapters.

Object Adapters

Object adapters adapt through object compositions, as shown in Figure 8-6. The adapter discussed so far is an example of an object adapter.

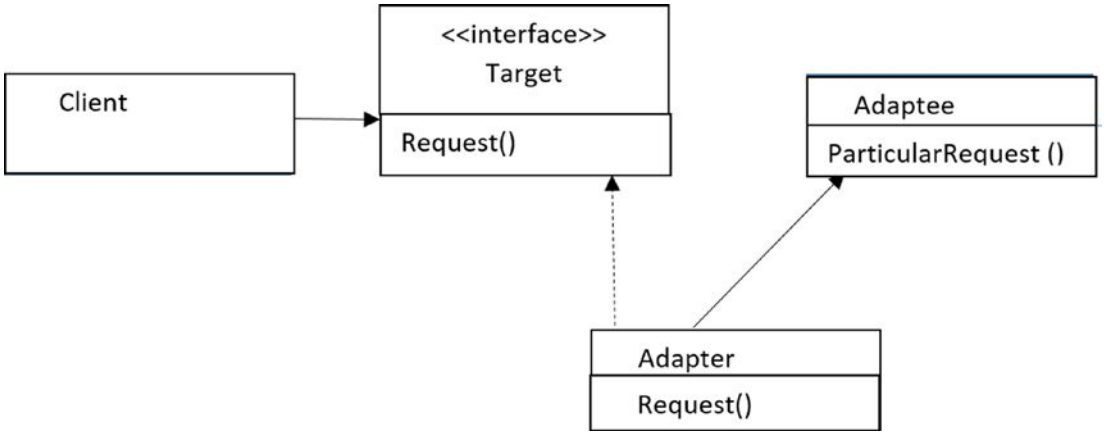


Figure 8-6. *A typical object adapter*

In our example, TriangleAdapter is the adapter that implements the RectInterface (Target interface). Triangle is the Adaptee interface. The adapter holds the adaptee instance.

Note So, if you follow the body of the TriangleAdapter class, you can conclude that to create an object adapter, you need to follow these general guidelines:

(1) Your class needs to implement the target interface (***adapting to*** interface). If the target is an abstract class, you need to extend it.

(2) Mention the class that you are ***adapting from*** in the constructor and store a reference to it in an instance variable.

(3) Override the interface methods to delegate the corresponding methods of the class you are adapting from.

Class Adapters

Class adapters adapt through subclassing. They are the promoters of multiple inheritance. But you know that in Java, multiple inheritance through classes is not supported. (You need interfaces to implement the concept of multiple inheritance.)

Figure 8-7 shows the typical class diagram for class adapters, which support multiple inheritance.

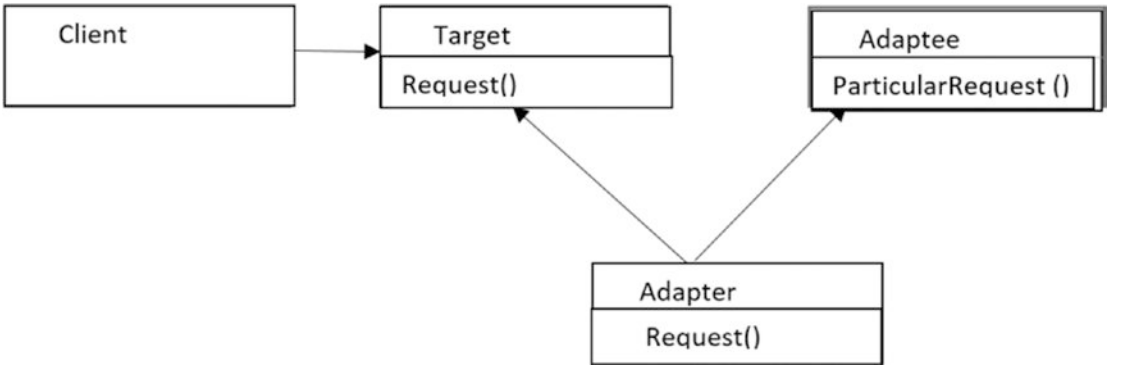


Figure 8-7. *A typical class adapter*

Q&A Session

1. **How can you implement class adapter design patterns in Java?**

   You can subclass an existing class and implement the desired interface. For example, if you want to use a class adapter instead of an object adapter in the modified implementation, then you can use the following code.

```
class TriangleClassAdapter extends Triangle implements
RectInterface
{
    public TriangleClassAdapter(double base, double height) {
        super(base, height);
    }

    @Override
    public void aboutRectangle()
    {
        aboutTriangle();
    }
    @Override
```

```
    public double calculateAreaOfRectangle()
    {
        return calculateAreaOfTriangle();
    }
}
```

But note that you cannot always apply this approach. For example, consider when the Triangle class is a final class (so, you cannot derive from it). Apart from this case , you will be blocked again when you notice that you need to adapt a method that is not specified in the interface. So, in cases like this, object adapters are useful.

2. **"Apart from this case, you will be blocked again when you notice that you need to adapt a method that is not specified in the interface." What do you mean by this?**

   In the modified implementation, you have used the `aboutRectangle()` and `aboutTriangle()` methods.These methods are actually telling about the objects of the Rectangle and Triangle classes. Now, say, instead of `aboutTriangle()` , there is a method called `aboutMe()`, which is doing the same but there is no such method in the RectInterface interface. Then it will be a challenging task for you to adapt the `aboutMe()` method from the Triangle class and write code similar to this:

```
for(RectInterface rectObjects:rectangleObjects)
{
    rectObjects.aboutMe();
}
```

3. **Which do you prefer—class adapters or object adapters?**

   In most cases, I prefer compositions over inheritance. Object adapters use compositions and are more flexible. Also, in many cases, you may not implement a true class adapter. (In this context, you may go through the answers to the previous questions again.)

4. **What are the drawbacks associated with this pattern?**

   I do not see any big challenges. I believe that you can make an adapter's job simple and straightforward, but you may need to write some additional code. But the payoff is great—particularly for those legacy systems that cannot be changed but you still need to use them for their stability.

   At the same time, experts suggest that you do not use different types of validations or add a new behavior to the adapter. Ideally, the job of an adaptar should be limited to only performing simple interface translations.

CHAPTER 9

Facade Pattern

This chapter covers the facade pattern.

GoF Definition

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Concept

Facades make a client's life easier. Suppose that there is a complex system where multiple objects need to perform a series of tasks, and you need to interact with the system. In a situation like this, facade can provide you a simplified interface that takes care of everything (the creation of those objects, providing the correct sequence of tasks, etc.). As a result, instead of interacting with multiple objects in a complicated way, you just interact with a single object.

It is one of those patterns that supports loose coupling. Here you emphasize the abstraction and hide the complex details by exposing a simple interface. As a result, the code becomes clearer and more attractive.

Real-World Example

Suppose that you are going to organize a birthday party, and you plan to invite 500 people. Nowadays, you can go to any party organizer and let them know the key information—party type, the date and time, number of attendees, and so forth. The organizer does the rest for you. You do not need to think about how the hall will be

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_9

decorated, whether attendees will get their food from a buffet table or be served by the caterer, and so forth. So, you do not need to buy items from the store or decorate the party hall yourself—you just pay the organizer and let them do the job properly.

Computer-World Example

Think about a situation where you use a method from a library (in the context of a programming language). You do not care how the method is implemented in the library. You just call the method to experiment the easy usage of it.

Note You can use the concept of facade design pattern effectively to make your JDBC application attractive. You can consider the java.net.URL class as an example of a facade pattern implementation.Consider the shorthand openStream() or getContent() methods in this class. The openStream() method returns openConnection().getInputStream() and the getContent() method returns openConnection.getContent().The getInputStream() and getContent() methods are further defined in the URLConnection class.

Illustration

In the following implementation, you create some robots, and later, you destroy those objects. (The word "destroy" is not used in the context of garbage collection in this example). Here you can construct or destroy a particular kind of robot by invoking simple methods like `constructMilanoRobot()` and `destroyMilanoRobot()` of the RobotFacade class.

From a client's point of view, he/she needs to interact only with the facade (see FacadePatternExample.java). RobotFacade is taking full responsibility in creating or destroying a particular kind of robot. This facade is talking to each of the subsystems (RobotHands, RobotBody, RobotColor) to fulfill the client's request. The RobotBody class includes two simple static methods that provide instructions prior to the creation or destruction of a robot.

So, in this implementation, the clients do not need to worry about the creation of the separate classes and the calling sequence of the methods.

Class Diagram

Figure 9-1 shows the class diagram.

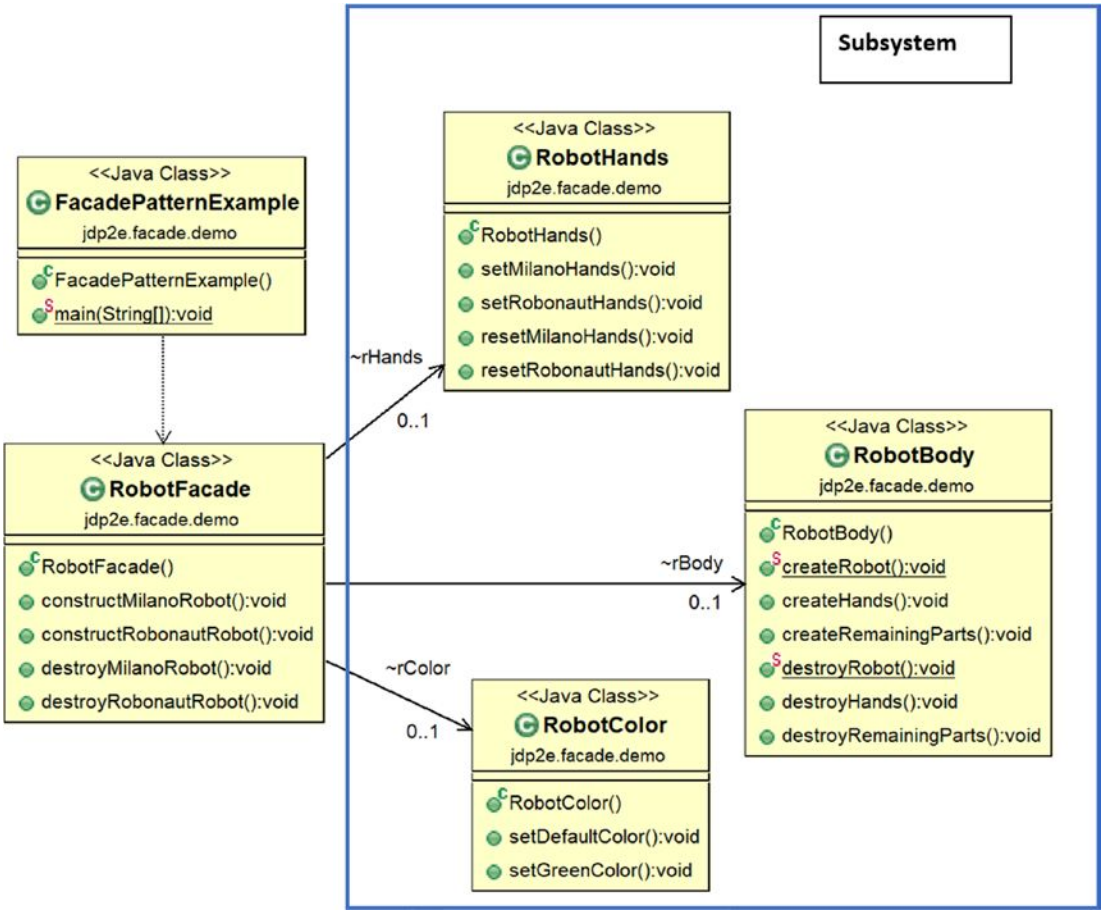


Figure 9-1. *Class diagram*

Package Explorer View

Figure 9-2 shows the high-level structure of the program.

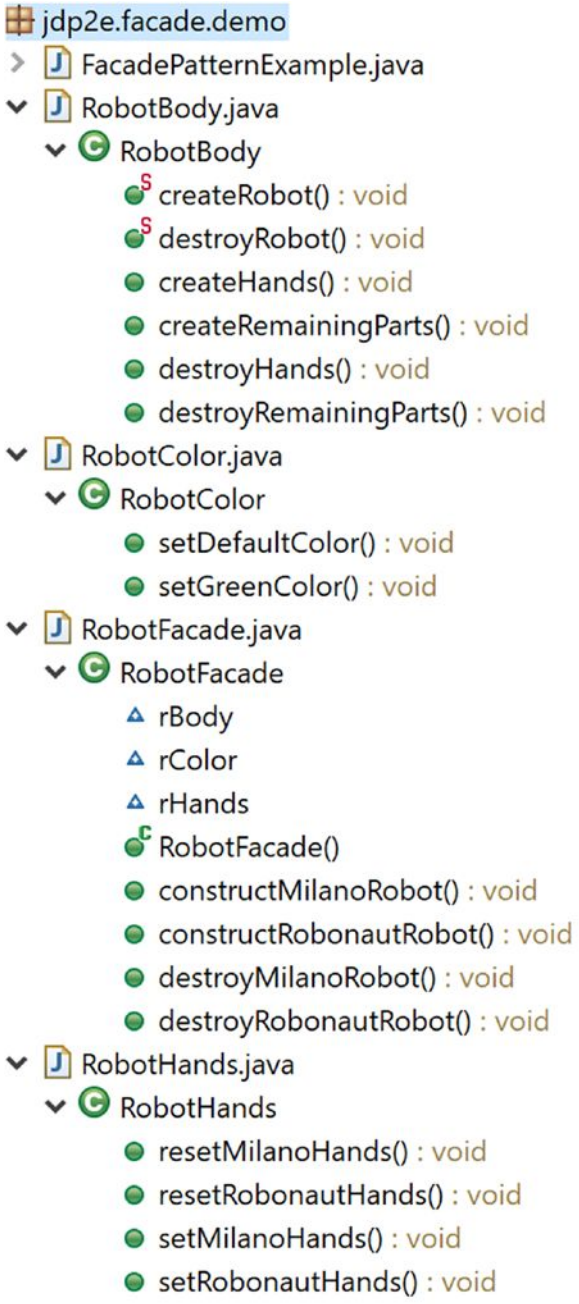


Figure 9-2. *Package Explorer view*

Implementation

Here's the implementation.

```
// RobotBody.java

package jdp2e.facade.demo;

public class RobotBody
{
    //Instruction manual -how to create a robot
    public static void createRobot()
    {
        System.out.println(" Refer the manual before creation of a
        robot.");
    }
    //Method to create hands of a robot
    public void createHands()
    {
        System.out.println(" Hands manufactured.");
    }
    //Method to create remaining parts (other than hands) of a robot
    public void createRemainingParts()
    {
        System.out.println(" Remaining parts (other than hands) are
        created.");
    }
    //Instruction manual -how to destroy a robot
    public static void destroyRobot()
    {
        System.out.println(" Refer the manual before destroying of a robot.");
    }
    //Method to destroy hands of a robot
    public void destroyHands()
    {
        System.out.println(" The robot's hands are destroyed.");
    }
```

```
    //Method to destroy remaining parts (other than hands) of a robot
    public void destroyRemainingParts()
    {
        System.out.println(" The robot's remaining parts are destroyed.");
    }

}
//RobotColor.java

package jdp2e.facade.demo;

public class RobotColor
{
    public void setDefaultColor()
    {
        System.out.println(" This is steel color robot.");
    }
    public void setGreenColor()
    {
        System.out.println(" This is a green color robot.");
    }
}

// RobotHands.java

package jdp2e.facade.demo;

public class RobotHands
{
    public void setMilanoHands()
    {
        System.out.println(" The robot will have EH1 Milano hands.");
    }
    public void setRobonautHands()
    {
        System.out.println(" The robot will have Robonaut hands.");
    }
```

```
    public void resetMilanoHands()
    {
        System.out.println(" EH1 Milano hands are about to be destroyed.");
    }
    public void resetRobonautHands()
    {
        System.out.println(" Robonaut hands are about to be destroyed.");
    }
}

// RobotFacade.java
package jdp2e.facade.demo;

public class RobotFacade
{
    RobotColor rColor;
    RobotHands rHands ;
    RobotBody rBody;
    public RobotFacade()
    {
        rColor = new RobotColor();
        rHands = new RobotHands();
        rBody = new RobotBody();

    }
    //Constructing a Milano Robot
    public void constructMilanoRobot()
    {
        RobotBody.createRobot();
        System.out.println("Creation of a Milano Robot Start.");
        rColor.setDefaultColor();
        rHands.setMilanoHands();
        rBody.createHands();
        rBody.createRemainingParts();
        System.out.println(" Milano Robot Creation End.");
        System.out.println();
    }
```

```
    //Constructing a Robonaut Robot
    public void constructRobonautRobot()
    {
        RobotBody.createRobot();
        System.out.println("Initiating the creational process of a Robonaut
        Robot.");
        rColor.setGreenColor();
        rHands.setRobonautHands();
        rBody.createHands();
        rBody.createRemainingParts();
        System.out.println("A Robonaut Robot is created.");
        System.out.println();
    }
    //Destroying a Milano Robot
    public void destroyMilanoRobot()
    {
        RobotBody.destroyRobot();
        System.out.println(" Milano Robot's destruction process is
        started.");
        rHands.resetMilanoHands();
        rBody.destroyHands();
        rBody.destroyRemainingParts();
        System.out.println(" Milano Robot's destruction process is over.");
        System.out.println();
    }
    //Destroying a Robonaut Robot
    public void destroyRobonautRobot()
    {
        RobotBody.destroyRobot();
        System.out.println(" Initiating a Robonaut Robot's destruction
        process.");
        rHands.resetRobonautHands();
        rBody.destroyHands();
        rBody.destroyRemainingParts();
        System.out.println(" A Robonaut Robot is destroyed.");
```

```
        System.out.println();
    }

}
//Client code
//FacadePatternExample.java
package jdp2e.facade.demo;

public class FacadePatternExample {
    public static void main(String[] args) {
        System.out.println("***Facade Pattern Demo***\n");
        //Creating Robots
        RobotFacade milanoRobotFacade = new RobotFacade();
        milanoRobotFacade.constructMilanoRobot();
        RobotFacade robonautRobotFacade = new RobotFacade();
        robonautRobotFacade.constructRobonautRobot();
        //Destroying robots
        milanoRobotFacade.destroyMilanoRobot();
        robonautRobotFacade.destroyRobonautRobot();

    }

}
```

Output

Here's the output.

```
***Facade Pattern Demo***

 Refer the manual before creation of a robot.
Creation of a Milano Robot Start.
 This is steel color robot.
 The robot will have EH1 Milano hands.
 Hands manufactured.
 Remaining parts (other than hands) are created.
 Milano Robot Creation End.
```

```
 Refer the manual before creation of a robot.
Initiating the creational process of a Robonaut Robot.
 This is a green color robot.
 The robot will have Robonaut hands.
 Hands manufactured.
 Remaining parts (other than hands) are created.
A Robonaut Robot is created.

 Refer the manual before destroying of a robot.
 Milano Robot's destruction process is started.
 EH1 Milano hands are about to be destroyed.
 The robot's hands are destroyed.
 The robot's remaining parts are destroyed.
 Milano Robot's destruction process is over.

 Refer the manual before destroying of a robot.
 Initiating a Robonaut Robot's destruction process.
 Robonaut hands are about to be destroyed.
 The robot's hands are destroyed.
 The robot's remaining parts are destroyed.
 A Robonaut Robot is destroyed.
```

Q&A Session

1. **What are key advantages of using a facade pattern?**

    - If a system consists of many subsystems, managing all those subsystems becomes very tough and clients may find their life difficult to communicate separately with each of these subsystems. In this scenario, facade patterns are very much handy. It provides a simple interface to clients. In simple words, instead of presenting complex subsystems, you present one simplified interface to clients. This approach also promotes weak coupling by separating a client from the subsystems.
    - It can also help you to reduce the number of objects that a client needs to deal with.

2. **I see that the facade class is using compositions. Is this intentional?**

   Yes. With this approach, you can easily access the methods in each subsystem.

3. **It appears to me that facades do not restrict us to directly connect with subsystems. Is this understanding correct?**

   Yes. A facade does not encapsulate the subsystem classes or interfaces. It just provides a simple interface (or layer) to make your life easier. You are free to expose any functionality of the subsystem, but in those cases, your code may look dirty, and at the same time, you lose all the benefits associated with this pattern.

4. **How is it different from adapter design pattern?**

   In the adapter pattern, you try to alter an interface so that the clients do not feel the difference between the interfaces. The facade pattern simplifies the interface. They present the client a simple interface to interact with (instead of a complex subsystem).

5. **There should be only one facade for a complex subsystem. Is this correct?**

   Not at all. You can create any number of facades for a particular subsystem.

6. **Can I add more stuffs/logic with a facade?**

   Yes, you can.

7. **What are the challenges associated with a facade pattern?**

   - Subsystems are connected with the facade layer. So, you need to take care of an additional layer of coding (i.e., your codebase increases).
   - When the internal structure of a subsystem changes, you need to incorporate the changes in the facade layer also.
   - Developers need to learn about this new layer, whereas some of them may already be aware of how to use the subsystems/APIs efficiently.

8. **How is it different from the mediator design pattern?**

   In a mediator pattern implementation, subsystems are aware of the mediator. They talk to each other. But in a facade, subsystems are not aware of the facade and the one-way communication is provided from facade to the subsystem(s). (The mediator pattern is discussed in Chapter 21 of this book).

9. **It appears to me that to implement a facade pattern, I have to write lots of code. Is this understanding correct?**

   Not at all. It depends on the system and corresponding functionalities. For example, in the preceding implementation, if you consider only one type of robot (either Milano or Robonaut), and if you do not want to provide the destruction mechanism of robots, and if you want to ignore the instruction manuals (two static methods in this example), your code size will drop significantly. I have kept all of these for complete illustration purposes.

CHAPTER 10

Flyweight Pattern

This chapter covers the flyweight pattern.

GoF Definition

Use sharing to support large numbers of fine-grained objects efficiently.

Concept

In their famous book *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995), the Gang of Four (GoF) wrote about flyweights as follows:

> *A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context—it's indistinguishable from an instance of the object that's not shared. Flyweights cannot make assumptions about the context in which they operate.*

When you consider flyweight pattern, you need to remember following points:

- The pattern is useful when you need a large number of similar objects that are unique in terms of only a few parameters and most of the stuffs are common in general.
- A flyweight is an object. It tries to minimize memory usage by sharing data as much as possible with other similar objects. Sharing objects may allow their usage at fine granularities with minimum costs.
- Two common terms are used in this context: extrinsic and intrinsic. An *intrinsic* state is stored/shared in the flyweight object, and it is independent of flyweight's context. On the other hand, an extrinsic

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_10

state varies with flyweight's context, which is why they cannot be shared. Client objects maintain the extrinsic state, and they need to pass this to a flyweight. Note that, if required, clients can also compute the extrinsic state on the fly when using flyweights.

- Experts suggest that while implementing this pattern, we should make intrinsic states immutable.

Real-World Example

Suppose that you have pen. You can replace different refills to write with different colors. So, a pen without refills is considered as a flyweight with intrinsic data, and a pen with refills is considered as extrinsic data.

Consider another example. Suppose that a company needs to print visiting cards for its employees. So, where does the process start? The company can create a common template with the company logo, address, and so forth (intrinsic), and later it adds each employee's particular contact information (extrinsic) on the cards.

Computer-World Example

Suppose that you want to make a website where different users can compile and execute the programs with their preferred computer languages, such as Java, C++, C#, and so forth. If you need to set up a unique environment for each individual user within a short period of time, your site will overload and the response time of the server will become so slow that no one will be interested in using your site. So, instead of creating a new programming environment for every user, you can make a common programming environment (which supports different programming language with/without minor changes) among them. And to check the existing/available programming environment and to make decisions whether you need to create a new one or not, you can maintain a factory.

Consider another example. Suppose that in a computer game, you have large number of participants whose core structures are same, but their appearances vary (e.g., different states, colors, weapons, etc.) Therefore, assume that if you need to create (or store) all of these objects with all of these variations/states, the memory requirement will be huge. So, instead of storing all of these objects, you can design your application in such way that you create these instances with common properties (flyweights with

intrinsic state) and your client object maintains all of these variations (extrinsic states). If you can successfully implement this concept, you can claim that you have followed the flyweight design pattern in your application.

Another common use of this pattern is seen in the graphical representation of characters in a word processor.

Note In Java, you may notice the use of this pattern when you use the wrapper classes, such as java.lang.Integer, java.lang.Short, java.lang.Byte, and java.lang.Character, where the static method valueof() replicates a factory method. (It is worth remembering that some of the wrapper classes, such as java.lang.Double and java.lang.Float, *do not* follow this pattern.) The String pool is another example of a flyweight.

Illustration

In the following example, I used three different types of objects: small, large, and fixed-size robots. These robots have two states: "robotTypeCreated" and "color". The first one can be shared among "similar" objects, so it is an *intrinsic* state. The second one (color) is supplied by the client and it varies with the context. So, it is an *extrinsic* state in this example.

For the fixed-size robots, it does not matter which color is supplied by the client. For these robots, I am ignoring the extrinsic state, so you can conclude that these fixed-size robots are representing *unshared flyweights*.

In this implementation, the robotFactory class caches these flyweights and provides a method to get them.

Lastly, these objects are similar. So, once a particular robot is created, you do not want to repeat the process from scratch. Instead, the next time onward, you will try to use these flyweights to serve your needs. Now go through the code with the comments for your ready reference.

Class Diagram

Figure 10-1 shows the class diagram.

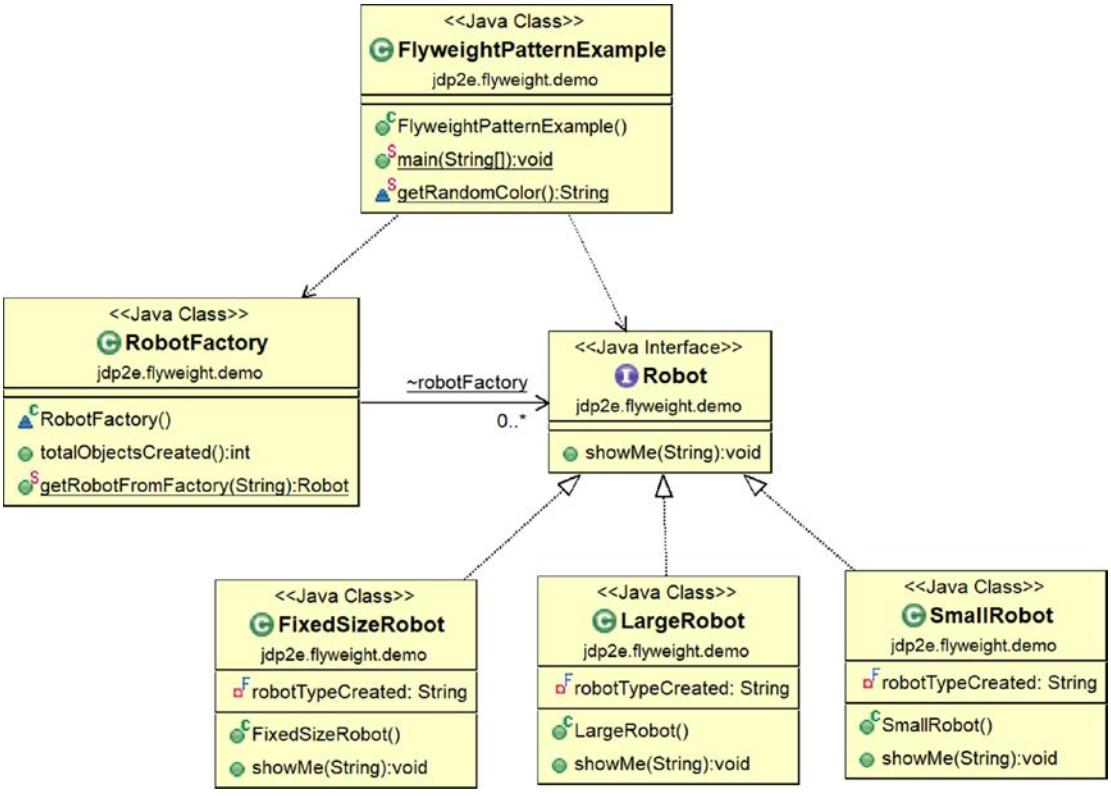


Figure 10-1. *Class diagram*

Package Explorer View

Figure 10-2 shows the high-level structure of the program.

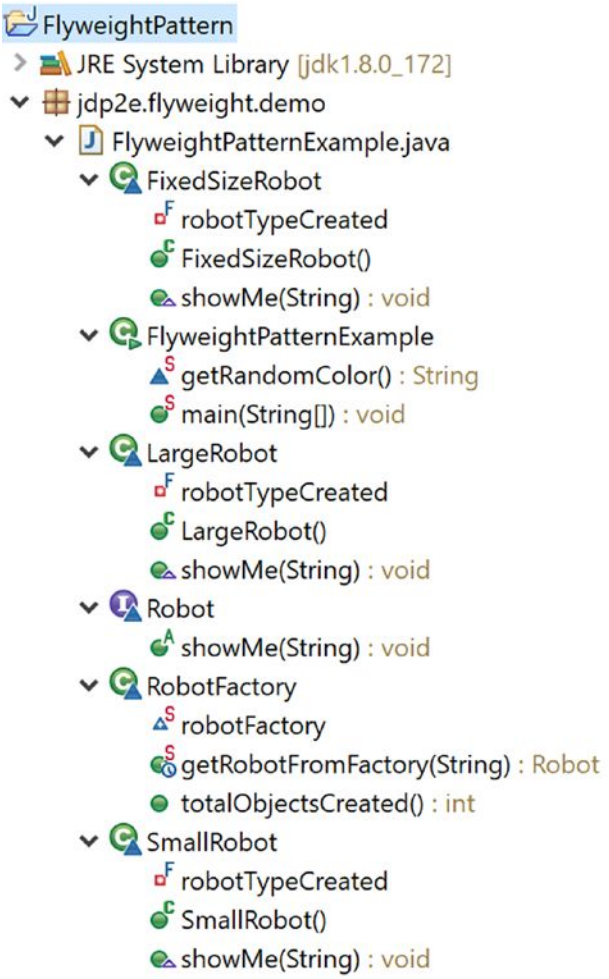


Figure 10-2. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.flyweight.demo;

import java.util.Map;
import java.util.HashMap;
import java.util.Random;
```

```
interface Robot
{
    //Color comes from client.It is extrinsic.
    void showMe(String color);
}
//A shared flyweight implementation
class SmallRobot implements Robot
{
    /*
     * Intrinsic state.
     * It is not supplied by client.
     * So, it is independent of the flyweight's context.
     * This can be shared across.
     * These data are often immutable.
     */
    private final String robotTypeCreated;
    public SmallRobot()
    {
           robotTypeCreated="A small robot created";
           System.out.print(robotTypeCreated);
    }
    @Override
    public void showMe(String color)
    {
        System.out.print(" with " +color + " color");
    }
}
//A shared flyweight implementation
class LargeRobot implements Robot
{
    /*
     * Intrinsic state.
     * It is not supplied by client.
     * So, it is independent of the flyweight's context.
     * This can be shared across.
```

```
     * These data are often immutable.
     */
    private final String robotTypeCreated;
    public LargeRobot()
    {
        robotTypeCreated="A large robot created";
        System.out.print(robotTypeCreated);
    }
    @Override
    public void showMe(String color)
    {
        System.out.print(" with " + color + " color");

    }
}
//An unshared flyweight implementation
class FixedSizeRobot implements Robot
{
    /*
     * Intrinsic state.
     * It is not supplied by client.
     * So, it is independent of the flyweight's context.
     * This can be shared acorss.
     */
    private final String robotTypeCreated;
    public FixedSizeRobot()
    {
        robotTypeCreated="A robot with a fixed size created";
        System.out.print(robotTypeCreated);
    }
    @Override
    //Ingoring the extrinsic state argument
    //Since it is an unshared flyweight
```

```
    public void showMe(String color)
    {
        System.out.print(" with " + " blue (default) color");
    }
}

class RobotFactory
{
    static Map<String, Robot> robotFactory = new HashMap<String, Robot>();
    public int totalObjectsCreated()
    {
        return robotFactory.size();
    }

    public static synchronized Robot getRobotFromFactory(String robotType)
    throws Exception
    {
        Robot robotCategory = robotFactory.get(robotType);
        if(robotCategory==null)
        {
            switch (robotType)
            {
            case "small":
                System.out.println("We do not have Small Robot at present.
                So we are creating a small robot now.") ;
                robotCategory = new SmallRobot();
                break;
            case "large":
                System.out.println("We do not have Large Robot at present.
                So we are creating a large robot now.");
                robotCategory = new LargeRobot();
                break;

            case "fixed":
                System.out.println("We do not have fixed size at present.
                So we are creating a fixed size robot now.");
```

```
                robotCategory = new FixedSizeRobot();
                break;
            default:
                throw new Exception(" Robot Factory can create only small
                ,large or fixed size robots");
            }
            robotFactory.put(robotType,robotCategory);
        }
        else
        {
            System.out.print("\n \t Using existing "+ robotType +" robot
            and coloring it" );
        }
        return robotCategory;
    }
}

public class FlyweightPatternExample {

    public static void main(String[] args) throws Exception {
        RobotFactory robotFactory = new RobotFactory();
        System.out.println("\n***Flyweight Pattern Example ***\n");
        Robot myRobot;
        //Here we are trying to get 3 Small type robots
        for (int i = 0; i < 3; i++)
        {
            myRobot = RobotFactory.getRobotFromFactory("small");
            /*
            Not required to add sleep().But it is included to
            increase the probability of getting a new random number
            to see the variations in the output.
             */
            Thread.sleep(1000);
            //The extrinsic property color is supplied by the client code.
            myRobot.showMe(getRandomColor());
        }
```

```
int numOfDistinctRobots = robotFactory.totalObjectsCreated();
System.out.println("\n Till now, total no of distinct robot objects
created: " + numOfDistinctRobots);

//Here we are trying to get 5 Large type robots
for (int i = 0; i < 5; i++)
{
    myRobot = RobotFactory.getRobotFromFactory("large");
    /*
    Not required to add sleep().But it is included to
    increase the probability of getting a new random number
    to see the variations in the output.
     */
    Thread.sleep(1000);
    //The extrinsic property color is supplied by the client code.
    myRobot.showMe(getRandomColor());
}
numOfDistinctRobots = robotFactory.totalObjectsCreated();
System.out.println("\n Till now, total no of distinct robot objects
created: " + numOfDistinctRobots);

//Here we are trying to get 4 fixed sizerobots
for (int i = 0; i < 4; i++)
{
    myRobot = RobotFactory.getRobotFromFactory("fixed");
    /*
    Not required to add sleep().But it is included to
    increase the probability of getting a new random number
    to see the variations in the output.
     */
    Thread.sleep(1000);
    //The extrinsic property color is supplied by the client code.
    myRobot.showMe(getRandomColor());
}
numOfDistinctRobots = robotFactory.totalObjectsCreated();
```

```
        System.out.println("\n Till now, total no of distinct robot objects
        created: " + numOfDistinctRobots);
    }

    static String getRandomColor()
    {
        Random r = new Random();
        /* I am simply checking the random number generated that can be
        either an even number or an odd number. And based on that we are
        choosing the color. For simplicity, I am using only two colors-red
        and green
         */
        int random = r.nextInt();
        if (random % 2 == 0)
        {
            return "red";
        }
        else
        {
            return "green";
        }
    }

}
```

Output

Here's the first run output.

```
***Flyweight Pattern Example ***

We do not have Small Robot at present.So we are creating a small robot now.
A small robot created with green color
      Using existing small robot and coloring it with green color
      Using existing small robot and coloring it with red color
```

```
 Till now, total no of distinct robot objects created: 1
We do not have Large Robot at present.So we are creating a large robot now.
A large robot created with green color
      Using existing large robot and coloring it with red color
      Using existing large robot and coloring it with green color
      Using existing large robot and coloring it with green color
      Using existing large robot and coloring it with green color
 Till now, total no of distinct robot objects created: 2
We do not have fixed size at present.So we are creating a fixed size robot
now.
A robot with a fixed size created with  blue (default) color
      Using existing fixed robot and coloring it with  blue (default) color
      Using existing fixed robot and coloring it with  blue (default) color
      Using existing fixed robot and coloring it with  blue (default) color
 Till now, total no of distinct robot objects created: 3
```

Here's the second run output.

```
***Flyweight Pattern Example ***

We do not have Small Robot at present.So we are creating a small robot now.
A small robot created with red color
      Using existing small robot and coloring it with green color
      Using existing small robot and coloring it with green color
 Till now, total no of distinct robot objects created: 1
We do not have Large Robot at present.So we are creating a large robot now.
A large robot created with red color
      Using existing large robot and coloring it with green color
      Using existing large robot and coloring it with green color
      Using existing large robot and coloring it with red color
      Using existing large robot and coloring it with green color
 Till now, total no of distinct robot objects created: 2
We do not have fixed size at present.So we are creating a fixed size robot
now.
A robot with a fixed size created with  blue (default) color
      Using existing fixed robot and coloring it with  blue (default) color
```

```
    Using existing fixed robot and coloring it with  blue (default) color
    Using existing fixed robot and coloring it with  blue (default) color
Till now, total no of distinct robot objects created: 3
```

Analysis

- The output varies because in this implementation, I am choosing color at random.
- The fixed-size robot's color never changes because the extrinsic state (color) is ignored to represent an unshared flyweight.
- The client needed to play with 12 robots (3 small, 5 large, 4 fixed-size) but these demands are served by only three distinct template objects (one from each category) and these were configured on the fly.

Q&A Session

1. **I notice some similarities between a singleton pattern and a flyweight pattern. Can you highlight the key differences between them?**

   The singleton pattern helps you maintain only one required object in the system. In other words, once the required object is created, you cannot create more. You need to reuse the existing object.

   The flyweight pattern is generally concerned about a large number of similar (which can be heavy) objects, because they may occupy big blocks of memory. So, you try to create a smaller set of template objects that can be configured on the fly to complete the creation of the heavy objects. These smaller and configurable objects are called flyweights. You can reuse them in your application to appear that you have many large objects. This approach helps you reduce the consumption of big chunks of memory. Basically, flyweights make one look like many. This is why the GoF tells us: *A flyweight is a shared object that can be*

used in multiple contexts simultaneously. The flyweight acts as an independent object in each context — it's indistinguishable from an instance of the object that's not shared.

Figure 10-3 visualizes the core concepts of the flyweight pattern before using flyweights.

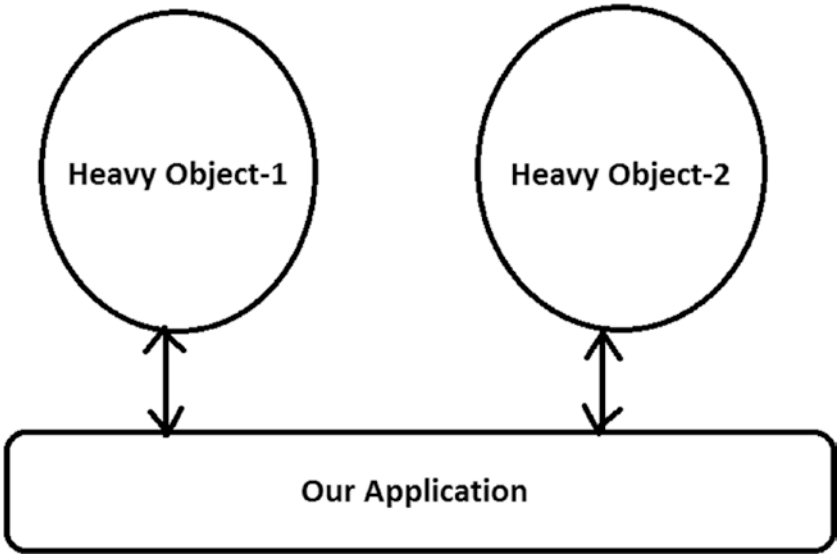


Figure 10-3. *Before using flyweights*

Figure 10-4 shows the design after using flyweights.

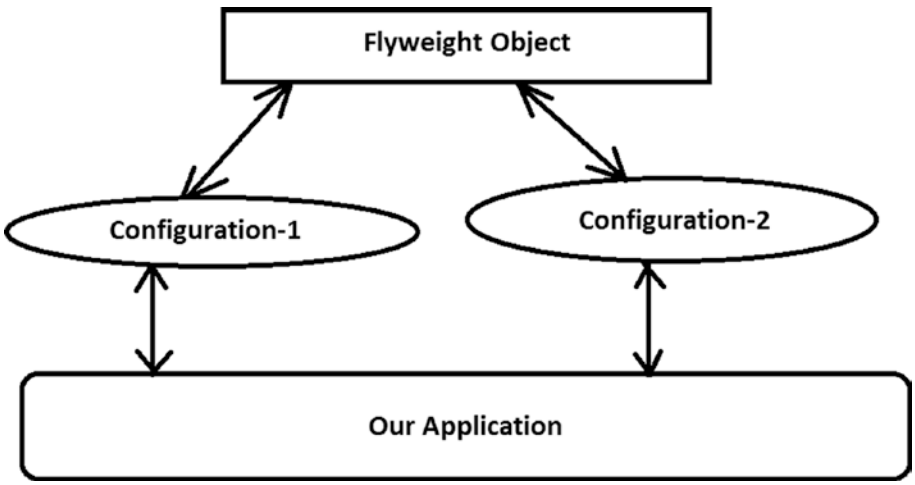


Figure 10-4. *After using flyweights*

In Figure 10-4, you can see that

- Heavy Object 1 = Flyweight Object (shared) + Configuration 1 (extrinsic and not shared)
- Heavy Object 2 = Flyweight Object(shared) + Configuration 2 (extrinsic and not shared)

By combining the intrinsic and extrinsic states, the flyweight objects provide the complete functionality.

2. **Can you observe any impact due to multithreading?**

   If you are creating objects with new operators in a multithreaded environment, you may end up with multiple unwanted objects (similar to singleton patterns). The remedy is similar to the way you handle multithreaded environment in a singleton pattern.

3. **What are the advantages of using flyweight design patterns?**

   - You can reduce memory consumptions of heavy objects that can be controlled identically.
   - You can reduce the total number of "complete but similar objects" in the system.
   - You can provide a centralized mechanism to control the states of many "virtual" objects.

4. **What are the challenges associated with using flyweight design patterns?**

   - In this pattern, you need to take the time to configure these flyweights. The configuration time can impact the overall performance of the application.
   - To create flyweights, you extract a common template class from the existing objects. This additional layer of programming can be tricky and sometimes hard to debug and maintain.
   - You can see that logical instances of a class cannot behave differently from other instances.

- The flyweight pattern is often combined with singleton factory implementation and to guard the singularity, additional cost is required (e.g., you may opt for a synchronized method or double-checked locking, but each of them are costly operations).

5. **Can I have non-shareable flyweight interface?**

   Yes. A flyweight interface does not enforce that it needs to always be shareable. In some cases, you may have non-shareable flyweights with concrete flyweight objects as children. In our example, you saw the use of non-shareable flyweights using fixed-size robots.

6. **Since intrinsic data of flyweights are the same, I can share them. Is this correct?**

   Yes.

7. **How do clients handle the extrinsic data of these flyweights?**

   They need to pass the information (states) to the flyweights. Clients either manage the data or compute them on the fly.

8. **Extrinsic data is not shareable. Is this correct?**

   Yes.

9. **You said that I should try to make intrinsic states immutable. How can I achieve that?**

   Yes, for thread safety and security, experts suggest that you implement that. In this case, it is already implemented. In Java, you must remember that String objects are inherently immutable.

   Also, you may notice that in the concrete flyweights (SmallRobot, LargeRobot, FixedSizeRobot), there are no setter methods to set/modify the value of robotTypeCreated. When you supply the data only through a constructor and there are no setter methods, you are following an approach that promotes immutability.

10. **You have tagged the final keyword with the intrinsic state robotTypeCreated to achieve immutability. Is this correct?**

    You need to remember that *final* and *immutability* are not synonymous. In the context of design patterns, the word *immutability* generally means that once created, you cannot change the state of the object. Although the keyword final can be applied to a class, a method, or a field, the aim is different.

    The final field can help you construct a thread-safe immutable object without synchronization, and it provides safety in a multithreaded environment. So, I used it in this example. The concept is described in detail in the article at `https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.5-110`.

11. **The getRobotFromFactory() method is synchronized here to provide thread safety. Is this understanding correct?**

    Exactly. In a single-threaded environment, it is not required.

12. **The getRobotFromFactory() method is static here. Is that mandatory?**

    No. You can implement a non-static factory method also. You may often notice the presence of a singleton factory with flyweight pattern implementations.

13. **What is the role of "RobotFactory" in this implementation?**

    It caches flyweights and provides a method to get them. In this example, there are many objects that can be shared. So, storing them in a central place is always a good idea.

CHAPTER 11

Composite Pattern

This chapter covers the composite pattern.

GoF Definition

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Concept

To help you understand this concept, I will start with an example. Consider a shop that sells different kinds of dry fruits and nuts; let's say cashews, dates, and walnuts. Each of these items is associated with a certain price. Let's assume that you can purchase any of these individual items or you can purchase "gift packs" (or boxed items) that contain different items. In this case, the cost of a packet is the sum of its component parts. The composite pattern is useful in a similar situation, where you treat both the individual parts and the combination of the parts in the same way so that you can process them uniformly.

This pattern is useful to represent part-whole hierarchies of objects. In object-oriented programming, a composite is an object with a composition of one-or-more similar objects, where each of these objects has similar functionalities. (This is also known as a "has-a" relationship among objects). The usage of this pattern is very common in a tree-like data structure. If you can apply it properly, you do not need to discriminate between a branch and the leaf-nodes. You can achieve two key goals with this pattern.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_11

- You can compose objects into a tree structure to represent a part-whole hierarchy.
- You can access both the composite objects (branches) and the individual objects (leaf-nodes) uniformly. As a result, you can reduce the complexity of codes and at the same time, you make your application less error prone.

Real-World Example

You can also think of an organization that consists of many departments. In general, an organization has many employees. Some of these employees are grouped together to form a department, and those departments can be further grouped together to build the final structure of the organization.

Computer-World Example

Any tree data structure can follow this concept. Clients can treat the *leaves of the tree* and the *non-leaves* (or branches of the tree) in the same way.

Note This pattern is commonly seen in various UI frameworks. In Java, the generic Abstract Window Toolkit (AWT) container object is a component that can contain other AWT components. For example, in java.awt.Container class (which extends java.awt.Component) you can see various overloaded version of add(Component comp) method. In JSF, UIViewRoot class acts like a composite node and UIOutput acts like a leaf node. When you traverse a tree, you often use the iterator design pattern, which is covered in Chapter 18.

Illustration

In this example, I am representing a college organization. Let's assume that there is a principal and two heads of departments—one for computer science and engineering (CSE) and one for mathematics (Maths). In the Maths department, there are two

teachers (or professors/lecturers), and in the CSE department, there are three teachers (or professors/lecturers). The tree structure for this organization is similar to Figure 11-1.

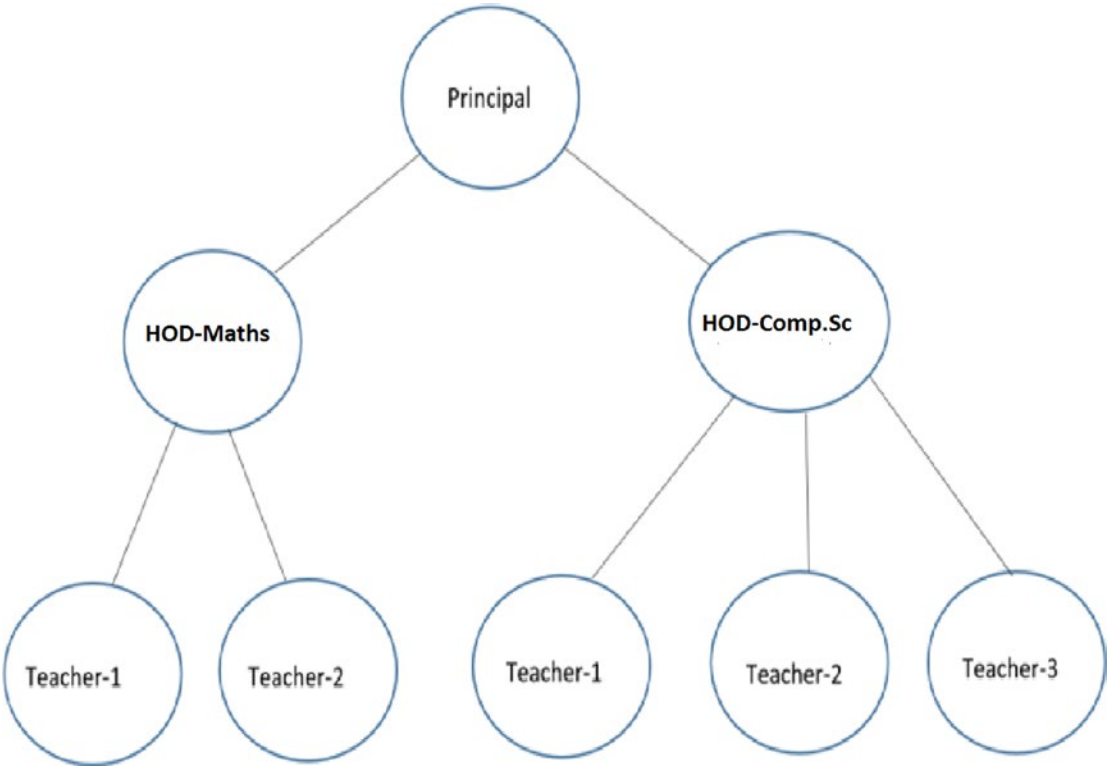


Figure 11-1. *A sample college organization*

Let's also assume that at the end, one lecturer from the CSE department retires. You'll examine all of these cases in the following sections.

Class Diagram

Figure 11-2 shows the class diagram.

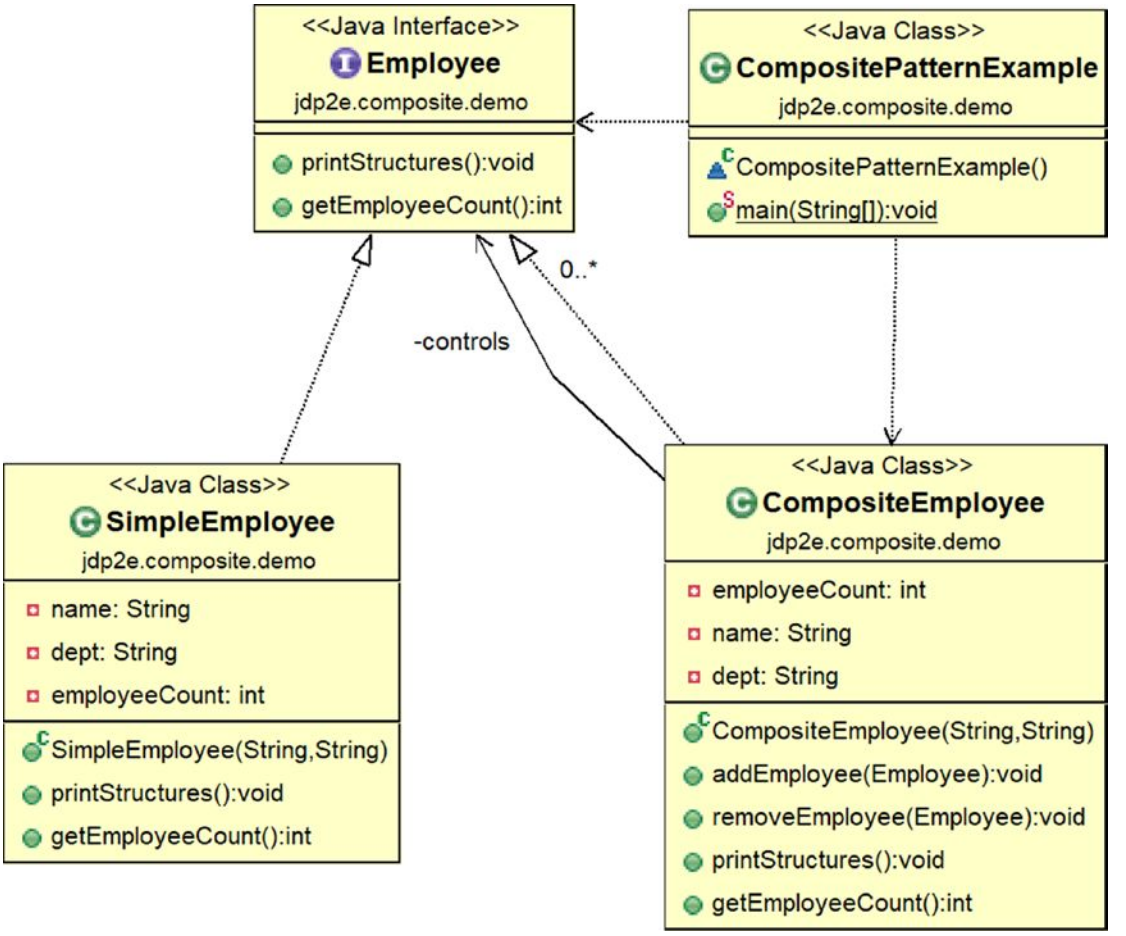


Figure 11-2. *Class diagram*

Package Explorer View

Figure 11-3 shows the high-level structure of the program.

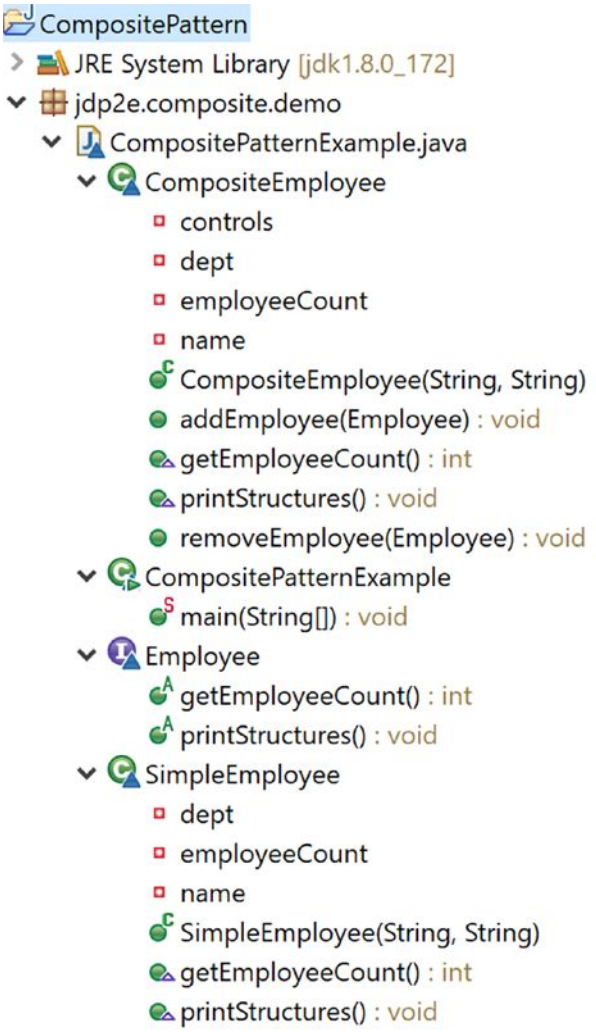


Figure 11-3. *Package Explorer view*

Implementation

Here is the implementation.

```
package jdp2e.composite.demo;

import java.util.ArrayList;
import java.util.List;
```

```
interface IEmployee
{
    void printStructures();
    int getEmployeeCount();
}
class CompositeEmployee implements IEmployee
{
    //private static int employeeCount=0;
    private int employeeCount=0;

    private String name;
    private String dept;
    //The container for child objects
    private List<IEmployee> controls;
    //Constructor
    public CompositeEmployee(String name, String dept)
    {
        this.name = name;
        this.dept = dept;
        controls = new ArrayList<IEmployee>();
    }

    public void addEmployee(IEmployee e)
    {
        controls.add(e);
    }

    public void removeEmployee(IEmployee e)
    {
        controls.remove(e);
    }
    @Override
    public void printStructures()
    {
        System.out.println("\t" + this.name + " works in  " + this.dept);
        for(IEmployee e: controls)
```

```
        {
            e.printStructures();
        }
    }
    @Override
    public int getEmployeeCount()
    {
        employeeCount=controls.size();
        for(IEmployee e: controls)
        {
            employeeCount+=e.getEmployeeCount();
        }
        return employeeCount;
    }
}
class Employee implements IEmployee
{
    private String name;
    private String dept;
    private int employeeCount=0;
    //Constructor
    public Employee(String name, String dept)
    {
        this.name = name;
        this.dept = dept;
    }
    @Override
    public void printStructures()
    {
        System.out.println("\t\t"+this.name + " works in  " + this.dept);
    }
    @Override
```

```
    public int getEmployeeCount()
    {
        return employeeCount;//0
    }

}
class CompositePatternExample {

    /**Principal is on top of college.
     *HOD -Maths and Comp. Sc directly reports to him
     *Teachers of Computer Sc. directly reports to HOD-CSE
     *Teachers of Mathematics directly reports to HOD-Maths
     */
    public static void main(String[] args) {
        System.out.println("***Composite Pattern Demo ***");
        //2 teachers other than HOD works in Mathematics department
        Employee mathTeacher1 = new Employee("Math Teacher-1","Maths");
        Employee mathTeacher2 = new Employee("Math Teacher-2","Maths");

        //teachers other than HOD works in Computer Sc. Department
        Employee cseTeacher1 = new Employee("CSE Teacher-1", "Computer Sc.");
        Employee cseTeacher2 = new Employee("CSE Teacher-2", "Computer Sc.");
        Employee cseTeacher3 = new Employee("CSE Teacher-3", "Computer Sc.");

        //The College has 2 Head of Departments-One from Mathematics, One
        //from Computer Sc.
        CompositeEmployee hodMaths = new CompositeEmployee("Mrs.S.Das(HOD-
        Maths)","Maths");
        CompositeEmployee hodCompSc = new CompositeEmployee(
        "Mr. V.Sarcar(HOD-CSE)", "Computer Sc.");

        //Principal of the college
        CompositeEmployee principal = new CompositeEmployee("Dr.S.Som
        (Principal)","Planning-Supervising-Managing");

        //Teachers of Mathematics directly reports to HOD-Maths
        hodMaths.addEmployee(mathTeacher1);
        hodMaths.addEmployee(mathTeacher2);
```

```
//Teachers of Computer Sc. directly reports to HOD-CSE

hodCompSc.addEmployee(cseTeacher1);
hodCompSc.addEmployee(cseTeacher2);
hodCompSc.addEmployee(cseTeacher3);

/*Principal is on top of college.HOD -Maths and Comp. Sc directly
reports to him*/
principal.addEmployee(hodMaths);
principal.addEmployee(hodCompSc);

/*Printing the leaf-nodes and branches in the same way i.e.
 in each case, we are calling PrintStructures() method
 */
System.out.println("\n Testing the structure of a Principal
object");
//Prints the complete structure
principal.printStructures();
System.out.println("At present Principal has control over "+
principal.getEmployeeCount()+ " number of employees.");

System.out.println("\n Testing the structure of a HOD-CSE
object:");
//Prints the details of Computer Sc, department
hodCompSc.printStructures();
System.out.println("At present HOD-CSE has control over "+
hodCompSc.getEmployeeCount()+ " number of employees.");

System.out.println("\n Testing the structure of a HOD-Maths
object:");
//Prints the details of Mathematics department
hodMaths.printStructures();
System.out.println("At present HOD-Maths has control over "+
hodMaths.getEmployeeCount()+ " number of employees.");
```

```
        //Leaf node
        System.out.println("\n Testing the structure of a leaf node:");
        mathTeacher1.printStructures();
        System.out.println("At present Math Teacher-1 has control over "+
        mathTeacher1.getEmployeeCount()+ " number of employees.");

        /*Suppose, one computer teacher is leaving now
         from the organization*/
        hodCompSc.removeEmployee(cseTeacher2);
        System.out.println("\n After CSE Teacher-2 resigned, the
        organization has following members:");
        principal.printStructures();

        System.out.println("At present Principal has control over "+
        principal.getEmployeeCount()+ " number of employees");
        System.out.println("At present HOD-CSE has control over "+
        hodCompSc.getEmployeeCount()+ " number of employees");
        System.out.println("At present HOD-Maths has control over "+
        hodMaths.getEmployeeCount()+ " number of employees");

    }

}
```

Output

Here is the output. The key changes are shown in bold.

```
***Composite Pattern Demo ***

 Testing the structure of a Principal object
    Dr.S.Som(Principal) works in  Planning-Supervising-Managing
    Mrs.S.Das(HOD-Maths) works in  Maths
        Math Teacher-1 works in  Maths
        Math Teacher-2 works in  Maths
    Mr. V.Sarcar(HOD-CSE) works in  Computer Sc.
        CSE Teacher-1 works in  Computer Sc.
        CSE Teacher-2 works in  Computer Sc.
        CSE Teacher-3 works in  Computer Sc.
```

At present Principal has control over 7 number of employees.

```
 Testing the structure of a HOD-CSE object:
    Mr. V.Sarcar(HOD-CSE) works in  Computer Sc.
        CSE Teacher-1 works in  Computer Sc.
        CSE Teacher-2 works in  Computer Sc.
        CSE Teacher-3 works in  Computer Sc.
```

At present HOD-CSE has control over 3 number of employees.

```
 Testing the structure of a HOD-Maths object:
    Mrs.S.Das(HOD-Maths) works in  Maths
        Math Teacher-1 works in  Maths
        Math Teacher-2 works in  Maths
At present HOD-Maths has control over 2 number of employees.

 Testing the structure of a leaf node:
        Math Teacher-1 works in  Maths
At present Math Teacher-1 has control over 0 number of employees.

 After CSE Teacher-2 resigned, the organization has following members:
    Dr.S.Som(Principal) works in  Planning-Supervising-Managing
    Mrs.S.Das(HOD-Maths) works in  Maths
        Math Teacher-1 works in  Maths
        Math Teacher-2 works in  Maths
    Mr. V.Sarcar(HOD-CSE) works in  Computer Sc.
        CSE Teacher-1 works in  Computer Sc.
        CSE Teacher-3 works in  Computer Sc.
```

At present Principal has control over 6 number of employees

At present HOD-CSE has control over 2 number of employees

```
At present HOD-Maths has control over 2 number of employees
```

Q&A Session

1. **What are the advantages of using composite design patterns?**

   - In a tree-like structure, you can treat both the composite objects (branches) and the individual objects (leaf-nodes) uniformly. Notice that in this example, I have used two common methods: `printStructures()` and `getEmployeeCount()` to print the structure and get the employee count from both the composite object structure (principal or HODs) and the single object structure (i.e., leaf nodes like Math Teacher 1.)
   - It is very common to implement a part-whole hierarchy using this design pattern.
   - You can easily add a new component to an existing architecture or delete an existing component from your architecture.

2. **What are the challenges associated with using composite design patterns?**

   - If you want to maintain the ordering of child nodes (e.g., if parse trees are represented as components), you may need to use additional efforts.
   - If you are dealing with immutable objects, you cannot simply delete those.
   - You can easily add a new component but that kind of support can cause maintenance overhead in the future. Sometimes, you want to deal with a composite object that has special components. This kind of constraint can cause additional development costs because you may need to implement a dynamic checking mechanism to support the concept.

3. **In this example, you have used list data structure. But I prefer to use other data structures. Is this okay?**

   Absolutely. There is no universal rule. You are free to use your preferred data structure. GoF confirmed that it is not necessary to use any general-purpose data structure.

4. **How can you connect the iterator design pattern to a composite design pattern?**

   Go through our example once again. If you want to examine composite object architecture, you may need to iterate over the objects. In other words, if you want to do special activities with branches, you may need to iterate over its leaf nodes and non-leaf nodes. Iterator patterns are often used with composite patterns.

5. **In the interface of your implementation, you defined only two methods: printStructures() and getEmployeeCount(). But you are using other methods for the addition and removal of objects in the Composite class (CompositeEmployee). Why didn't you put these methods in the interface?**

   Nice observation. GoF discussed this. Let's look at what happens if you put the `addEmployee (…)` and `removeEmployee (…)` methods in the interface. The leaf nodes need to implement the addition and removal operations. But will it be meaningful in this case? The obvious answer is no. It may appear that you lose transparency, but I believe that you have more safety because I have blocked the meaningless operations in the leaf nodes. This is why the GoF mentioned that this kind of decision involves a trade-off between safety and transparency.

6. **I want to use an abstract class instead of an interface. Is this allowed?**

   In most of the cases, the simple answer is yes. But you need to understand the difference between an abstract class and an interface. In a typical scenario, you find one of them more useful than the other one. Since I am presenting only simple and easy to understand examples, you may not see much difference between the two. Particularly in this example, if I use the abstract class instead of the interface, I may put a default implementation of `getEmployeeCount()` in the abstract class definition. Although you can still argue that with Java's default keyword, you could achieve the same, as in the following:

```
interface IEmployee
{
    void printStructures();
    //int getEmployeeCount();
    default public int getEmployeeCount()
    {
        return 0;
    }
}
```

Note In the Q&A session of the builder pattern (see Chapter 3), I discussed how to decide between an abstract class and an interface.

CHAPTER 12

Bridge Pattern

This chapter covers the bridge pattern.

GoF Definition

Decouple an abstraction from its implementation so that the two can vary independently.

Concept

This pattern is also known as the *handle/body pattern*, in which you separate an implementation from its abstraction and build separate inheritance structures for them. Finally, you connect them through a bridge.

You must note that the abstraction and the implementation can be represented either through an interface or an abstract class, but the abstraction contains a reference to its implementer. Normally, a child of an abstraction is called a *refined abstraction* and a child of an implementation is called a *concrete implementation*.

This bridge interface makes the functionality of concrete classes independent from the interface implementer classes. You can alter different kinds of classes structurally without affecting each other.

Real-World Example

In a software product development company, the development team and the marketing team both play a crucial role. Marketing teams do market surveys and gather customers' needs, which may vary depending on the nature of the customers. Development teams implement those requirements in their products to fulfill the customers' needs.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_12

Any change (e.g., in the operational strategy) in one team should not have a direct impact on the other team. Also, when new requirements come from the customer side, it should not change the way that developers work in their organization. In a software organization, the marketing team plays the role of the bridge between the clients and the development team.

Computer-World Example

GUI frameworks can use the bridge pattern to separate abstractions from platform-specific implementation. For example, using this pattern, it can separate a window abstraction from a window implementation for Linux or macOS.

Note In Java, you may notice the use of JDBC, which provides a bridge between your application and a particular database. For example, the java.sql.DriverManager class and the java.sql.Driver interface can form a bridge pattern where the first one plays the role of abstraction and the second one plays the role of implementor. The concrete implementors are com.mysql.jdbc.Driver or oracle.jdbc.driver.OracleDriver, and so forth.

Illustration

Suppose that you are a remote-control maker and you need to make remote controls for different electronic items. For simplicity, let's assume that you are presently getting orders to make remote controls for televisions and DVD players. Let's also assume that your remote control has two major functionalities: on and off.

You may want to start with the design shown in Figure 12-1 or the one shown in Figure 12-2.

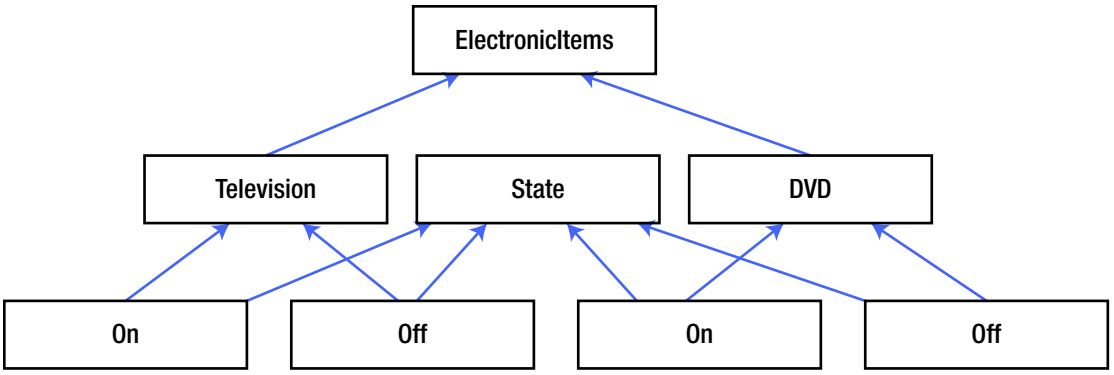


Figure 12-1. *Approach 1*

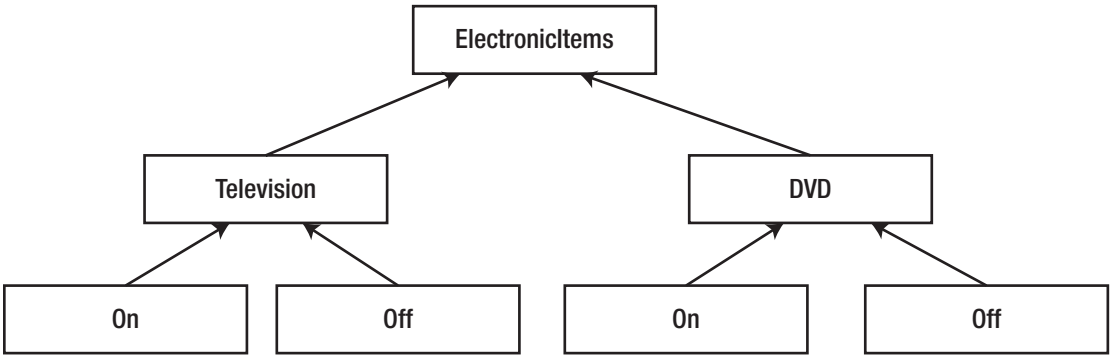


Figure 12-2. *Approach 2*

On further analysis, you discover that Approach 1 is truly messy and difficult to maintain.

At first, Approach 2 looks cleaner, but if you want to include new states, such as sleep, mute, and so forth, or if you want to include new electronic items, such as AC, DVD, and so on, you face new challenges because the elements are tightly coupled in this design approach. But in a real-world scenario, this kind of enhancement is often required.

This is why, you need to start with a loosely coupled system for future enhancements so that either of the two hierarchies (electronics items and their states) can grow independently. The bridge pattern perfectly fits this scenario.

Let's start with the most common bridge pattern class diagram (see Figure 12-3).

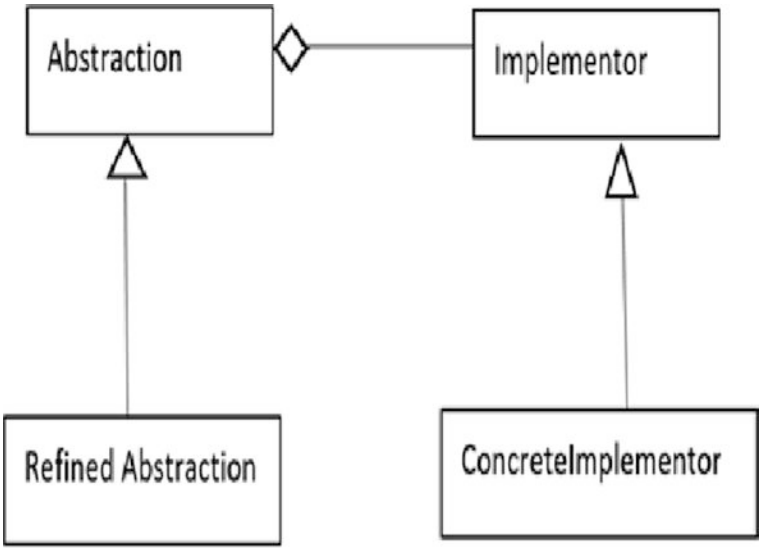


Figure 12-3. *A classic bridge pattern*

- *Abstraction* (an abstract class) defines the abstract interface and it maintains the Implementor reference.
- *RefinedAbstraction* (a concrete class) extends the interface defined by Abstraction.
- *Implementor* (an interface) defines the interface for implementation classes.
- *ConcreteImplementor* (Concrete class) implements the `Implementor` interface.

I followed a similar architecture in the following implementation. For your ready reference, I have pointed out all the participants in the following implementation with comments.

Class Diagram

Figure 12-4 shows the class diagram.

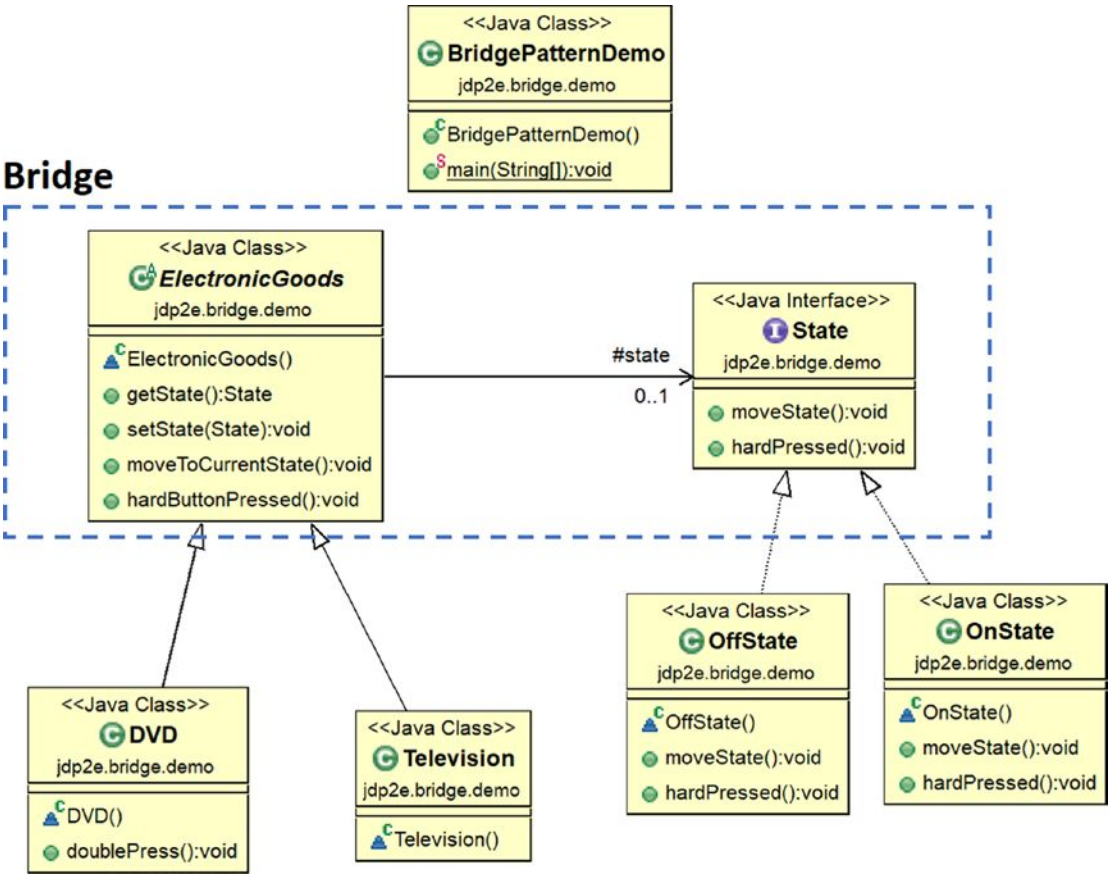


Figure 12-4. *Class diagram*

Package Explorer View

Figure 12-5 shows the high-level structure of the program.

Figure 12-5. *Package Explorer view*

Key Characteristics

Here are the key characteristics of the following implementation.

- The ElectronicGoods abstract class plays the role of abstraction. The State interface plays the role of the implementor.
- The concrete implementors are OnState class and OffState class. They have implemented the `moveState()` and `hardPressed()interface` methods as per their requirements.
- The ElectronicGoods abstract class holds a reference of the State implementor.
- The abstraction methods are delegating the implementation to the implementor object. For example, notice that `hardButtonPressed()` is actually shorthand for `state.hardPressed()`, where state is the implementor object.
- There are two refined abstractions: Television and DVD. The class is happy with the methods that it inherits from its parent. But the DVD class wants to provide an additional feature, so it implements a DVD-specific method: `doublePress()`. *The doublePress() method is coded in terms of superclass abstraction only*.

Implementation

Here is the implementation.

```
package jdp2e.bridge.demo;

//Implementor
interface State
{
    void moveState();
    void hardPressed();
}
//A Concrete Implementor.
class OnState implements State
{
```

```
    @Override
    public void moveState()
    {
        System.out.print("On State\n");
    }

    @Override
    public void hardPressed()
    {
        System.out.print("\tThe device is already On.Do not press the
        button so hard.\n");

    }
}
//Another Concrete Implementor.
class OffState implements State
{
    @Override
    public void moveState()
    {
        System.out.print("Off State\n");
    }

    @Override
    public void hardPressed()
    {
        System.out.print("\tThe device is Offline now.Do not press the off
        button again.\n");

    }
}
//Abstraction
abstract class ElectronicGoods
{
    //Composition - implementor
    protected State state;
    /*Alternative approach:
```

```
      We can also pass an implementor (as input argument) inside a
constructor.
     */
    /*public ElectronicGoods(State state)
    {
        this.state = state;
    }*/
    public State getState()
    {
        return state;
    }

    public void setState(State state)
    {
        this.state = state;
    }
    /*Implementation specific:
      We are delegating the implementation to the Implementor object.
     */
    public void moveToCurrentState()
    {
        System.out.print("The electronic item is functioning at : ");
        state.moveState();
    }
    public void hardButtonPressed()
    {
        state.hardPressed();
    }

}
//Refined Abstraction
//Television does not want to modify any superclass method.
class Television extends ElectronicGoods
{
```

```
    /*public Television(State state)
    {
        super(state);
    }*/
}
/*DVD class also ok with the super class method.
In addition to this, it uses one additional method*/
class DVD extends ElectronicGoods
{

    /*public DVD(State state)
    {
        super(state);
    }*/
    /* Notice that following DVD specific method is coded with superclass
    methods but not with the implementor (State) method.So, this approach
    will allow to  vary the abstraction and implementation independently.
     */
    public void doublePress() {
        hardButtonPressed();
        hardButtonPressed();
    }
}
public class BridgePatternDemo {

    public static void main(String[] args) {
        System.out.println("***Bridge Pattern Demo***");

        System.out.println("\n Dealing with a Television at present.");

        State presentState = new OnState();
        //ElectronicGoods eItem = new Television(presentState);
        ElectronicGoods eItem = new Television();
        eItem.setState(presentState);
        eItem.moveToCurrentState();
        //hard press
        eItem.hardButtonPressed();
```

```
        //Verifying Off state of the Television now
        presentState = new OffState();
        //eItem = new Television(presentState);
        eItem.setState(presentState);
        eItem.moveToCurrentState();

        System.out.println("\n Dealing with a DVD now.");
        presentState = new OnState();
        //eItem = new DVD(presentState);
        eItem = new DVD();
        eItem.setState(presentState);
        eItem.moveToCurrentState();

        presentState = new OffState();
        //eItem = new DVD(presentState);
        eItem = new DVD();
        eItem.setState(presentState);
        eItem.moveToCurrentState();

        //hard press-A DVD specific method
        //(new DVD(presentState)).doublePress();
        ((DVD)eItem).doublePress();

        /*The following line of code will cause error because a television
        object does not have this method.*/
        //(new Television(presentState)).doublePress();
    }
}
```

Output

Here is the output.

```
***Bridge Pattern Demo***

 Dealing with a Television at present.
The electronic item is functioning at : On State
The device is already On.Do not press the button so hard.
The electronic item is functioning at : Off State
```

```
 Dealing with a DVD now.
The electronic item is functioning at : On State
The electronic item is functioning at : Off State
    The device is Offline now.Do not press the off button again.
    The device is Offline now.Do not press the off button again.
```

Q&A Session

1. **This pattern looks similar to a state pattern. Is this correct?**

   No. The state pattern falls into the behavioral pattern and its intent is different. In this chapter, you have seen an example where the electronic items can be in different states, but the key intent was to show that

   - How you can avoid tight coupling between the items and their states.
   - How you can maintain two different hierarchies and both of them can extend without making an impact to each other.

   In addition to these points, you are dealing with multiple objects in which implementations are shared among themselves.

   For a better understanding, go through the comments that are attached with this implementation. I am also drawing your attention to the DVD-specific `doublePress()` method. Notice that it is constructed with superclass methods, which in turn delegate the implementation to the implementor object (a state object in this case). This approach allows you to vary the abstraction and implementation independently, which is the key objective of the bridge pattern.

2. **You could use simple subclassing instead of this kind of design. Is this correct?**

   No. With simple subclassing, your implementations cannot vary dynamically. It may appear that the implementations behave differently with subclassing techniques, but actually, those kinds of variations are already bound to the abstraction at compile time.

3. **In this example, I see lots of dead code. Why are you keeping those?**

   Some developers prefer constructors over Getter/Setter methods. You can see the variations in different implementations. I am keeping those for your ready reference. You are free to use any of them.

4. **What are key advantages of using a bridge design pattern?**

   - The implementations are not bound to the abstractions.
   - Both the abstractions and the implementations can grow independently.
   - Concrete classes are independent from the interface implementer classes (i.e., changes in one of these does not affect the other). You can also vary the interface and the concrete implementations in different ways.

5. **What are the challenges associated with this pattern?**

   - The overall structure may become complex.
   - Sometimes it is confused with the adapter pattern. (The key purpose of an adapter pattern is to deal with incompatible interfaces only.)

6. **Suppose I have only one state; for example, either OnState or OffState. In this case, do I need to use the State interface?**

   No, it is not mandatory. GoF classified this case as a degenerate case of the bridge pattern.

7. **In this example, an abstract class is used to represent an abstraction and an interface is used for an implementation. Is it mandatory?**

   No. You can also use an interface for abstraction. Basically, you can use either of an abstract class or an interface for any of the abstractions or implementations. I simply used this format for better readability.

CHAPTER 13

Visitor Pattern

This chapter covers the visitor pattern.

GoF Definition

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Concept

This pattern helps you add new operations on the objects without modifying the corresponding classes, especially when your operations change very often. Ideally, visitors define class-specific methods, which work with an object of that class to support new functionalities. Here you separate an algorithm from an object structure, and you add new operations using a new hierarchy. Therefore, this pattern can support the open/close principle (extension is allowed but modification is disallowed for entities like class, function, modules, etc.). The upcoming implementations will make the concept clearer to you.

Note You can experience the true power of this design pattern when you combine it with the composite pattern, as shown in the modified implementation later in this chapter.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_13

Real-World Example

Think of a taxi-booking scenario. When the tax arrives and you get into it, the taxi driver takes the control of the transportation. The taxi may take you to your destination through a new route that you are not familiar with. So, you can explore the new route with the help of the taxi driver. But you should use the visitor pattern carefully, otherwise, you may encounter some problem. (For example, consider a case when your taxi driver alters the destination unknowingly, and you face the trouble).

Computer-World Example

This pattern is very useful when public APIs need to support plugging operations. Clients can then perform their intended operations on a class (with the visiting class) without modifying the source.

Note In Java, you may notice the use of this pattern when you use the abstract class org.dom4j.VisitorSupport, which extends Object and implements the org.dom4j.Visitor interface. Also, when you work with the javax.lang.model.element.Element interface or javax.lang.model.element.ElementVisitor<R,P> (where R is the return type of visitor's method and P is the type of additional parameter to the visitor's method), you may notice the use of visitor design pattern.

Illustration

Here our discussion will start with a simple example of the visitor design pattern. Let's assume that you have an inheritance hierarchy where a MyClass concrete class implements the OriginalInterface interface. MyClass has an integer, myInt. When you create an instance of MyClass, it is initialized with a value, 5. Now suppose, you want to update this initialized value and display it. You can do it in two different ways: you can add a method inside MyClass to do your job or use a visitor pattern, which I am about to explain.

In the following implementation, I am multiplying the existing value by 2 and displaying this double value of myInt using the visitor design pattern. If I do not use this pattern, I need to add an operation (or method) inside MyClass, which does the same.

But there is a problem with the later approach. If you want to further update the logic (e.g., you want to triple myInt and display the value), you need to modify the operation in MyClass. One drawback with this approach is that if there are many classes involved, it will be tedious to implement this updated logic in all of them.

But in a visitor pattern, you can just update the visitor's method. The advantage is that you do not need to change the original classes. This approach helps you when your operations change quite often.

So, let's start with an example. Let's assume that in this example, you want to double the initial integer value in MyClass and manipulate it, but your constraint is that you cannot change the original codes in the OriginalInterface hierarchy. So, you are using a visitor pattern in this case.

To achieve the goal, in the following example, I am separating the functionality implementations (i.e., algorithms) from the original class hierarchy.

Class Diagram

Figure 13-1 shows the class diagram.

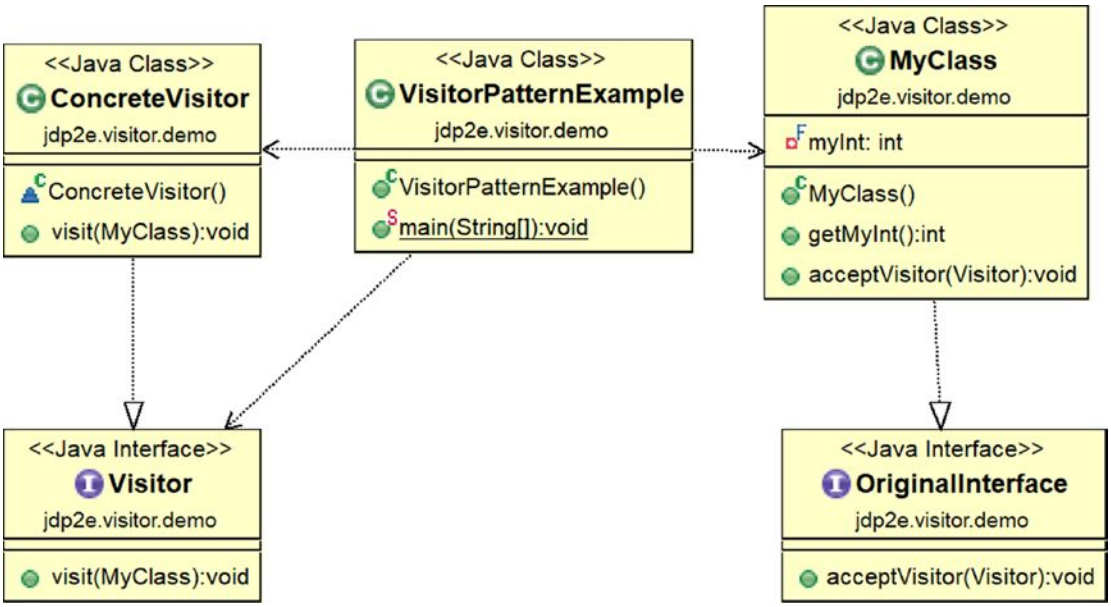


Figure 13-1. *Class diagram*

Package Explorer View

Figure 13-2 shows the high-level structure of the program.

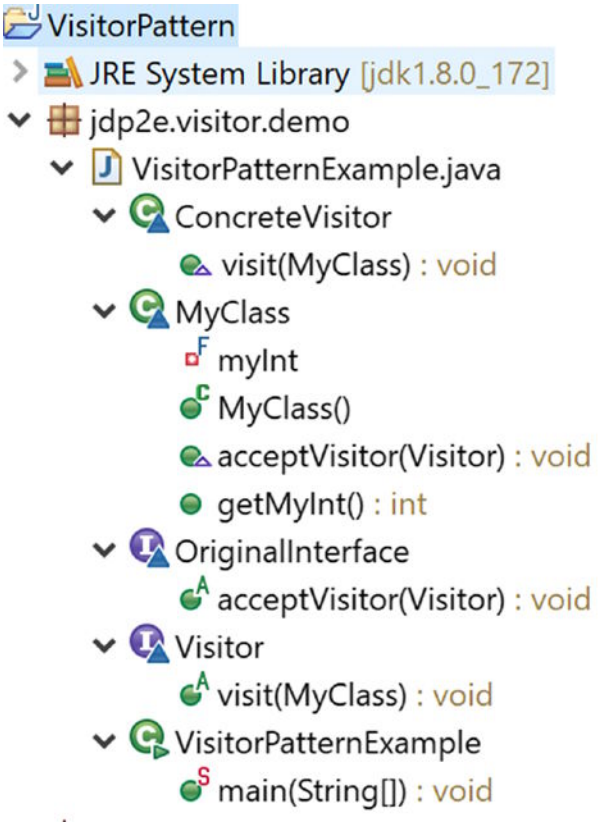


Figure 13-2. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.visitor.demo;
interface OriginalInterface
{
    //The following method has a Visitor argument.
    void acceptVisitor(Visitor visitor);
}
```

```
class MyClass implements OriginalInterface
{
    //Here "myInt" is final.So, once initialized, it should not be changed.
    private final int myInt;
    public MyClass()
    {
        myInt=5;//Initial or default value
    }
    public int getMyInt()
    {
        return myInt;
    }

    @Override
    public void acceptVisitor(Visitor visitor)
    {
        visitor.visit(this);
    }
}

interface Visitor
{
    //The method to visit MyClass
    void visit(MyClass myClassObject);
}
class ConcreteVisitor implements Visitor
{
    @Override
    public void visit(MyClass myClassObject)
    {
        System.out.println("Current value of myInt="+ myClassObject.
        getMyInt());
        System.out.println("Visitor will make it double and display it.");
        System.out.println("Current value of myInt="+ 2*myClassObject.
        getMyInt());
```

```
        System.out.println("Exiting from Visitor.");
    }

}

public class VisitorPatternExample {

    public static void main(String[] args) {
        System.out.println("***Visitor Pattern Demo***\n");
        Visitor visitor = new ConcreteVisitor();
        MyClass myClass = new MyClass();
        myClass.acceptVisitor(visitor);
    }
}
```

Output

Here's the output.

```
***Visitor Pattern Demo***

Current value of myInt=5
Visitor will make it double and display it.
Current value of myInt=10
Exiting from Visitor.
```

Modified Illustration

You have already seen a very simple example of the visitor design pattern. But you can exercise the true power of this design pattern when you combine it with the composite pattern (see Chapter 11). So, let's examine a scenario where you need to combine both the composite pattern and the visitor pattern.

Key Characteristic of the Modified Example

Let's revisit the example of our composite design pattern from Chapter 11. In that example, there is a college with two different departments. Each of these departments has one head of department (HOD) and multiple teachers (or professors/lecturers). Each HOD reports to the principal of the college. Figure 13-3 shows the tree structure that I discussed in that chapter.

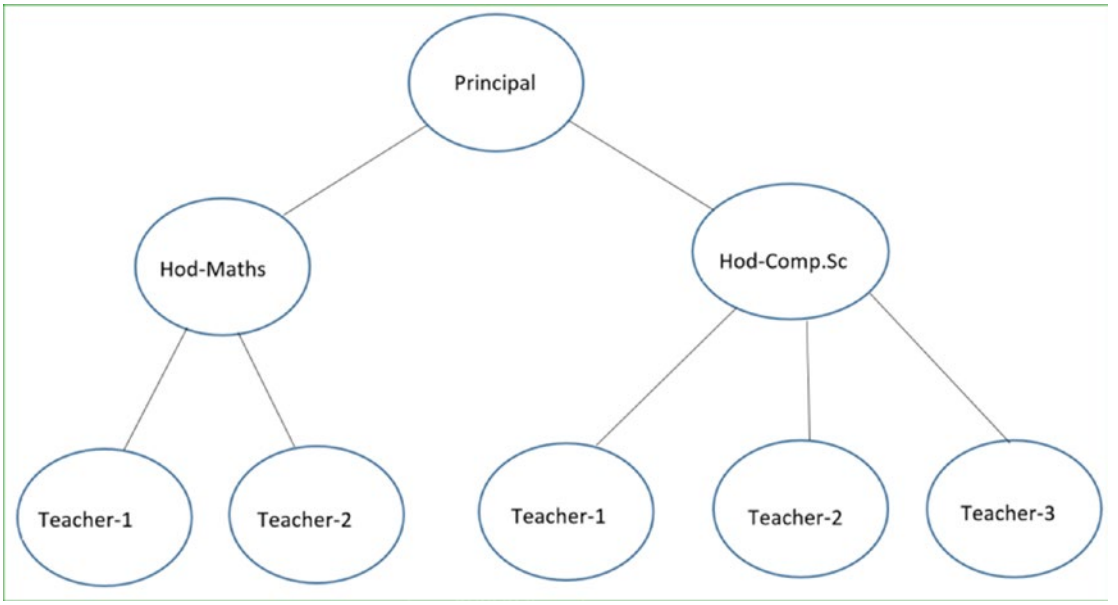


Figure 13-3. *Tree structure of a composite design example*

Now suppose that the principal of the college wants to promote a few employees. Let's consider that teaching experience is the only criteria to promote someone. Ideally, the criteria should vary among senior teachers and junior teachers. *So, let's assume that for a junior teacher, the minimum criteria for promotion is 12 years and for senior teachers, it is 15 years.*

To accomplish this, you need to introduce a new field, yearsOfExperience. So, when a visitor gathers the necessary information from the college, it shows the eligible candidates for promotion.

The visitor is collecting the data from the original college structure without making any modifications to it, and once the collection process is over, it analyses the data to display the intended results. To understand this visually, you can follow the arrows in the upcoming figures. *The principal is at the top of the organization, so you can assume that no promotion is required for that person.*

Step 1

Figure 13-4 shows step 1.

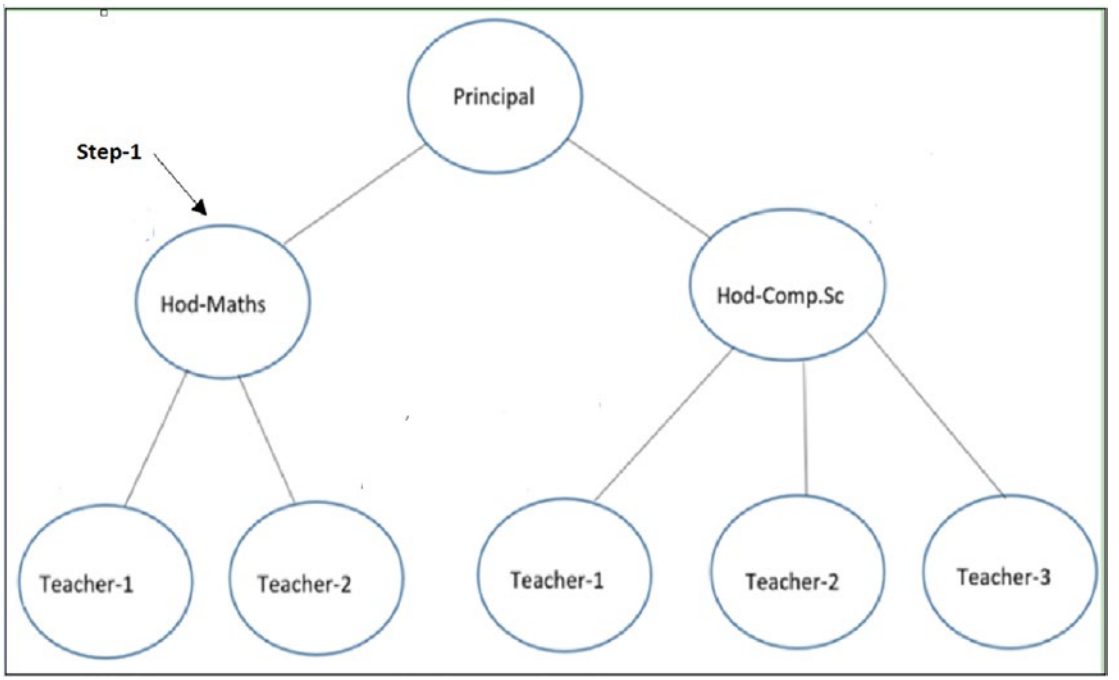


Figure 13-4. *Step 1*

Step 2

Figure 13-5 shows step 2.

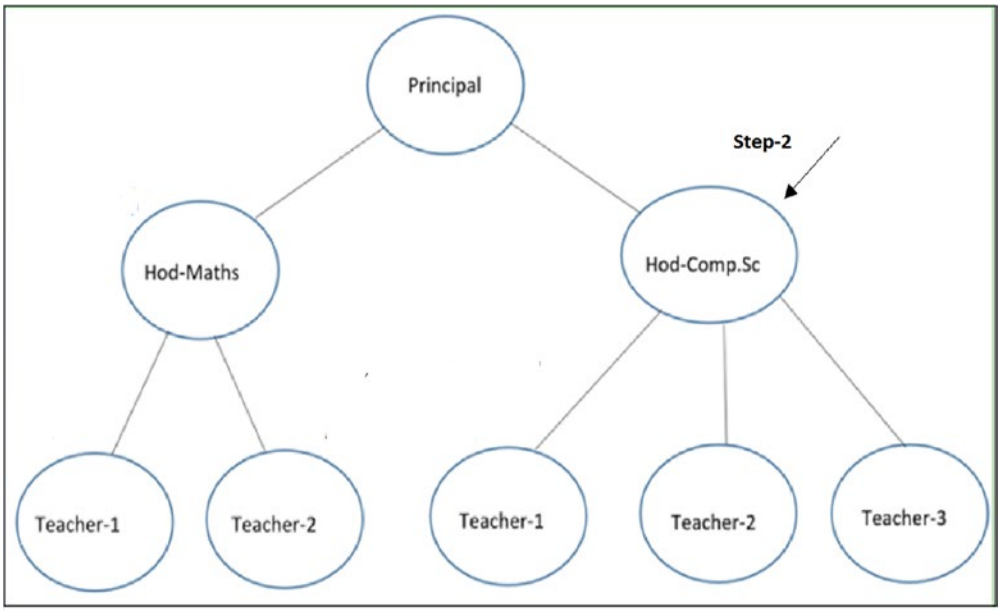


Figure 13-5. *Step 2*

Step 3

Figure 13-6 shows step 3.

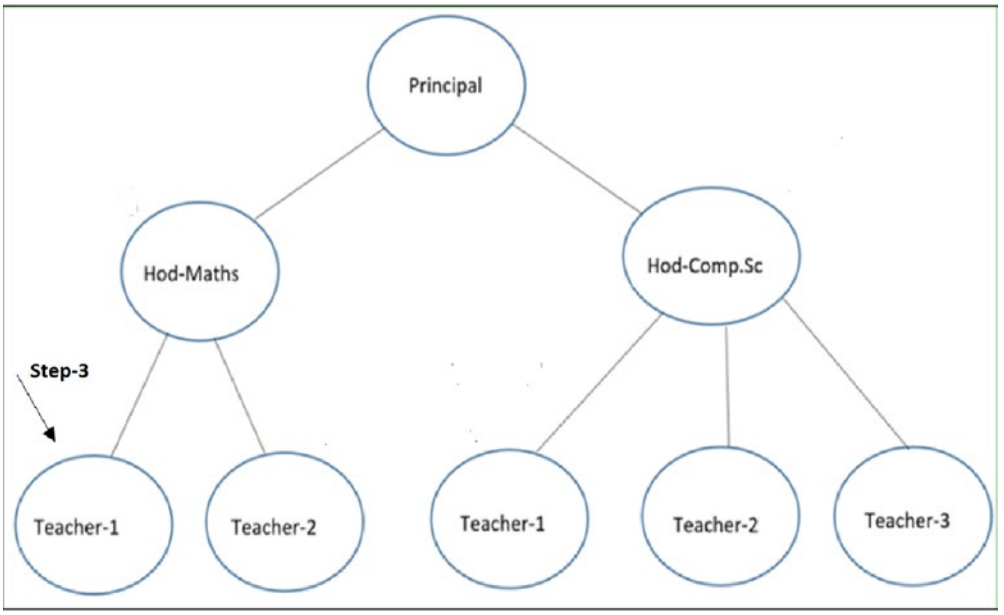


Figure 13-6. *Step 3*

Step 4

Figure 13-7 shows step 4.

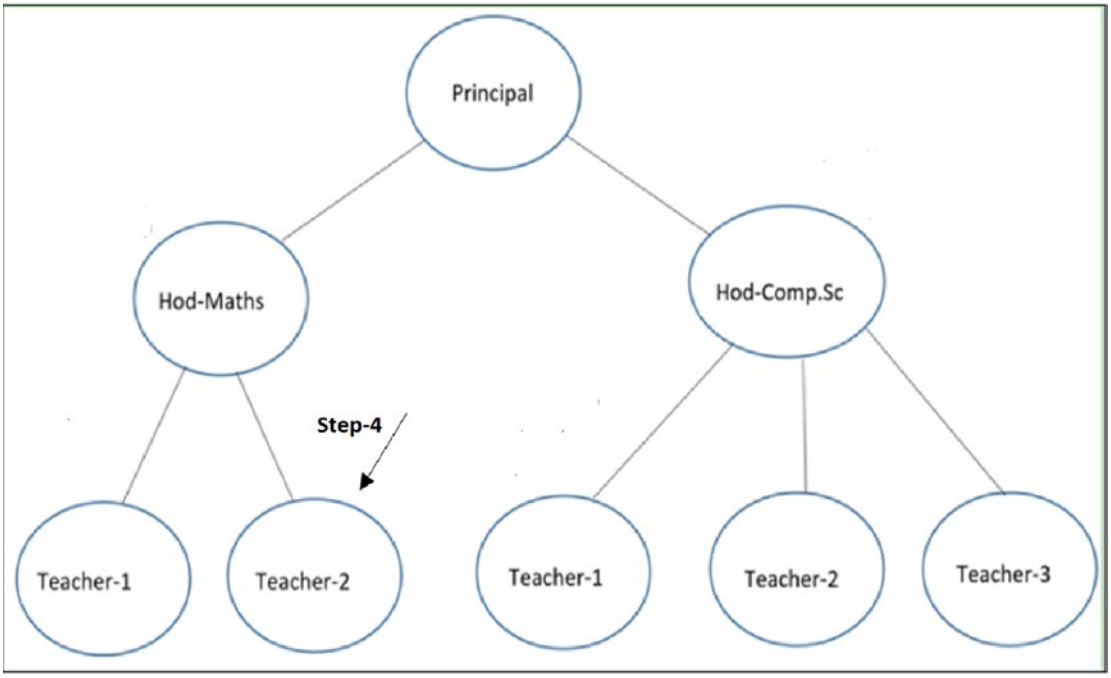


Figure 13-7. *Step 4*

Step 5

Figure 13-8 shows step 5.

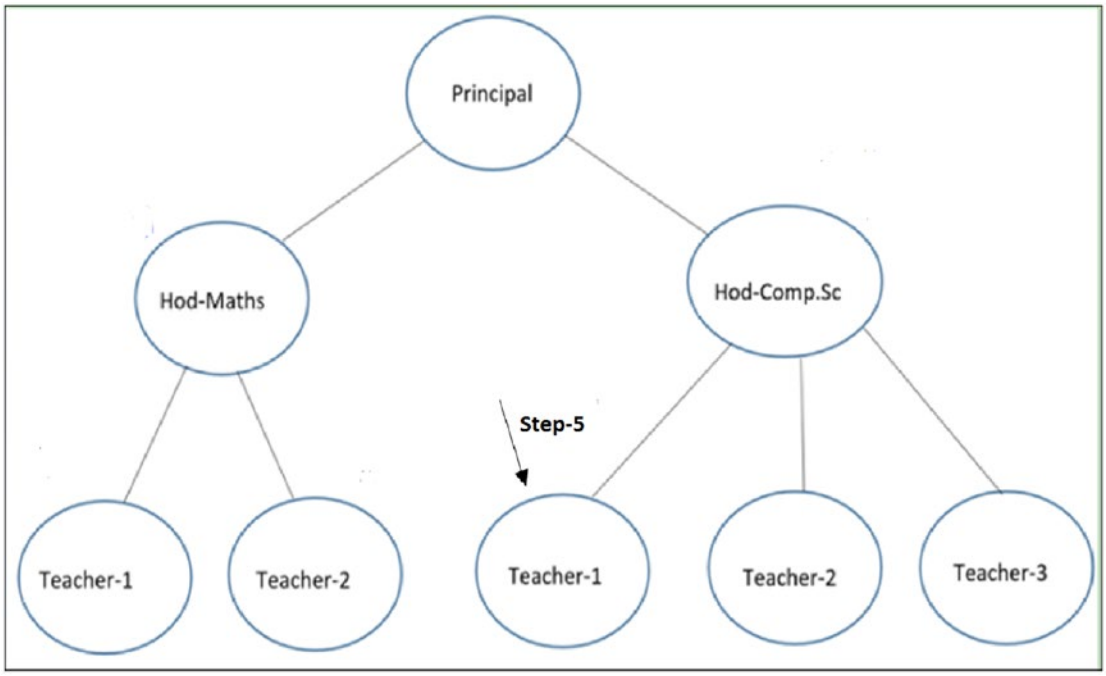


Figure 13-8. *Step 5*

And so on…

In the following implementation, there are code blocks like the following.

```
@Override
public void acceptVisitor(Visitor visitor)
{
    visitor.visitTheElement(this);
}
```

From this structure, you can see two important things.

- Each time a visitor visits a particular object, the object invokes a method on the visitor, passing itself as an argument. The visitor has methods that are specific to a particular class.
- Objects of the concrete employee classes (CompositeEmployee, SimpleEmployee) only implement the acceptVisitor(Visitor visitor) method. *These objects know about the specific method of the visitor (which is passed as an argument here) that it should invoke.*

So, let's start.

Modified Class Diagram

Figure 13-9 shows the modified class diagram.

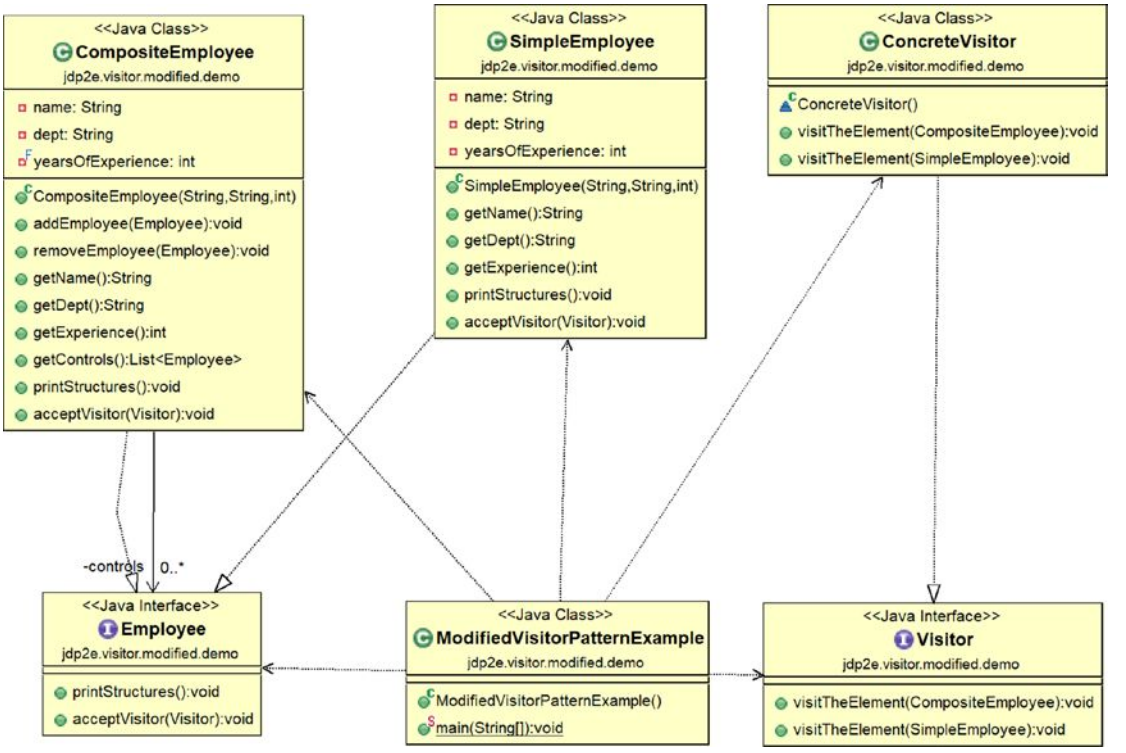


Figure 13-9. *Modified class diagram*

Modified Package Explorer View

Figure 13-10 shows the high-level structure of the modified program.

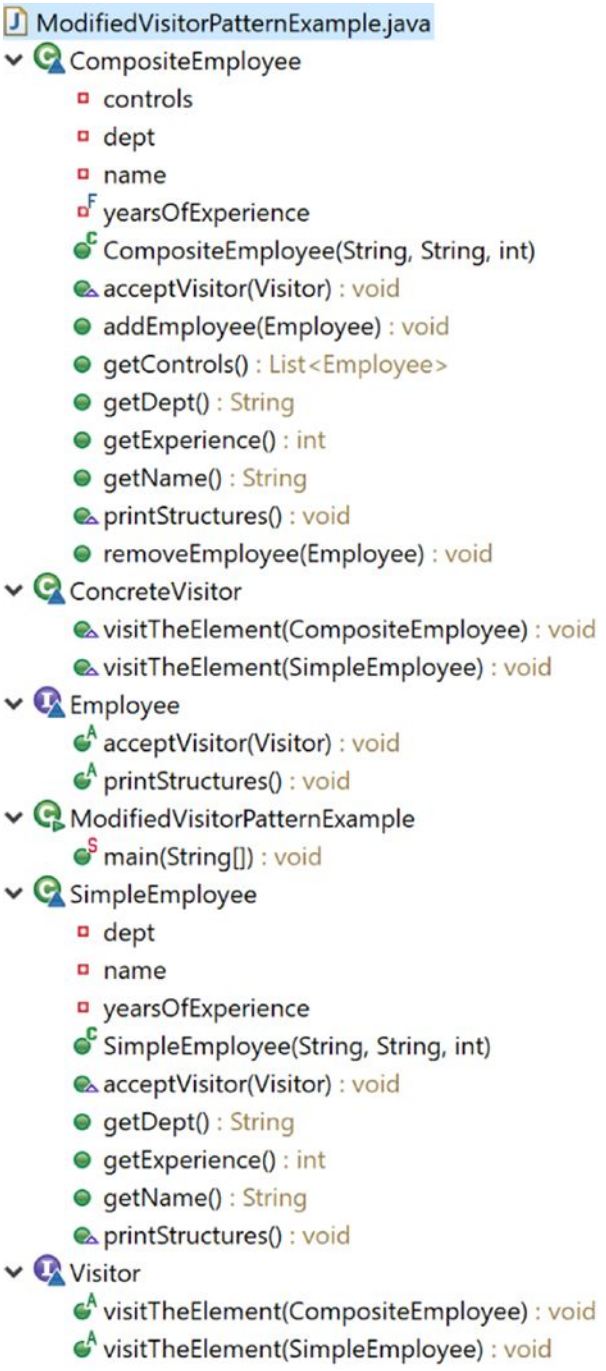


Figure 13-10. *Modified Package Explorer view*

Modified Implementation

Here's the modified implementation.

```
package jdp2e.visitor.modified.demo;
import java.util.ArrayList;
import java.util.List;

interface Employee
{
    void printStructures();
    //The following method has a Visitor argument.
    void acceptVisitor(Visitor visitor);
}
//Employees who have Subordinates
class CompositeEmployee implements Employee
{
    private String name;
    private String dept;
    //New field for this example.
    //It is tagged with "final", so visitor cannot modify it.
    private final int  yearsOfExperience;
    //The container for child objects
    private List<Employee> controls;
    // Constructor
    public CompositeEmployee(String name,String dept, int experience)
    {
        this.name = name;
        this.dept = dept;
        this.yearsOfExperience = experience;
        controls = new ArrayList<Employee>();
    }
    public void addEmployee(Employee e)
    {
        controls.add(e);
    }
```

```
public void removeEmployee(Employee e)
{
    controls.remove(e);
}
// Gets the name
public String getName()
{
    return name;
}
// Gets the department name

public String getDept()
{
    return dept;
}

// Gets the yrs. of experience
public int getExperience()
{
    return yearsOfExperience;

}
public List<Employee> getControls()
{
    return this.controls;
}
@Override
public void printStructures()
{
    System.out.println("\t" + getName() + " works in  " + getDept() + "
    Experience :" + getExperience() + " years");
    for(Employee e: controls)
    {
        e.printStructures();
    }

}
```

```
    @Override
    public void acceptVisitor(Visitor visitor)
    {
        visitor.visitTheElement(this);
    }
}
class SimpleEmployee implements Employee
{
    private String name;
    private String dept;
    //New field for this example
    private int yearsOfExperience;
    //Constructor
    public SimpleEmployee(String name, String dept, int experience)
    {
        this.name = name;
        this.dept = dept;
        this.yearsOfExperience = experience;
    }
    // Gets the name
    public String getName()
    {
        return name;
    }
    // Gets the department name
    public String getDept()
    {
        return this.dept;
    }
    // Gets the yrs. of experience
    public int getExperience()
    {
        return yearsOfExperience;
    }
```

```
    @Override
    public void printStructures()
    {
        System.out.println("\t\t" + getName() + " works in  " + getDept() +
        " Experience :" + getExperience() + " years");
    }
    @Override
    public void acceptVisitor(Visitor visitor)
    {
        visitor.visitTheElement(this);
    }
}

interface Visitor
{
    void visitTheElement(CompositeEmployee employees);
    void visitTheElement(SimpleEmployee employee);
}
class ConcreteVisitor implements Visitor
{
    @Override
    public void visitTheElement(CompositeEmployee employee)
    {
        //We'll promote them if experience is greater than 15 years
        boolean eligibleForPromotion = employee.getExperience() > 15 ?
        true : false;
        System.out.println("\t\t" + employee.getName() + " from  "
        + employee.getDept() + " is eligible for promotion? " +
        eligibleForPromotion);
    }
    @Override
    public void visitTheElement(SimpleEmployee employee)
    {
        //We'll promote them if experience is greater than 12 years
        boolean eligibleForPromotion = employee.getExperience() > 12 ?
        true : false;
```

```
        System.out.println("\t\t" + employee.getName() + " from  "
        + employee.getDept() + " is eligible for promotion? " +
        eligibleForPromotion);
    }
}

public class ModifiedVisitorPatternExample {

    public static void main(String[] args) {
        System.out.println("***Visitor Pattern combined with Composite
        Pattern Demo***\n");
        /*2 teachers other than HOD works in
         Mathematics department*/
        SimpleEmployee mathTeacher1 = new SimpleEmployee("Math Teacher-1",
        "Maths",13);
        SimpleEmployee mathTeacher2 = new SimpleEmployee("Math Teacher-2",
        "Maths",6);

        /* 3 teachers other than HOD works in
          Computer Sc. department*/
        SimpleEmployee cseTeacher1 = new SimpleEmployee("CSE Teacher-1",
        "Computer Sc.",10);
        SimpleEmployee cseTeacher2 = new SimpleEmployee("CSE Teacher-2",
        "Computer Sc.",13);
        SimpleEmployee cseTeacher3 = new SimpleEmployee("CSE Teacher-3",
        "Computer Sc.",7);

        //The College has 2 Head of Departments-One from Mathematics, One
        from Computer Sc.
        CompositeEmployee hodMaths = new CompositeEmployee("Mrs.S.Das(HOD-
        Maths)","Maths",14);
        CompositeEmployee hodCompSc = new CompositeEmployee("Mr.
        V.Sarcar(HOD-CSE)", "Computer Sc.",16);

        //Principal of the college
        CompositeEmployee principal = new CompositeEmployee("Dr.S.Som
        (Principal)","Planning-Supervising-Managing",20);
```

```
//Teachers of Mathematics directly reports to HOD-Maths
hodMaths.addEmployee(mathTeacher1);
hodMaths.addEmployee(mathTeacher2);

//Teachers of Computer Sc. directly reports to HOD-CSE

hodCompSc.addEmployee(cseTeacher1);
hodCompSc.addEmployee(cseTeacher2);
hodCompSc.addEmployee(cseTeacher3);

/*Principal is on top of college.HOD -Maths and Comp. Sc directly
reports to him */
principal.addEmployee(hodMaths);
principal.addEmployee(hodCompSc);

System.out.println("\n Testing the overall structure");
//Prints the complete structure
principal.printStructures();

System.out.println("\n***Visitor starts visiting our composite
structure***\n");
System.out.println("---The minimum criteria for promotion is as
follows ---");
System.out.println("--Junior Teachers-12 years and Senior
teachers-15 years.--\n");
Visitor myVisitor = new ConcreteVisitor();
/*
 * At first, we are building a container for employees who will be
 considered for promotion.
 Principal is holding the highest position.So, he is not considered
 for promotion.
 */
List<Employee> employeeContainer= new ArrayList<Employee>();
//For employees who directly reports to Principal
for (Employee e : principal.getControls())
{
    employeeContainer.add(e);
}
```

```
        //For employees who directly reports to HOD-Maths
        for (Employee e : hodMaths.getControls())
        {
            employeeContainer.add(e);
        }
        //For employees who directly reports to HOD-Comp.Sc
        for (Employee e : hodCompSc.getControls())
        {
            employeeContainer.add(e);
        }
        //Now visitor can traverse through the container.
        for (Employee e :employeeContainer)
        {
            e.acceptVisitor(myVisitor);
        }
    }
}
```

Modified Output

Here's the modified output.

```
***Visitor Pattern combined with Composite Pattern Demo***

 Testing the overall structure
 Dr.S.Som(Principal) works in  Planning-Supervising-Managing Experience :20
 years
   Mrs.S.Das(HOD-Maths) works in  Maths Experience :14 years
     Math Teacher-1 works in  Maths Experience :13 years
     Math Teacher-2 works in  Maths Experience :6 years
   Mr. V.Sarcar(HOD-CSE) works in  Computer Sc. Experience :16 years
     CSE Teacher-1 works in  Computer Sc. Experience :10 years
     CSE Teacher-2 works in  Computer Sc. Experience :13 years
     CSE Teacher-3 works in  Computer Sc. Experience :7 years

***Visitor starts visiting our composite structure***
```

```
---The minimum criteria for promotion is as follows ---
--Junior Teachers-12 years and Senior teachers-15 years.--

   Mrs.S.Das(HOD-Maths) from  Maths is eligible for promotion? false
  Mr. V.Sarcar(HOD-CSE) from  Computer Sc. is eligible for promotion? true
   Math Teacher-1 from  Maths is eligible for promotion? true
   Math Teacher-2 from  Maths is eligible for promotion? false
  CSE Teacher-1 from  Computer Sc. is eligible for promotion? false
  CSE Teacher-2 from  Computer Sc. is eligible for promotion? true
  CSE Teacher-3 from  Computer Sc. is eligible for promotion? false
```

Q&A Session

1. **When should you consider implementing visitor design patterns?**

   You need to add new operations to a set of objects without changing their corresponding classes. It is one of the primary aims to implement a visitor pattern. When the operations change very often, this approach can be your savior. In this pattern, encapsulation is not the primary concern.

   If you need to change the logic of various operations, you can simply do it through visitor implementation.

2. **Are there any drawbacks associated with this pattern?**

   - Encapsulation is not its key concern. So, you can break the power of encapsulation by using visitors.
   - If you need to frequently add new concrete classes to an existing architecture, the visitor hierarchy becomes difficult to maintain. For example, suppose you want to add another concrete class in the original hierarchy now. Then in this case, you need to modify visitor class hierarchy accordingly to fulfill the purpose.

3. **Why are you saying that a visitor class can violate the encapsulation?**

   In our illustration, I have tested a very simple visitor design pattern in which I show an updated integer value of myInt through the visitor class. Also, in many cases, you may see that the visitor needs to move around a composite structure to gather information from them, and then it can modify that information. So, when you provide this kind of support, you violate the core aim of encapsulation.

4. **Why does this pattern compromise the encapsulation?**

   Here you perform some operations on a set of objects that can be heterogeneous. But your constraint is that you cannot change their corresponding classes. So, your visitor needs a way to access the members of these objects. As a result, you need to expose the information to the visitor.

5. **In the visitor interfaces of the modified implementation, you are using the concept of method overloading (i.e., method names are same). Is this mandatory?**

   No. In my book *Design Patterns in C#*, I used method names like `VisitCompositeElement()` and `VisitLeafNode()` in a similar context. Remember that these interface methods should target the specific classes, such as SimpleEmployee or CompositeEmployee.

6. **Suppose that in the modified implementation, I add a concrete subclass of Employee called UndefinedEmployee. How should I proceed? Should I have another specific method in the visitor interface?**

   Exactly. You need to define a new method that is specific to this new class. So, your interface may look like the following.

```
interface Visitor
{
    void visitTheElement(CompositeEmployee employees);
    void visitTheElement(SimpleEmployee employee);
    void visitTheElement(UndefinedEmployee employee);
}
```

And later you need to implement this new method in the concrete visitor class.

7. **Suppose that I need to support new operations in the existing architecture. How should I proceed with the visitor pattern?**

   For each new operation, create a new visitor subclass and implement the operation in it. Then, visit your existing structure the way that was shown in the preceding examples.

8. **In the client code, you made a container of employees first, and then it starts visiting. Is it mandatory to create such a structure?**

   No. It just helps clients to visit smoothly in one shot. If you do not create any such structure, you can always call it separately.

CHAPTER 14

Observer Pattern

This chapter covers the observer pattern.

GoF Definition

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Concept

In this pattern, there are many observers (objects) that are observing a particular subject (also an object). Observers register themselves to a subject to get a notification when there is a change made inside that subject. When they lose interest of the subject, they simply unregister from the subject. It is also referred to as the *publish-subscribe pattern*. The whole idea can be summarized as follows: *Using this pattern, an object (subject) can send notifications to multiple observers (a set of objects) at the same time.*

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_14

You can visualize the scenarios in the following diagrams.

Step 1. Observers are requesting a subject to get notifications (see Figure 14-1).

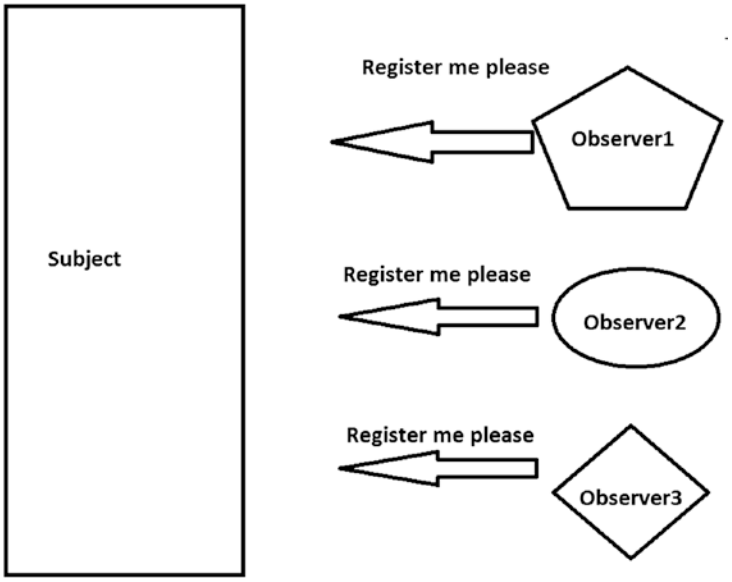


Figure 14-1. *Step 1*

Step 2. The subject grants the requests and the connection is established (see Figure 14-2).

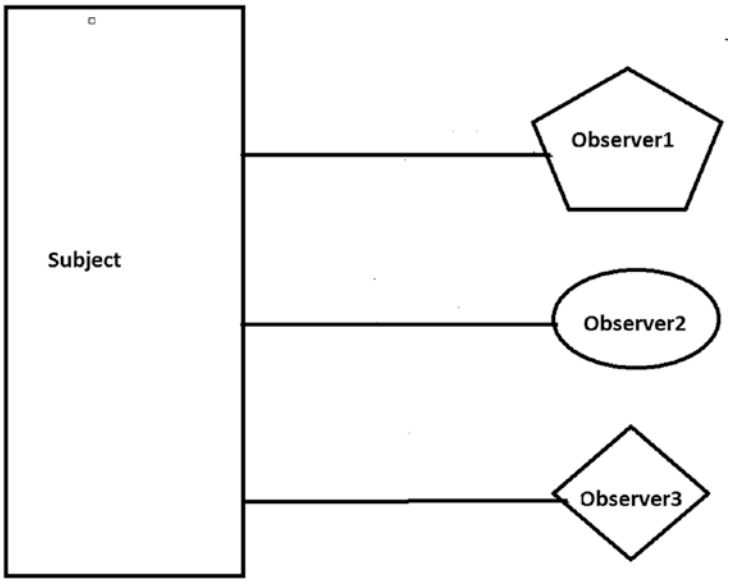


Figure 14-2. *Step 2*

Step 3. The subject sends notifications to the registered users (in case a typical event occurs in the subject and it wants to notify others) (see Figure 14-3).

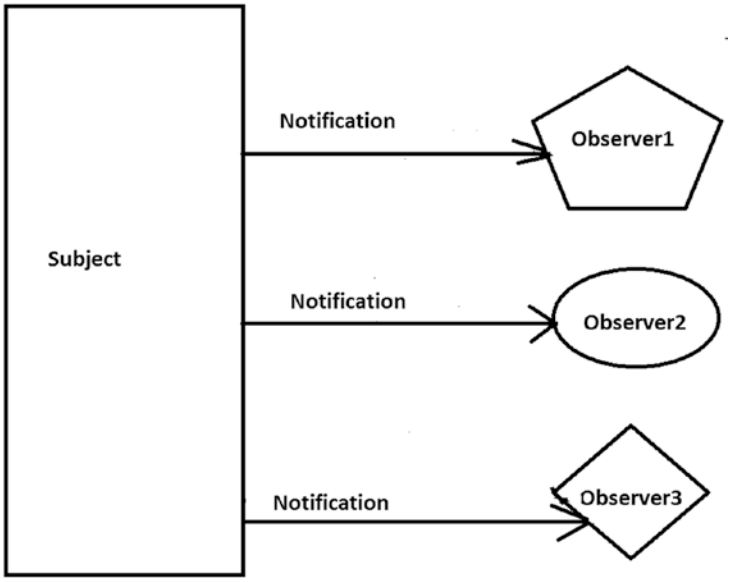


Figure 14-3. *Step 3*

Step 4 (optional). Observer2 does not want to get further notification, so it unregisters itself (see Figure 14-4).

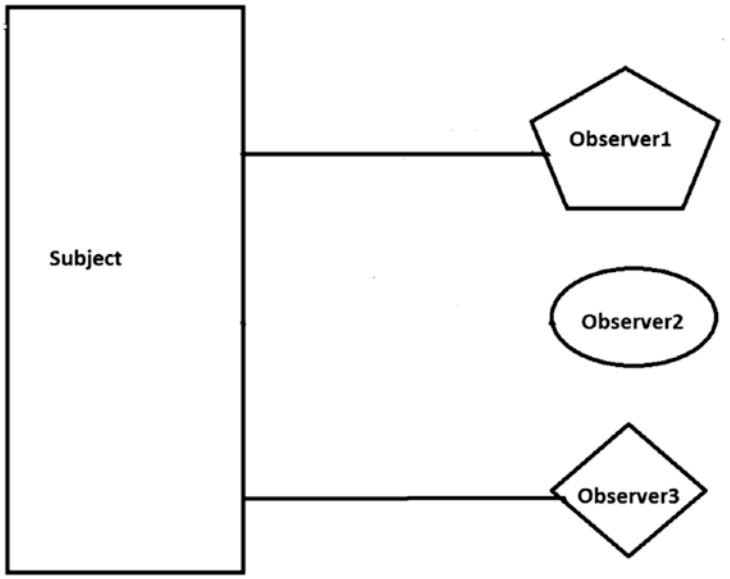


Figure 14-4. *Step 4*

Step 5. Now onward, only Observer1 and Observer3 get notifications from the subject (see Figure 14-5).

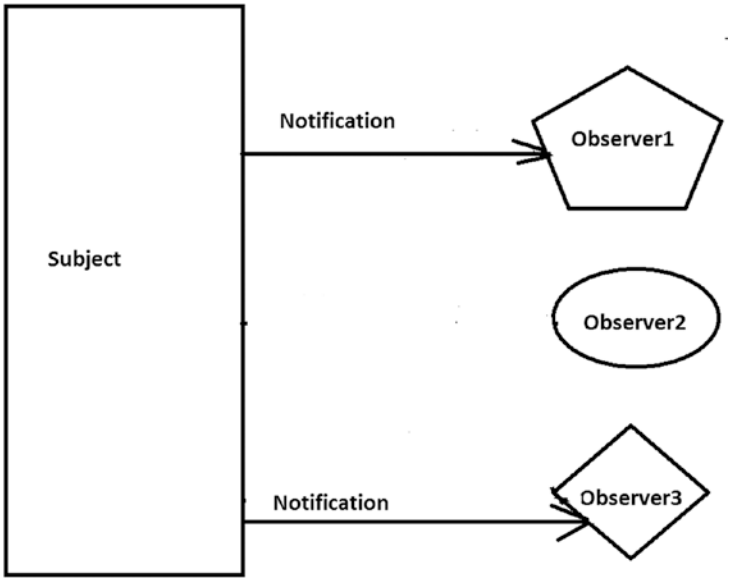


Figure 14-5. *Step 5*

Real-World Example

Think about a celebrity who has many followers on social media. Each of these followers wants all the latest updates from their favorite celebrity. So, they follow the celebrity until their interest wanes. When they lose interest, they simply do not follow that celebrity any longer. You can think each of these fans or followers as an observer and the celebrity as a subject.

Computer-World Example

In the world of computer science, consider a simple UI-based example. Let's assume that this UI is connected to a database. A user can execute a query through that UI, and after searching the database, the result is returned in the UI. Here you segregate the UI from the database in such a way that if a change occurs in the database, the UI is notified, and it updates its display according to the change.

To simplify this scenario, assume that you are the person responsible for maintaining a particular database in your organization. Whenever there is a change made inside the database, you want a notification so that you can take action if necessary.

Note In general, you see the presence of this pattern in event-driven software. Modern languages like C#, Java, and so forth have built-in support for handling events following this pattern. Those constructs make your life easy.

In Java, you can see the use of event listeners. These listeners are observers only. In Java, you have a ready-made class called Observable, which can have multiple observers. These observers need to implement the Observer interface. The Observer interface has an "update" method: void update(Observable o,Object arg). This method is invoked whenever a change occurs in the observed object. Your application needs to call the Observable object's notifyObservers method to notify about the change to the observers. The addObserver(Observer o) and deleteObserver(Observer o) methods add or and delete an observer, similar to the register and unregister methods discussed earlier. You can learn more from https://docs.oracle.com/javase/8/docs/api/java/util/Observer.html and https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Observable.html.

If you are familiar with the .NET Framework, you see that in C#, you have the generic System.IObservable<T> and System.IObserver<T> interfaces, where the generic type parameter provides notifications.

Illustration

Let's consider the following example and go through the post analysis of the output. I have created three observers and one subject. The subject maintains a list for all its registered users. Our observers want to receive notification when a flag value changes in the subject. In the output, you discover that the observers are getting the notifications when flag values are changed to 5, 50, and 100, respectively. But one of them did not receive any notification when the flag value changed to 50, because at that moment, he was not a registered user in subject. But in the end, he is getting notifications because he registered himself again.

In this implementation, the `register()`, `unregister()`, and `notifyRegisteredUsers()` methods have their typical meanings. The `register()` method registers an observer in the subject's notification list, the `unregister()` method removes an observer from the subject's notification list, and `notifyRegisteredUsers()` notifies all the registered users when a typical event occurs in the subject.

Class Diagram

Figure 14-6 shows the class diagram.

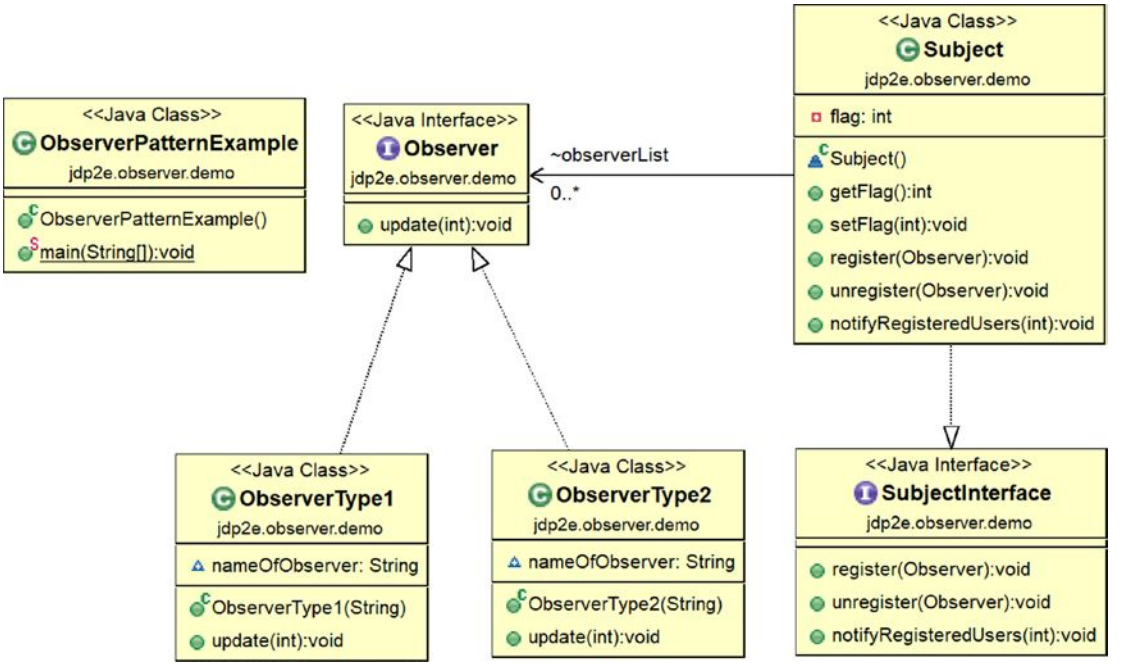


Figure 14-6. *Class diagram*

Package Explorer View

Figure 14-7 shows high-level structure of the program.

Figure 14-7. *Package Explorer view*

Implementation

Here is the implementation.

```
package jdp2e.observer.demo;

import java.util.*;

interface Observer
{
    void update(int updatedValue);
}
class ObserverType1 implements Observer
{
    String nameOfObserver;
    public ObserverType1(String name)
    {
        this.nameOfObserver = name;
    }
    @Override
    public void update(int updatedValue)
    {
        System.out.println( nameOfObserver+" has received an alert: Updated
        myValue in Subject is: "+ updatedValue);
    }
}
class ObserverType2 implements Observer
{
    String nameOfObserver;
    public ObserverType2(String name)
    {
        this.nameOfObserver = name;
    }
    @Override
```

```
    public void update(int updatedValue)
    {
        System.out.println( nameOfObserver+" has received an alert: The
        current value of myValue in Subject is: "+ updatedValue);
    }
}

interface SubjectInterface
{
    void register(Observer anObserver);
    void unregister(Observer anObserver);
    void notifyRegisteredUsers(int notifiedValue);
}
class Subject implements SubjectInterface
{
    private int flag;
    public int getFlag()
    {
        return flag;
    }
    public void setFlag(int flag)
    {
        this.flag = flag;
        //Flag value changed. So notify registered users/observers.
        notifyRegisteredUsers(flag);
    }
    List<Observer> observerList = new ArrayList<Observer>();
    @Override
    public void register(Observer anObserver) {
        observerList.add(anObserver);

    }
    @Override
    public void unregister(Observer anObserver) {
        observerList.remove(anObserver);
    }
```

```
    @Override
    public void notifyRegisteredUsers(int updatedValue)
    {
        for (Observer observer : observerList)
            observer.update(updatedValue);
    }
}
public class ObserverPatternExample {

    public static void main(String[] args) {
        System.out.println(" ***Observer Pattern Demo***\n");
        //We have 3 observers- 2 of them are ObserverType1, 1 of them is of
        //ObserverType2
        Observer myObserver1 = new ObserverType1("Roy");
        Observer myObserver2 = new ObserverType1("Kevin");
        Observer myObserver3 = new ObserverType2("Bose");
        Subject subject = new Subject();
        //Registering the observers-Roy,Kevin,Bose
        subject.register(myObserver1);
        subject.register(myObserver2);
        subject.register(myObserver3);
        System.out.println(" Setting Flag = 5 ");
        subject.setFlag(5);
        //Unregistering an observer(Roy))
        subject.unregister(myObserver1);
        //No notification this time Roy. Since it is unregistered.
        System.out.println("\n Setting Flag = 50 ");
        subject.setFlag(50);
        //Roy is registering himself again
        subject.register(myObserver1);
        System.out.println("\n Setting Flag = 100 ");
        subject.setFlag(100);
    }
}
```

Output

Here is the output.

```
***Observer Pattern Demo***

 Setting Flag = 5
Roy has received an alert: Updated myValue in Subject is: 5
Kevin has received an alert: Updated myValue in Subject is: 5
Bose has received an alert: The current value of myValue in Subject is: 5

 Setting Flag = 50
Kevin has received an alert: Updated myValue in Subject is: 50
Bose has received an alert: The current value of myValue in Subject is: 50

 Setting Flag = 100
Kevin has received an alert: Updated myValue in Subject is: 100
Bose has received an alert: The current value of myValue in Subject is: 100
Roy has received an alert: Updated myValue in Subject is: 100
```

Analysis

Initially, all three observers—Roy, Kevin and Bose—registered for notifications from the subject. So, in the initial phase, all of them received notifications. At some point, Roy became disinterested in notifications, so he unregistered himself. So, from this time onward, only Kevin and Bose received notifications (notice when I set the flag value to 50). But Roy has changed his mind and he re-registered himself to get notifications from the subject. So, in the final case, all of them received notifications from the subject.

Q&A Session

1. **If I have only one observer, then I may not need to set up the interface. Is this correct?**

   Yes. But if you want to follow the pure object-oriented programming guidelines, programming to an interface/abstract class is always considered a better practice. So, you should prefer

interfaces (or abstract classes) over concrete classes. Also, usually, you have multiple observers, and you want them to implement the methods in a systematic manner that follows the contract. You get benefit from this kind of design.

2. **Can you have different types of observers in the same application?**

   Yes. This is why I have played with three observers from two different classes. But you should not feel that for each observer; you need to create a different class.

   Consider a real-world scenario. When a company releases or updates new software, the company business partners and the customers who purchased the software get notifications. In this case, the business partners and the customers are two different types of observers.

3. **Can I add or remove observers at runtime?**

   Yes. At the beginning our program, Roy registered to get notifications; then he unregistered and later reregistered.

4. **It appears that there are similarities between the observer pattern and the chain of responsibility pattern. Is this correct?**

   In an observer pattern, all registered users get notifications at the same time, but in a chain of responsibility pattern, objects in the chain are notified one by one, and this process continues until the object fully handles the notification. Figure 14-8 and Figure 14-9 summarize the differences.

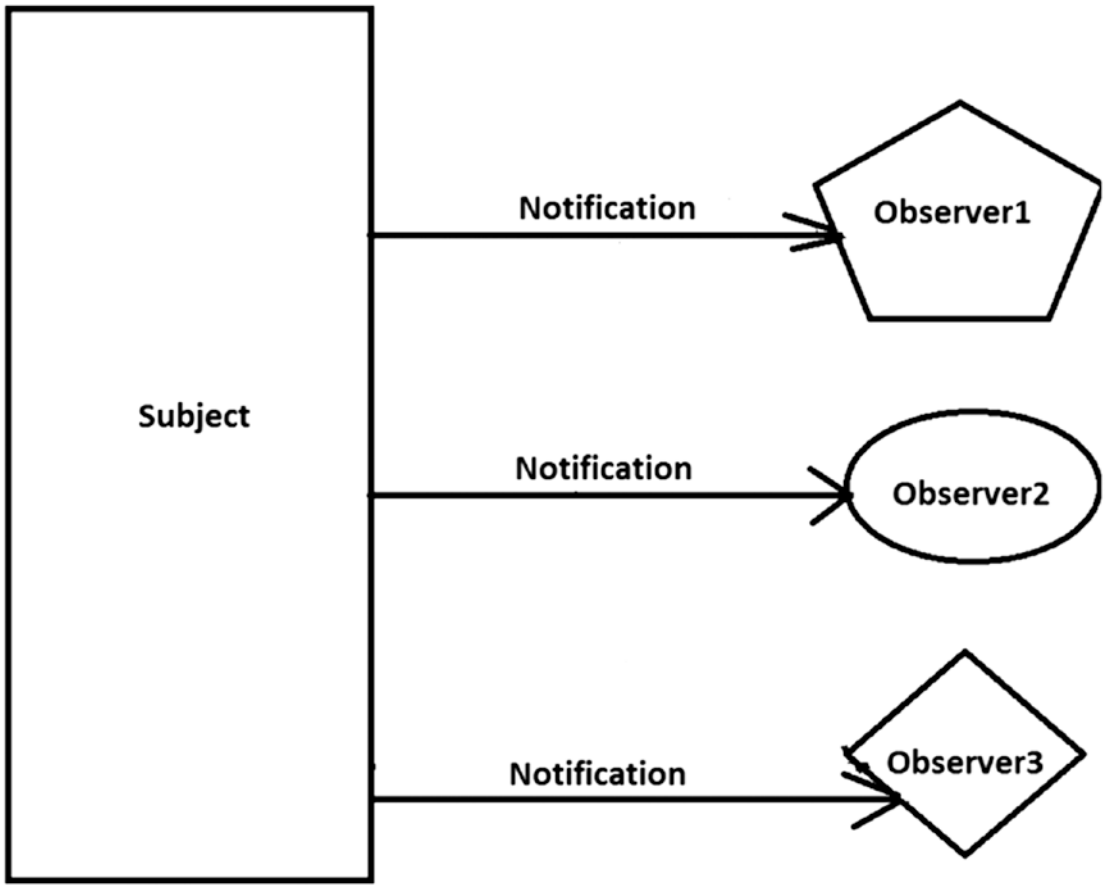


Figure 14-8. *The basic workflow of an observer pattern*

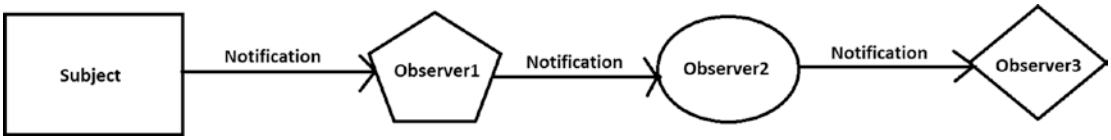


Figure 14-9. *The basic workflow of a chain of responsibility pattern*

5. **This model supports one-to-many relationships. Is this correct?**

   Yes. Since a subject can send notifications to multiple observers, this kind of dependency is clearly depicting a one-to-many relationship.

6. **If you already have these ready-made constructs, why are you writing your own code?**

   Changing the ready-made constructs to your preference is not always easy. In many cases, you cannot change the built-in functionalities at all. When you try to implement the concept yourself, you may have a better understanding of how to use those ready-made constructs.

   Consider some typical scenarios.

   - In Java, Observable is a concrete class. It does not implement an interface. So, you can't create your own implementation that works with Java's built-in Observer API.
   - Java does not allow multiple inheritance. So, when you have to extend the Observable class, you have to keep in mind the restriction. This may limit the reuse potential.
   - The signature of the `setChanged` method in an Observable is as follows: `protected void setChanged()`. That means to use it, you need to subclass Observable class. This violates one of the key design principles, which basically says to prefer composition over inheritance.

7. **What are the key benefits of the observer pattern?**

   - The subject and its registered users(observers) are making a loosely coupled system. They do not need to know each other explicitly.
   - No modification is required in subjects when you add or remove an observer from its notification lists.
   - Also, you can independently add or remove observers at any time.

8. **What are the key challenges associated with an observer pattern?**

    - Undoubtedly, memory leak is the greatest concern when you deal with any event-based mechanism. An automatic garbage collector may not always help you in this context. You can consider such a case where the deregister/unregister operations are not performed properly.
    - The order of notification is not dependable.
    - Java's built-in support for the observer pattern has some key restrictions, which I discussed earlier. (Revisit the answer to question 6 .) One of them forces you to prefer inheritance over composition, so it clearly violates one of the key design principles that prefers the opposite.

CHAPTER 15

Strategy (Policy) Pattern

This chapter covers the strategy pattern.

GoF Definition

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

Concept

Suppose there is an application where you have multiple algorithms and each of these algorithms can perform a specific task. A client can dynamically pick any of these algorithms to serve its current need.

The strategy pattern suggests that you implement these algorithms in separate classes. When you encapsulate an algorithm in a separate class, you call it a *strategy*. An object that uses the strategy object is often referred to as a *context object*. These "algorithms" are also called *behaviors* in some applications.

Real-World Example

Generally at the end of a soccer match, if team A is leading 1–0 over team B, instead of attacking they become defensive to maintain the lead. On the other hand, team B goes for an all-out attack to score the equalizer.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_15

Computer world Example

Suppose that you have a list of integers and you want to sort them. You do this by using various algorithms; for example, Bubble Sort, Merge Sort, Quick Sort, Insertion Sort, and so forth. So, you can have a sorting algorithm with many different variations. Now you can implement each of these variations (algorithms) in separate classes and pass the objects of these classes in client code to sort your integer list.

Note You can consider the java.util.Comparator interface in this context. You can implement this interface and provide multiple implementations of comparators with different algorithms to do various comparisons using the *compare()* method. This comparison result can be further used in various sorting techniques. The Comparator interface plays the role of a strategy interface in this context.

Illustration

Before you proceed, let's keep in mind the following points.

- The strategy pattern encourages you to use object composition instead of subclassing. So, it suggests you do *not* override parent class behaviors in different subclasses. Instead, you put these behaviors in separate classes (called a *strategy*) that share a common interface.
- The client class only decides which algorithm to use; the context class does not decide that.
- A context object contains reference variables for the strategy objects' interface type. So, you can obtain different behaviors by changing the strategy in the context.

In the following implementation, the Vehicle class is an abstract class that plays the role of a context. Boat and Aeroplane are two concrete implementations of the Vehicle class. You know that they are associated with different behaviors: one travels through water and the other one travels through air.

These behaviors are placed in two concrete classes: AirTransport and WaterTransport. These classes share a common interface, TransportMedium. So, these

concrete classes are playing the role of the strategy classes where different behaviors are reflected through the `transport()` method implementations.

In the Vehicle class, there is a method called `showTransportMedium()`. Using this method, I am delegating the task to the corresponding behavior class. So, once you pick your strategy, the corresponding behavior can be invoked. Notice that in the Vehicle class, there is a method called `commonJob()`,which is not supposed to vary in the future, so its behavior is not treated as a volatile behavior.

Class Diagram

Figure 15-1 shows the class diagram.

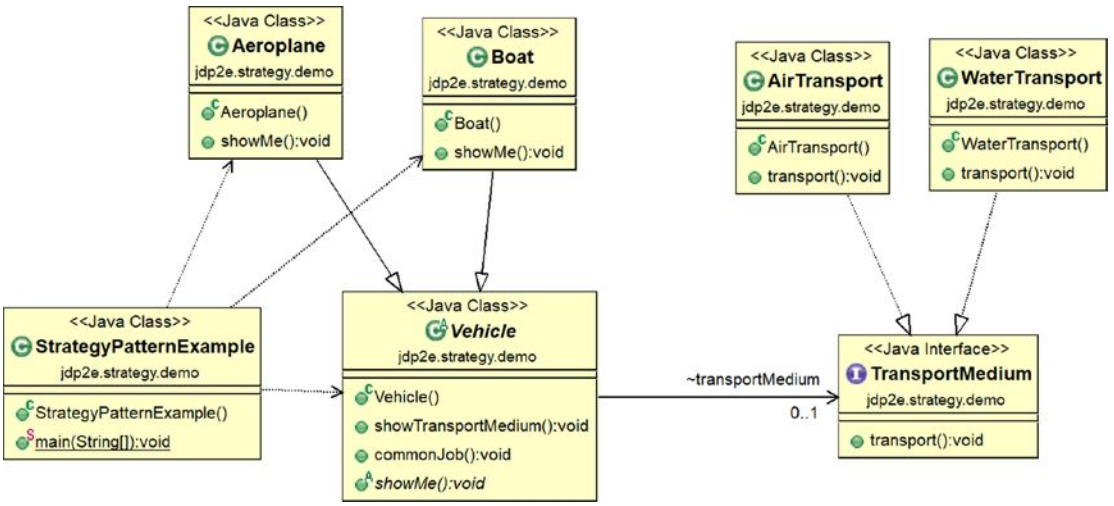


Figure 15-1. *Class diagram*

Package Explorer View

Figure 15-2 shows the high-level structure of the program.

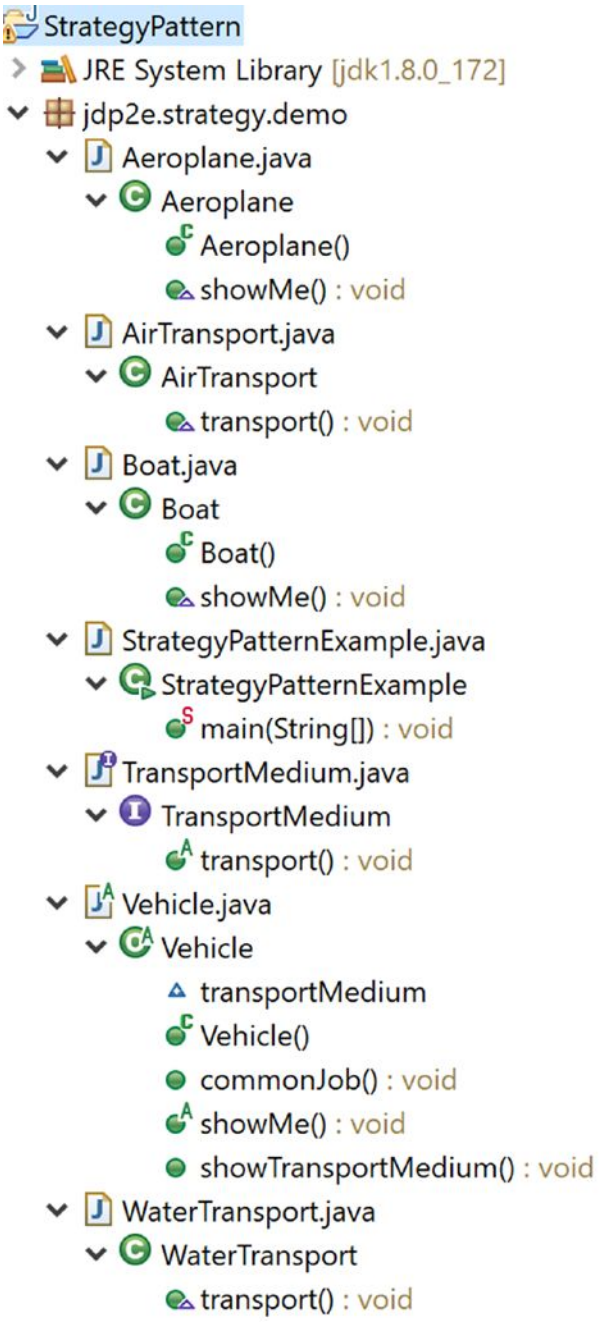


Figure 15-2. *Package Explorer view*

Implementation

Here's the implementation.

```
   // Vehicle.java

package jdp2e.strategy.demo;

//Context class
public abstract class Vehicle
{
    /*A context object contains reference variable/s for the strategy
    object/s interface type.*/
    TransportMedium transportMedium;
    public Vehicle()
    {

    }
    public void showTransportMedium()
    {
        //Delegate the task to the //corresponding behavior class.
        transportMedium.transport();
    }
    //The code that does not vary.
    public void commonJob()
    {
        System.out.println("We all can be used to transport");
    }
    public abstract void showMe();

}
```

```
// Boat.java

package jdp2e.strategy.demo;

public class Boat extends Vehicle
{
    public Boat()
    {
        transportMedium= new WaterTransport();
    }
    @Override
    public void showMe() {
        System.out.println("I am a boat.");

    }
}
// Aeroplane.java

package jdp2e.strategy.demo;

public class Aeroplane extends Vehicle
{
    public Aeroplane()
    {
        transportMedium= new AirTransport();
    }

    @Override
    public void showMe() {
        System.out.println("I am an aeroplane.");

    }
}
// TransportMedium.java
package jdp2e.strategy.demo;

public interface TransportMedium
{
    public void transport();
}
```

```
//WaterTransport.java
package jdp2e.strategy.demo;
//This class represents an algorithm/behavior.
public class WaterTransport implements TransportMedium
{
    @Override
    public void transport()
    {
        System.out.println("I am transporting in water.");
    }
}
//AirTransport.java
package jdp2e.strategy.demo;
//This class represents an algorithm/behavior.
public class AirTransport implements TransportMedium
{
    @Override
    public void transport()
    {
        System.out.println("I am transporting in air.");
    }
}

// StrategyPatternExample.java

package jdp2e.strategy.demo;

//Client code
public class StrategyPatternExample {

    public static void main(String[] args) {
        System.out.println("***Strategy Pattern Demo***");
        Vehicle vehicleContext=new Boat();
        vehicleContext.showMe();
        vehicleContext.showTransportMedium();
        System.out.println("________");
```

```
        vehicleContext=new Aeroplane();
        vehicleContext.showMe();
        vehicleContext.showTransportMedium();

    }

}
```

Output

Here's the output.

```
***Strategy Pattern Demo***
I am a boat.
I am transporting in water.

_______
I am an aeroplane.
I am transporting in air.
```

Q&A Session

1. **Why are you complicating the example by avoiding simple subclassing of these behaviors?**

   In object-oriented programming, you may prefer to use the concept of polymorphism so that your code can pick the intended object (among different object types) at runtime, leaving your code unchanged.

   When you are familiar with design patterns, most often, you prefer composition over inheritance.

   Strategy patterns help you combine composition with polymorphism. Let's examine the reasons behind this.

   It is assumed that you try to use the following guidelines in any application you write:

- Separate the code that varies a lot from the part of code that does not vary.
- Try to maintain the varying parts as freestanding as possible (for easy maintenance).
- Try to reuse them as much as possible.

  Following these guidelines, I have used composition to extract and encapsulate the volatile/varying parts of the code, so that the whole task can be handled easily, and you can reuse them.

  But when you use inheritance, your parent class can provide a default implementation, and then the derived class changes it (Java calls it *overriding it*). The next derived class can further modify the implementation, so you are basically spreading out the tasks over different levels, which may cause severe maintenance and extensibility issues in the future. Let's examine such a case.

  Let's assume that your vehicle class has the following construct.

```
abstract class Vehicle
{
    //Default implementation
    public void showTransportMedium()
    {
        System.out.println("I am transporting in air.");
    }
    //The code that does not vary.
    public void commonJob()
    {
        System.out.println("We all can be used to transport");
    }
    public abstract void showMe();
}
```

So, make a concrete implementation of Vehicle, like this:

```
class Aeroplane extends Vehicle
{
    @Override
    public void showMe() {
        System.out.println("I am an aeroplane.");

    }
}
```

And use following lines of codes in client class.

```
Aeroplane aeroplane=new Aeroplane();
aeroplane.showMe();
aeroplane.showTransportMedium();
```

You will receive following output:

```
I am an aeroplane.
I am transporting in air.
```

So far, it looks good. Now suppose that you have introduced another class, Boat, like in the following.

```
class Boat extends Vehicle
{
    @Override
    public void showMe() {
        System.out.println("I am a boat.");

    }
}
```

Use the following lines of codes in the client class (new lines are shown in bold).

```
Aeroplane aeroplane=new Aeroplane();
aeroplane.showMe();
aeroplane.showTransportMedium();
```

```
Boat boat=new Boat();
boat.showMe();
boat.showTransportMedium();
```

You receive the following output.

```
I am an aeroplane.
I am transporting in air.
I am a boat.
I am transporting in air.
```

You can see that your boat is moving into the air now. To prevent this ugly situation, you need to override it properly.

Now further assume that you need to introduce another class, SpeedBoat, which can also transport through water at high speed. You need to guard the situations like this:

```
class Boat extends Vehicle
{
    @Override
    public void showMe()
    {
        System.out.println("I am a boat.");

    }
    @Override
    public void showTransportMedium() {
        System.out.println("I am transporting in water.");

    }
}
class SpeedBoat extends Vehicle
{
    @Override
    public void showMe() {
        System.out.println("I am a speedboat.");

    }
```

```
    @Override
    public void showTransportMedium() {
        System.out.println("I am transporting in water with high
        speed.");
    }
}
```

You can see that if you spread out the task that can vary across different classes (and their subclasses), in the long run, maintenance becomes very costly. You can experience a lot of pain if you want to accommodate similar changes very often, because you need to keep updating the `showTransportMedium()` method in each case.

2. **If this is the case, you could create a separate interface, TransportInterface, and place the showTransportMedium() method in that interface. Now any class that wants to get the method can implement that interface also. Is this understanding correct?**

   Yes, you can do that. But this is what the code looks like:

```
abstract class Vehicle
{
    //The code that does not vary.
    public void commonJob()
    {
        System.out.println("We all can be used to transport");
    }
    public abstract void showMe();
}
interface TransportInterface
{
    void showTransportMedium();
}
```

```
class Aeroplane extends Vehicle implements TransportInterface
{
    @Override
    public void showMe() {
        System.out.println("I am an aeroplane.");

    }
    @Override
    public void showTransportMedium() {
        System.out.println("I am transporting in air.");
    }
}
class Boat extends Vehicle implements TransportInterface
{
    @Override
    public void showMe()
    {
        System.out.println("I am a boat.");

    }

    @Override
    public void showTransportMedium() {
        System.out.println("I am transporting in water.");

    }
}
```

You can see that each class and its subclasses may need to provide its own implementations for the `showTransportMedium()` method. So, you cannot reuse your code, which is as bad as inheritance in this case.

3. **Can you modify the default behavior at runtime in your implementation?**

   Yes, you can. Let's introduce a special vehicle that can transport in both water and air, as follows.

```
public class SpecialVehicle extends Vehicle
{
    public SpecialVehicle()
    {
        //Initialized with AirTransport
        transportMedium= new AirTransport();
    }

    @Override
    public void showMe()
    {
        System.out.println("I am a special vehicle who can
        transport both in air and water.");
    }
}
```

   And add a setter method in the Vehicle class(changes are shown in bold).

```
//Context class
public abstract class Vehicle
{
    //A context object contains reference variable/s
    //for the strategy object/s interface type
    TransportMedium transportMedium;
    public Vehicle()
    {

    }
    public void showTransportMedium()
    {
        //Delegate the task to the corresponding behavior class.
        transportMedium.transport();
```

```
    }
    //The code that does not vary.
    public void commonJob()
    {
        System.out.println("We all can be used to transport");
    }
    public abstract void showMe();

    //Additional code to explain the answer of question no 3 in
    //the "Q&A session"

    public void setTransportMedium(TransportMedium
    transportMedium)
    {
        this.transportMedium=transportMedium;
    }
}
```

To test this, add a few lines of code in the client class, as well.

```
//Client code
public class StrategyPatternExample {

    public static void main(String[] args) {
        System.out.println("***Strategy Pattern Demo***");
        Vehicle vehicleContext=new Boat();
        vehicleContext.showMe();
        vehicleContext.showTransportMedium();
        System.out.println("________");

        vehicleContext=new Aeroplane();
        vehicleContext.showMe();
        vehicleContext.showTransportMedium();
        System.out.println("________");

        //Additional code to explain the answer of question no
        //3 in the "Q&A session"
        vehicleContext=new SpecialVehicle();
```

```
        vehicleContext.showMe();
        vehicleContext.showTransportMedium();
        System.out.println("- - - - -");
        //Changing the behavior of Special vehicle
        vehicleContext.setTransportMedium(new WaterTransport());
        vehicleContext.showTransportMedium();

    }
}
```

Now if you execute this modified program, you get the following output.

```
***Strategy Pattern Demo***
***Strategy Pattern Demo***
I am a boat.
I am transporting in water.

_______
I am an aeroplane.
I am transporting in air.

_______
I am a special vehicle who can transport both in air and water.
I am transporting in air.
- - - - -
I am transporting in water.
```

The initial behavior is modified dynamically in a later phase.

4. **Can you use an abstract class instead of an interface?**

   Yes. It is suitable in some cases where you may want to put common behaviors in the abstract class. I discussed it in detail in the "Q&A Session" section on the builder pattern.

5. **What are the key advantages of using a strategy design pattern?**

   - This pattern makes your classes independent from algorithms. Here, a class delegates the algorithms to the strategy object (that encapsulates the algorithm) dynamically at runtime. So, you can simply say that the choice of the algorithm is not bound at compile time.

- Easier maintenance of your codebase.
- It is easily extendable. (Refer to the answers for questions 2 and 3 in this context.)

6. **What are key challenges associated with a strategy design pattern?**
    - The addition of context classes causes more objects in our application.
    - Users of the application must be aware of different strategies; otherwise, the output may surprise them. So, there exists a tight coupling between the client code and the implementation of different strategies.
    - When you introduce a new behavior/algorithm, you may need to change the client code also.

CHAPTER 16

Template Method Pattern

This chapter covers the Template Method pattern.

GoF Definition

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Concept

In a template method, you define the minimum or essential structure of an algorithm. Then you defer some responsibilities to the subclasses. The key intent is that you can redefine certain steps of an algorithm, but those changes should not impact the basic flow of the algorithm.

So, this design pattern is useful when you implement a multistep algorithm and you want to allow customization through subclasses.

Real-World Example

Suppose that you are ordering a pizza from a restaurant. For the chef, the basic preparation of the pizza is the same; he includes some final toppings based on customer choice. For example, you can opt for a veggie pizza or a non-veggie pizza. You can also choose toppings like bacons, onions, extra cheese, mushrooms, and so forth. The chef prepares the final product according to your preferences.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_16

Computer-World Example

Suppose that you are making a program to design engineering courses. Let's assume that the first semester is common for all streams. In subsequent semesters, you need to add new papers/subjects to the application based on the course. You see a similar situation in the upcoming illustration. Remember that this pattern makes sense when you want to avoid duplicate codes in your application. At the same time, you may want to allow subclasses to change some specific details of the base class workflow to provide varying behaviors in the application.

Note The removeAll() method of java.util.AbstractSet is an example of the template method pattern. Apart from this, there are many non-abstract methods in java.util.AbstractMap and java.util.AbstractSet classes, which can also be considered as the examples of the template method pattern.

Illustration

In the following implementation, I assume that each engineering student needs to complete the course of Mathematics and then Soft skills (This subject may deal with communication skills, character traits, people management skills etc.) in their initial semesters to obtain the degree. Later you will add special paper/s to these courses (Computer Science or Electronics).

To serve the purpose, a method completeCourse() is defined in an abstract class BasicEngineering. I have also marked the method final, so that subclasses of BasicEngineering cannot override the completeCourse() method to alter the sequence of course completion order.

Two other concrete classes- ComputerScience and Electronics are the subclasses of BasicEngineering class and they are completing the abstract method completeSpecialPaper() as per their needs.

Class Diagram

Figure 16-1 shows the class diagram.

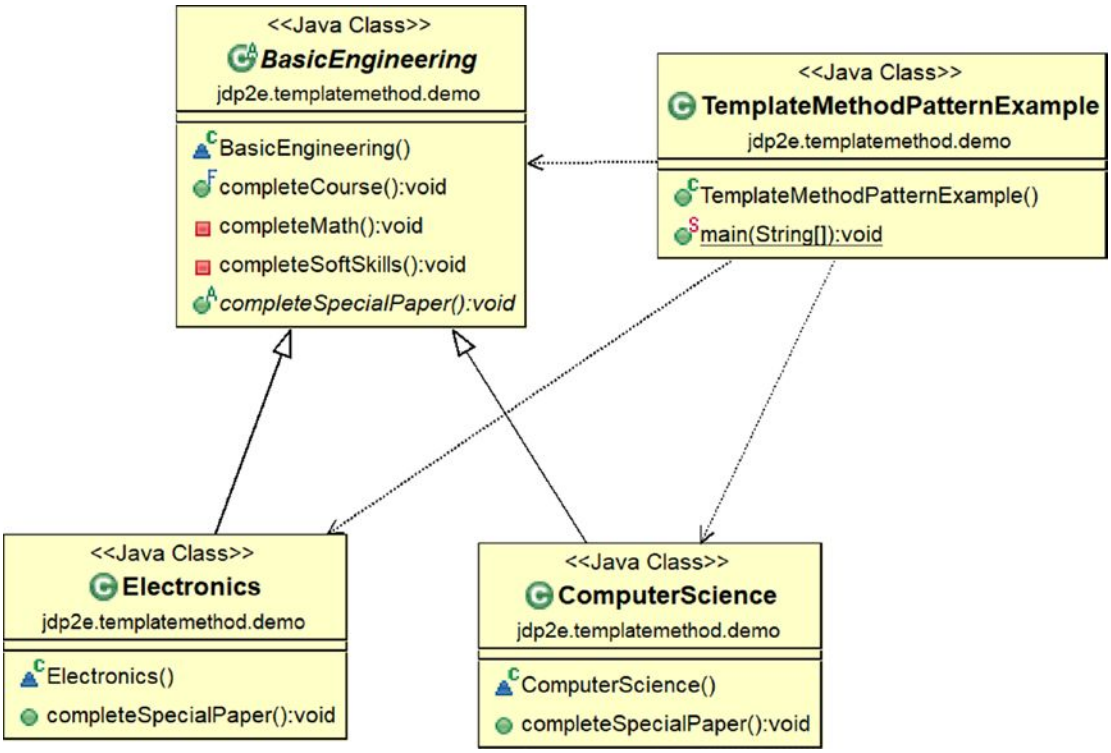


Figure 16-1. *Class diagram*

Package Explorer View

Figure 16-2 shows the high-level structure of the program.

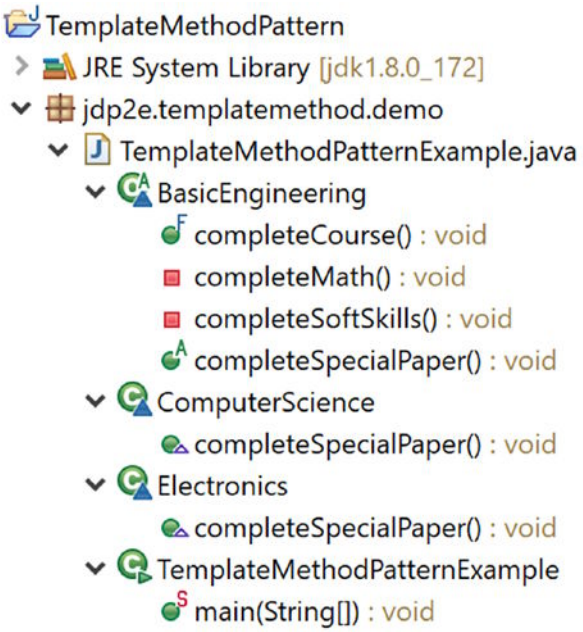


Figure 16-2. *Package Explorer View*

Implementation

Here's the implementation:

```
package jdp2e.templatemethod.demo;

abstract class BasicEngineering
{
    //Making the method final to prevent overriding.
    public final void completeCourse()
    {
        //The course needs to be completed in the following sequence
        //1.Math-2.SoftSkills-3.Special Paper
        //Common Papers:
        completeMath();
        completeSoftSkills();
        //Specialization Paper:
        completeSpecialPaper();
    }
```

```
    private void completeMath()
    {
        System.out.println("1.Mathematics");
    }
    private void completeSoftSkills()
    {
        System.out.println("2.SoftSkills");
    }
    public abstract void completeSpecialPaper();
}
class ComputerScience extends BasicEngineering
{
    @Override
    public void completeSpecialPaper() {
        System.out.println("3.Object-Oriented Programming");

    }
}
class Electronics extends BasicEngineering
{
    @Override
    public void completeSpecialPaper()
    {
        System.out.println("3.Digital Logic and Circuit Theory");
    }
}

public class TemplateMethodPatternExample {

    public static void main(String[] args) {
        System.out.println("***Template Method Pattern Demo***\n");
        BasicEngineering preferrredCourse = new ComputerScience();
        System.out.println("Computer Sc. course will be completed in
        following order:");
        preferrredCourse.completeCourse();
        System.out.println();
```

```
        preferrredCourse = new Electronics();
        System.out.println("Electronics course will be completed in
        following order:");
        preferrredCourse.completeCourse();
    }
}
```

Output

Here's the output:

```
***Template Method Pattern Demo***

Computer Sc. course will be completed in following order:
1.Mathematics
2.SoftSkills
3.Object-Oriented Programming

Electronics course will be completed in following order:
1.Mathematics
2.SoftSkills
3.Digital Logic and Circuit Theory
```

Q&A Session

1. **In this pattern, I am seeing that subclasses can simply redefine the methods as per their need. Is the understanding correct?**

   Yes.

2. **In the abstract class BasicEngineering, only one method is abstract, other two methods are concrete methods. What is the reason behind it?**

   It is a simple example with only 3 methods where I wanted the subclasses to override only the `completeSpecialPaper()` method. Other methods are common to both stream and they do not need to be overridden by the subclasses.

3. **Consider a situation like this: Suppose you want to add some more methods in the BasicEngineering class but you want to work on those methods if and only if, the child classes need them otherwise you will ignore them. This type of situation is very common in some PhD courses where some courses are not mandatory for all candidates. For example, if a student has certain qualifications, he/she may not need to attend the lectures of those subjects. Can you design this kind of situation with the Template Method Pattern?**

   Yes, we can. Basically, you may need to put a **hook** which is nothing but a method that can help us to control the flow in an algorithm.

   To show an example of this kind of design, I am adding one more method in `BasicEngineering` called is **`AdditionalPapersNeeded()`**. Let us assume that Computer science students need to complete this course, but Electronics students can opt out this paper. Let's go through the program and output.

Modified Implementation

Here's the modified implementation. Key changes are shown in bold.

```
package jdp2e.templatemethod.questions_answers;

abstract class BasicEngineering
{
    //Making the method final to prevent overriding.
    public final void completeCourse()
    {
        //The course needs to be completed in the following sequence
        //1.Math-2.SoftSkills-3.Special Paper-4.Additional Papers(if any)
        //Common Papers:
        completeMath();
        completeSoftSkills();
```

```
        //Specialization Paper:
        completeSpecialPaper();
        if (isAdditionalPapersNeeded())
        {
            completeAdditionalPapers();
        }
    }

    private void completeMath()
    {
        System.out.println("1.Mathematics");
    }
    private void completeSoftSkills()
    {
        System.out.println("2.SoftSkills");
    }
    public abstract void completeSpecialPaper();
    private void completeAdditionalPapers()
    {
        System.out.println("4.Additional Papers are needed for this
        course.");
    }
    //By default, AdditionalPapers are needed for a course.
    boolean isAdditionalPapersNeeded()
    {
        return true;
    }
}
class ComputerScience extends BasicEngineering
{
    @Override
    public void completeSpecialPaper()
    {
        System.out.println("3.Object-Oriented Programming");
    }
```

```
    //Additional papers are needed for ComputerScience
    //So, there is no change for the hook method.
}
class Electronics extends BasicEngineering
{
    @Override
    public void completeSpecialPaper()
    {
        System.out.println("3.Digital Logic and Circuit Theory");
    }
    //Overriding the hook method:
    //Indicating that AdditionalPapers are not needed for Electronics.
    @Override
    public  boolean isAdditionalPapersNeeded()
    {
        return false;
    }
}

public class TemplateMethodPatternModifiedExample {

    public static void main(String[] args) {
        System.out.println("***Template Method Pattern Modified
        Demo***\n");
        BasicEngineering preferrredCourse = new ComputerScience();
        System.out.println("Computer Sc. course will be completed in
        following order:");
        preferrredCourse.completeCourse();
        System.out.println();
        preferrredCourse = new Electronics();
        System.out.println("Electronics course will be completed in
        following order:");
        preferrredCourse.completeCourse();
    }
}
```

Modified Output

Here's the modified output:

```
***Template Method Pattern Modified Demo***

Computer Sc. course will be completed in following order:
1.Mathematics
2.SoftSkills
3.Object-Oriented Programming
4.Additional Papers are needed for this course.

Electronics course will be completed in following order:
1.Mathematics
2.SoftSkills
3.Digital Logic and Circuit Theory
```

Note You may prefer an alternative approach. For example, you could make a default method isAdditionalPapersNeeded() in BasicEngineering. Then you could override the method in Electronics class and then you could make the method body empty. But this approach does not look better if you compare it to the previous approach.

---

4. **Looks like this pattern is similar to Builder pattern.Is the understanding correct ?**

   No. You should not forget the core intent;Template Method is a behavioral design patterns, and Builder is a creational design pattern. In Builder Patterns, the clients/customers are the boss-they can control the order of the algorithm. On the other hand, in Template Method pattern, you are the boss-you put your code in a central location and you only provide the corresponding behavior (For example, notice the completeCourse() method in BasicEngineering and see how the course completion order is defined there).So, you have absolute control over the flow of the execution. You can also alter your template as per your need and then other participants need to follow you.

5. **What are the key advantages of using a template design pattern?**

   - You can control the flow of the algorithms. Clients cannot change them. (Notice that completeCourse() is marked with final keyword in the abstract class BasicEngineering)
   - Common operations are placed in a centralized location, for example, in an abstract class. The subclasses can redefine only the varying parts, so that, you can avoid redundant codes.

6. **What are key challenges associated with a template design pattern?**

   - Client code cannot direct the sequence of steps (If you need that approach, you may follow the Builder pattern).
   - A subclass can override a method defined in the parent class (i.e. hiding the original definition in parent class) which can go against Liskov Substitution Principle that basically says: If S is a subtype of T, then objects of type T can be replaced with objects of type S. You can learn the details from the following link: https://en.wikipedia.org/wiki/Liskov_substitution_principle
   - More subclass means more scattered codes and difficult maintenance.

7. **Looks like the subclasses can override other parent methods also in the BasicEngineering. Is the understanding correct?**

   You can do this but ideally that should not be your intent. In this pattern, you may not want to override all the parent methods entirely to bring the radical changes in the subclasses. In this way, it differs from simple polymorphism.

CHAPTER 17

Command Pattern

This chapter covers the command pattern.

GoF Definition

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queues, or log requests, and supports undoable operations.

Concept

Here you encapsulate a method invocation process. In general, four terms are associated: *invoker*, *client*, *command*, and *receiver*. A command object can invoke a method of the receiver in a way that is specific to that receiver's class. The receiver then starts processing the job. A command object is separately passed to the invoker object to invoke the command. The client object holds the invoker object and the command objects. The client only makes the decision—which commands to execute—and then it passes the command to the invoker object (for that execution).

Real-World Example

When you draw something with a pencil, you may need to undo (erase and redraw) some parts to make it better.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_17

Computer-World Example

The real-world scenario for painting applies to Microsoft Paint. You can use the Menu or Shortcut keys to perform the undo/redo operations in those contexts.

In general, you can observe this pattern in the menu system of an editor or IDE (integrated development environment). So, if you want to make an application that needs to support undos, multiple undos, or similar operations, then the command pattern can be your savior.

Microsoft used this pattern in Windows Presentation Foundation (WPF). The online source at https://visualstudiomagazine.com/articles/2012/04/10/command-pattern-in-net.aspx describes it in detail: "The command pattern is well suited for handling GUI interactions. It works so well that Microsoft has integrated it tightly into the Windows Presentation Foundation (WPF) stack. The most important piece is the ICommand interface from the System. Windows.Input namespace. Any class that implements the ICommand interface can be used to handle a keyboard or mouse event through the common WPF controls. This linking can be done either in XAML or in a code-behind."

Note When you implement the run() method of java.lang.Runnable interface , you are basically using the command design pattern. Another interface, java.swing. Action, also represents the command design pattern. It is important to note that the implementation of undos varies and can be complex. The memento design pattern also supports undo operations. You may need to use both of these design patterns in your application to implement a complex undo operation.

Illustration

Consider the following example. For an easy understanding, I am following similar class names to the concept described earlier. You can refer to the associated comments for a better understanding.

Class Diagram

Figure 17-1 shows the class diagram.

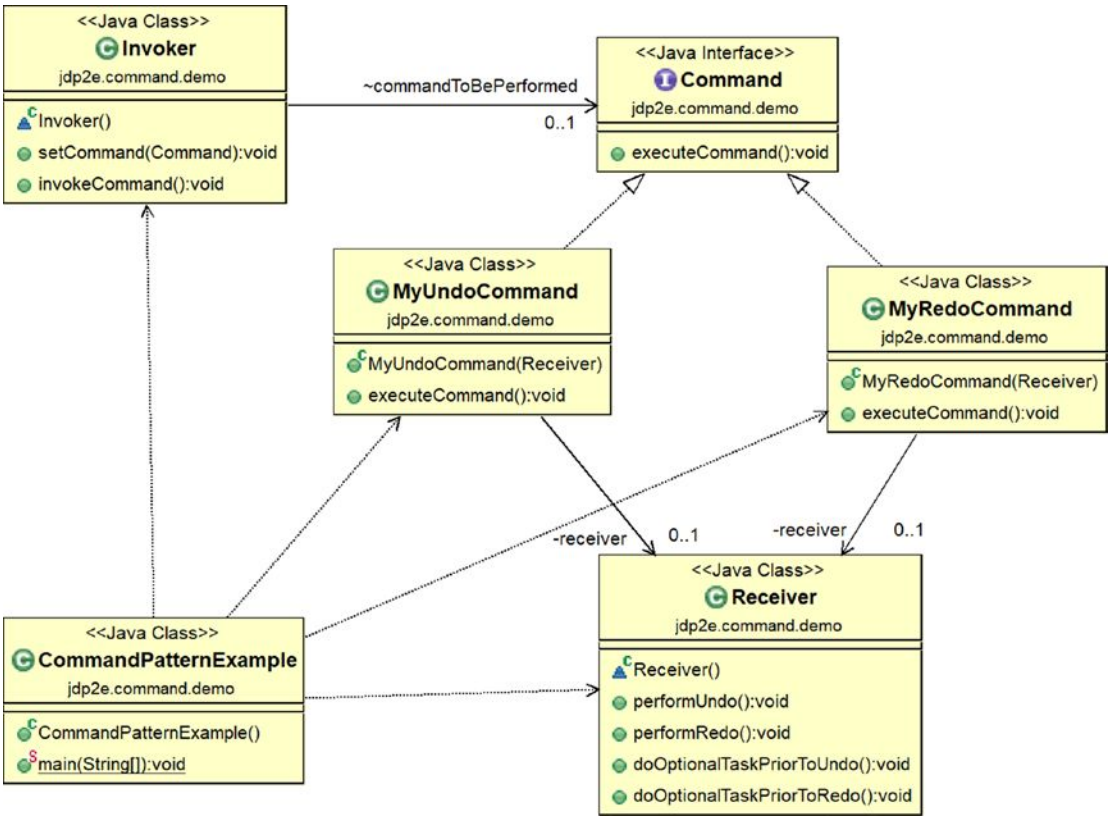


Figure 17-1. *Class diagram*

Package Explorer View

Figure 17-2 shows the high-level structure of the program.

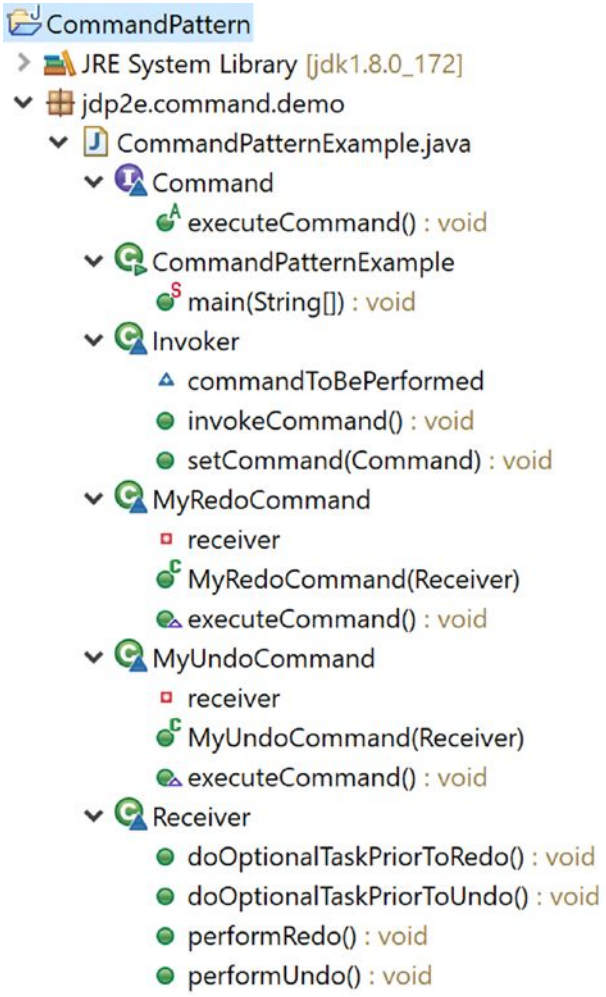


Figure 17-2. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.command.demo;

interface Command
{
    //Typically this method does not take any argument.
    //Some of the reasons are:
    //1.We supply all the information when it is created.
    //2.Invoker may reside in different address space.etc.
    void executeCommand();
}

class MyUndoCommand implements Command
{
    private Receiver receiver;
    public MyUndoCommand(Receiver receiver)
    {
        this.receiver=receiver;
    }
    @Override
    public void executeCommand()
    {
        //Perform any optional task prior to UnDo
        receiver.doOptionalTaskPriorToUndo();
        //Call UnDo in receiver now
        receiver.performUndo();
    }
}
class MyRedoCommand implements Command
{
    private Receiver receiver;
    public MyRedoCommand(Receiver receiver)
    {
        this.receiver=receiver;
    }
```

```
    @Override
    public void executeCommand()
    {
        //Perform any optional task prior to ReDo
        receiver.doOptionalTaskPriorToRedo();
        //Call ReDo in receiver now
        receiver.performRedo();
    }
}
//Receiver Class
class Receiver
{
    public void performUndo()
    {
        System.out.println("Performing an undo command in Receiver.");
    }
    public void performRedo()
    {
        System.out.println("Performing an redo command in Receiver.");
    }
    /*Optional method-If you want to perform
     any prior tasks before undo operations.*/
    public void doOptionalTaskPriorToUndo()
    {
        System.out.println("Executing -Optional Task/s prior to  execute
        undo command.");
    }
    /*Optional method-If you want to perform
     any prior tasks before redo operations*/
    public void doOptionalTaskPriorToRedo()
    {
        System.out.println("Executing -Optional Task/s prior to  execute
        redo command.");
    }

}
```

```
//Invoker class
class Invoker
{
    Command commandToBePerformed;
    //Alternative approach:
    //You can also initialize the invoker with a command object
    /*public Invoker(Command command)
    {
        this.commandToBePerformed = command;
    }*/

    //Set the command
    public void setCommand(Command command)
    {
        this.commandToBePerformed = command;
    }
    //Invoke the command
    public void invokeCommand()
    {
        commandToBePerformed.executeCommand();
    }
}

//Client
public class CommandPatternExample {

    public static void main(String[] args) {
        System.out.println("***Command Pattern Demo***\n");
        /*Client holds both the Invoker and Command Objects*/
        Receiver intendedReceiver = new Receiver();
        MyUndoCommand undoCmd = new MyUndoCommand(intendedReceiver);
        //If you use parameterized constructor of Invoker
        //use the following line of code.
        //Invoker invoker = new Invoker(undoCmd);
        Invoker invoker = new Invoker();
        invoker.setCommand(undoCmd);
        invoker.invokeCommand();
```

```
        MyRedoCommand redoCmd = new MyRedoCommand(intendedReceiver);
        invoker.setCommand(redoCmd);
        invoker.invokeCommand();
    }
}
```

Output

Here's the output.

```
***Command Pattern Demo***

Executing -Optional Task/s prior to  execute undo command.
Performing an undo command in Receiver.
Executing -Optional Task/s prior to  execute redo command.
Performing an redo command in Receiver.
```

Q&A Session

1. **I have two questions. In this example, you are dealing with a single receiver only. How can you deal with multiple receivers? And the GoF definition says that this pattern supports undoable operations. Can you show an example with a true undo operation using this pattern?**

   Consider the following program. The key characteristics of this program are as follows:

- Here you have two different receivers (Receiver1 and Receiver2). Each of them implements the Receiver interface methods. Since I am dealing with multiple receivers, I introduced a common interface, Receiver.

- In an undo operation, you generally want to reverse the last action or operation. A typical undo operation may involve complex logic. But in the upcoming implementation, I am presenting a simple example that supports undo operations with the following assumptions.
    - A Receiver1 object is initialized with the value 10 (the myNumber instance variable is used for this purpose) and a Receiver2 object is initialized with the "power off" status (the status instance variable is used for this purpose). Any Receiver1 object can keep adding 2 to an existing integer.
    - I have put a checkmark on the value 10, so that when you process an undo operation, if you notice that a Receiver1 object's myNumber is 10, you will not go beyond (because you started at 10).
    - A Receiver2 object does different things. It can switch a machine on or off. If the machine is already powered on, then by requesting an undo operation, you can switch off the machine and vice versa. But if your machine is already in switch on mode, then a further "switch on" request is ignored.

Modified Class Diagram

There are many participants and dependencies in the modified class diagram shown in Figure 17-3. To illustrate the main design and keep the diagram neat and clean, I do not show the client code dependencies.

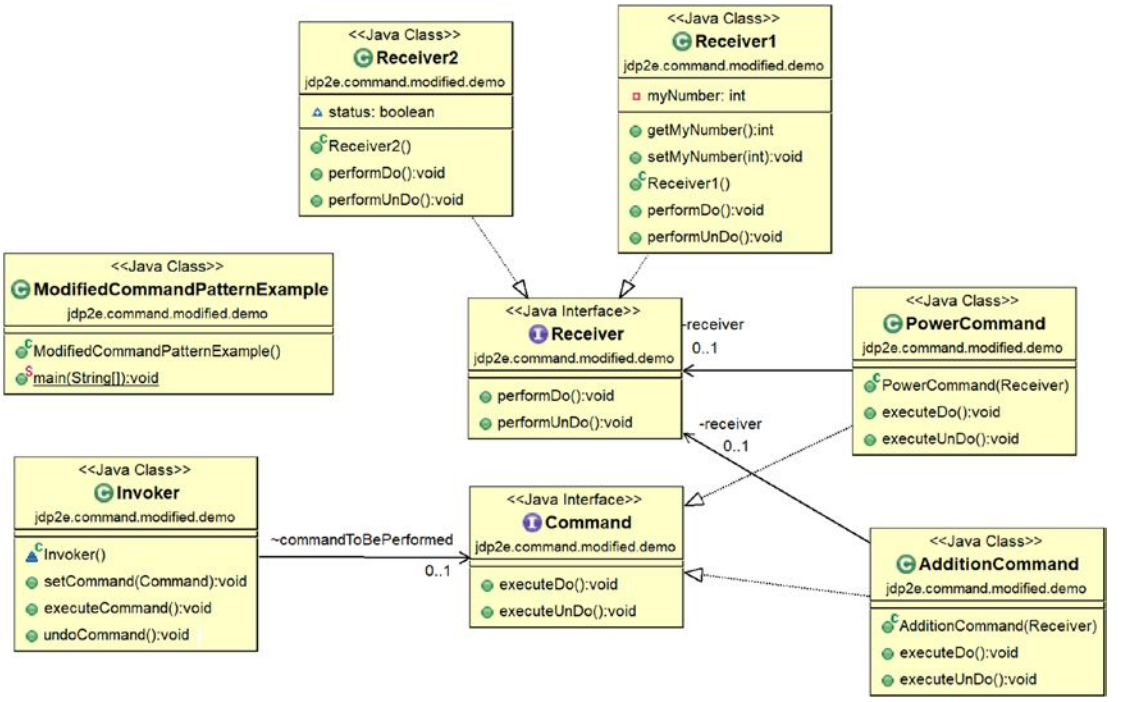


Figure 17-3. *Modified class diagram*

Modified Package Explorer View

Figure 17-4 shows the modified Package Explorer view.

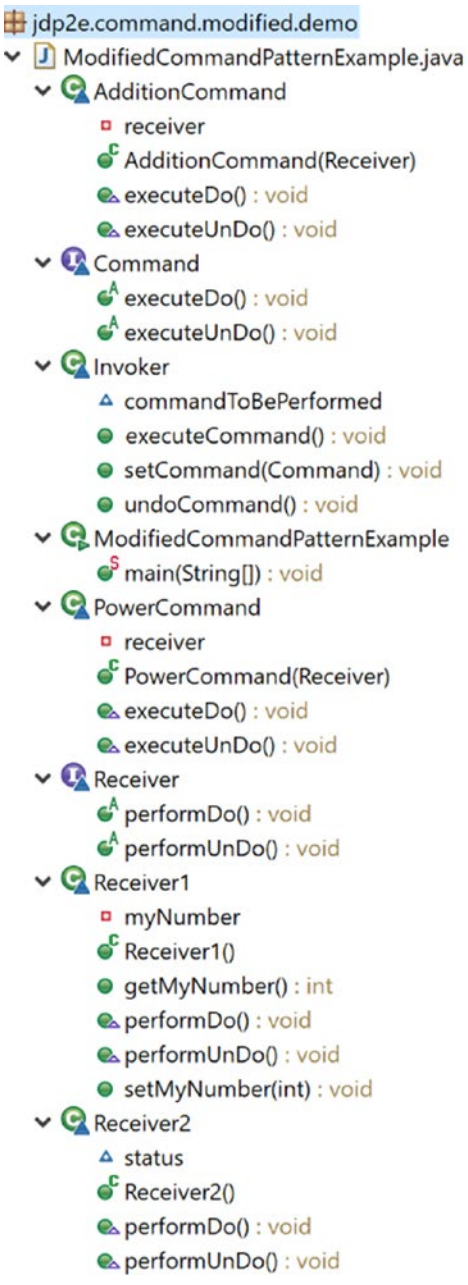


Figure 17-4. *Modified Package Explorer view*

Modified Implementation

Here's the modified implementation

```
package jdp2e.command.modified.demo;

/**
 *In general, an undo operation involves complex logic.
 But for simplicity, in this example,I assume that executeDo() can either
 add 2 with a given integer or it can switch on a machine.
 Similarly, executeUnDo() can either subtract 2 from a given number() or,
 it will switch off a machine.But you cannot go beyond the initialized
 value(i.e.10 in this case)*/

interface Command
{
    void executeDo();
    void executeUnDo();
}
class AdditionCommand implements Command
{
    private Receiver receiver;
    public AdditionCommand(Receiver receiver)
    {
        this.receiver = receiver;
    }
    @Override
    public void executeDo()
    {
        receiver.performDo();
    }
    @Override
    public void executeUnDo()
    {
        receiver.performUnDo();
    }
}
```

```
class PowerCommand implements Command
{
    private Receiver receiver;
    public PowerCommand(Receiver receiver)
    {
        this.receiver = receiver;
    }
    @Override
    public void executeDo()
    {
        receiver.performDo();
    }
    @Override
    public void executeUnDo()
    {
        receiver.performUnDo();
    }
}

//To deal with multiple receivers , we are using interfaces here
interface Receiver
{
    //It will add 2 with a number or switch on the m/c
    void performDo();
    //It will subtract 2 from a number or switch off the m/c
    void performUnDo();
}
//Receiver Class
class Receiver1 implements Receiver
{
    private int myNumber;

    public int getMyNumber()
    {
        return myNumber;
    }
```

```
    public void setMyNumber(int myNumber)
    {
        this.myNumber = myNumber;
    }
    public Receiver1()
    {
        myNumber = 10;
        System.out.println("Receiver1 initialized with " + myNumber);
        System.out.println("The objects of receiver1 cannot set beyond "+
        myNumber);
    }
    @Override
    public void performDo()
    {
        System.out.println("Received an addition request.");
        int presentNumber = getMyNumber();
        setMyNumber(presentNumber + 2);
        System.out.println(presentNumber +" + 2 ="+ this.myNumber);
    }
    @Override
    public void performUnDo()
    {
        System.out.println("Received an undo addition request.");
        int presentNumber = this.myNumber;
        //We started with number 10.We'll not decrease further.
        if (presentNumber > 10)
        {
            setMyNumber(this.myNumber - 2);
            System.out.println(presentNumber +" - 2 ="+ this.myNumber);
            System.out.println("\t Undo request processed.");
        }
        else
        {
            System.out.println("Nothing more to undo...");
        }
```

```
    }
}
//Receiver2 Class

class Receiver2 implements Receiver
{
    boolean status;

    public Receiver2()
    {
        System.out.println("Receiver2 initialized ");
        status=false;
    }
    @Override
    public void performDo()
    {
        System.out.println("Received a system power on request.");
        if( status==false)
        {
            System.out.println("System is starting up.");
            status=true;
        }
        else
        {
            System.out.println("System is already running.So, power on
            request is ignored.");

        }

    }
    @Override
    public void performUnDo()
    {
        System.out.println("Received a undo request.");
        if( status==true)
        {
            System.out.println("System is currently powered on.");
```

```
            status=false;
            System.out.println("\t Undo request processed.System is
            switched off now.");
        }
        else
        {
            System.out.println("System is switched off at present.");
            status=true;
            System.out.println("\t Undo request processed.System is powered
            on now.");

        }
    }
}

//Invoker class

class Invoker
{
    Command commandToBePerformed;
    public void setCommand(Command command)
    {
        this.commandToBePerformed = command;
    }
    public void executeCommand()
    {
        commandToBePerformed.executeDo();
    }
    public void undoCommand()
    {
        commandToBePerformed.executeUnDo();
    }
}
```

```
//Client
public class ModifiedCommandPatternExample {
    public static void main(String[] args) {

        System.out.println("***Command Pattern Q&As***");
        System.out.println("***A simple demo with undo supported
        operations***\n");
        //Client holds  both the Invoker and Command Objects

        //Testing receiver -Receiver1
        System.out.println("-----Testing operations in Receiver1-----");
        Receiver intendedreceiver = new Receiver1();
        Command currentCmd = new AdditionCommand(intendedreceiver);

        Invoker invoker = new Invoker();
        invoker.setCommand(currentCmd);
        System.out.println("*Testing single do/undo operation*");
        invoker.executeCommand();
        invoker.undoCommand();
        System.out.println("_______");
        System.out.println("**Testing a series of do/undo operations**");
        //Executed the command 2 times
        invoker.executeCommand();
        //invoker.undoCommand();
        invoker.executeCommand();
        //Trying to undo 3 times
        invoker.undoCommand();
        invoker.undoCommand();
        invoker.undoCommand();

        System.out.println("\n-----Testing operations in Receiver2-----");
        intendedreceiver = new Receiver2();
        currentCmd = new PowerCommand(intendedreceiver);
        invoker.setCommand(currentCmd);

        System.out.println("*Testing single do/undo operation*");
        invoker.executeCommand();
        invoker.undoCommand();
```

```
        System.out.println("_______");
        System.out.println("**Testing a series of do/undo operations**");
        //Executing the command 2 times
        invoker.executeCommand();
        invoker.executeCommand();
        //Trying to undo 3 times
        invoker.undoCommand();
        invoker.undoCommand();
        invoker.undoCommand();

    }

}
```

Modified Output

Here's the modified output.

```
***Command Pattern Q&As***
***A simple demo with undo supported operations***

-----Testing operations in Receiver1-----
Receiver1 initialized with 10
The objects of receiver1 cannot set beyond 10
*Testing single do/undo operation*
Received an addition request.
10 + 2 =12
Received an undo addition request.
12 - 2 =10
     Undo request processed.

_______
**Testing a series of do/undo operations**
Received an addition request.
10 + 2 =12
Received an addition request.
12 + 2 =14
Received an undo addition request.
```

```
14 - 2 =12
     Undo request processed.
Received an undo addition request.
12 - 2 =10
     Undo request processed.
Received an undo addition request.
Nothing more to undo...

-----Testing operations in Receiver2-----
Receiver2 initialized
*Testing single do/undo operation*
Received a system power on request.
System is starting up.
Received a undo request.
System is currently powered on.
     Undo request processed.System is switched off now.

_______
**Testing a series of do/undo operations**
Received a system power on request.
System is starting up.
Received a system power on request.
System is already running.So, power on request is ignored.
Received a undo request.
System is currently powered on.
     Undo request processed.System is switched off now.
Received a undo request.
System is switched off at present.
     Undo request processed.System is powered on now.
Received a undo request.
System is currently powered on.
     Undo request processed.System is switched off now.
```

2. **In this modified program, two receivers are doing different things. Is this intentional?**

   Yes. It shows the power and flexibilities provided by the command design pattern. You can see that `performDo()` in these receivers actually performs different actions. For Receiver1, it is adding 2 with an existing integer, and for Receiver2, it is switching on a machine. So, you may think that some other names like `addNumber()` and `powerOn()` would be more appropriate for them.

   But in this case, I needed to work with both the receivers and their corresponding methods. So, I needed to use a common interface and common names that could be used by both receivers.

   So, if you need to work with two different receivers that have different method names, you can replace them with a common name, use a common interface, and through polymorphism, you can invoke those methods easily.

3. **Why do you need the invoker?**

   Most of the time, programmers try to encapsulate data and corresponding methods in object-oriented programming. But if you look carefully, you find that in this pattern, you are trying to encapsulate command objects. In other words, you are implementing encapsulation from a different perspective.

   This approach makes sense when you deal with a complex set of commands.

   Now let's review the terms again. You create command objects to shoot them to receivers and invoke some methods. But you execute those commands through an invoker, which calls the methods of the command object (e.g., executeCommand). But for a simple case, this invoker class is not mandatory; for example, consider a case in which a command object has only one method to execute and you are trying to dispense with the invoker to invoke the method. But the invokers may play an important role when you want to keep track of multiple commands in a log file (or in a queue).

4. **Why are you interested in keeping track of these logs?**

   They are useful if you want to do the undo or redo operations.

5. **What are the key advantages associated with command patterns?**

   - Requests for creation and the ultimate execution are decoupled. Clients may not know how an invoker is performing the operations.
   - You can create macros (sequence of commands).
   - New commands can be added without affecting the existing system.
   - Most importantly, you can support the undo/redo operations.

6. **What are the challenges associated with command patterns?**

   - To support more commands, you need to create more classes. So, maintenance can be difficult as time goes on.
   - How to handle errors or make a decision about what to do with return values when an erroneous situation occurs becomes tricky. A client may want to know about those. But here you decouple the command with client codes, so these situations are difficult to handle. The challenge becomes significant in a multithreaded environment where the invoker is also running in a different thread.

CHAPTER 18

Iterator Pattern

This chapter covers the iterator pattern.

GoF Definition

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Concept

Using iterators, a client object can traverse a container (or a collection of objects) to access its elements without knowing how these data are stored internally. The concept is very useful when you need to traverse different kinds of collection objects in a standard and uniform way. The following are some important points about this pattern.

- It is often used to traverse the nodes of a tree-like structure. So, in many scenarios, you may notice the use of iterator patterns with composite patterns.
- The role of an iterator is not limited to traversing. This role can vary to support various requirements.
- Clients cannot see the actual traversal mechanism. A client program only uses the iterator methods that are public in nature.
- Figure 18-1 shows a sample diagram for an iterator pattern.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_18

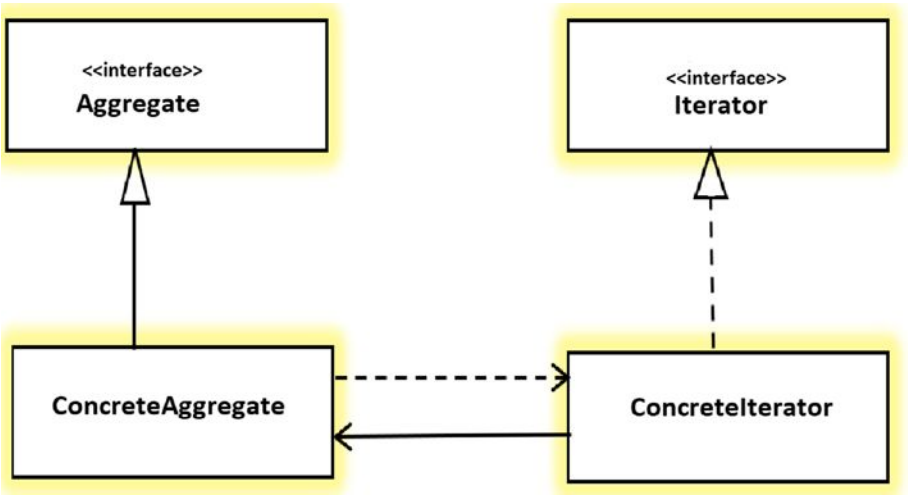


Figure 18-1. *A sample diagram for an iterator pattern*

The participants are as follows:

- *Iterator*: An interface to access or traverse elements.
- *ConcreteIterator*: Implements the Iterator interface methods. It can also keep track of the current position in the traversal of the aggregate.
- *Aggregate*: Defines an interface that can create an Iterator object.
- *ConcreteAggregate*: Implements the Aggregate interface. It returns an instance of ConcreteIterator.

Real-World Example

Suppose there are two companies: company A and company B. Company A stores its employee records (i.e., name, address, salary details, etc.) in a linked list data structure. Company B stores its employee data in an array data structure. One day, the two companies decide to merge to form a large organization. The iterator pattern is handy in such a situation because the developers do not want to write code from scratch. They can create a common interface so that they can access the data for both companies and invoke the methods in a uniform way.

Consider another example. Suppose that your company has decided to promote some employees based on their performances. So, all the managers get together and set a common criterion for promotion. Then they iterate over the past records of each employee to mark potential candidates for promotion.

Lastly, when you store songs in your preferred audio devices— an MP3 player or your mobile devices, for example, you can iterate over them through various button press or swipe movements. The basic idea is to provide you some mechanism to smoothly iterate over your list.

Computer-World Example

Similarly, let's assume that, a college arts department is using an array data structure to maintain its students' records. The science department is using a linked list data structure to keep their students' records. The administrative department does not care about the different data structures, they are simply interested in getting the data from each of the departments and they want to access the data in a universal way.

Note The iterator classes in Java's collection framework are iterator examples. When you use the interfaces like java.util.Iterator or java.util.Enumeration , you basically use this pattern. The java.util.Scanner class also follows this pattern. If you are familiar with C#, you may use C#'s own iterators that were introduced in Visual Studio 2005.The foreach statement is frequently used in this context.

Illustration

In this chapter, there are three different implementations of the iterator pattern. I'll start with an example that follows the core theory of this pattern. In the next example, I'll modify the example using Java's built-in support of the iterator pattern. In the third and final example, you use this pattern with a different data structure. In the first two examples, I'll simply use "String" data types but in the final example, I'll use a complex data type.

Before you start, I suggest that you note the structure in the Package Explorer view for your immediate reference.

In the first implementation, let's assume that in a particular college, an arts department student needs to study four papers (or subjects)—English, history, geography, and psychology. The details of these papers are stored in an array data structure. And your job is to print the curriculum using an iterator.

Let's assume that your iterator currently supports four basic methods: `first()`, `next()`, `currentItem()`, and `hasNext()`.

- The `first()` method resets the pointer to the first element before you start traversing a data structure.
- The `next()` method returns the next element in the container.
- The `currentItem()` method returns the current element of the container that the iterator is pointing at a particular point of time.
- The `hasNext()` validates whether any next element is available for further processing. So, it helps you determine whether you have reached the end of your container.

Class Diagram

Figure 18-2 shows the class diagram.

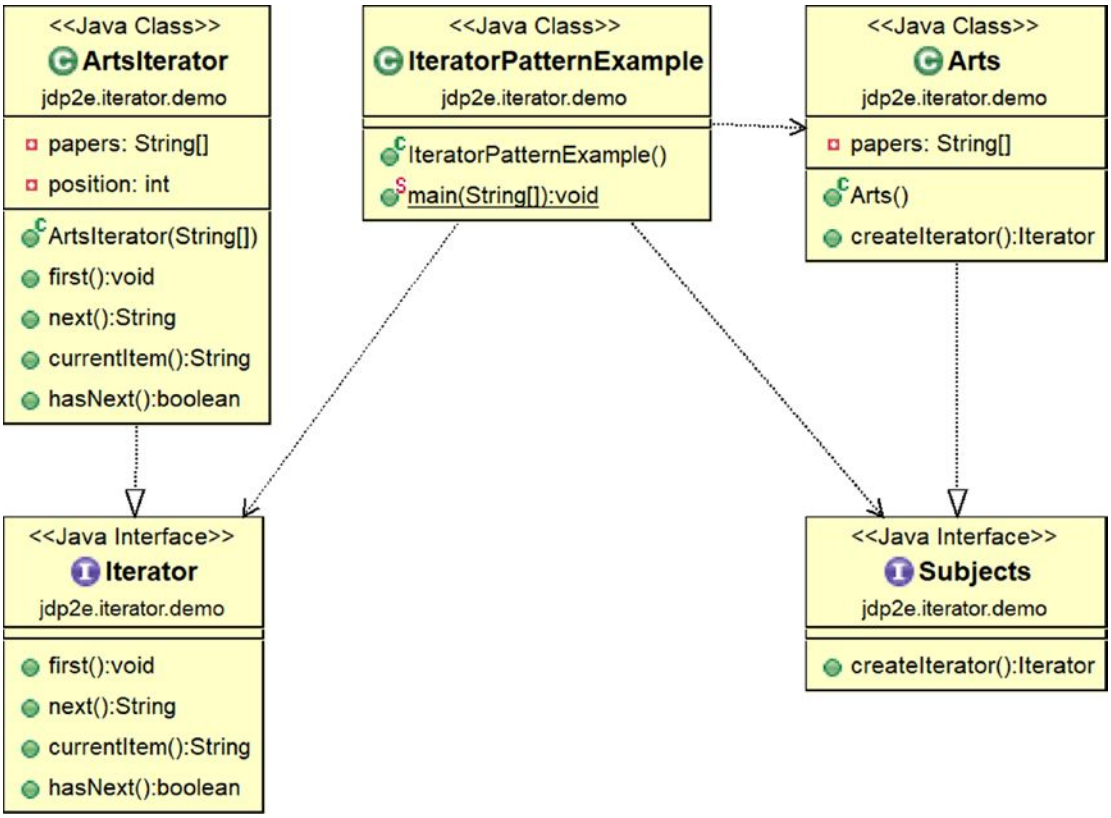


Figure 18-2. *Class diagram*

Note Like many of the previous examples in this book, to present a clean class diagram, I have shown only client code dependencies. For any ObjectAid class diagrams shown in the Eclipse editor, you can always see other dependencies by selecting an element in the diagram, right-clicking it, and selecting Add ➤ Dependencies.

Package Explorer View

Figure 18-3 shows the high-level structure of the program.

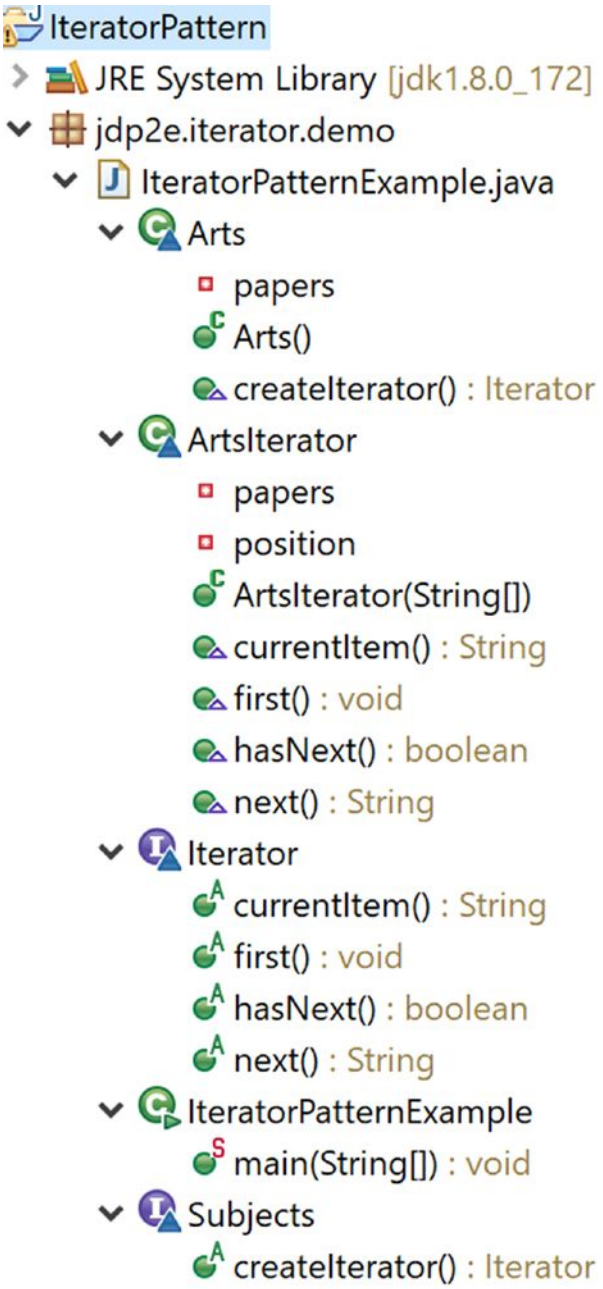


Figure 18-3. *Package Explorer view*

First Implementation

Here's the first implementation.

```
package jdp2e.iterator.demo;

interface Subjects
{
    Iterator createIterator();
}
class Arts implements Subjects
{
    private String[] papers;

    public Arts()
    {
        papers = new String[] { "English","History",
        "Geography","Psychology" };
    }

    public Iterator createIterator()
    {
        return new ArtsIterator(papers);
    }
}
interface Iterator
{
    void first();//Reset to first element
    String next();//To get the next element
    String currentItem();//To retrieve the current element
    boolean hasNext();//To check whether there is any next element or not.
}
class ArtsIterator implements Iterator
{
    private String[] papers;
    private int position;
    public ArtsIterator(String[] papers)
```

```
    {
        this.papers = papers;
        position = 0;
    }
    @Override
    public void first()
    {
        position = 0;
    }
    @Override
    public String next()
    {
        //System.out.println("Currently pointing to: "+ this.
        currentItem());
        return papers[position++];
    }
    @Override
    public String currentItem()
    {
        return papers[position];
    }
    @Override
    public boolean hasNext()
    {
        if(position >= papers.length)
            return false;
        return true;
    }
}

public class IteratorPatternExample {

    public static void main(String[] args) {
        System.out.println("***Iterator Pattern Demo***");
        Subjects artsSubjects = new Arts();
```

```
        Iterator iteratorForArts = artsSubjects.createIterator();
        System.out.println("\n Arts subjects are as follows:");
        while (iteratorForArts.hasNext())
        {
            System.out.println(iteratorForArts.next());
        }
        //Moving back to first element
        iteratorForArts.first();
        System.out.println(" Currently pointing back to: "+
        iteratorForArts.currentItem());
    }
}
```

Output

Here's the output.

```
***Iterator Pattern Demo***

 Arts subjects are as follows:
English
History
Geography
Psychology
 Currently pointing back to: English
```

Note If you want to see the current element that the iterator is pointing to, you can uncomment the line in the next() method: // System.out.println("Currently pointing to: "+ this.currentItem());

Now let's modify the previous implementation using Java's built-in Iterator interface.

Key Characteristics of the Second Implementation

I used Java's built-in support for the iterator pattern. Note the inclusion of the following line at the beginning of the program.

```
import java.util.Iterator;
```

If you open the source code, you see that this interface has three methods: `hasNext()`, `next()`, and `remove()`. But the `remove()` method has a default implementation already. So, in the following example, I needed to override the `hasNext()` and `next()` methods only.

Here you are using the Java's Iterator interface, so there is no need to define your own Iterator interface.

In this modified implementation, key changes are shown in bold.

Second Implementation

Here's the second implementation.

```
package jdp2e.iterator.modified.demo;

import java.util.Iterator;

interface Subjects
{
    //Iterator CreateIterator();
    ArtsIterator createIterator();
}
class Arts implements Subjects
{
    private String[] papers;

    public Arts()
    {
        papers = new String[] { "English","History",
        "Geography","Psychology" };
    }
```

```
    //public Iterator CreateIterator()
    public ArtsIterator createIterator()
    {
        return new ArtsIterator(papers);
    }
}

class ArtsIterator implements Iterator<String>
{
    private String[] papers;
    private int position;
    public ArtsIterator(String[] papers)
    {
        this.papers = papers;
        position = 0;
    }
    public void first()
    {
        position = 0;
    }
    public String currentItem()
    {
        return papers[position];
    }
    @Override
    public boolean hasNext()
    {
        if(position >= papers.length)
            return false;
        return true;
    }
```

```
    @Override
    public String next()
    {
        return papers[position++];
    }
}

public class ModifiedIteratorPatternExample {

    public static void main(String[] args) {
        System.out.println("***Modified Iterator Pattern Demo.***");
        Subjects artsSubjects = new Arts();

        //Iterator IteratorForArts = artsSubjects.createIterator();
        ArtsIterator iteratorForArts = artsSubjects.createIterator();
        System.out.println("\nArts subjects are as follows:");
        while (iteratorForArts.hasNext())
        {
            System.out.println(iteratorForArts.next());
        }
        //Moving back to first element
        iteratorForArts.first();
        System.out.println("Currently pointing to: "+ iteratorForArts.
        currentItem());
    }
}
```

Output

Here's the modified output.

```
***Modified Iterator Pattern Demo.***

Arts subjects are as follows:
English
History
Geography
Psychology
Currently pointing to: English
```

Q&A Session

1. **What is the use of an iterator pattern?**

   - You can traverse an object structure without knowing its internal details. As a result, if you have a collection of different subcollections (e.g., your container is mixed up with arrays, lists, or linked lists, etc.), you can still traverse the overall collection and deal with the elements in a universal way without knowing the internal details or differences among them.
   - You can traverse a collection in different ways. You can also provide an implementation that supports multiple traversals simultaneously.

2. **What are the key challenges associated with this pattern?**

   Ideally, during a traversal/iteration process, you should not perform any accidental modification to the core architecture.

3. **But to deal with the challenge mentioned earlier, you could make a backup and then proceed. Is this correct?**

   Making a backup and reexamining later is a costly operation.

4. **Throughout the discussion, you have talked about collections. What is a collection?**

   It is a group of individual objects that are presented in a single unit. You may often see the use of the interfaces like java.util. Collection, java.util.Map, and so forth, in Java programs. These are some common interfaces for Java's collection classes, which were introduced in JDK 1.2.

   Prior to collections, you had choices like arrays, vectors, and so forth, to store or manipulate a group of objects. But these classes did not have a common interface; the way you needed to access elements in an array were quite different from the way you needed to access the elements of a vector. That is why it was difficult to

write a common algorithm to access different elements from these different implementations. Also, many of these methods were final, so you could not extend them.

The collection framework was introduced to address these kinds of difficulties. At the same time, they provided high-performance implementations to make a programmer's life easier.

5. **In the modified implementation, why am I not seeing the @Override annotation for the first() and currentItem() methods?**

   These two methods are not present in the java.util.Iterator interface. The built-in Iterator interface has the `hasNext()` and `next()` methods. So, I used the @Override annotation for these methods. There is another method, `remove()`, in this interface. It has a default implementation. Since I have not used it, I did not need to modify this method.

6. **In these implementations, I am seeing that you are only using strings of arrays to store and manipulate data. Can you show an iterator pattern implementation that uses a relatively complex data type and a different data structure?**

   To make these examples simple and straightforward, I only used strings and an array data structure. You can always choose your preferred data structure and apply the same process when you consider a complex data type. For example, consider the following illustration (third implementation) with these key characteristics.

   - Here I am using a relatively complex data type, Employee. Each employee object has three things: a name, an identification number (id), and the salary.
   - Instead of an array, I used a different data structure, LinkedList, in the following implementation. So, I need to include the following line in this implementation.

     ```
     import java.util.LinkedList;
     ```

   - I have followed the same approach that I used in the previous example.

Third Implementation

Here's the third implementation.

```
package jdp2e.iterator.questions_answers;
import java.util.Iterator;
import java.util.LinkedList;

class Employee
{
    private String name;
    private int id;
    private double salary;
    public Employee(String name, int id, double salary )
    {
        this.name=name;
        this.id=id;
        this.salary=salary;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public double getSalary() {
        return salary;
    }
```

```
    public void setSalary(double salary) {
        this.salary = salary;
    }

    @Override
    public String toString(){
        return "Employee Name: "+this.getName()+", ID: "+this.getId()+ "
        and salary: "+this.getSalary()+"$";
    }
}
interface DataBase
{
    EmployeeIterator createIterator();
}
class EmployeeDatabase implements DataBase
{
    private LinkedList<Employee> employeeList;

    public EmployeeDatabase()
    {
        employeeList = new LinkedList<Employee>();
        employeeList.add(new Employee("Ron",1, 1000.25));
        employeeList.add(new Employee("Jack",2, 2000.5));
        employeeList.add(new Employee("Ambrose",3, 3000.75));
        employeeList.add(new Employee("Jian",4, 2550.0));
        employeeList.add(new Employee("Alex",5, 753.83));
    }
    public EmployeeIterator createIterator()
    {
        return new EmployeeIterator(employeeList);
    }
}

class EmployeeIterator implements Iterator<Employee>
{
    private LinkedList<Employee> employeeList;
    private int position;
```

```
    public EmployeeIterator(LinkedList<Employee> employeeList)
    {
        this.employeeList= employeeList;
        position = 0;
    }
    //@Override
    public void first()
    {
        position = 0;
    }

    //@Override
    public Employee currentItem()
    {
        return employeeList.get(position);
    }

    @Override
    public Employee next()
    {
        return employeeList.get(position++);
    }
    @Override
    public boolean hasNext() {
        if(position >= employeeList.size())
            return false;
        return true;
    }
}

public class ModifiedIteratorPatternExample2 {

    public static void main(String[] args) {
        System.out.println("***Modified Iterator Pattern Demo.
        Example-2.***");
        DataBase employeesList = new EmployeeDatabase();
```

```
        EmployeeIterator iteratorForEmployee = employeesList.
        createIterator();
        System.out.println("\n -----Employee details are as
        follows-----\n");

        while (iteratorForEmployee.hasNext())
        {
            System.out.println(iteratorForEmployee.next());
        }

    }
}
```

Output

Here's the output from the third implementation.

```
***Modified Iterator Pattern Demo.Example-2.***

 -----Employee details are as follows-----

Employee Name: Ron, ID: 1 and salary: 1000.25$
Employee Name: Jack, ID: 2 and salary: 2000.5$
Employee Name: Ambrose, ID: 3 and salary: 3000.75$
Employee Name: Jian, ID: 4 and salary: 2550.0$
Employee Name: Alex, ID: 5 and salary: 753.83$
```

Note You may use two or more different data structures in an implementation to demonstrate the power of this pattern. You have seen that across these different implementations, I have used the first(), next(), hasNext(), and currentItem() methods with different implementations that vary due to their internal data structures.

CHAPTER 19

Memento Pattern

This chapter covers the memento pattern.

GoF Definition

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Concept

In your application, you may need to support "undo" operations. To achieve this, you need to record the internal state of an object. So, you must save this state information in a place that can be referred again to revert back the old state of the object. But in general, objects encapsulate their states, and those states are inaccessible to the outer world. So, if you expose the state information, then you violate encapsulation.

The dictionary meaning of *memento* is reminder (of past events). So, you can guess that using this pattern, you can restore an object to its previous state, but it ensures that you achieve your goal without violating the encapsulation.

Real-World Example

A classic example in this category is noticed in a finite state machine. It is a mathematical model, but one of its simple applications can be found in a turnstile. It has rotating arms, which initially are locked. If you are allowed to pass through it (for example, when you insert coins or when a security person allows you to go through a security check), the locks are opened. Once you pass through, the turnstile returns to the locked state again.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_19

Computer-World Example

In a drawing application, you may need to revert back to a prior state.

Note You notice a similar pattern when you consider the JTextField class, which extends the javax.swing.text.JTextComponent abstract class and provides an undo support mechanism. Here javax.swing.undo.UndoManager can act as a caretaker, an implementation of javax.swing.undo. UndoableEdit can act like a memento, and an implementation of javax.swing.text.Document can act like an originator. You will learn about the terms originator, caretaker, and memento shortly. Also, java.io.Serializable is often called an example of a memento but although you can serialize a memento object, it is not a mandatory requirement for the memento design pattern.

Illustration

Go through the code and follow the comments for your ready reference. In this example, three objects are involved: a memento, an originator, and a caretaker. (These names are very common, so I have kept the same naming convention in our implementation.)

The *originator* object has an internal state. A client can set a state in it. A *memento* object may store as much or as little of the originator's state, at the originator's discretion. When a *caretaker* wants to record the state of the originator, it requests the current state from it. So, it first asks the originator for a memento object.

In the following example, the caretaker object confirms the save operation by displaying a console message. Suppose that the client makes some changes and then wants to revert back to the previous state. Since the originator object's state is already changed, to roll back to the previous state requires help from the caretaker object, which saved the state earlier. The caretaker object returns the memento object (with the previous state) to the originator. The memento object itself is an opaque object (one which the caretaker is not allowed to make any change to, and ideally, only the originator, who created the memento can access the memento's internal state).

So, you can conclude that caretaker has a narrow view/interface to the memento because it can only pass it to other objects. In contrast, the originator sees the wide interface because it can access the data necessary to return to a previous state.

Class Diagram

Figure 19-1 shows the class diagram.

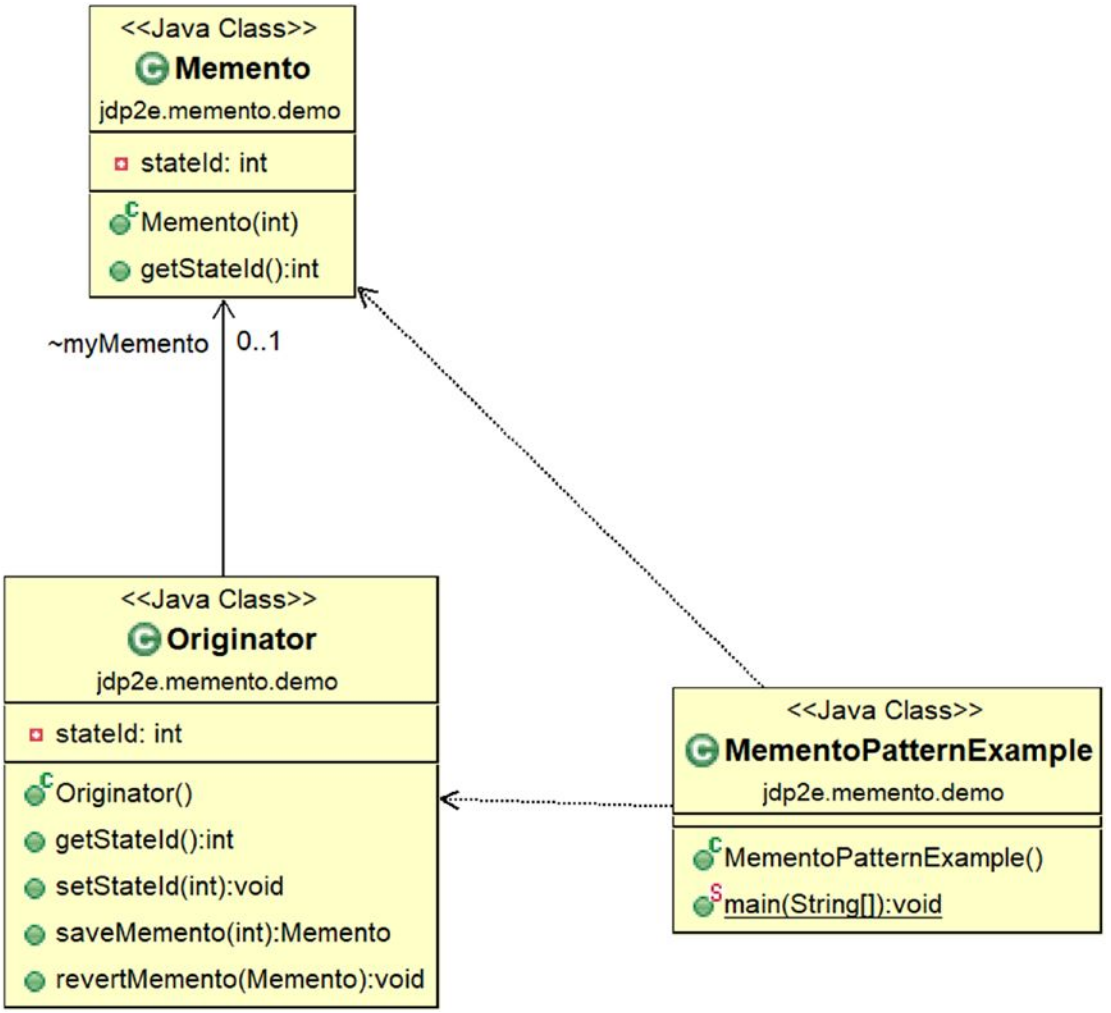


Figure 19-1. *Class diagram*

Package Explorer View

Figure 19-2 shows the high-level structure of the parts of the program.

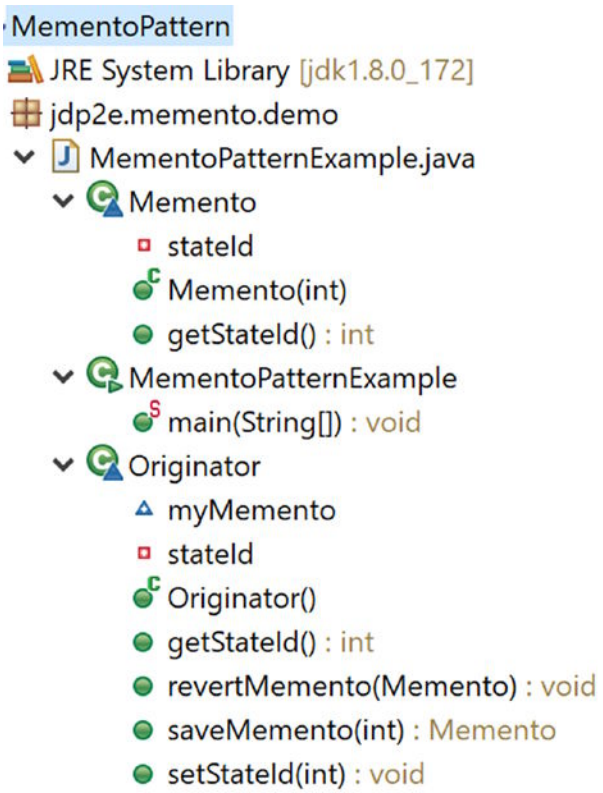


Figure 19-2. *Package Explorer view*

Implementation

Here is the implementation.

```
package jdp2e.memento.demo;

class Memento
{
    private int stateId;
    public Memento(int stateId)
    {
        this.stateId = stateId;
    }
```

```
    public int getStateId() {
        return stateId;
    }
    /*This class does not have the
    setter method.We need to use this class
    to get the state of the object only.*/

    /*public void setState(String state) {
        this.state = state;
    }*/

}

/*
The 'Originator' class
WikiPedia notes( for your reference):
Make an object (originator) itself responsible for:
1.Saving its internal state to a(memento) object and
2.Restoring to a previous state from a(memento) object.
3.Only the originator that created a memento is allowed to access it.
 */
class Originator
{
    private int stateId;
    public Originator()
    {
        this.stateId = 0;
        System.out.println(" Originator is created with state id :
        "+ stateId);
    }

    public int getStateId()
    {
        return stateId;
    }
```

```
    public void setStateId(int stateId)
    {
        System.out.println(" Setting the state id of the originator to :
        "+ stateId);
        this.stateId= stateId;
    }
    //Saving its internal state to a(memento) object
    public Memento saveMemento(int stateId)
    {
        System.out.println(" Saving originator's current state id. ");
        //Create memento with the current state and return it.
        return new Memento(stateId);
    }

    //Restoring to a previous state from a(memento) object.
    public void revertMemento(Memento previousMemento)
    {
        System.out.println(" Restoring to state id..."+ previousMemento.
        getStateId());
        this.stateId = previousMemento.getStateId();
        System.out.println(" Current state id of originator : "+ stateId);
    }
}
/*
The 'Caretaker' class.
WikiPedia notes( for your reference):
1.A client (caretaker) can request a memento from the originator  to save
the internal state of the originator and
2.Pass a memento back to the originator (to restore to a previous state)
This enables to save and restore the internal state of an originator
without violating its encapsulation.
 */
public class MementoPatternExample {

    public static void main(String[] args) {
        System.out.println("***Memento Pattern Demo***\n");
```

```
        //Originator is initialized with a state
        Originator originatorObject = new Originator();
        Memento mementoObject;
        originatorObject.setStateId(1);
        // A client (caretaker) can request a memento from the originator
        //to save the internal state of the originator
        mementoObject=originatorObject.saveMemento(originatorObject.
        getStateId());
        System.out.println(" Snapshot #1: Originator's current state id is
        saved in caretaker.");
        //A client (or caretaker) cannot set/modify the memento's state
        //mementoObject.setState("arbitratyState");//error

        //Changing the state id of Originator
        originatorObject.setStateId(2);
        mementoObject=originatorObject.saveMemento(originatorObject.
        getStateId());
        System.out.println(" Snapshot #2: Originator's current state id is
        saved in caretaker.");

        //Changing the state id of Originator again.
        originatorObject.setStateId(3);
        //Reverting back to previous state id.
        originatorObject.revertMemento(mementoObject);
    }

}
```

Output

Here is the output.

```
***Memento Pattern Demo***

 Originator is created with state id : 0
 Setting the state id of the originator to : 1
 Saving originator's current state id.
 Snapshot #1: Originator's current state id is saved in caretaker.
```

```
Setting the state id of the originator to : 2
Saving originator's current state id.
Snapshot #2: Originator's current state id is saved in caretaker.
Setting the state id of the originator to : 3
Restoring to state id...2
Current state id of originator : 2
```

Note If you deal with a state that is a mutable reference type, you may need to do a deep copy to store the state inside the Memento object.

Q&A Session

1. **I can restore the previous snapshot/restore point. But in a real-life scenario, I may have multiple restore points. How can you implement that using this design pattern?**

   You can use an ArrayList in such a case. Consider the following program.

   The Originator class and Memento class are same as before, so I am presenting the modified Caretaker class only. I am using the following line of code in the upcoming implementation.

   ```
   List<Memento> savedStateIds = new ArrayList<Memento>();
   ```

   So, you need to include these two lines of code at the beginning.

   ```
   import java.util.ArrayList;
   import java.util.List;
   ```

Modified Caretaker Class

This is the modified Caretaker class.

```
   /*
The modified 'Caretaker' class.
WikiPedia notes( for your reference):
1.A client (caretaker) can request a memento from the originator to save
the internal state of the originator and
2.Pass a memento back to the originator (to restore to a previous state)
This enables to save and restore the internal state of an originator
without violating its encapsulation.
 */

public class ModifiedMementoPatternExample {

    public static void main(String[] args) {
        System.out.println("***Modified Memento Pattern Demo***\n");
        List<Memento> savedStateIds = new ArrayList<Memento>();
        //Originator is initialized with a state
        Originator originatorObject = new Originator();
        Memento mementoObject;
        originatorObject.setStateId(1);
        mementoObject=originatorObject.saveMemento(originatorObject.
        getStateId());
        savedStateIds.add( mementoObject);
        System.out.println(" Snapshot #1: Originator's current state id is
        saved in caretaker.");
        //A client or caretaker cannot set/modify the memento's state
        //mementoObject.setState("arbitratyState");//error

        //Changing the state id of Originator
        originatorObject.setStateId(2);
        mementoObject=originatorObject.saveMemento(originatorObject.
        getStateId());
        savedStateIds.add( mementoObject);
        System.out.println(" Snapshot #2: Originator's current state id is
        saved in caretaker.");
```

```
        //Changing the state id of Originator
        originatorObject.setStateId(3);
        mementoObject=originatorObject.saveMemento(originatorObject.
        getStateId());
        savedStateIds.add( mementoObject);
        System.out.println(" Snapshot #3: Originator's current state id is
        saved in caretaker (client).");

        //Reverting back to previous state id.
        //originatorObject.revertMemento(mementoObject);
        //Reverting back to specific id -say, Snapshot #1)
        //originatorObject.revertMemento(savedStateIds.get(0));

        //Roll back everything...
        System.out.println("Started restoring process...");
        for (int i = savedStateIds.size(); i > 0; i--)
        {
            originatorObject.revertMemento(savedStateIds.get(i-1));
        }
    }
}
```

Modified Output

Once you run this modified program, you get the following output.

```
***Modified Memento Pattern Demo***

 Originator is created with state id : 0
 Setting the state id of the originator to : 1
 Saving originator's current state id.
 Snapshot #1: Originator's current state id is saved in caretaker.
 Setting the state id of the originator to : 2
 Saving originator's current state id.
 Snapshot #2: Originator's current state id is saved in caretaker.
 Setting the state id of the originator to : 3
 Saving originator's current state id.
```

```
 Snapshot #3: Originator's current state id is saved in caretaker (client).
Started restoring process...
 Restoring to state id...3
 Current state id of originator : 3
 Restoring to state id...2
 Current state id of originator : 2
 Restoring to state id...1
 Current state id of originator : 1
```

Analysis

In this modified program, you can see three different variations of "undo" operations.

- You can just go back to the previous restore point.
- You can go back to your specified restore point.
- You can revert back to all restore points.

To see cases 1 and 2, uncomment the lines in the previous implementation.

2. **In many applications, I notice that the memento class is presented as an inner class of Originator. Why are you not following that approach?**

   The memento design pattern can be implemented in many different ways (for example, using package-private visibility or using object serialization techniques). But in each case, if you analyze the key aim, you find that once the memento instance is created by an originator, no one else besides its creator is allowed to access the internal state (this includes the caretaker/client). A caretaker's job is to store the memento instance (restore points in our example) and supply them back when you are in need. So, there is no harm if your memento class is public. You can just block the public setter method for the memento. I believe that it is sufficient enough.

3. **But you are still using the getter method getStateId(). Does it not violate the encapsulation?**

   There is a lot of discussion and debate around this area—whether you should use getter/setter or not, particularly when you consider encapsulation. I believe that it depends on the amount of strictness that you want to impose. For example, if you just provide getter/setters for all fields without any reason, that is surely a bad design. The same thing applies when you use all the public fields inside the objects. But sometimes the accessor methods are required and useful. In this book, my aim is to encourage you learn design patterns with simple examples. If I need to consider each and every minute detail such as this, you may lose interest. So, in these examples, I show a simple way to promote encapsulation using the memento pattern. But, if you want to be stricter, you may prefer to implement the memento class as an inner class of the originator and modify the initial implementation, like in the following.

```
package jdp2e.memento.questions_answers;

/*
The 'Originator' class
WikiPedia notes( for your reference):
Make an object (originator) itself responsible for:
1.Saving its internal state to a(memento) object and
2.Restoring to a previous state from a(memento) object.
3.Only the originator that created a memento is allowed to access it.
 */
class Originator
{
    private int stateId;
    Memento myMemento;
    public Originator()
    {
        this.stateId = 0;
```

```
    System.out.println(" Originator is created with state id :
    "+ stateId);
}

public int getStateId()
{
    return stateId;
}

public void setStateId(int stateId)
{
    System.out.println(" Setting the state id of the
    originator to : "+ stateId);
    this.stateId= stateId;
}
//Saving its internal state to a(memento) object
public Memento saveMemento()
{
    System.out.println(" Saving originator's current state id. ");
    //Create memento with the current state and return it.
    return new Memento(this.stateId);
}

//Restoring to a previous state from a(memento) object.
public void revertMemento(Memento previousMemento)
{
    System.out.println(" Restoring to state id..."+
    previousMemento.getStateId());
    this.stateId = previousMemento.getStateId();
    System.out.println(" Current state id of originator : "+
    stateId);
}
```

```
    //A memento class implemented as an inner class of Originator
    static class Memento
    {
        private int stateId;
        public Memento(int stateId)
        {
            this.stateId = stateId;
        }
        //Only outer class can access now
        public int getStateId() {
            return stateId;
        }
        /*This class does not have the
        setter method.We need to use this class
        to get the state of the object only.*/

        /*public void setState(String state) {
            this.state = state;
        }*/

    }
}
/*
The 'Caretaker' class.
WikiPedia notes( for your reference):
1.A client (caretaker) can request a memento from the originator
to save the internal state of the originator and
2.Pass a memento back to the originator (to restore to a previous
state)
This enables to save and restore the internal state of an
originator without violating its encapsulation.
 */
public class MementoAsInnerClassExample {

    public static void main(String[] args) {
        System.out.println("***Memento Pattern Demo***\n");
```

```
        //Originator is initialized with a state
        Originator originatorObject = new Originator();
        Originator.Memento mementoObject;
        originatorObject.setStateId(1);
        // A client (caretaker) can request a memento from the
        originator
        //to save the internal state of the originator
        mementoObject=originatorObject.saveMemento();
        System.out.println(" Snapshot #1: Originator's current
        state id is saved in caretaker.");
        //A client (or caretaker) cannot set/modify the memento's
        state

        //Changing the state id of Originator
        originatorObject.setStateId(2);
        mementoObject=originatorObject.saveMemento();
        System.out.println(" Snapshot #2: Originator's current
        state id is saved in caretaker.");

        //Changing the state id of Originator again.
        originatorObject.setStateId(3);
        //Reverting back to previous state id.
        originatorObject.revertMemento(mementoObject);
    }

}
```

4. **What are the key advantages of using a memento design pattern?**

   - The biggest advantage is that you can always discard the unwanted changes and restore it to an intended or stable state.
   - You do not compromise the encapsulation associated with the key objects that are participating in this model.
   - Maintains high cohesion.
   - Provides an easy recovery technique.

5. **What are key challenges associated with a memento design pattern?**

   - A high number of mementos require more storage. At the same time, they put additional burdens on a caretaker.
   - The preceding point increases maintenance costs in parallel.
   - You cannot ignore the time to save these states. The additional time to save the states decreases the overall performance of the system.

Note In a language like C# or Java, developers may prefer the serialization/deserialization techniques instead of directly implementing a memento design pattern. Both techniques have their own pros/cons. But you can also combine both techniques in your application.

---

6. **In these implementations, if you make the originator's state public, then our clients also could directly access the states. Is this correct?**

   Yes. But you should not try to break the encapsulation. Notice the GoF definition that begins "without violating *encapsulation*…"

7. **In these implementations, the memento class does not have a public setter method. What is the reason behind this?**

   Go through the answer of question 2 again. And read the comment in the code that says, "Only the originator that created a memento is allowed to access it." So, if you do not provide a public setter method for your memento class, the caretaker or client cannot modify the memento instances that are created by an originator.

8. **In these implementations, you could ignore the getter method of the memento by using package-private visibility for stateId. For example, you could code memento class like the following.**

```
class Memento
{
    //private int stateId;
    int stateId;//←-Change is here
    public Memento(int stateId)
    {
        this.stateId = stateId;
    }
    public int getStateId() {
        return stateId;
    }
    /*This class does not have the
    setter method.We need to use this class
    to get the state of the object only.*/

    /*public void setState(String state) {
        this.state = state;
    }*/

}
```

And then you can use the following line.

```
//System.out.println(" Restoring to state id..."+
previousMemento.getStateId());
System.out.println(" Restoring to state id..."+
previousMemento.stateId);//←The change is shown in bold
```

Is this correct?

Yes. In many application, other classes (except originator) cannot even read the memento's state. When you use package-private visibility, you do not need any accessor method. In other words, you are simply using the default modifier in this case.

So, this kind of visibility is slightly more open than private visibility and other classes in the same package can access a class member. So, in this case, the intended classes need to be placed inside the same package. At the same time, you need to accept that all other classes inside the same package will have the direct access to this state. So, you need to be careful enough when you place the classes in your special package.

9. **I am confused. To support undo operations, which pattern should I prefer—memento or command?**

   The GoF told us that these are related patterns. It primarily depends on how you want to handle the situation. For example, suppose that you are adding 10 to an integer. After this addition, you want to undo the operation by doing the reverse operation (i.e., 50 + 10 = 60, so to go back, you do 60 –10 = 50). In this type of operation, we do not need to store the previous state.

   But consider a situation where you need to store the state of your objects prior to the operations. In this case, memento is your savior. So, in a paint application, you can avoid the cost of undoing a paint operation. You can store the list of objects prior to executing the commands. This stored list can be treated as a memento in this case. You can keep this list with the associated commands. I suggest that you read the nice online article at www.developer.com/design/article.php/3720566/Working-With-Design-Patterns-Memento.htm.

   So, an application can use both patterns to support undo operations.

   Finally, you must remember that storing a memento object is mandatory in a memento pattern, so that you can roll back to a previous state; but in a command pattern, it is not necessary to store the commands. Once you execute a command, its job is done. If you do not support “undo” operations, you may not be interested in storing these commands at all.

10. **You talked about deep copy after the first implementation. Why do I need that?**

    In Chapter 2 (the prototype pattern), I discussed shallow copy and deep copy. You can refer to this discussion for your reference. To answer your question, let's analyze what is special about deep copy with a simple example. Consider the following example.

Shallow Copy vs. Deep Copy in Java

You clone with the `clone()` method in Java, but at the same time, you need to implement the Cloneable interface (which is a marker interface) because the Java objects that implement this Cloneable interface are only eligible for cloning. The default version of `clone()`creates a shallow copy. To create the deep copy, you need to override the `clone()` method.

Key Characteristics of the Following Program

In the following example, you have two classes: Employee and EmpAddress.

- The Employee class has three fields: id, name, and EmpAddress. So, you may notice that to form an Employee object, you need to pass an EmpAddress object also. So, in the following example, you will notice the line of code:

  ```
  Employee emp=new Employee(1,"John",initialAddress);
  ```

- EmpAddress has only a field called address, which is a String datatype.

- In the client code, you create an Employee object emp and then you create another object, empClone, through cloning. So, you will notice the line of code as follows:

  ```
  Employee empClone=(Employee)emp.clone();
  ```

- Then you change the field values of the emp object. But as a side effect of this change, the address of empClone object also changes, but this is not wanted.

Implementation

Here is the implementation.

```
package jdp2e.memento.questions_answers;
class EmpAddress implements Cloneable
{
    String address;
    public EmpAddress(String address)
    {
        this.address=address;
    }
    public String getAddress()
    {
        return address;
    }
    public void setAddress(String address)
    {
        this.address = address;
    }
    @Override
    public String toString()
    {
        return  this.address;
    }
    @Override
    public Object clone() throws CloneNotSupportedException
    {
        //Shallow Copy
        return super.clone();
    }
}
class Employee implements Cloneable
{
    int id;
    String name;
```

```
EmpAddress empAddress;
public Employee(int id,String name,EmpAddress empAddress)
{
    this.id=id;
    this.name=name;
    this.empAddress=empAddress;
}
public int getId()
{
    return id;
}
public void setId(int id)
{
    this.id = id;
}
public String getName()
{
    return name;
}
public void setName(String name)
{
    this.name = name;
}
public EmpAddress getAddress()
{
    return this.empAddress;
}
public void setAddress(EmpAddress newAddress)
{
    this.empAddress=newAddress;
}
@Override
public String toString()
{
```

```
        return "EmpId=" +this.id+ " EmpName="+ this.name+ "
        EmpAddressName="+ this.empAddress;
    }
    @Override
    public Object clone() throws CloneNotSupportedException
    {
        //Shallow Copy
        return super.clone();
    }
}

public class ShallowVsDeepCopy {

    public static void main(String[] args) throws
    CloneNotSupportedException  {
        System.out.println("***Shallow vs Deep Copy Demo***\n");
        EmpAddress initialAddress=new EmpAddress("21, abc Road, USA");
        Employee emp=new Employee(1,"John",initialAddress);
        System.out.println("emp1 object is as follows:");
        System.out.println(emp);
        Employee empClone=(Employee)emp.clone();
        System.out.println("empClone object is as follows:");
        System.out.println(empClone);
        System.out.println("\n Now changing the name, id and address of the
        emp object ");
        emp.setId(10);
        emp.setName("Sam");
        emp.empAddress.setAddress("221, xyz Road, Canada");
        System.out.println("Now emp1 object is as follows:");
        System.out.println(emp);
        System.out.println("And emp1Clone object is as follows:");
        System.out.println(empClone);
    }

}
```

Output

Here is the output.

```
***Shallow vs Deep Copy Demo***

emp1 object is as follows:
EmpId=1 EmpName=John EmpAddressName=21, abc Road, USA
empClone object is as follows:
EmpId=1 EmpName=John EmpAddressName=21, abc Road, USA

 Now changing the name and id of emp object
Now emp1 object is as follows:
EmpId=10 EmpName=Sam EmpAddressName=221, xyz Road, Canada
And emp1Clone object is as follows:
EmpId=1 EmpName=John EmpAddressName=221, xyz Road, Canada
```

Analysis

Notice the last line of the output. You see an unwanted side effect. The address of the cloned object is modified due the modification to the emp object. This is because the original object and the cloned object both point to the same address, and they are not 100% disjoined. Figure 19-3 depicts the scenario.

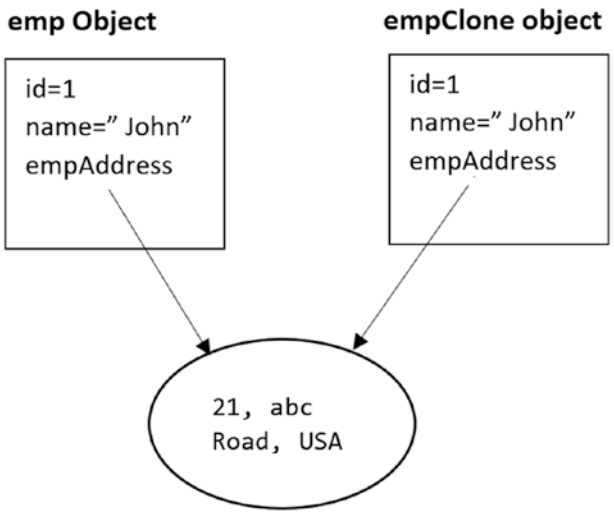


Figure 19-3. *Shallow copy*

So, now let's experiment with a deep copy implementation. Let's modify the clone method of the Employee class as follows.

```
@Override
    public Object clone() throws CloneNotSupportedException
    {
        //Shallow Copy
        //return super.clone();

        //For deep copy
        Employee employee = (Employee)  super.clone();
        employee.empAddress = (EmpAddress) empAddress.clone();
        return employee;
    }
```

Modified Output

Here is the modified output.

```
***Shallow vs Deep Copy Demo***

emp1 object is as follows:
EmpId=1 EmpName=John EmpAddressName=21, abc Road, USA
empClone object is as follows:
EmpId=1 EmpName=John EmpAddressName=21, abc Road, USA

 Now changing the name, id and address of the emp object
Now emp1 object is as follows:
EmpId=10 EmpName=Sam EmpAddressName=221, xyz Road, Canada
And emp1Clone object is as follows:
EmpId=1 EmpName=John EmpAddressName=21, abc Road, USA
```

Analysis

Notice the last line of the output. Now you do not see the unwanted side effect due to the modification to the emp object. This is because the original object and the cloned object are totally different and independent of each other. Figure 19-4 depicts the scenario.

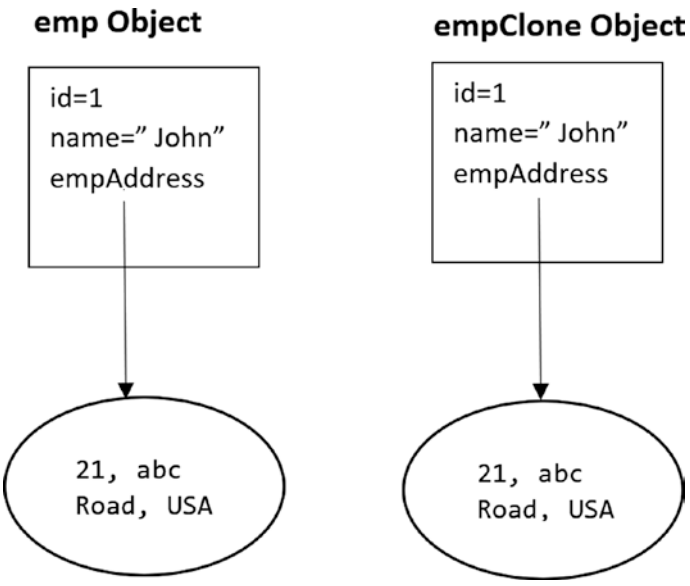


Figure 19-4. *Deep copy*

Note You saw the theoretical parts of a shallow copy and a deep copy in the "Q&A Session" of Chapter 2.

CHAPTER 20

State Pattern

This chapter covers the state pattern.

GoF Definition

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Concept

Suppose that you are dealing with a large-scale application where the codebase is rapidly growing. As a result, the situation becomes complex and you may need to introduce lots of `if-else` blocks/switch statements to guard the various conditions. The state pattern fits in such a context. It allows your objects to behave differently based on the current state, and you can define state-specific behaviors with different classes.

So, in this pattern, you start thinking in terms of possible states of your application, and you segregate the code accordingly. Ideally, each of the states is independent of other states. You keep track of these states, and your code responds as per the behavior of the current state. For example, suppose that you are watching a television (TV) program/show. If you press the mute button on the TV's remote control, you notice a state change in your TV. But you cannot notice any change if the TV is already turned off.

So, the basic idea is that if your code can track the current state of the application, you can centralize the task, segregate your code, and respond accordingly.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_20

Real-World Example

Consider the scenario of a network connection—a TCP connection. An object can be in various states; for example, a connection might already be established, the connection might be closed, or the object already started listening through the connection. When this connection receives a request from other objects, it responds as per its present state.

The functionalities of a traffic signal or a television (TV) can also be considered in this category. For example, you can change channels if the TV is already in a switched-on mode. It will not respond to the channel change requests if it is in a switched-off mode.

Computer-World Example

Suppose that you have a job-processing system that can process a certain number of jobs at a time. When a new job appears, either the system processes the job, or it signals that the system is busy with the maximum number of jobs that it can process at one time. In other words, the system sends a busy signal when its total number of job-processing capabilities has been reached.

Note In the javax.faces.lifecycle package, there is class called Lifecycle. This class has a method called *execute(FacesContext context),* in which you may notice an implementation of the state design pattern. FacesServlet can invoke the execute method of a LifeCycle and a LifeCycle object communicates with different phases (states).

Illustration

The following implementation models the functionalities of a television and its remote control. Suppose that you have a remote control to support the operations of a TV. You can simply assume that at any given time, the TV is in either of these three states: On, Off, or Mute. Initially, the TV is in the Off state. When you press the On button on the remote control, the TV goes into the On state. If you press the Mute button, it goes into the Mute state.

You can assume that if you press the Off button when the TV is already in the Off state, or if you press the On button when the TV is already in the On state, or if you press the Mute button when the TV is already in Mute mode, there is no state change for the TV.

The TV can go into the Off state from the On state if you press the Off button, or it goes into a Mute state if you press the Mute button. Figure 20-1 shows the state diagram that reflects all of these possible scenarios.

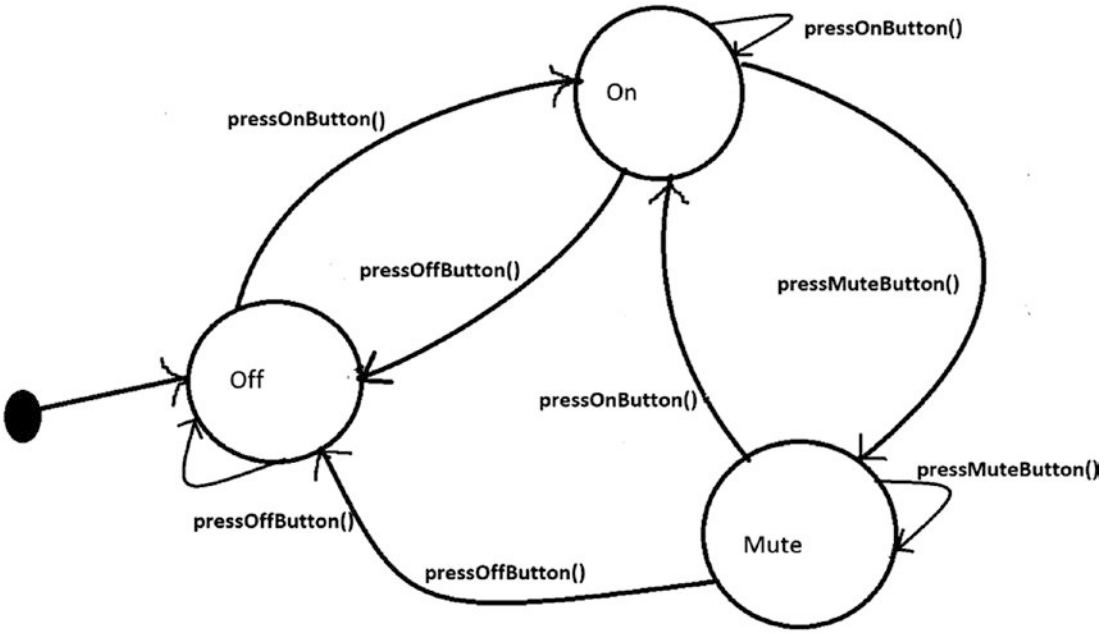


Figure 20-1. *Different states of a TV*

Note In this diagram, I have not marked any state as the final state, though in this illustration, at the end, I am switching off the TV. To make the design simple, I assume that if you press the Off button when the TV is already in the Off state, or if you press the On button when the TV is already in On state, or if you press the Mute button when the TV is already in Mute mode, there will be no state change to the TV. But in the real-world, a remote control may work differently. For example, if the TV is currently in the On state and you press the Mute button, the TV can go to Mute mode, and then if press Mute button again, the TV may come back to the On state again. So, you may need to put additional logic to support this behavior.

Key Characteristics

The key characteristics of the following implementations are as follows.

- For a state-specific behavior, you have separate classes. For example, here you have classes like On, Off, and Mute.
- The TV class is the main class here (the word *main* does not mean that it includes the `main()` method) and the client code only talks to it. In design pattern terms, TV is the context class here.
- Operations defined in the TV class are delegating the behaviour to the current state's object implementation.
- PossibleState is the interface that defines the methods/operations that are called when you own an object. On, Off, and Mute are concrete states that implement this interface.
- States are triggering state transitions (one state to another state) themselves.

Class Diagram

Figure 20-2 shows the class diagram.

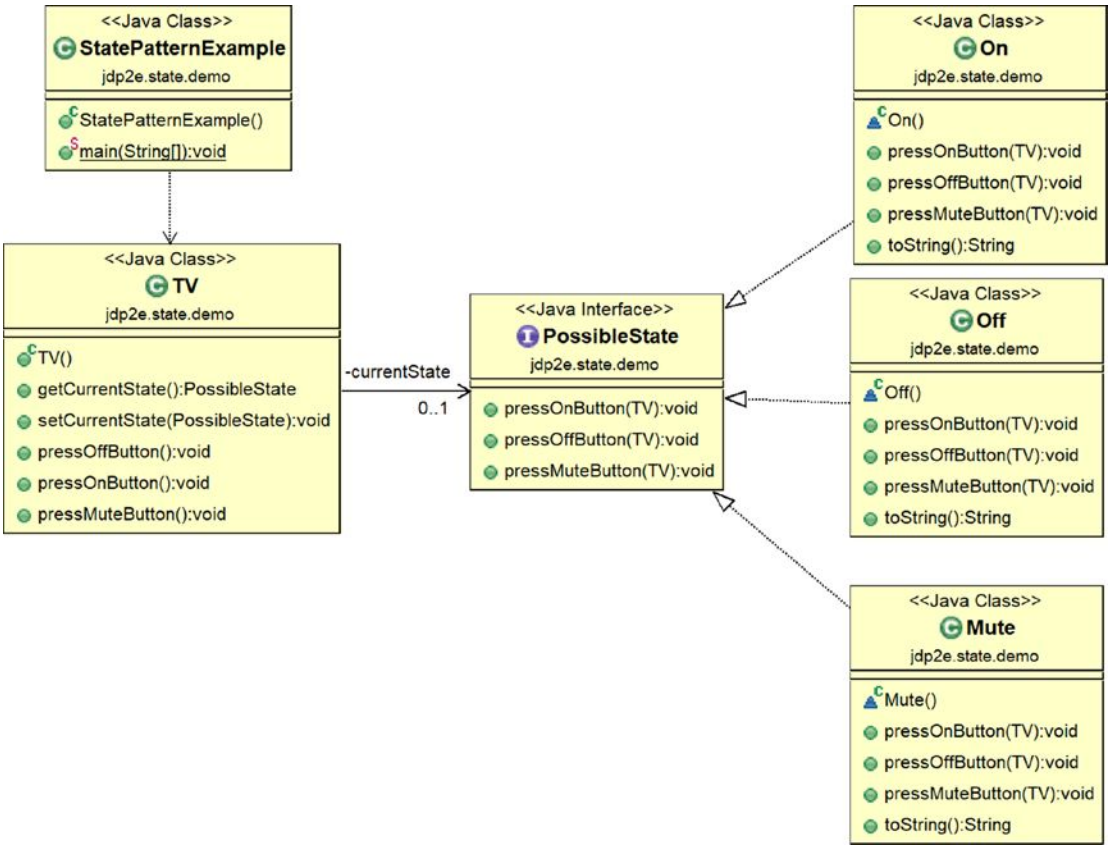


Figure 20-2. *Class diagram*

Package Explorer View

Figure 20-3 shows the high-level structure of the program.

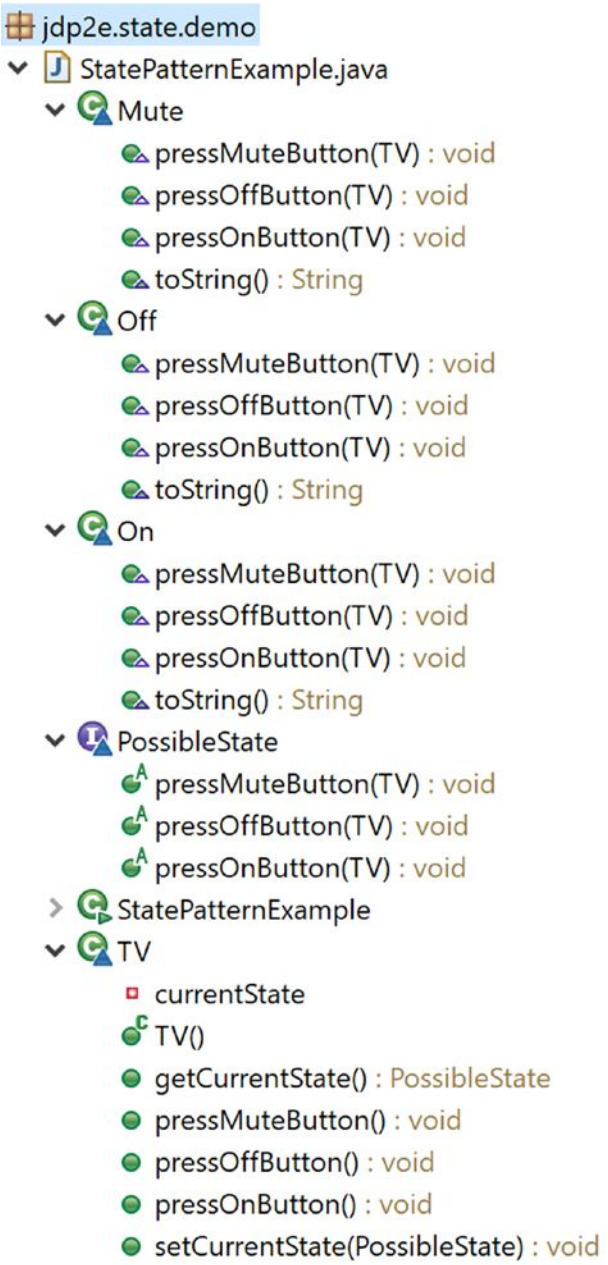


Figure 20-3. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.state.demo;
interface PossibleState
{
    void pressOnButton(TV context);
    void pressOffButton(TV context);
    void pressMuteButton(TV context);
}
//Off state
class Off implements  PossibleState
{
    //User is pressing Off button when the TV is in Off state
    @Override
    public void pressOnButton(TV context)
    {
        System.out.println("You pressed On button. Going from Off to On
        state.");
        context.setCurrentState(new On());
        System.out.println(context.getCurrentState().toString());
    }
    //TV is Off already, user is pressing Off button again
    @Override
    public void pressOffButton(TV context)
    {
        System.out.println("You pressed Off button. TV is already in Off
        state.");
    }
    //User is pressing Mute button when the TV is in Off state
    @Override
    public void pressMuteButton(TV context)
    {
        System.out.println("You pressed Mute button.TV is already in Off
        state, so Mute operation will not work.");
```

```
    }
    public String toString()
    {
        return "\t**TV is switched off now.**";
    }
}
//On state
class On implements PossibleState
{
    //TV is On already, user is pressing On button again
    @Override
    public void pressOnButton(TV context)
    {
        System.out.println("You pressed On button. TV is already in On
        state.");
    }
    //User is pressing Off button when the TV is in On state
    @Override
    public void pressOffButton(TV context)
    {
        System.out.println("You pressed Off button.Going from On to Off
        state.");
        context.setCurrentState(new Off());
        System.out.println(context.getCurrentState().toString());
    }
    //User is pressing Mute button when the TV is in On state
    @Override
    public void pressMuteButton(TV context)
    {
        System.out.println("You pressed Mute button.Going from On to Mute
        mode.");
        context.setCurrentState(new Mute());
        System.out.println(context.getCurrentState().toString());
    }
```

```
    public String toString()
    {
        return "\t**TV is switched on now.**";
    }
}
//Mute state
class Mute implements PossibleState
{
    //User is pressing On button when the TV is in Mute mode
    @Override
    public void pressOnButton(TV context)
    {
        System.out.println("You pressed On button.Going from Mute mode to
        On state.");
        context.setCurrentState(new On());
        System.out.println(context.getCurrentState().toString());
    }
    //User is pressing Off button when the TV is in Mute mode
    @Override
    public void pressOffButton(TV context)
    {
        System.out.println("You pressed Off button. Going from Mute mode to
        Off state.");
        context.setCurrentState(new Off());
        System.out.println(context.getCurrentState().toString());
    }
    //TV is in mute mode already, user is pressing mute button again
    @Override
    public void pressMuteButton(TV context)
    {
        System.out.println("You pressed Mute button.TV is already in Mute
        mode.");
    }
```

```
    public String toString()
    {
        return "\t**TV is silent(mute) now**";
    }
}
class TV
{
    private PossibleState currentState;
    public TV()
    {
        //Initially TV is initialized with Off state
        this.setCurrentState(new Off());
    }
    public PossibleState getCurrentState()
    {
        return currentState;
    }
    public void setCurrentState(PossibleState currentState)
    {
        this.currentState = currentState;
    }
    public void pressOffButton()
    {
        currentState.pressOffButton(this);//Delegating the state
    }
    public void pressOnButton()
    {
        currentState.pressOnButton(this);//Delegating the state
    }
    public void pressMuteButton()
    {
        currentState.pressMuteButton(this);//Delegating the state
    }
}
```

```
//Client
public class StatePatternExample {

    public static void main(String[] args) {
        System.out.println("***State Pattern Demo***\n");
        //Initially TV is Off.
        TV tv = new TV();
        System.out.println("User is pressing buttons in the following
        sequence:");
        System.out.println("Off->Mute->On->On->Mute->Mute->Off\n");
        //TV is already in Off state.Again Off button is pressed.
        tv.pressOffButton();
        //TV is already in Off state.Again Mute button is pressed.
        tv.pressMuteButton();
        //Making the TV on
        tv.pressOnButton();
        //TV is already in On state.Again On button is pressed.
        tv.pressOnButton();
        //Putting the TV in Mute mode
        tv.pressMuteButton();
        //TV is already in Mute mode.Again Mute button is pressed.
        tv.pressMuteButton();
        //Making the TV off
        tv.pressOffButton();
    }

}
```

Output

Here's the output.

```
***State Pattern Demo***
```

User is pressing buttons in the following sequence:
Off->Mute->On->On->Mute->Mute->Off

```
You pressed Off button. TV is already in Off state.
```

```
You pressed Mute button.TV is already in Off state, so Mute operation will
not work.
You pressed On button. Going from Off to On state.
    **TV is switched on now.**
You pressed On button. TV is already in On state.
You pressed Mute button.Going from On to Mute mode.
    **TV is silent(mute) now**
You pressed Mute button.TV is already in Mute mode.
You pressed Off button. Going from Mute mode to Off state.
    **TV is switched off now.**
```

Q&A Session

1. **Can you elaborate how this pattern is useful with another real-world scenario?**

   Psychologists repeatedly documented the fact that human beings can *perform their best when they are in a relaxed mode and they are free of tension* but in the reverse scenario, when their minds are filled with tension, they cannot produce great results. That is why psychologists always suggest that you should work in relaxed mood. You can relate this simple philosophy with the TV illustration. If the TV is on, it can entertain you; if it is off, it cannot. Right? *So, if you want to design similar kinds of behavior changes of an object when it's internal state changes, this pattern is useful.*

   Apart from this example, you can consider the scenario where a customer buys an online ticket and in some later phase he/she cancels it. The refund amount may vary with different conditions; for example, the number of days before you can cancel the ticket.

2. **In this example, you have considered only three states of a TV: On, Off, or Mute. There are many other states, for example, there may be a state that deals with connection issues or display conditions. Why have you ignored those?**

   The straightforward answer is to represent simplicity. If the number of states increases significantly in the system, then it becomes difficult to maintain the system (and it is one of the key challenges associated with this design pattern). But if you understand this implementation, you can easily add any states you want.

3. **I noticed that the GoF represented a similar structure for both the state pattern and the strategy pattern in their famous book. I am confused to see that.**

   Yes, the structures are similar, but you need to note that the intents are different. Apart from this key distinction, you can simply think like this: with a strategy pattern provides a better alternative to subclassing. On the other hand, in a state design pattern, different types of behaviors can be encapsulated in a state object and the context is delegated to any of these states. When a context's internal states change, its behavior also changes.

   State patterns can also help us avoid lots of `if` conditions in some contexts. (Consider our example once again. If the TV is in the Off state, it cannot go to the Mute state. From this state, it can move to the On state only.) So, if you do not like state design pattern, you may need to code like this for a On button press.

```
class TV
{
//Some code before
public void pressOnButton()
{
if(currentState==Off )
{
```

```
System.out.println (" You pressed Onbutton. Going from Off to
OnState");
//Do some operations
}
if(currentState==On )
 {
  System.out.println (" You pressed On button. TV is already  in
  On state");
 }
//TV presently is in mute mode
else
 {
  System.out.println (" You pressed On button . Going from Mute
  mode to On State");
 }
//Do some operations
}
```

Notice that you need to repeat these checks for different kinds of button presses. (For example, for the `pressOffButton()` and `pressMuteButton()` methods, you need to repeat these checks and perform accordingly.)

If you do not think in terms of states, if your code base grows, maintenance becomes difficult.

4. **How are you supporting the open-close principle in our implementation?**

   Each of these TV states are closed for modification, but you can add brand-new states to the TV class.

5. **What are the common characteristics between the strategy pattern and the state pattern?**

   Both can promote composition and delegation.

6. **It appears to me that these state objects are acting like singletons. Is this correct?**

   Yes. Most times they act in this way.

7. **Can you avoid the use of "contexts" in the method parameters. For example, can you avoid them in the following statements?**

   ```
   void pressOnButton(TV context);

   ....
   ```

   If you do not want to use the context parameter like this, you may need to modify the implementation. To give a quick overview, I am presenting the modified Package Explorer view with a modified implementation only.

   One of the key changes in the following implementation can be seen in the class TV. The `TV()` constructor is initialized with all possible state objects, which are used for the change of states in later phases. The getter methods are invoked for this purpose. Consider the following implementation.

Modified Package Explorer View

In this case, all three possible states have similar components. So, to keep the diagram short, I am showing only one of them in the following Package Explorer view.

Figure 20-4 shows the modified high-level structure of the program.

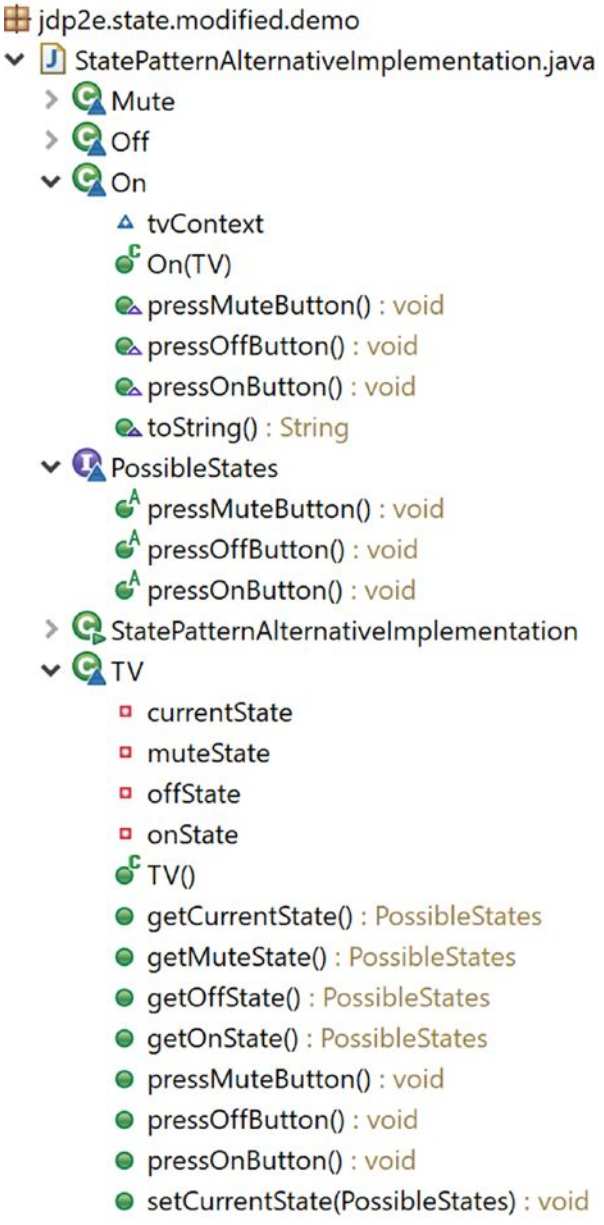


Figure 20-4. *Modified Package Explorer View*

Modified Implementation

Here is the modified implementation.

```
package jdp2e.state.modified.demo;

interface PossibleStates
{
    void pressOnButton();
    void pressOffButton();
    void pressMuteButton();
}

class Off implements  PossibleStates
{
    TV tvContext;
    //Initially we'll start from Off state
    public Off(TV context)
    {
        //System.out.println(" TV is Off now.");
        this.tvContext = context;
    }
    //Users can press any of these buttons at this state-On, Off or Mute
    //TV is Off now, user is pressing On button
    @Override
    public void pressOnButton()
    {
        System.out.println(" You pressed On button. Going from Off to On
        state");
        tvContext.setCurrentState(tvContext.getOnState());
        System.out.println(tvContext.getCurrentState().toString());
    }
    //TV is Off already, user is pressing Off button again
    @Override
```

```
    public void pressOffButton()
    {
        System.out.println(" You pressed Off button. TV is already in Off
        state");
    }
    //TV is Off now, user is pressing Mute button
    @Override
    public void pressMuteButton()
    {
        System.out.println(" You pressed Mute button.TV is already in Off
        state, so Mute operation will not work.");
    }
    public String toString()
    {
        return "\t**TV is switched off now.**";
    }

}
class On implements PossibleStates
{
    TV tvContext;
    public On(TV context)
    {
        //System.out.println(" TV is On now.");
        this.tvContext = context;
    }
    //Users can press any of these buttons at this state-On, Off or Mute
    //TV is On already, user is pressing On button again
    @Override
    public void pressOnButton()
    {
        System.out.println("You pressed On button. TV is already in On
        state.");
    }
    //TV is On now, user is pressing Off button
    @Override
```

```
    public void pressOffButton()
    {
        System.out.println(" You pressed Off button.Going from On to Off
        state.");
        tvContext.setCurrentState(tvContext.getOffState());
        System.out.println(tvContext.getCurrentState().toString());
    }
    //TV is On now, user is pressing Mute button
    @Override
    public void pressMuteButton()
    {
        System.out.println("You pressed Mute button.Going from On to Mute
        mode.");
        tvContext.setCurrentState(tvContext.getMuteState());
        System.out.println(tvContext.getCurrentState().toString());
    }
    public String toString()
    {
        return "\t**TV is switched on now.**";
    }
}
class Mute implements PossibleStates
{
    TV tvContext;
    public Mute(TV context)
    {
        this.tvContext = context;
    }
    //Users can press any of these buttons at this state-On, Off or Mute
    //TV is in mute, user is pressing On button
    @Override
    public void pressOnButton()
    {
        System.out.println("You pressed On button.Going from Mute mode to
        On state.");
```

```
        tvContext.setCurrentState(tvContext.getOnState());
        System.out.println(tvContext.getCurrentState().toString());
    }
    //TV is in mute, user is pressing Off button
    @Override
    public void pressOffButton()
    {
        System.out.println("You pressed Off button. Going from Mute mode to
        Off state.");
        tvContext.setCurrentState(tvContext.getOffState());
        System.out.println(tvContext.getCurrentState().toString());
    }
    //TV is in mute already, user is pressing mute button again
    @Override
    public void pressMuteButton()
    {
        System.out.println(" You pressed Mute button.TV is already in Mute
        mode.");
    }
    public String toString()
    {
        return "\t**TV is silent(mute) now**";
    }
}
class TV
{
    private PossibleStates currentState;
    private PossibleStates onState;
    private PossibleStates offState;
    private PossibleStates muteState;
    public TV()
    {
        onState=new On(this);
        offState=new Off(this);
        muteState=new Mute(this);
        setCurrentState(offState);
```

```
    }
    public PossibleStates getCurrentState()
    {
        return currentState;
    }
    public void setCurrentState(PossibleStates currentState)
    {
        this.currentState = currentState;
    }
    public void pressOffButton()
    {
        currentState.pressOffButton();
    }
    public void pressOnButton()
    {
        currentState.pressOnButton();
    }
    public void pressMuteButton()
    {
        currentState.pressMuteButton();
    }
    public PossibleStates getOnState()
    {
        return onState;
    }
    public PossibleStates getOffState()
    {
        return offState;
    }
    public PossibleStates getMuteState()
    {
        return muteState;
    }
}
```

```
//Client
public class StatePatternAlternativeImplementation {

    public static void main(String[] args) {
        System.out.println("***State Pattern Alternative Implementation
        Demo***\n");
        //Initially TV is Off.
        TV tv = new TV();
        System.out.println("User is pressing buttons in the following
        sequence:");
        System.out.println("Off->Mute->On->On->Mute->Mute->Off\n");
        //TV is already in Off state.Again Off button is pressed.
        tv.pressOffButton();
        //TV is already in Off state.Again Mute button is pressed.
        tv.pressMuteButton();
        //Making the TV on
        tv.pressOnButton();
        //TV is already in On state.Again On button is pressed.
        tv.pressOnButton();
        //Putting the TV in Mute mode
        tv.pressMuteButton();
        //TV is already in Mute mode.Again Mute button is pressed.
        tv.pressMuteButton();
        //Making the TV off
        tv.pressOffButton();

    }
}
```

Modified Output

Here is the output from the modified implementation.

```
 ***State Pattern Alternative Implementation Demo***

User is pressing buttons in the following sequence:
Off->Mute->On->On->Mute->Mute->Off
```

```
 You pressed Off button. TV is already in Off state
 You pressed Mute button.TV is already in Off state, so Mute operation will
 not work.
 You pressed On button. Going from Off to On state
    **TV is switched on now.**
You pressed On button. TV is already in On state.
You pressed Mute button.Going from On to Mute mode.
    **TV is silent(mute) now**
 You pressed Mute button.TV is already in Mute mode.
You pressed Off button. Going from Mute mode to Off state.
    **TV is switched off now.**
```

8. **In these implementations, TV is a concrete class. Why are you not programming to interface in this case?**

   I assume that the TV class is not going to change, and so I ignored that part to reduce some code size of the program. But yes, you can always start from an interface in which you can define the contracts.

9. **What are the pros and cons of a state design pattern?**

   Pros

   - You have already seen that following the open/close principle, you can easily add new states and new behaviors. Also, a state behavior can be extended without hassle. For example, in this implementation, you can add a new state and a new behavior for a TV class without changing the TV class itself.
   - Reduces the use of `if-else` statements (i.e., conditional complexity is reduced. (Refer to the answer in question 3).

Cons

- The state pattern is also known as *objects for states*. So, you can assume that more states need more codes, and the obvious side effect is difficult maintenance for you.

10. **In the TV class constructor, you are initializing the TV with an Off state. So, can both the states and the context class trigger the state transitions?**

    Yes.

CHAPTER 21

Mediator Pattern

This chapter covers the mediator pattern.

GoF Definition

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Concept

A mediator takes the responsibility of controlling and coordinating the interactions of a specific group of objects that cannot refer to each other explicitly. So, you can imagine a mediator as an intermediary through whom these objects talk to each other. This kind of implementation helps reduce the number of interconnections among different objects. As a result, you can promote loose coupling in the system.

So, in this design, object communications are encapsulated with a mediator object so that they cannot communicate directly with each other and you reduce the dependencies among them.

Real-World Example

When a flight needs to take off, a series of verifications takes place. These kinds of verifications conform that all components/parts (which are dependent on each other) are in perfect condition.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_21

Also consider when airplane pilots (approaching or departing the terminal area) communicate with the towers. They do not explicitly communicate with other pilots from different airlines. They only send their status to the tower. These towers also send the signals to conform which airplane can take-off or land. You must note that these towers do not control the whole flight. They only enforce constraints in the terminal areas.

Computer-World Example

When a client processes a business application, the developer may need to put some constraints on it. For example, a form in which a client needs to supply a user ID and password to access their account. On the same form, the client must supply other information, such as email, address, age, and so forth. Let's assume that the developer applied the constraints as follows.

Initially, the application checks whether the ID supplied by the user is valid or not. If it is a valid user ID, then only the password field is enabled. After supplying these two fields, the application form needs to check whether the email address was provided by the user. Let's further assume that after providing all of this information (a valid user ID, password, a correctly formatted email, etc.), the Submit button is enabled. So, basically the Submit button is enabled if the client supplies a valid user ID, password, email, and other mandatory details in the correct order. The developer may also enforce that the user ID must be an integer, so if the user mistakenly places any characters in that field, the Submit button stays in disabled mode. The mediator pattern becomes handy in such a scenario.

So, when a program consists of many classes and the logic is distributed among them, the code becomes harder to read and maintain. In those scenarios, if you want to bring new changes in the system's behavior, it can be difficult unless you use the mediator pattern.

Note The execute() method inside the java.util.concurrent.Executor interface follows this pattern.

The javax.swing.ButtonGroup class is another example that supports this pattern. This class has a method setSelected() that ensures that the user provides a new selection.

The different overloaded versions of various schedule() methods of the java.util.Timer class also can be considered to follow this pattern.

Illustration

A common structure of the mediator pattern (which is basically adopted from the GoF's *Design Patterns: Elements of Reusable Object-Oriented Software*) is often described with the diagram shown in Figure 21-1.

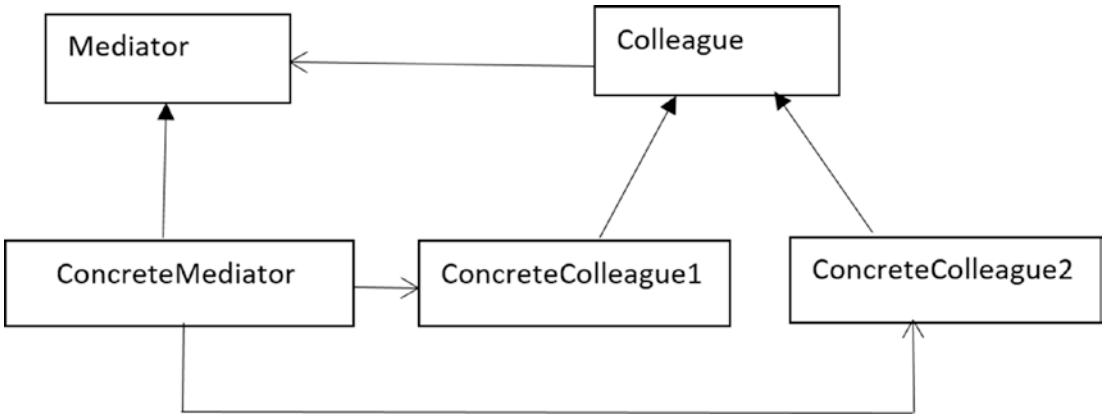


Figure 21-1. *Mediator pattern example*

The participants are described as follows.

- *Mediator*: Defines the interface to provide the communication among Colleague objects.
- *ConcreteMediator*: Maintains the list of the Colleague objects. It implements the Mediator interface and coordinates the communication among the Colleague objects.
- *Colleague*: Defines the interface for communication with other Colleagues.
- *ConcreteColleague1 and ConcreteColleague2*: Implements the Colleague interface. These objects communicate with each other through the mediator.

In this chapter, I provide two implementations of this pattern. In the first implementation, I replaced the word *Colleague* with *Employee*. Also, ConcreteColleague1 and ConcreteColleague2 are replaced with JuniorEmployee and SeniorEmployee, respectively. Let's assume that you have three employees: Amit, Sohel, and Raghu, where Amit and Sohel are junior employees who report to their boss, Raghu, who is a senior

employee. Raghu wants to smoothly coordinate things. Let's further assume that they can communicate with each other through a chat server.

In the following implementation, Mediator is an interface that has two methods: `register()` and `sendMessage()`. The `register()` method registers an employee with the mediator and `sendMessage()` posts messages to the server. The ConcreteMediator class is the concrete implementation of the Mediator interface.

Employee is an abstract class and the JuniorEmployee and SeniorEmployee classes are the concrete implementations of it. The `sendMessage()` method of the Employee class is described as follows.

```
public void sendMessage(String msg) throws InterruptedException
{
    mediator.sendMessage(this, msg);
}
```

You can see that when an employee invokes the `sendMessage()` method, it is invoking mediator's `sendMessage()` method. So, the actual communication process is conducted through the mediator.

In the client code, I introduced another person, Jack. But he did not register himself with the mediator object. So, the mediator is not allowing him to post any messages to this server.

Now go through the code and the corresponding output.

Class Diagram

Figure 21-2 shows the class diagram.

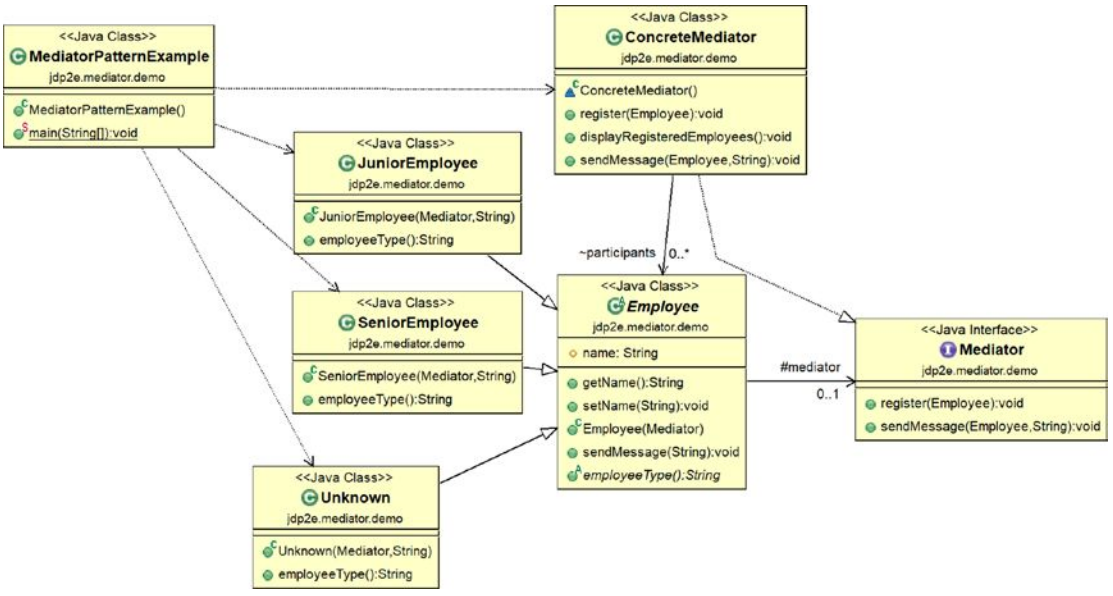


Figure 21-2. *Class diagram*

Package Explorer View

Figure 21-3 shows the high-level structure of the program.

Figure 21-3. *Package Explorer view*

Implementation

Here's the first implementation.

```
package jdp2e.mediator.demo;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

interface Mediator
{
    void register(Employee employee);
    void sendMessage(Employee employee, String msg) throws
    InterruptedException;
}
// ConcreteMediator
class ConcreteMediator implements Mediator
{
    List<Employee> participants = new ArrayList<Employee>();
    @Override
    public void register(Employee employee)
    {
        participants.add(employee);
    }
    public void displayRegisteredEmployees()
    {
        System.out.println("At present,registered employees are:");
        for (Employee employee: participants)
        {

            System.out.println(employee.getName());
        }
    }
    @Override
    public void sendMessage(Employee employee, String msg) throws
    InterruptedException
    {
```

```
        if (participants.contains(employee))
        {
            System.out.println(employee.getName() +" posts:"+ msg+"Last
            message posted at "+LocalDateTime.now());
            Thread.sleep(1000);
        }
        else
        {
            System.out.println("An outsider named "+ employee.getName()+"
            is trying to send some messages.");
        }
    }
}
// The abstract class-Employee
abstract class Employee
{
    protected Mediator mediator;
    protected String name;
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    // Constructor
    public Employee(Mediator mediator)
    {
        this.mediator = mediator;
    }
    public void sendMessage(String msg) throws InterruptedException
    {
        mediator.sendMessage(this, msg);
    }
    public abstract String employeeType();
}
```

```
// Junior Employee
class JuniorEmployee extends Employee
{
    public JuniorEmployee(Mediator mediator, String name)
    {
        super(mediator);
        this.name = name;
    }

    @Override
    public String employeeType()
    {
        return "JuniorEmployee";
    }
}

//Senior Employee
class SeniorEmployee extends Employee
{
    // Constructor
    public SeniorEmployee(Mediator mediator, String name)
    {
        super(mediator);
        this.name = name;
    }
    @Override
    public String employeeType()
    {
        return "SeniorEmployee";
    }
}
// Unknown participant.
class Unknown extends Employee
{
    // Constructor
    public Unknown(Mediator mediator, String name)
```

```
    {
        super(mediator);
        this.name = name;
    }
    @Override
    public String employeeType()
    {
        return "Outsider";
    }
}

public class MediatorPatternExample {

    public static void main(String[] args) throws InterruptedException {
        System.out.println("***Mediator Pattern Demo***\n");

        ConcreteMediator mediator = new ConcreteMediator();

        JuniorEmployee amit = new JuniorEmployee(mediator, "Amit");
        JuniorEmployee sohel = new JuniorEmployee(mediator, "Sohel");
        SeniorEmployee raghu = new SeniorEmployee(mediator, "Raghu");

        //Registering participants
        mediator.register(amit);
        mediator.register(sohel);
        mediator.register(raghu);
        //Displaying the participant's list
        mediator.displayRegisteredEmployees();

        System.out.println("Communication starts among participants...");
        amit.sendMessage("Hi Sohel,can we discuss the mediator pattern?");
        sohel.sendMessage("Hi Amit,yup, we can discuss now.");
        raghu.sendMessage("Please get back to work quickly.");
        //An outsider/unknown person tries to participate
        Unknown unknown = new Unknown(mediator, "Jack");
        unknown.sendMessage("Hello Guys..");
    }
}
```

Output

Here's the output.

```
***Mediator Pattern Demo***

At present,registered employees are:
Amit
Sohel
Raghu
Communication starts among participants...
Amit posts:Hi Sohel,can we discuss the mediator pattern?Last message posted
at 2018-09-09T17:41:21.868
Sohel posts:Hi Amit,yup, we can discuss now.Last message posted at 2018-09-
09T17:41:23.369
Raghu posts:Please get back to work quickly.Last message posted at 2018-09-
09T17:41:24.870
An outsider named Jack is trying to send some messages.
```

Analysis

Note that only registered users can communicate with each other and successfully post messages on the chat server. The mediator does not allow any outsiders into the system. (Notice the last line of the output.)

Modified Illustration

You have just seen a simple example of the mediator pattern. But you can make it better. You identified the following points.

- The messages are only passing in one direction.
- When one participant posts a message, everyone can see the message. So, there is no privacy.

- If an employee forgets to register himself, he is not allowed to send a message. It is fine, but he should not be treated like an outsider. In a normal scenario, an organization outsider should be treated differently from an employee of the organization who forgets to register himself on the server.
- The client code needed to register the participants to the mediator. Though you may argue that it is not a drawback, you may opt for a better approach. For example, you may register the participants automatically to a mediator when you create an Employee object inside the client code.
- You have not used the `employeeType()` method in client code.

So, keeping these points in mind, let's modify the previous example. Here are some key characteristics of the modified implementation.

- The JuniorEmployee and SeniorEmployee classes are replaced with a single ConcreteEmployee class. It helps us easily identify who belongs to the organization and who does not (in other words, outsiders).
- In the modified implementation, each of these participants can see who is posting messages, but it is not disclosed to whom it is targeted or what the actual message is. So, there is privacy between two participants, but this approach can help someone like Raghu to coordinate things because he may interfere if he sees that employees are chatting too much.
- In the client code, you create participants like the following.

```
Employee Amit = new ConcreteEmployee(mediator, "Amit", true);
```

The third argument (true/false) is used to determine whether a participant wants to register himself or not to the mediator. He is treated accordingly when he tries to post messages.

- The `employeeType()` method determines whether a participant is from inside the organization or if he or she is an outsider. In this context, you may also note that instead of using the following line

```
if( fromEmployee.employeeType()=="UnauthorizedUser")
```

you could directly use this line of code:

```
if( fromEmployee.getClass().getSimpleName().equals("UnauthorizedUser"))
```

I used the former one for better readability.

Modified Class Diagram

Figure 21-4 shows the modified class diagram. To show the key changes and to present a neat diagram, I do not show the client code dependencies in the following diagram.

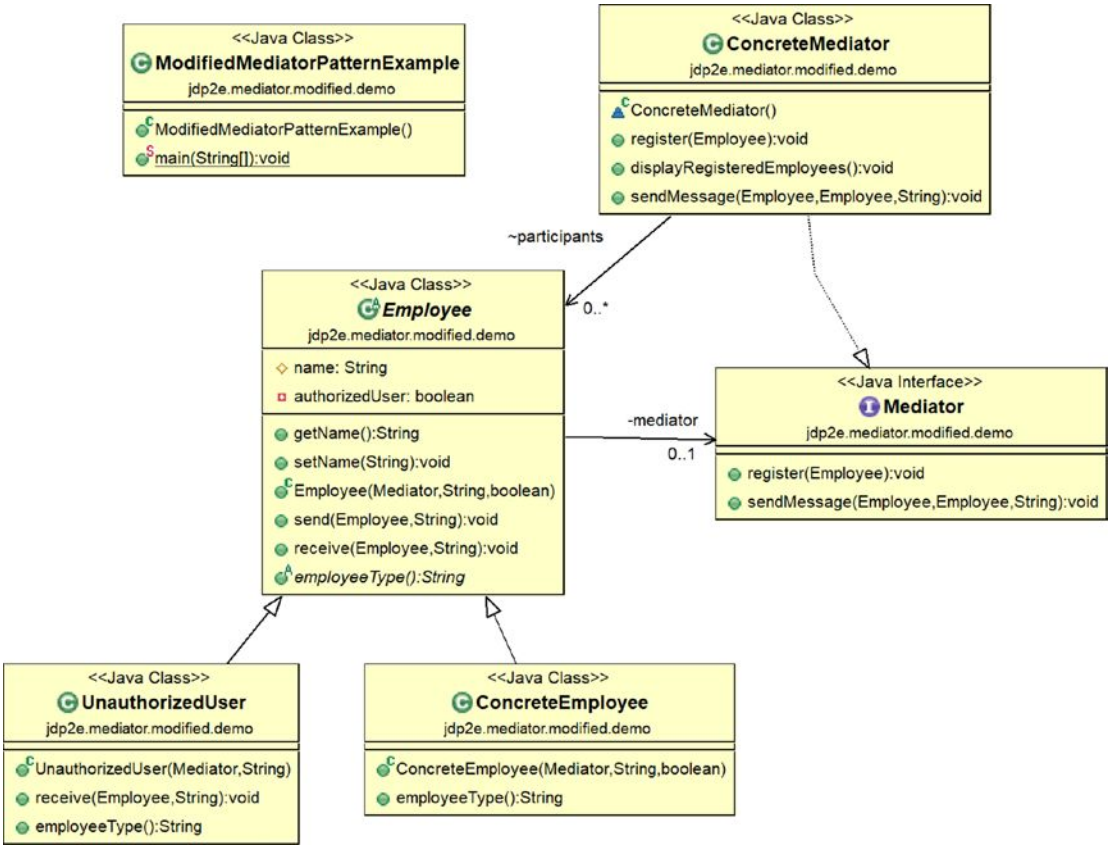


Figure 21-4. *Class diagram*

Modified Package Explorer View

Figure 21-5 shows the modified Package Explorer view.

Figure 21-5. *Modified Package Explorer view*

Modified Implementation

Here is the modified implementation.

```
package jdp2e.mediator.modified.demo;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

interface Mediator
{
    void register(Employee employee);
    void sendMessage(Employee fromEmployee, Employee toEmployee,String msg)
    throws InterruptedException;
}
// ConcreteMediator
class ConcreteMediator implements Mediator
{
    List<Employee> participants = new ArrayList<Employee>();
    @Override
    public void register(Employee employee)
    {
        participants.add(employee);
    }
    public void displayRegisteredEmployees()
    {
        System.out.println("At present ,registered participants are:");
        for (Employee employee: participants)
        {
            System.out.println(employee.getName());
        }
    }
    @Override
    public void sendMessage(Employee fromEmployee,Employee
    toEmployee,String msg) throws InterruptedException
    {
```

```
        /*if( fromEmployee.getClass().getSimpleName().
        equals("UnauthorizedUser"))*/
        if( fromEmployee.employeeType()=="UnauthorizedUser")
        {
            System.out.println("[ALERT Everyone] An outsider named "+
            fromEmployee.getName()+" trying to send some messages to "+
            toEmployee.getName());
            fromEmployee.receive(fromEmployee, ",you are not allowed to
            enter here.");
        }
        else if (participants.contains(fromEmployee))
        {
            System.out.println("-----"+fromEmployee.getName() +" posts some
            message at: "+LocalDateTime.now()+"-----");
            Thread.sleep(1000);
            //No need to inform everyone or himself
            //Only let the target receiver know
            if(participants.contains(toEmployee))
            {
                toEmployee.receive(fromEmployee,msg);
            }
            //If target receipient does not exist
            else
            {
                System.out.println(fromEmployee.getName() +" , your target
                recipient does not exist");
            }
        }
        //An outsider tries to send message.
        else
        {
            System.out.println("[ALERT] An unregistered employee named "+
            fromEmployee.getName()+" trying to send some messages to "+
            toEmployee.getName());
```

```
            System.out.println(fromEmployee.getName()+", you need to
            register yourself first.");
        }
    }
}
// Employee
abstract class Employee
{
    private Mediator mediator;
    protected String name;
    private boolean authorizedUser;
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    // Constructor
    public Employee(Mediator mediator, String name, boolean authorizedUser)
    {
        this.mediator = mediator;
        this.name=name;
        this.authorizedUser=authorizedUser;
        if(authorizedUser)
        {
            mediator.register(this);
        }
    }
    //The following method name need not be same as the Mediator's method name
    public void send(Employee toFriend,String msg) throws
    InterruptedException
    {
        mediator.sendMessage(this,toFriend, msg);
    }
```

```
    //public abstract void receive(Friend fromFriend,String message);

    public void receive(Employee fromFriend,String message)
    {
        System.out.println(this.name+" received a message : " + message +"
        from an employee "+ fromFriend.getName() +".");

    }
    public abstract String employeeType();
}
//A concrete friend
class ConcreteEmployee extends Employee
{

    public ConcreteEmployee(Mediator mediator, String name,boolean
    authorizedUser)
    {
        super(mediator,name, authorizedUser);
    }
    @Override
    public String employeeType()
    {
        return "ConcreteEmployee";
    }
}
//Unauthorized user
class UnauthorizedUser extends Employee
{
    public UnauthorizedUser(Mediator mediator, String name)
    {
        //The user is always treated an unauthorized user.So, the flag is
        //false always.
        super(mediator,name, false);
    }

    @Override
    public void receive(Employee fromEmployee,String message)
```

```
    {
        System.out.println(this.name + message);

    }
    @Override
    public String employeeType()
    {
        return "UnauthorizedUser";
    }
}

public class ModifiedMediatorPatternExample {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("***Mediator Pattern Demo***\n");

        ConcreteMediator mediator = new ConcreteMediator();

        Employee Amit = new ConcreteEmployee(mediator, "Amit", true);
        Employee Sohel = new ConcreteEmployee(mediator, "Sohel",true);
        Employee Raghu = new ConcreteEmployee(mediator, "Raghu",true);
        //Unauthorized user
        Employee Jack = new ConcreteEmployee(mediator, "Jack",false);
        //Only two parameter needed to pass in the following case.
        Employee Divya = new UnauthorizedUser(mediator, "Divya");

        //Displaying the participant's list
        mediator.displayRegisteredEmployees();

        System.out.println("Communication starts among participants...");
        Amit.send(Sohel,"Hi Sohel,can we discuss the mediator pattern?");
        Sohel.send(Amit,"Hi Amit,Yup, we can discuss now.");
        //Boss is sending messages to each of them individually
        Raghu.send(Amit,"Please get back to work quickly.");
        Raghu.send(Sohel,"Please get back to work quickly.");
```

```
        //An unregistered employee(Jack) and an outsider(Divya) are also
        //trying to participate.
        Jack.send(Amit,"Hello Guys..");
        Divya.send(Raghu, "Hi Raghu");
    }
}
```

Modified Output

Here is the modified output.

```
***Mediator Pattern Demo***

At present ,registered participants are:
Amit
Sohel
Raghu
Communication starts among participants...
-----Amit posts some message at: 2018-09-04T20:37:00.999-----
Sohel received a message : Hi Sohel,can we discuss the mediator pattern?
from an employee Amit.
-----Sohel posts some message at: 2018-09-04T20:37:01.999-----
Amit received a message : Hi Amit,Yup, we can discuss now. from an employee
Sohel.
-----Raghu posts some message at: 2018-09-04T20:37:03.002-----
Amit received a message : Please get back to work quickly. from an employee
Raghu.
-----Raghu posts some message at: 2018-09-04T20:37:04.016-----
Sohel received a message : Please get back to work quickly. from an
employee Raghu.
[ALERT] An unregistered employee named Jack trying to send some messages to
Amit
Jack, you need to register yourself first.
[ALERT Everyone] An outsider named Divya trying to send some messages to
Raghu
Divya,you are not allowed to enter here.
```

Analysis

Notice that when the employee named Jack (who belongs to the organization) sends a message without registering himself, the system prevents him from posting the message but gives him a suggestion. But Divya, who is an organization outsider, is told that she is not allowed to enter into the system. It also warns others.

Q&A Session

1. **Why are you complicating the things? In the first example, each of the participants could talk to each other directly and you could bypass the use of mediator. Is this correct?**

   In this example, you have only three *registered* participants, so it may appear that they can communicate with each other directly. But you may need to consider a relatively complicated scenario. For example, a participant can send a message to a target participant only if the target participant is in online mode (which is the common scenario for a chat server). So, with your proposed architecture, if they try to communicate with each other, each of them needs to maintain the status of all other participants before sending a message. And if the number of participants keeps growing, can you imagine the complexity of the system?

   So, a mediator can certainly help you deal with a scenario like this. Figure 21-6 and Figure 21-7 depict the scenario.

Case 1. Communication without a mediator.

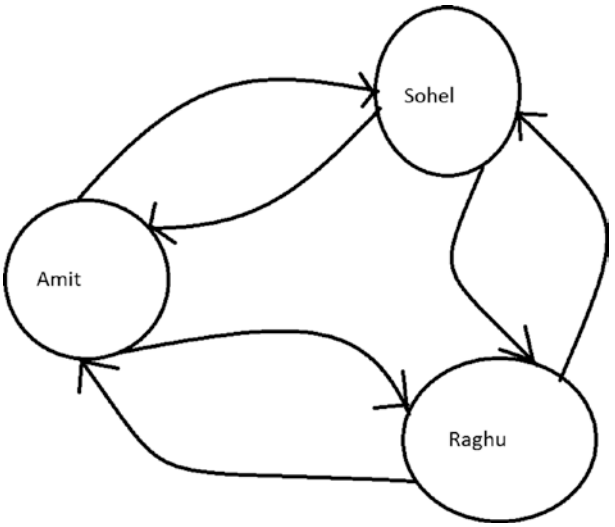


Figure 21-6. *Communication without using a mediator*

Case 2. Communication with a mediator.

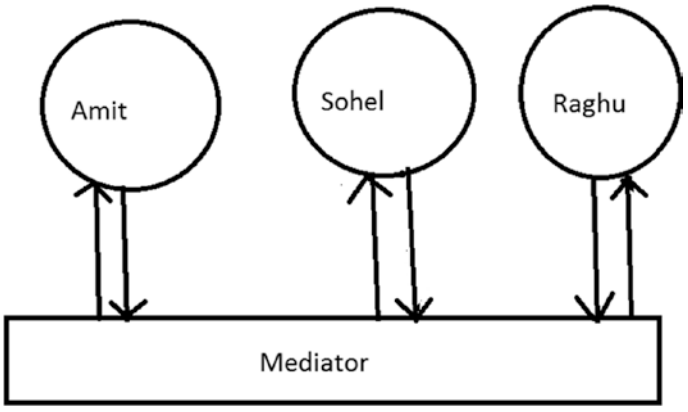


Figure 21-7. *Communication using a mediator*

Also, you can consider the modified implementation in this context. In the modified implementation, you can see that the mediator is maintaining the logic—who should be allowed to post messages on the server and how he/she should be treated.

2. **What are advantages of using mediator patterns?**

   - You can reduce the complexity of objects' communication in a system.
   - The pattern promotes loose coupling.
   - It reduces number of subclasses in the system.
   - You can replace "many-to-many" relationship with "one-to-many" relationships, so it is much easier to read and understand. (Consider our first illustration in this context). And as an obvious effect, maintenance becomes easy.
   - You can provide a centralized control through the mediator with this pattern.
   - In short, it is always our aim to remove tight coupling (among objects) from our code and this pattern scores high in this context.

3. **What are the disadvantages of using mediator patterns?**

   - In some cases, implementing the proper encapsulation is tricky.
   - The mediator object's architecture may become complex if you put too much logic inside it. An inappropriate use of the mediator pattern may end up with a "God Class" antipattern. (You'll learn about antipatterns in Chapter 28).
   - Sometimes maintaining the mediator becomes a big concern.

4. **If you need to add a new rule or logic, you can directly add it to the mediator. Is this correct?**

   Yes.

5. **I am finding some similarities in the facade pattern and the mediator pattern. Is this correct?**

   Yes. In his book *Design Pattern for Dummies* (Wiley Publishing, 2006), Steve Holzner mentions the similarity by describing the mediator pattern as a multiplexed facade pattern. In mediator, instead of working with an interface of a single object, you are making a multiplexed interface among multiple objects to provide smooth transitions.

6. **In this pattern, you are reducing the number of interconnections among various objects. What key benefits have you achieved due to this reduction?**

   More interconnections among objects can make a monolithic system where the system's behavior is difficult to change (the system's behavior is distributed among many objects). As a side effect, you may need to create many subclasses to bring those changes in the system.

7. **In the modified implementations, you are using Thread.Sleep(1000). What is the reason for this?**

   You can ignore that. I used it to mimic a real-life scenario. I assume that participants are posting messages after reading a message properly and this activity takes a minimum of 1 second.

8. **In some applications, I have seen the use of a concrete mediator only. Is this approach OK?**

   The mediator pattern does not restrict you to use only a concrete mediator. But I like to follow the experts' advice that says, "programming to the supertype (abstract class/interface) is a better approach," and it can provide more flexibility in the long run.

9. **Can I simply say that if a class simply calls methods from multiple objects, it is a mediator?**

   Not at all. The key purpose of a mediator is to simplify the complex communications among objects in a system. I suggest that you always keep in mind the GoF definition and the corresponding concepts.

10. **In the first implementation, both send methods (mediator and employees) are named sendMessage() but in the modified implementation, they are different—one is send() and the other is sendMessage(). Do I need to follow any specific naming convention?**

    No. Both are fine. It's your choice.

CHAPTER 22

Chain-of-Responsibility Pattern

This chapter covers the chain-of-responsibility pattern.

GoF Definition

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Concept

In this pattern, you form a chain of objects where each object in the chain handles a particular kind of request. If an object cannot handle the request fully, it passes the request to the next object in the chain. The same process may follow until the end of a chain is reached. This kind of request handling mechanism gives you the flexibility to add a new processing object (handler) at the end of the chain. Figure 22-1 depicts such a chain with N number of handlers.

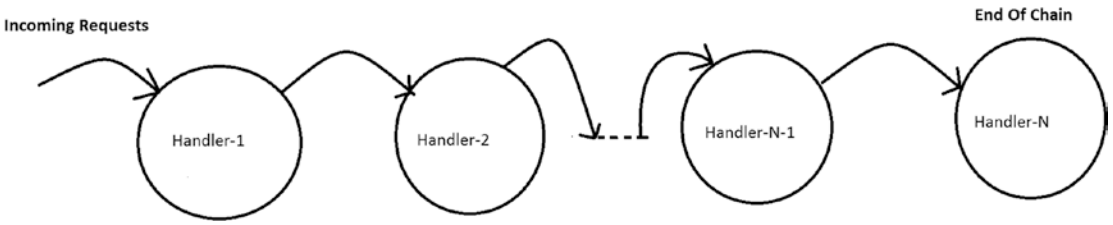


Figure 22-1. *The concept of a chain-of-responsibility pattern*

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_22

Real-World Example

- Each organization employs customer care executives who receive feedback or complaints directly from the customers. If the employees cannot answer the customers' issues properly, they forward these issues/escalations to the appropriate departments in the organization. These departments do not try to fix an issue simultaneously. In the first phase of investigation, the department that seems responsible analyzes the case, and if they believe that the issue should be forwarded to another department, they do that.
- A similar scenario occurs when a patient visits a hospital. Doctors from one department can refer a patient to a different department for further diagnosis.

Computer-World Example

Consider a software application (e.g., a printer) that can send emails and faxes. So, customers can report faxing issues or email issues. Let's assume that these issues are handled by handlers. So, you introduce two different types of error handlers: `EmailErrorHandler` and `FaxErrorHandler`. You can assume that `EmailErrorHandler` handles email errors only; it cannot fix the fax errors. In a similar manner, `FaxErrorHandler` handles fax errors and does not care about email errors.

So, you may form a chain like this: whenever the application finds an error, it raises a ticket and forwards the error with a hope that one of the handlers will handle it. Let's assume that the request first comes to `FaxErrorhandler`. If this handler agrees that the error is a fax issue, it handles it; otherwise, the handler forwards the issue to `EmailErrorHandler`.

Note that the chain ends with `EmailErrorHandler`. But if you need to handle a different type of issue—for example, authentication issues (which can occur due to security vulnerabilities), you can make a handler called `AuthenticationErrorHandler` and place it after `EmailErrorHandler`. Now if an `EmailErrorHandler` cannot fix the issue completely, it forwards the issue to `AuthenticationErrorHandler`, and the chain ends there.

Note You are free to place these handlers in any order you choose in your application.

So, the bottom line is that the processing chain may end in any of the following scenarios.

- Any of these handlers could process the request completely and control comes back.
- A handler cannot handle the request completely, so it passes the request to the next handlers. This way, you reach the end of the chain. So, the request is handled there. But if the request cannot be processed there, you cannot pass it further. (You may need to take special care for such a situation.)

You notice a similar mechanism when you are implementing an exception handling mechanism with multiple catch blocks in your Java application. If an exception occurs in a `try` block, the first catch block tries to handle it. If it cannot handle that type of exception, the next `catch` block tries to handle it, and the same mechanism is followed until the exception is handled properly by handlers (`catch` blocks). If the last catch block in your application is unable to handle it, an exception is thrown outside of this chain.

Note In java.util.logging.Logger, you can see a different overloaded version of log() methods that supports a similar concept.

Another built-in support can be seen in the doFilter (ServletRequest request, ServletResponse response, FilterChain chain) interface method in javax.Servlet.Filter.

Illustration

Let's consider the scenario that is discussed in the computer-world example. Let's further assume that in the following example, you can process both normal and high-priority issues that may come from either the email or fax pillar.

Class Diagram

Figure 22-2 shows the class diagram.

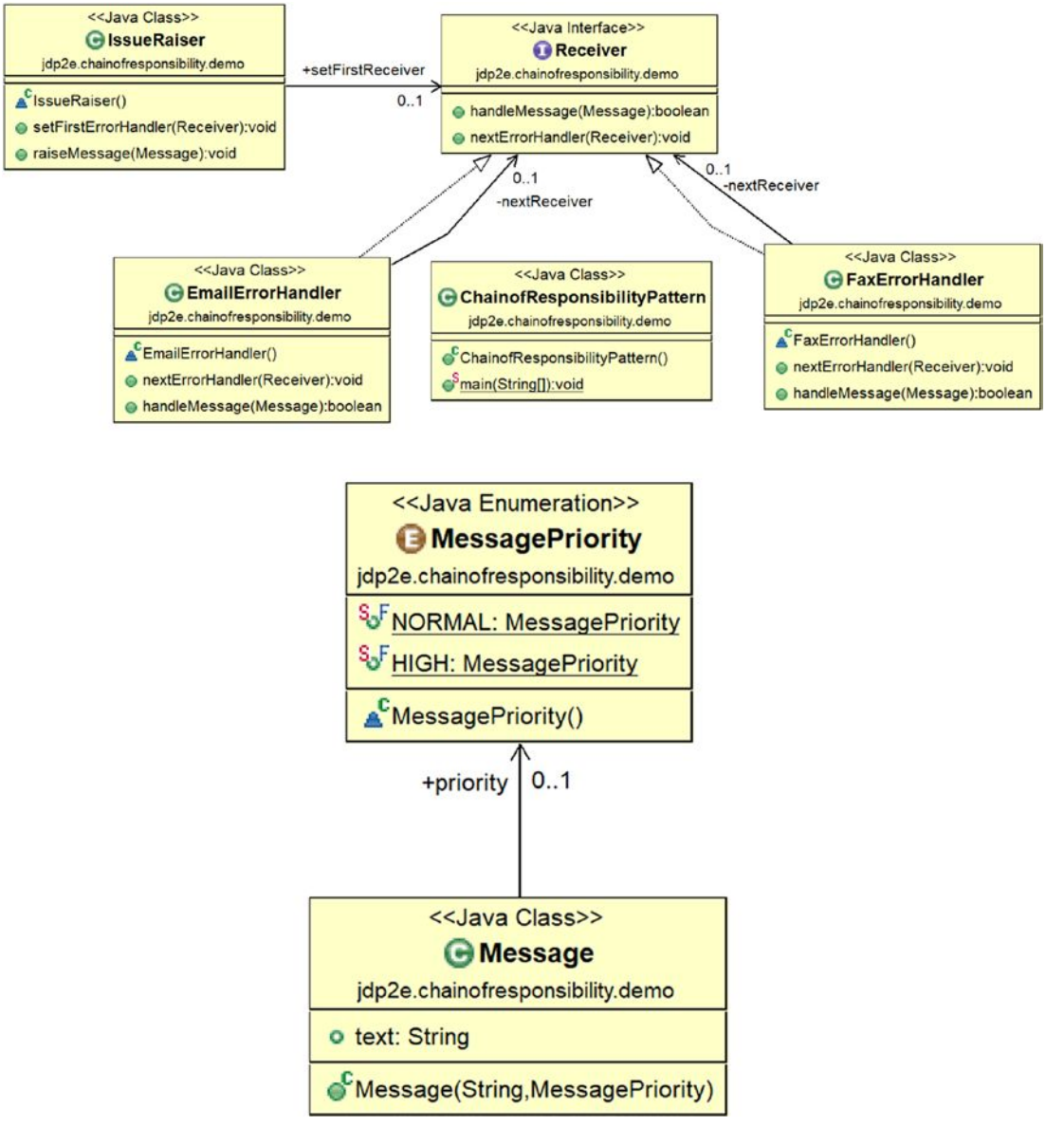


Figure 22-2. *Class diagram*

Package Explorer View

Figure 22-3 shows the high-level structure of the program.

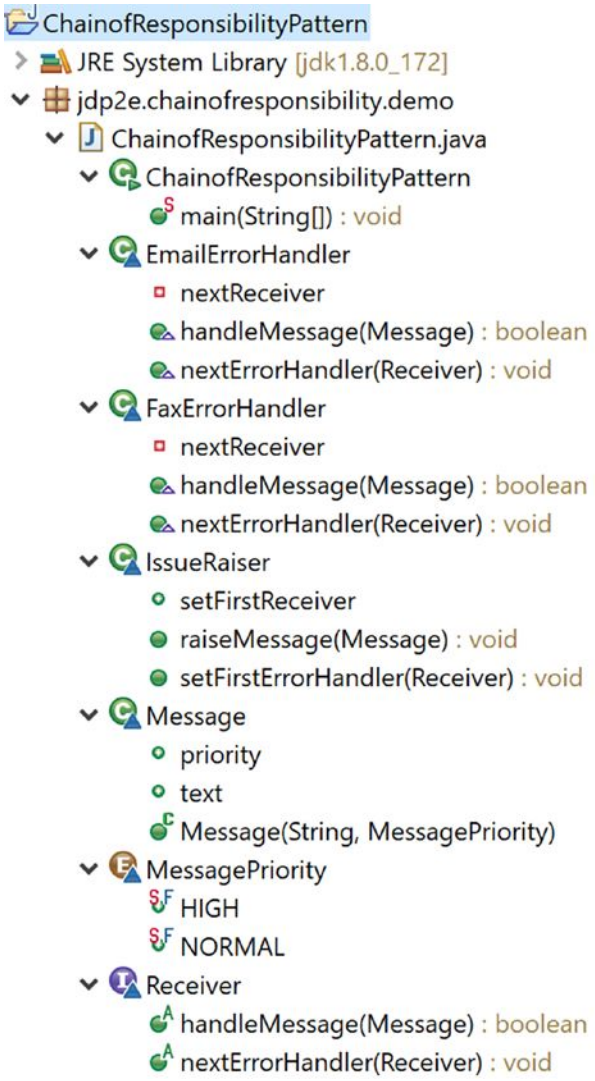


Figure 22-3. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.chainofresponsibility.demo;

enum MessagePriority
{
    NORMAL,
    HIGH
}
class Message
{
    public String text;
    public MessagePriority priority;
    public Message(String msg, MessagePriority p)
    {
        text = msg;
        this.priority = p;
    }
}

interface Receiver
{
    boolean handleMessage(Message message);
    void nextErrorHandler(Receiver nextReceiver);
}
class IssueRaiser
{
    public Receiver setFirstReceiver;
    public void setFirstErrorHandler(Receiver firstErrorHandler)
    {
        this.setFirstReceiver = firstErrorHandler;
    }
    public void raiseMessage(Message message)
    {
        if (setFirstReceiver != null)
```

```
            setFirstReceiver.handleMessage(message);
    }
}
// FaxErrorHandler class
class FaxErrorHandler implements Receiver
{
    private Receiver nextReceiver;
    @Override
    public void nextErrorHandler(Receiver nextReceiver)
    {
        this.nextReceiver = nextReceiver;
    }
    @Override
    public boolean handleMessage(Message message)
    {
        if (message.text.contains("Fax"))
        {
            System.out.println(" FaxErrorHandler processed " +message.
            priority +" priority issue :"+ message.text);
            return true;
        }
        else
        {
            if (nextReceiver != null)
                nextReceiver.handleMessage(message);
        }
        return false;
    }
}
// EmailErrorHandler class
class EmailErrorHandler implements Receiver
{
    private Receiver nextReceiver;
    @Override
    public void nextErrorHandler(Receiver nextReceiver)
```

```
    {
        this.nextReceiver = nextReceiver;
    }
    @Override
    public boolean handleMessage(Message message)
    {
        if (message.text.contains("Email"))
        {
            System.out.println(" EmailErrorHandler processed "+message.
            priority+ " priority issue: "+message.text);
            return true;
        }
        else
        {
            if (nextReceiver != null)
                nextReceiver.handleMessage(message);
        }
        return false;
    }
}
//Client code
public class ChainofResponsibilityPattern {

    public static void main(String[] args) {
        System.out.println("\n ***Chain of Responsibility Pattern
        Demo***\n");
        /* Forming the chain as IssueRaiser->FaxErrorhandler->
        EmailErrorHandler*/
        Receiver faxHandler, emailHandler;
        //Objects of the chains
        IssueRaiser issueRaiser = new IssueRaiser();
        faxHandler = new FaxErrorHandler();
        emailHandler = new EmailErrorHandler();
        //Making the chain
        //Starting point:IssueRaiser will raise issues and set the first
        //handler
```

```
        issueRaiser.setFirstErrorHandler(faxHandler);
        //FaxErrorHandler will pass the error to EmailHandler if needed.
        faxHandler.nextErrorHandler(emailHandler);
        //EmailErrorHandler will be placed at the last position in the chain
        emailHandler.nextErrorHandler(null);

        Message m1 = new Message("Fax is going slow.",
        MessagePriority.NORMAL);
        Message m2 = new Message("Emails are not reaching.",
        MessagePriority.HIGH);
        Message m3 = new Message("In Email, CC field is disabled always.",
        MessagePriority.NORMAL);
        Message m4 = new Message("Fax is not reaching destinations.",
        MessagePriority.HIGH);

        issueRaiser.raiseMessage(m1);
        issueRaiser.raiseMessage(m2);
        issueRaiser.raiseMessage(m3);
        issueRaiser.raiseMessage(m4);
    }
}
```

Output

Here's the output.

```
***Chain of Responsibility Pattern Demo***

 FaxErrorHandler processed NORMAL priority issue :Fax is going slow.
 EmailErrorHandler processed HIGH priority issue: Emails are not reaching.
 EmailErrorHandler processed NORMAL priority issue: In Email, CC field is
 disabled always.
 FaxErrorHandler processed HIGH priority issue :Fax is not reaching
 destinations.
```

Q&A Session

1. **In the example, what is the purpose of message priorities?**

   Good catch. Actually, you could ignore them because, for simplicity in the handlers, you are just searching for the words *email* or *fax*. These priorities are added to beautify the code. But instead of using separate handlers for email and fax, you could make a different kind of chain that handles the messages based on the priorities. In such a case, these priorities can be used more effectively.

2. **What are the advantages of using a chain-of-responsibility design pattern?**

   - You can have more than one object to handle a request. (Notice that if a handler cannot handle the whole request, it may forward the responsibility to the next handler in the chain).
   - The nodes of the chain can be added or removed dynamically. Also, you can shuffle the order. For example, if you notice that the majority of issues are with email processing, then you may place EmailErrorHandler as the first handler in the chain to save the average processing time of the application.
   - A handler does not need to know how the next handler in the chain will handle the request. It focuses only on its own handling mechanism.
   - In this pattern, you are promoting loose coupling because it decouples the senders (of requests) from the receivers.

3. **What are the challenges associated with using the chain-of-responsibility design pattern?**

   - There is no guarantee that the request will be handled (fully or partially) because you may reach the end of the chain; but it is possible that you have not found any explicit receiver to handle the request.
   - Debugging may become tricky with this kind of design.

4. **How can you handle the scenario where you have reached at the end of chain, but the request is not handled at all?**

   One simple solution is to use `try/catch` (or `try/finally` or `try/catch/finally`) blocks. You may put the handlers in these constructs. You may notice that a `try` block can be associated with multiple `catch` blocks also.

   In the end, if no one can handle the request, you may raise an exception with the appropriate messages and catch the exception in your intended `catch` block to draw your attention (or handle it in some different way).

   The GoF talked about Smalltalk's automatic forwarding mechanism, `doesNotUnderstand`, in a similar context. If a message cannot find a proper handler, it is caught in `doesNotUnderstand` implementations that can be overridden to forward the message in the object's successor, log it in a file, and store it in a queue for later processing, or you can simply perform any other intended operations. But you must make a note that by default, this method raises an exception that needs to be handled in a proper way.

5. **In short, if a handler cannot handle the request fully, it will pass it to the next handler. Is this correct?**

   Yes.

6. **It appears that there are similarities between the observer pattern and the chain-of-responsibility pattern. Is this correct?**

   In an observer pattern, all registered users get notifications in parallel; but in a chain-of-responsibility pattern, objects in the chain are notified, one by one, in a sequential manner. This process continues until an object handles the notification fully (or you reach the end of the chain). I show the comparisons in diagrams in the "Q&A Session" in Chapter 14.

CHAPTER 23

Interpreter Pattern

This chapter covers the interpreter pattern.

GoF Definition

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Concept

To understand this pattern, you need to be familiar with some key terms, like *sentences*, *grammar, languages*, and so forth. So, you may need to visit the topics of formal languages in Automata, if you are not familiar with them.

Normally, this pattern deals with *how to evaluate sentences in a language*. So, you first need to define a grammar to represent the language. Then the interpreter deals with that grammar. This pattern is best if the grammar is simple.

Each class in this pattern may represent a rule in that language, and it should have a method to interpret an expression. So, to handle a greater number of rules, you need to create a greater number of classes. This is why an interpreter pattern should not be used to handle complex grammar.

Let's consider different arithmetic expressions in a calculator program. Though these expressions are different, they are all constructed using some basic rules, which are defined in the grammar of the language (of these arithmetic expressions). So, it is best if you can interpret a generic combination of these rules rather than treat each combination of rules as separate cases. An interpreter pattern can be used in such a scenario.

A typical structure of this pattern is often described with a diagram similar to Figure 23-1.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_23

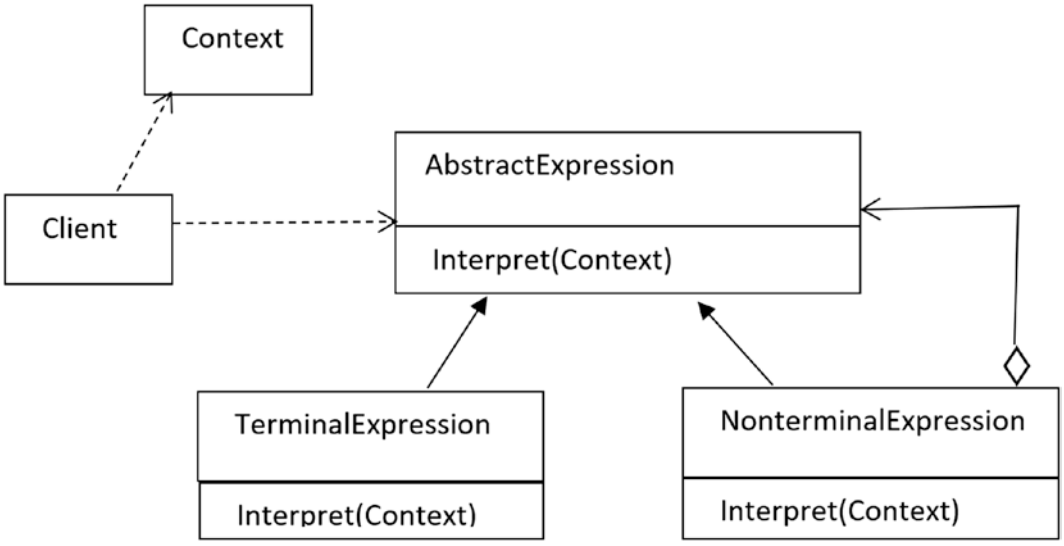


Figure 23-1. *Structure of a typical interpreter pattern*

The terms are described as follows.

- *AbstractExpression*: Typically an interface with an interpret method. You need to pass a context object to this method.
- *TerminalExpression*: Used for terminal expressions. A terminal expression does not need other expressions to interpret. These are basically leaf nodes (i.e., they do not have child nodes) in the data structure.
- *NonterminalExpression*: Used for nonterminal expressions. Also known as *AlternationExpression*, *RepititionExpression*, or *SequenceExpression*. These are like composites that can contain both the terminal and nonterminal expressions. When you call `interpret()` method on this, you basically call it on all of its children.
- *Context*: Holds the global information that the interpreter needs.
- *Client*: Calls the `interpret()` method. It can optionally build a syntax tree based on the rules of the language.

Note An interpreter is used to process a language with simple rules or grammar. Ideally, developers do not want to create their own languages. This is the reason why they seldom use this pattern.

Real-World Example

- A translator who translates a foreign language.
- Consider music notes as grammar, where musicians play the role of interpreters.

Computer-World Example

- Java compiler interprets the Java source code into byte code that is understandable by JVM.
- In C#, the source code is converted to MSIL code that is interpreted by CLR. Upon execution, this MSIL (intermediate code) is converted to native code (binary executable code) by JIT compiler.

Note In Java, you may also notice the java.util.regex.Pattern class that acts as an interpreter. You can create an instance of this class by invoking the compile() method and then you can use a Matcher instance to evaluate a sentence against the grammar.

Illustration

These are some important steps to implement this pattern.

- **Step 1**. Define the rules of the language for which you want to build an interpreter.
- **Step 2**. Define an abstract class or an interface to represent an expression. It should contain a method to interpret an expression.

    - Step 2A. Identify terminal and nonterminal expressions. For example, in the upcoming example, IndividualEmployee class is a terminal expression class.
    - Step2B. Create nonterminal expression classes. Each of them calls interpret method on their children. For example, in the upcoming example, OrExpression and AndExpression classes are nonterminal expression classes.
- **Step 3**. Build the abstract syntax tree using these classes. *You can do this inside the client code or you can create a separate class to accomplish the task.*
- **Step 4**. A client now uses this tree to interpret a sentence.
- **Step 5**. Pass the context to the interpreter. It typically has the sentences that are to be interpreted. An interpreter can do additional tasks using this context.

In the upcoming program, I use the interpreter pattern as a rule validator. I am instantiating different employees with their "years of experience" and current grades. Note the following lines.

```
Employee emp1 = new IndividualEmployee(5,"G1");
Employee emp2 = new IndividualEmployee(10,"G2");
Employee emp3 = new IndividualEmployee(15,"G3");
Employee emp4 = new IndividualEmployee(20,"G4");
```

For simplicity, four employees with four different grades—G1,G2,G3, and G4—are considered here.

Also note the context, as follows.

```
//Minimum Criteria for promoton is:
//The year of experience is minimum 10 yrs. and
//Employee grade should be either G2 or G3
Context context=new Context(10,"G2","G3");
```

So, you can assume that I want to validate some condition against the context, which basically tells you that to be promoted, an employee should have a minimum of 10 years of experience and he/she should be either from the G2 grade or the G3 grade. Once these expressions are interpreted, you see the output in terms of a boolean value.

One important point to note is that this design pattern does not instruct you how to build the syntax tree or how to parse the sentences. It gives you freedom on how to proceed. So, to present a simple scenario, I used an EmployeeBuilder class with a method called `buildExpression()` to accomplish my task.

Class Diagram

Figure 23-2 shows the class diagram.

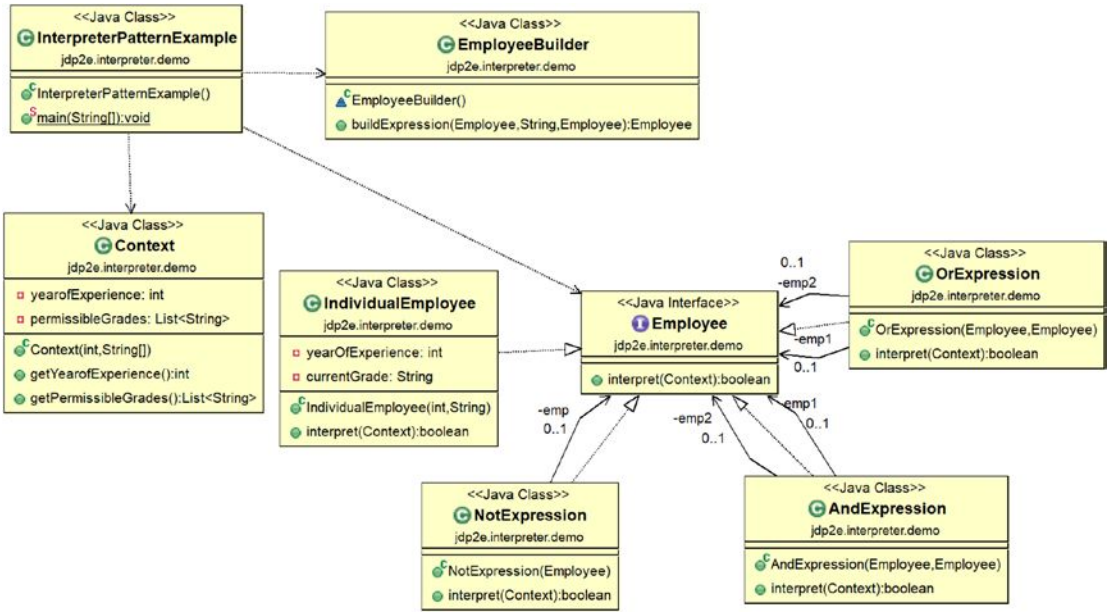


Figure 23-2. *Class diagram*

Package Explorer View

Figure 23-3 shows the high-level structure of the program.

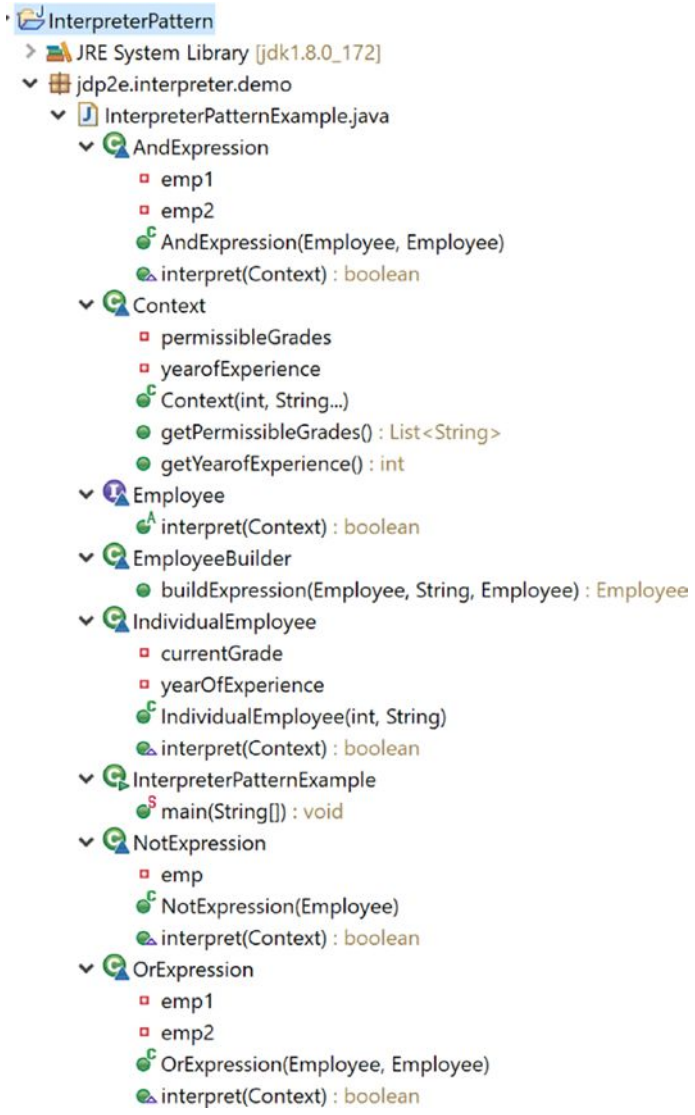


Figure 23-3. *Package Explorer view*

Implementation

Here is the implementation.

```
package jdp2e.interpreter.demo;

import java.util.ArrayList;
import java.util.List;

interface Employee
{
    public boolean interpret(Context context);
}
class IndividualEmployee implements Employee
{
    private int yearOfExperience;

    private String currentGrade;


    public IndividualEmployee(int experience, String grade){
        this.yearOfExperience=experience;
        this.currentGrade=grade;
    }
    @Override
    public boolean interpret(Context context)
    {

        if(this.yearOfExperience>=context.getYearofExperience() && context.
        getPermissibleGrades().contains(this.currentGrade))
        {
            return true;
        }
        return false;
    }
}
```

```
class OrExpression implements Employee
{

    private Employee  emp1;
    private Employee  emp2;

    public OrExpression(Employee emp1, Employee emp2)
    {
        this.emp1 = emp1;
        this.emp2 = emp2;
    }

    @Override
    public boolean interpret(Context context)
    {
        return emp1.interpret(context) || emp2.interpret(context);
    }
}
class AndExpression implements Employee
{

    private Employee  emp1;
    private Employee  emp2;

    public AndExpression(Employee emp1, Employee emp2)
    {
        this.emp1 = emp1;
        this.emp2 = emp2;
    }

    @Override
    public boolean interpret(Context context)
    {
        return emp1.interpret(context) && emp2.interpret(context);
    }
}
```

```
class NotExpression implements Employee
{
    private Employee  emp;

    public NotExpression(Employee  expr)
    {
        this.emp = expr;
    }

    @Override
    public boolean interpret(Context context)
    {
        return !emp.interpret(context);
    }
}
class Context
{
    private int yearofExperience;
    private List<String> permissibleGrades;
    public Context(int experience,String... allowedGrades)
    {
        this.yearofExperience=experience;
        this.permissibleGrades=new ArrayList<>();
        for( String grade:allowedGrades)
        {
            permissibleGrades.add(grade);
        }
    }
    public int getYearofExperience()
    {
        return yearofExperience;
    }
    public List<String> getPermissibleGrades()
    {
        return permissibleGrades;
    }
}
```

```
class EmployeeBuilder
{
    public Employee buildExpression(Employee emp1,  String operator,
    Employee emp2)
    {
        //Whatever the input,converting it to lowarcase
        switch(operator.toLowerCase())
        {
        case "or":
            return new OrExpression(emp1,emp2);
        case "and":
            return new AndExpression(emp1,emp2);
        case "not":
            return new NotExpression(emp1);
        default:
            System.out.println("Only AND,OR and NOT operators are allowed
            at present");
            return null;
        }
    }
}
public class InterpreterPatternExample {

    public static void main(String[] args) {
        System.out.println("***Interpreter Pattern Demo***\n");

        //Minimum Criteria for promoton is:
        //The year of experience is minimum 10 yrs. and
        //Employee grade should be either G2 or G3
        Context context=new Context(10,"G2","G3");

        //Different employees with grades
        Employee emp1 = new IndividualEmployee(5,"G1");
        Employee emp2 = new IndividualEmployee(10,"G2");
        Employee emp3 = new IndividualEmployee(15,"G3");
        Employee emp4 = new IndividualEmployee(20,"G4");

        EmployeeBuilder builder=new EmployeeBuilder();
```

```
        System.out.println("emp1 is eligible for promotion. " + emp1.
        interpret(context));
        System.out.println("emp2 is eligible for promotion. " + emp2.
        interpret(context));
        System.out.println("emp3 is eligible for promotion. " + emp3.
        interpret(context));
        System.out.println("emp4 is eligible for promotion. " + emp4.
        interpret(context));

        System.out.println("Is either emp1 or emp3 is eligible
        for promotion?" +builder.buildExpression(emp1,"Or",emp3).
        interpret(context));
        System.out.println("Is both emp2 and emp4 are eligible for
        promotion? ?" + builder.buildExpression(emp2,"And",emp4).
        interpret(context));
        System.out.println("The statement 'emp3 is NOT eligible for
        promotion' is true? " + builder.buildExpression(emp3, "Not",null).
        interpret(context));
        //Invalid input expression
        //System.out.println("Is either emp1 or emp3 is eligible for
        promotion?" +builder.buildExpression(emp1,"Wrong",emp3).
        interpret(context));
    }
}
```

Output

Here is the output.

```
***Interpreter Pattern Demo***

emp1 is eligible for promotion. false
emp2 is eligible for promotion. true
emp3 is eligible for promotion. true
emp4 is eligible for promotion. false
Is either emp1 or emp3 is eligible for promotion?true
Is both emp2 and emp4 are eligible for promotion? ?false
The statement 'emp3 is NOT eligible for promotion' is true? false
```

Analysis

You can see that each of the composite expressions are invoking the `interpret()` method on all of its children.

Modified Illustration

You have just seen a simple example of the interpreter pattern. From this implementation, it may appear to you that you have handled some easy and straightforward expressions. So, lets handle some complex rules or expressions in the modified implementation.

Modified Class Diagram

In the modified implementation, the key changes are made only in the EmployeeBuilder class. So, let's have a quick look of the class diagram for this class only (see Figure 23-4).

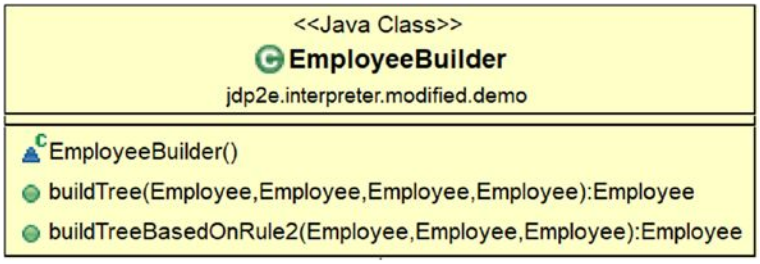


Figure 23-4. *Modified Class diagram for EmployeeBuilder class*

Modified Package Explorer View

In the modified implementation, the key changes are reflected only in the EmployeeBuilder class. So, in this section I expanded this class only. Figure 23-5 shows the modified Package Explorer view.

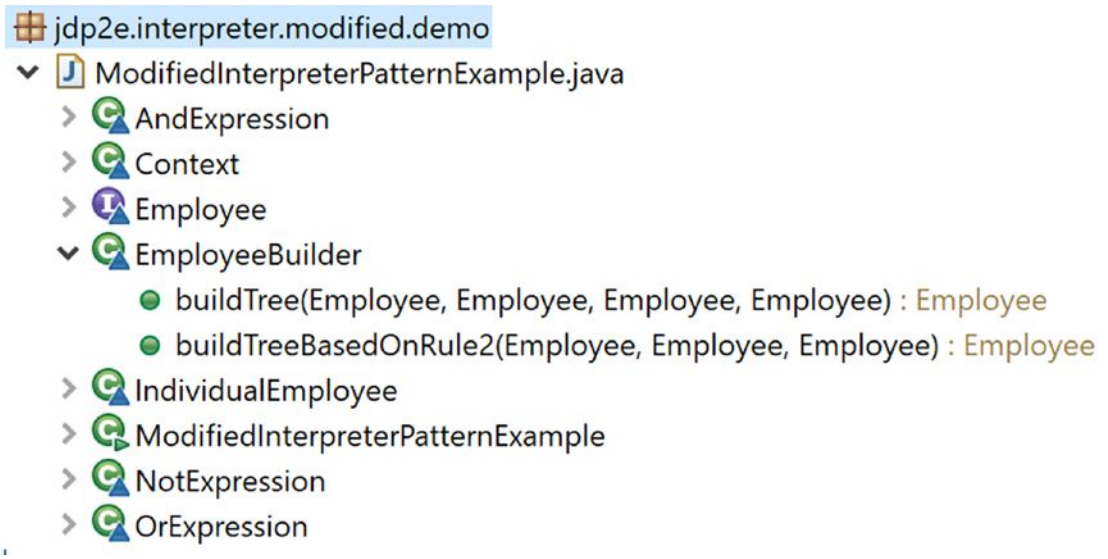


Figure 23-5. *Modified Package Explorer View*

Modified Implementation

Here is the modified implementation. Key changes are shown in bold.

```
package jdp2e.interpreter.modified.demo;

import java.util.ArrayList;
import java.util.List;

interface Employee
{
    public boolean interpret(Context context);
}
class IndividualEmployee implements Employee
{
    private int yearOfExperience;

    private String currentGrade;

    public IndividualEmployee(int experience, String grade){
        this.yearOfExperience=experience;
        this.currentGrade=grade;
    }
```

```
    @Override
    public boolean interpret(Context context)
    {

        if(this.yearOfExperience>=context.getYearofExperience() && context.
        getPermissibleGrades().contains(this.currentGrade))
        {
            return true;
        }
        return false;
    }
}
class OrExpression implements Employee
{

    private Employee  emp1;
    private Employee  emp2;

    public OrExpression(Employee emp1, Employee emp2)
    {
        this.emp1 = emp1;
        this.emp2 = emp2;
    }

    @Override
    public boolean interpret(Context context)
    {
        return emp1.interpret(context) || emp2.interpret(context);
    }
}
class AndExpression implements Employee
{

    private Employee  emp1;
    private Employee  emp2;

    public AndExpression(Employee emp1, Employee emp2)
    {
```

```
        this.emp1 = emp1;
        this.emp2 = emp2;
    }

    @Override
    public boolean interpret(Context context)
    {
        return emp1.interpret(context) && emp2.interpret(context);
    }
}
class NotExpression implements Employee
{
    private Employee  emp;

    public NotExpression(Employee  expr)
    {
        this.emp = expr;
    }

    @Override
    public boolean interpret(Context context)
    {
        return !emp.interpret(context);
    }
}
class Context
{
    private int yearofExperience;
    private List<String> permissibleGrades;
    public Context(int experience,String... allowedGrades)
    {
        this.yearofExperience=experience;
        this.permissibleGrades=new ArrayList<>();
        for( String grade:allowedGrades)
        {
            permissibleGrades.add(grade);
        }
```

```
    }
    public int getYearofExperience()
    {
        return yearofExperience;
    }
    public List<String> getPermissibleGrades()
    {
        return permissibleGrades;
    }
}
class EmployeeBuilder
{

    // Building the tree
    //Complex Rule-1: emp1 and (emp2 or (emp3 or emp4))

    public Employee buildTree(Employee emp1, Employee emp2,Employee
    emp3,Employee emp4)
    {
        //emp3 or emp4
        Employee firstPhase=new OrExpression(emp3,emp4);
        //emp2 or (emp3 or emp4)
        Employee secondPhase=new OrExpression(emp2,firstPhase);
        //emp1 and (emp2 or (emp3 or emp4))
        Employee finalPhase=new AndExpression(emp1,secondPhase);
        return finalPhase;

    }
    //Complex Rule-2: emp1 or (emp2 and (not emp3 ))
    public Employee buildTreeBasedOnRule2(Employee emp1, Employee
    emp2,Employee emp3)
    {
        //Not emp3
        Employee firstPhase=new NotExpression(emp3);
        //emp2 or (not emp3)
        Employee secondPhase=new AndExpression(emp2,firstPhase);
```

```
        //emp1 and (emp2 or (not emp3 ))
        Employee finalPhase=new OrExpression(emp1,secondPhase);
        return finalPhase;

    }

}
public class ModifiedInterpreterPatternExample {

    public static void main(String[] args) {
        System.out.println("***Modified Interpreter Pattern Demo***\n");

        //Minimum Criteria for promoton is:
        //The year of experience is minimum 10 yrs. and
        //Employee grade should be either G2 or G3
        Context context=new Context(10,"G2","G3");
        //Different Employees with grades
        Employee emp1 = new IndividualEmployee(5,"G1");
        Employee emp2 = new IndividualEmployee(10,"G2");
        Employee emp3 = new IndividualEmployee(15,"G3");
        Employee emp4 = new IndividualEmployee(20,"G4");

        EmployeeBuilder builder=new EmployeeBuilder();

        //Validating the 1st complex rule
        System.out.println("Is emp1 and any of emp2,emp3, emp4 is eligible
        for promotion?" +builder.buildTree(emp1,emp2, emp3,emp4).
        interpret(context));
        System.out.println("Is emp2 and any of emp1,emp3, emp4 is eligible
        for promotion?" +builder.buildTree(emp2,emp1, emp3,emp4).
        interpret(context));
        System.out.println("Is emp3 and any of emp1,emp2, emp3 is eligible
        for promotion?" +builder.buildTree(emp3,emp1, emp2,emp4).
        interpret(context));
```

```
        System.out.println("Is emp4 and any of emp1,emp2, emp3 is eligible
        for promotion?" +builder.buildTree(emp4,emp1, emp2,emp3).
        interpret(context));

        System.out.println("");
        //Validating the 2nd complex rule
        System.out.println("Is emp1 or (emp2 but not emp3) is eligible
        for promotion?" +builder.buildTreeBasedOnRule2(emp1, emp2, emp3).
        interpret(context));
        System.out.println("Is emp2 or (emp3 but not emp4) is eligible
        for promotion?" +builder.buildTreeBasedOnRule2(emp2, emp3, emp4).
        interpret(context));
    }
}
```

Modified Output

Here is the modified output.

```
***Modified Interpreter Pattern Demo***

Is emp1 and any of emp2,emp3, emp4 is eligible for promotion?false
Is emp2 and any of emp1,emp3, emp4 is eligible for promotion?true
Is emp3 and any of emp1,emp2, emp4 is eligible for promotion?true
Is emp4 and any of emp1,emp2, emp3 is eligible for promotion?false

Is emp1 or (emp2 but not emp3) is eligible for promotion?false
Is emp2 or (emp3 but not emp4) is eligible for promotion?true
```

Analysis

Now you have an idea of how to handle complex rules that follow the approach shown by using an interpreter pattern.

Q&A Session

1. **When should I use this pattern?**

   In daily programming, it is not needed very much. Though in some rare situations, you may need to work with your own programming language to define specific protocols. In a situation like this, this pattern may become handy. But before you proceed, you must ask yourself about the return on investment (ROI).

2. **What are the advantages of using an interpreter design pattern?**

   - You are very much involved in the process of how to define grammar for your language and how to represent and interpret those sentences. You can change and extend your grammar also.
   - You have full freedom over how to interpret these expressions.

3. **What are the challenges associated with using interpreter design patterns?**

   I believe that the amount of work is the biggest concern. Also maintaining complex grammar becomes tricky because you may need to create (and maintain) separate classes to deal with different rules.

PART II

Additional Design Patterns

CHAPTER 24

Simple Factory Pattern

This chapter covers the simple factory pattern.

Intent

Create an object without exposing the instantiation logic to the client.

Concept

In object-oriented programming, a factory is a special kind of object that can create other objects. A factory can be invoked in many ways, but most often, it uses a method that can return objects with varying prototypes. Any subroutine that can help create these new objects is considered a factory. The ultimate purpose of using a factory method is to abstract the object creational mechanism (or process) from the consumers of the application.

Real-World Example

Consider a car manufacturing company that manufactures different models of a car. They must have a factory with different production units. Some of these units can produce the parts that are common to all models, while other units are dedicated to produce the model-specific parts. When they make the final product, they assemble the model-specific parts with the common parts. From a client's point of view, a car is built from a car factory; the client does not know how the car is built. But if you investigate further, you see that based on the model of the car, a production unit of the factory varies the parts. For example, a particular car model can support a manual gearbox only and

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_24

another model can support both the automatic and manual gearbox. So, based on the model of the car, the car factory constructs the particular gearbox for the car.

Consider a simpler example. When a kid demands a toy from his/her parent, the child does not know how the parent will fulfill the demand. The parent, in this case, is considered a factory for their small child. Now think from the parent's point of view. The parent can make the toy himself/herself or purchase a toy from a shop to make their kid happy.

Computer-World example

The simple factory pattern is very common to software applications, but before we proceed further, you must remember these points.

- A simple factory is not treated as a standard design pattern in the GoF's famous book, but the approach is common to any application that you write where you want to separate the code that varies a lot from the part of code that does not vary. It is assumed that you try to follow this approach in any application you write.
- A simple factory is considered the simplest form of factory method patterns (or abstract factory patterns). So, you can assume that any application that follows either the factory method pattern or the abstract factory pattern, also supports the concept of simple factory pattern's design goals.

Note The static getInstance()method of the java.text.NumberFormat class is an example of this category.

Let's follow the implementation in which I discuss this pattern in a common use case.

Illustration

The following are the important characteristics of the following implementation.

- In this example, there are two types of animals: dogs and tigers. The object creational process depends on users' input.
- I assume that each of them can speak and they prefer to perform some actions.
- SimpleFactory is the factory class and simpleFactory (note that the "s" is not in caps) is an object of the class. In the client code (SimpleFactoryPatternExample class), you see the following line.

```
preferredType = simpleFactory.createAnimal();
```

This means that to get a preferredType object, you need to invoke the `createAnimal()` method of the simpleFactory object. So, using this approach, you are not directly using a "new" operator in the client code to get an object.

- I have separated the code that varies from the code that are least likely to vary. This approach helps you remove tight coupling in the system. (How? Follow the "Q&A Session" section.)

Note In some applications, you may notice a slight variation of this pattern where use of parameterized constructors is suggested. So, in those applications, to get a preferredType object, you may need to use a line of code similar to this line: preferredType=simpleFactory.createAnimal("Tiger").

Class Diagram

Figure 24-1 shows a class diagram for the simple factory pattern.

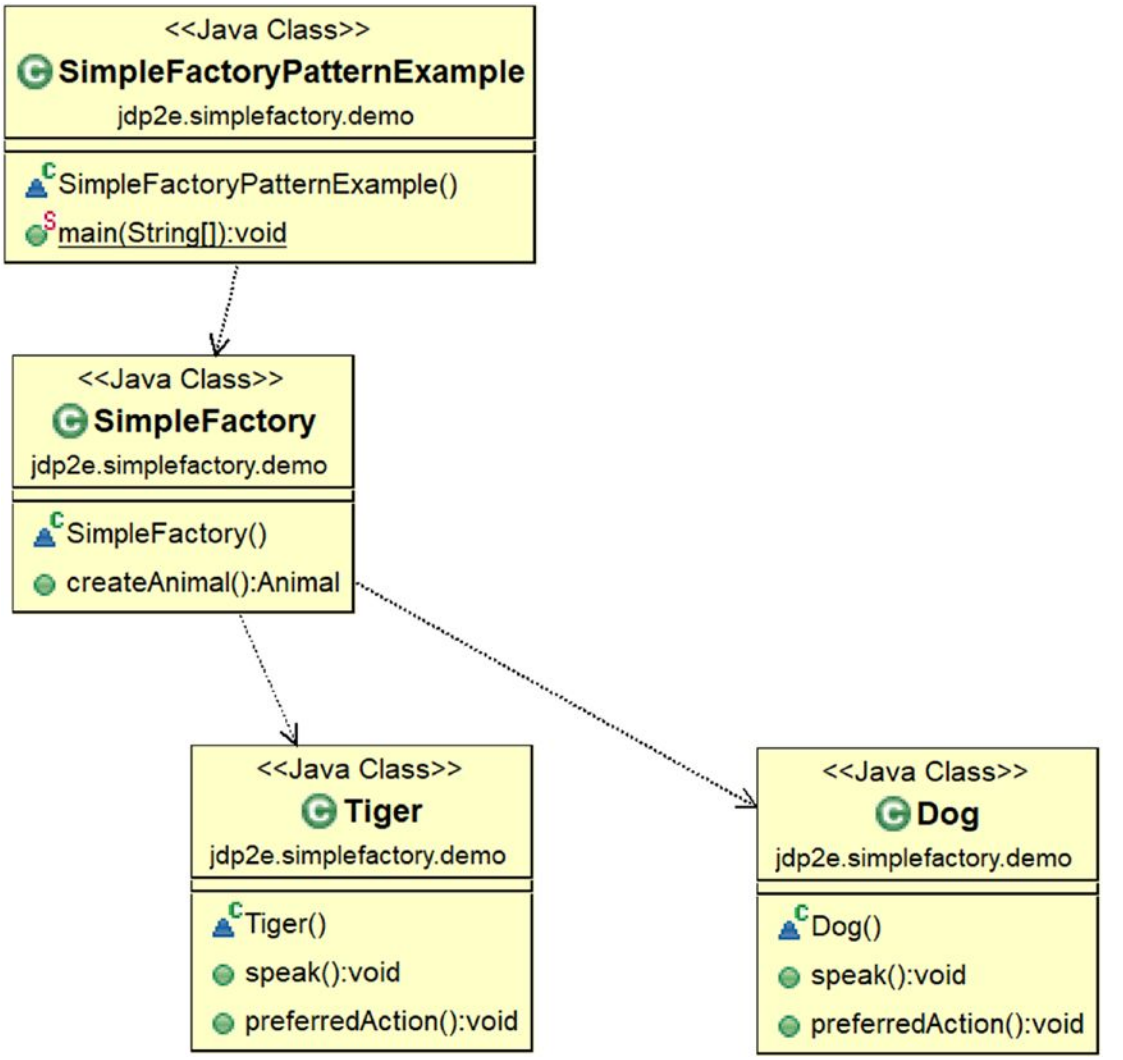


Figure 24-1. *Class diagram*

Package Explorer View

Figure 24-2 shows the high-level structure of the program.

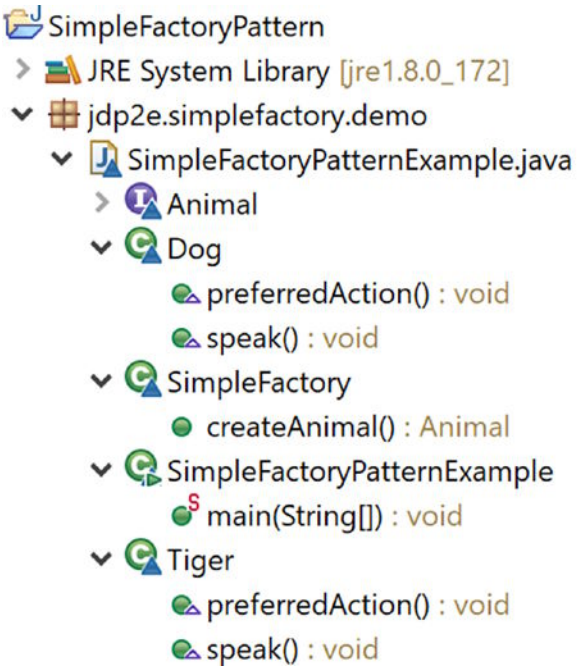


Figure 24-2. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.simplefactory.demo;
import java.util.Scanner;//Available Java5 onwards

interface Animal
{
    void speak();
    void preferredAction();
}
class Dog implements Animal
{
    public void speak()
    {
        System.out.println("Dog says: Bow-Wow.");
    }
```

```
    public void preferredAction()
    {
        System.out.println ("Dogs prefer barking...");
    }
}
class Tiger implements Animal
{
    public void speak()
    {
        System.out.println("Tiger says: Halum.");
    }
    public void preferredAction()
    {
        System.out.println("Tigers prefer hunting...");
    }
}
class SimpleFactory
{
    public Animal createAnimal()
    {
        Animal intendedAnimal=null;
        System.out.println("Enter your choice( 0 for Dog, 1 for Tiger)");
        /* To suppress the warning message:Resource leak:'input' is never
        closed. So,the following line is optional in this case*/
        @SuppressWarnings("resource")
        Scanner input=new Scanner(System.in);
        int choice=Integer.parseInt(input.nextLine());
        System.out.println("You have entered :"+ choice);
        switch (choice)
        {
        case 0:
            intendedAnimal = new Dog();
            break;
```

```
        case 1:
            intendedAnimal = new Tiger();
            break;
        default:
            System.out.println("You must enter either 0 or 1");
            //We'll throw a runtime exception for any other choices.
            throw new IllegalArgumentException(" Your choice tries to
            create an unknown Animal");
        }

        return intendedAnimal;
    }
}
//A client is interested to get an animal who can speak and perform an
//action.
class SimpleFactoryPatternExample
{
    public static void main(String[] args)     {
        System.out.println("*** Simple Factory Pattern Demo***\n");
        Animal preferredType=null;
        SimpleFactory simpleFactory = new SimpleFactory();
        // The code that will vary based on users preference.
        preferredType = simpleFactory.createAnimal();
        //The codes that do not change frequently.
        //These animals can speak and prefer to do some specific actions.
        preferredType.speak();
        preferredType.preferredAction();

    }
}
```

Output

Here's the output.

Case1. User input:0

```
*** Simple Factory Pattern Demo***

Enter your choice( 0 for Dog, 1 for Tiger)
0
You have entered :0
Dog says: Bow-Wow.
Dogs prefer barking...
```

Case2. User input:1

```
*** Simple Factory Pattern Demo***

Enter your choice( 0 for Dog, 1 for Tiger)
1
You have entered :1
Tiger says: Halum.
Tigers prefer hunting...
```

Case3. An unwanted user input:2

```
*** Simple Factory Pattern Demo***

Enter your choice( 0 for Dog, 1 for Tiger)
2
You have entered :2
You must enter either 0 or 1Exception in thread "main"
java.lang.IllegalArgumentException:  Your choice tries to create an unknown
Animal
    at jdp2e.simplefactory.demo.SimpleFactory.createAnimal(SimpleFactoryPat
ternExample.java:54)
    at jdp2e.simplefactory.demo.SimpleFactoryPatternExample.main(SimpleFact
oryPatternExample.java:68)
```

Q&A Session

1. **In this example, the clients are delegating the objects' creation through the SimpleFactory. But instead, they could directly create the objects with the "new" operator. Is this correct?**

   No. These are the key reasons behind the preceding design.

   - An important object-oriented design principle is to separate the part of your code that is most likely to change from the rest.
   - In this case, only "the objects creational part" varies. I assume that these animals must speak and perform actions, and I do not need to vary that portion of code inside the client. So, in the future, if you need to modify the creational process, you need to change only the `createAnimal()` method of SimpleFactory class. This client code is unaffected due to those modifications.
   - "How are you creating objects?" is hidden in the client code. This kind of abstraction promotes security.
   - This approach can help you avoid lots of `if/else` blocks (or switch statements) inside the client code because they make your code look clumsy.

2. **What are the challenges associated with this pattern?**

   - Deciding which object to instantiate becomes complex over time. In those cases, you should prefer the factory method pattern.
   - If you want to add a new animal or delete an existing one, you need to modify the `createAnimal()` method of the factory class. This approach clearly violates the open-closed principle (which basically says that your code should be open for extension but closed for modification) of SOLID principles.

Note SOLID principles were promoted by Robert C. Martin. You can learn about them at https://en.wikipedia.org/wiki/SOLID.

---

3. **I learned that programming with an abstract class or interface is always a better practice. So, to make a better implementation, you could write something like this:**

```
abstract class ISimpleFactory
{
    public abstract IAnimal createAnimal() throws IOException;
}
class SimpleFactory extends ISimpleFactory
{

    //rest of the code
}
```

Is this correct?

Yes. Programming with the abstract class or an interface is always a better practice. This approach can prevent you from future changes because any newly added classes can simply implement the interface and settle down in the architecture through polymorphism. But if you solely depend on concrete classes, you need to change your code when you want to integrate a new class in the architecture, and in such a case, *you violate the rule that says that your code should be closed for modifications*.

So, your understanding is correct. You could use such a construct to make it a better program. But ultimately, you learn the factory method pattern (see Chapter 4), where you need to defer the instantiation process to subclasses. So, you are advised to write programs with an abstract class or an interface in such a case.

4. **Can you make the factory class (SimpleFactory) static?**

No. In Java, you are not allowed to tag the word *static* with a top-level class. In other words, by design, the compiler always complains about the top-level static classes in Java.

CHAPTER 25

Null Object Pattern

Wikipedia says, "In object-oriented computer programming, a null object is an object with no referenced value or with defined neutral (null) behavior. The null object design pattern describes the uses of such objects and their behavior (or lack thereof). It was first published in the *Pattern Languages of Program Design* book series." The Hillside Group sponsors Pattern Languages of Programs (PLoP) annual conferences.

The pattern can implement a "do-nothing" relationship or it can provide a default behavior when an application encounter with a null object instead of a real object. In simple words, the core aim is to make a better solution by avoiding "null objects check" or "null collaborations check" through `if` blocks. Using this pattern, you try to encapsulate the absence of an object by providing a default behavior that does nothing.

Concept

The notable characteristic of this pattern is that you do not need to do anything (or store nothing) when you invoke an operation on a null object. Consider the following program and the corresponding output. Let's try to understand the problems associated with the following program segment, analyze the probable solutions and at the end of this chapter, you see a better implementation that uses this design pattern.

In the following implementation, let's assume that you have two types of vehicles: bus and train. A client can opt for a bus or a train object through different input, like "a" or "b". Let's further assume that the application considers these two as the valid input only.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_25

A Faulty Program

Here is a faulty program.

```
package jdp2e.nullobject.context.demo;

import java.util.Scanner;

interface Vehicle
{
    void travel();
}
class Bus implements Vehicle
{
    public static int busCount = 0;
    public Bus()
    {
        busCount++;
    }
    @Override
    public void travel()
    {
        System.out.println("Let us travel with a bus");
    }
}
class Train implements Vehicle
{
    public static int trainCount = 0;
    public Train()
    {
        trainCount++;
    }
    @Override
    public void travel()
    {
        System.out.println("Let us travel with a train");
    }
}
```

```
public class NeedForNullObjectPattern {

    public static void main(String[] args) {
        System.out.println("***Need for Null Object Pattern Demo***\n");
        String input = null;
        int totalObjects = 0;

        while (true)
        {
            System.out.println("Enter your choice( Type 'a' for Bus, 'b'
            for Train ) ");
            Scanner scanner=new Scanner(System.in);
            input = scanner.nextLine();
            Vehicle vehicle = null;
            switch (input.toLowerCase())
            {
            case "a":
                vehicle = new Bus();
                break;
            case "b":
                vehicle = new Train();
                break;
            }
            totalObjects = Bus.busCount + Train.trainCount;

            vehicle.travel();
            System.out.println("Total number of objects created in the
            system is : "+ totalObjects);
        }
    }
}
```

Output with Valid Inputs

```
***Need for Null Object Pattern Demo***

Enter your choice( Type 'a' for Bus, 'b' for Train )
a
Let us travel with a bus
Total number of objects created in the system is : 1
Enter your choice( Type 'a' for Bus, 'b' for Train )
b
Let us travel with a train
Total number of objects created in the system is : 2
Enter your choice( Type 'a' for Bus, 'b' for Train )
b
Let us travel with a train
Total number of objects created in the system is : 3
Enter your choice( Type 'a' for Bus, 'b' for Train )
```

Analysis with an Unwanted Input

Let's assume that by mistake, the user has supplied a different character 'd' now as shown below:

```
***Need for Null Object Pattern Demo***

Enter your choice( Type 'a' for Bus, 'b' for Train )
a
Let us travel with a bus
Total number of objects created in the system is : 1
Enter your choice( Type 'a' for Bus, 'b' for Train )
b
Let us travel with a train
Total number of objects created in the system is : 2
Enter your choice( Type 'a' for Bus, 'b' for Train )
b
```

```
Let us travel with a train
Total number of objects created in the system is : 3
Enter your choice( Type 'a' for Bus, 'b' for Train )
d
```

Encountered Exception

This time, you receive the `System.NullPointerException` runtime exception.

```
Enter your choice( Type 'a' for Bus, 'b' for Train )
d
Exception in thread "main" java.lang.NullPointerException
    at jdp2e.nullobject.context.demo.NeedForNullObjectPattern.
    main(NeedForNullObjectPattern.java:61)
```

Immediate Remedy

The immediate remedy that may come in your mind is to do a null check before you invoke the operation as follows:

```
//A immediate remedy
if(vehicle !=null)
{
    vehicle.travel();
}
```

Analysis

The prior solution works in this case. But think of an enterprise application. If you need to do null checks for each possible scenario, you may need to have a larger number of `if` conditions to evaluate each time you proceed, and this approach makes your code dirty. At the same time, you may notice the side effects of a difficult maintenance also. The concept of null object pattern is useful in similar cases.

Real-World Example

Let's consider a real-life scenario with a washing machine. A washing machine can wash properly if the door is closed and there is a smooth water supply without any internal leakage. But suppose, in one occasion, you forget to close the door or stopped the water supply in between. The washing machine should not damage itself in those situations. It can beep some alarm to draw your attention and indicate that there is no water at present or the door is still open.

Computer-World Example

Assume that in a client server architecture, the server does some kinds of processing based on the client input. The server should be intelligent enough, so that it does not initiate any calculation unnecessarily. Prior processing the input, it may want to do a cross verification to ensure whether it needs to start the process at all or it should ignore an invalid input. You may notice the use of the command pattern with a null object pattern in such a case.

Basically, in an enterprise application, you can avoid a large number of null checks and `if/else` blocks using this design pattern. The following implementation can give you a nice overview about this pattern.

Note In Java, you may have seen the use of various adapter classes in java.awt.event package. These classes can be thought closer to null object pattern. For example, consider the MouseMotionAdapter class. It is an abstract class but contains methods with empty bodies like mouseDragged(MouseEvent e){}, mouseMoved(MouseEvent e){}. But since the adapter class is tagged with abstract keyword, you cannot directly create objects of the class.

Illustration

As before, in the following implementation, let's assume that you have two types of vehicles: bus and train. A client can opt for a bus or a train through different input: "a" or "b". If by mistake, the user supplies any invalid data (i.e., any input other than "a"

or "b" in this case), he cannot travel at all. The application ignores an invalid input by doing nothing *using a NullVehicle object.* In the following example, I'll not create these NullVehicle objects repeatedly. Once it is created, I'll simply reuse that object.

Class Diagram

Figure 25-1 shows the class diagram. (The concept is implemented with a singleton pattern, so that, you can avoid unnecessary object creations).

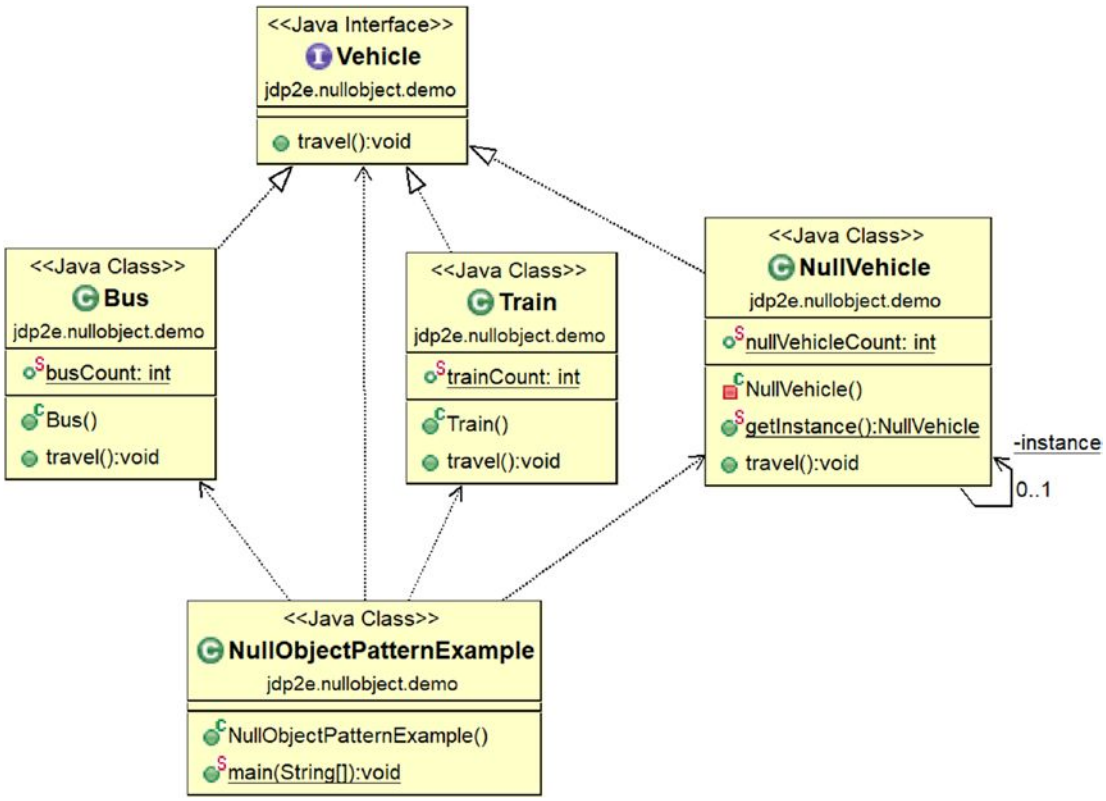


Figure 25-1. *Class diagram*

Package Explorer View

Figure 25-2 shows the high-level structure of the program.

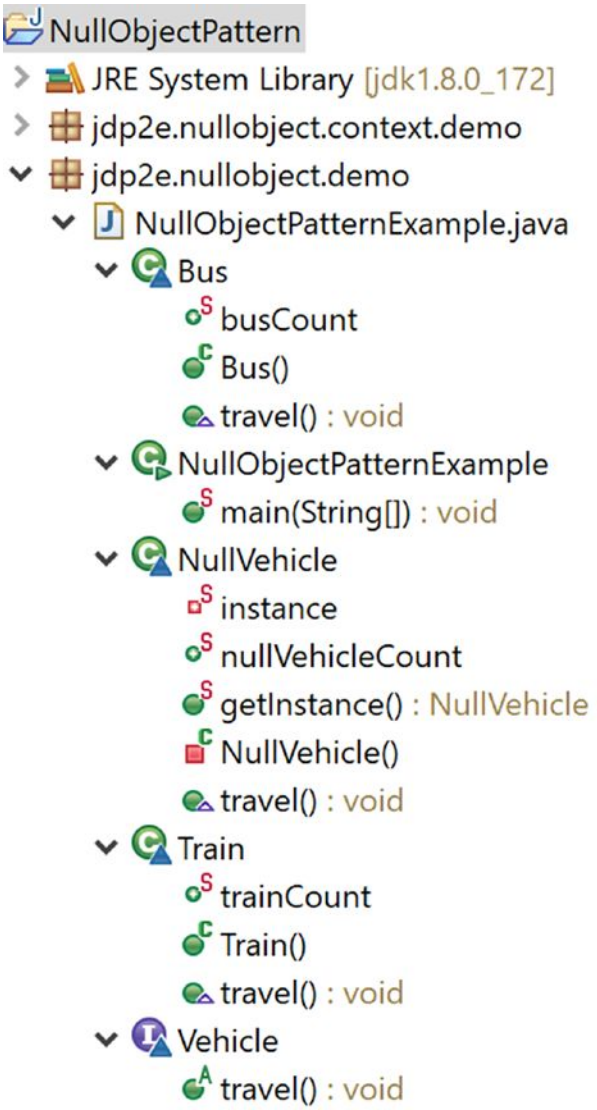


Figure 25-2. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.nullobject.demo;
import java.util.Scanner;

interface Vehicle
{
    void travel();
}
class Bus implements Vehicle
{
    public static int busCount = 0;
    public Bus()
    {
        busCount++;
    }
    @Override
    public void travel()
    {
        System.out.println("Let us travel with a bus");
    }
}
class Train implements Vehicle
{
    public static int trainCount = 0;
    public Train()
    {
        trainCount++;
    }
    @Override
    public void travel()
    {
        System.out.println("Let us travel with a train");
    }
}
```

```
class NullVehicle implements Vehicle
{
    //Early initialization
    private static  NullVehicle instance = new NullVehicle();
    public static int nullVehicleCount;
    //Making constructor private to prevent the use of "new"
    private NullVehicle()
    {
        nullVehicleCount++;
        System.out.println(" A null vehicle object created.Currently null
        vehicle count is :  "+nullVehicleCount);
    }
    // Global point of access.
    public static NullVehicle getInstance()
    {
        //System.out.println("We already have an instance now. Use it.");
        return instance;
    }
    @Override
    public void travel()
    {
        //Do Nothing
    }
}

public class NullObjectPatternExample  {

    public static void main(String[] args) {
        System.out.println("***Null Object Pattern Demo***\n");
        String input = "dummyInput";
        int totalObjects = 0;
        Scanner scanner;
        while(!input.toLowerCase().contains("exit"))
        {
            System.out.println("Enter your choice( Type 'a' for Bus, 'b'
            for Train.Type 'exit' to close the application. ) ");
            scanner=new Scanner(System.in);
```

```
            if(scanner.hasNextLine())
            {
                input = scanner.nextLine();
            }
            Vehicle vehicle = null;
            switch (input.toLowerCase())
            {
            case "a":
                vehicle = new Bus();
                break;
            case "b":
                vehicle = new Train();
                break;
            case "exit":
                System.out.println("Closing the application");
                vehicle = NullVehicle.getInstance();
                break;
            default:
                System.out.println("Invalid input");
                vehicle =  NullVehicle.getInstance();
            }
            totalObjects = Bus.busCount + Train.trainCount+NullVehicle.null
            VehicleCount;
            //A immediate remedy
            //if(vehicle !=null)
            //{
            vehicle.travel();
            //}
            System.out.println("Total number of objects created in the
            system is : "+ totalObjects);
        }

    }
}
```

Output

Here's the output.

```
***Null Object Pattern Demo***

Enter your choice( Type 'a' for Bus, 'b' for Train.Type 'exit' to close the
application. )
a
 A null vehicle object created.Currently null vehicle count is :  1
Let us travel with a bus
Total number of objects created in the system is : 2
Enter your choice( Type 'a' for Bus, 'b' for Train.Type 'exit' to close the
application. )
b
Let us travel with a train
Total number of objects created in the system is : 3
Enter your choice( Type 'a' for Bus, 'b' for Train.Type 'exit' to close the
application. )
c
Invalid input
Total number of objects created in the system is : 3
Enter your choice( Type 'a' for Bus, 'b' for Train.Type 'exit' to close the
application. )
dfh
Invalid input
Total number of objects created in the system is : 3
Enter your choice( Type 'a' for Bus, 'b' for Train.Type 'exit' to close the
application. )
exit
Closing the application
Total number of objects created in the system is : 3
```

Analysis

- Invalid input and their effects are shown in bold.
- Apart from the initial case, notice that object count has not increased due to null vehicle objects or invalid input.
- I did not perform any null check this time (notice the commented line in the following segment of code).

```
//A immediate remedy
  //if(vehicle !=null)
  //{
   vehicle.travel();
  //}
```

- This time program execution is not interrupted due to the invalid user input.

Q&A Session

1. **At the beginning, I see that an additional object is created. Is it intentional?**

   To save memory, I followed a singleton design pattern mechanism that supports early initialization in the structure of the NullVehicle class. I do not want to create a NullVehicle object for each invalid input. It is very likely that the application may need to deal with a larger number of invalid input. If you do not guard this situation, a large number of NullVehicle objects reside in the system (which are basically useless) and those occupy more memory. As a result, you may notice some typical side effects (for example, the system becomes slow, etc.).

2. **To implement a simple null object pattern, I can ignore different object counters(that used in the prior example) and reduce lots of code. Is this correct?**

   Yes. Ideally, consider the following code segment.

```
//Another context
List<Vehicle> vehicleList=new ArrayList<Vehicle>();
vehicleList.add(new Bus());
vehicleList.add(new Train());
vehicleList.add(null);
for( Vehicle vehicle : vehicleList)
  {
      vehicle.travel();
  }
```

   You cannot loop through this code because you encounter the `java.lang.NullPointerException`.

   Note a class like the following.

```
class NullVehicle implements Vehicle
{
    @Override
    public void travel()
    {
        //Do nothing
    }
}
```

   And you code like this:

```
//Another context discussed in Q&A session
List<Vehicle> vehicleList=new ArrayList<Vehicle>();
vehicleList.add(new Bus());
vehicleList.add(new Train());
//vehicleList.add(null);
vehicleList.add(new NullVehicle());
```

```
for( Vehicle vehicle : vehicleList)
{
 vehicle.travel();
}
```

This time you can loop through smoothly. So, remember that the following structure prior to implementing a null object pattern (see Figure 25-3).

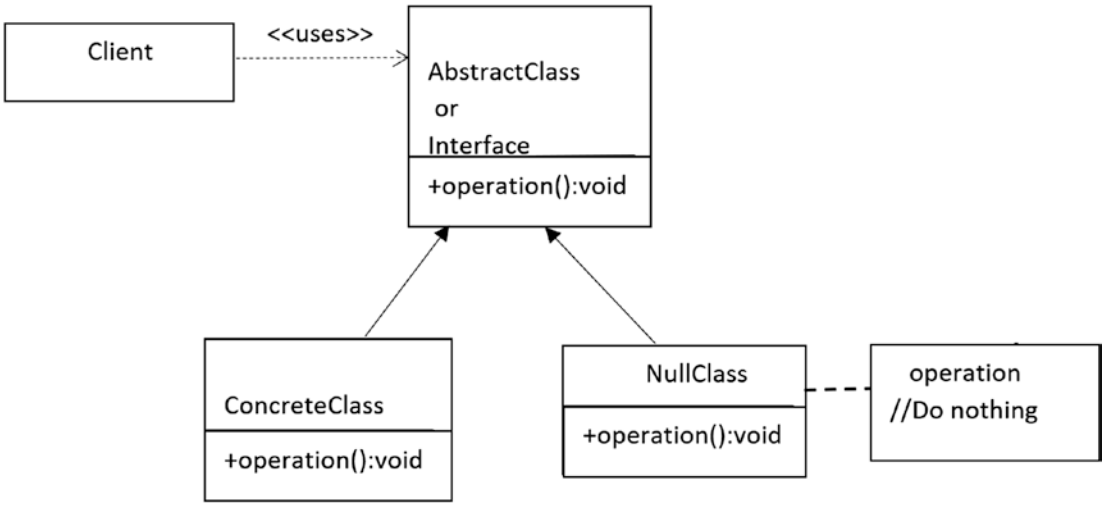


Figure 25-3. *The basic structure of a null object pattern*

3. **When should I use this pattern?**

   - The pattern is useful if you do not want to encounter with a NullPointerException in Java in some typical scenarios. (For example, if by mistake, you try to invoke a method of a null object.)
   - You can ignore lots of "null checks" in your code.
   - Absence of these null checks make your code cleaner and easily maintainable.

4. **What are the challenges associated with null object patterns?**

    - In some cases, you may want to get closure to the root cause of failure. So, if you throw a NullPointerException that makes more sense to you, you can always handle those exceptions in a `try/catch` or in a `try/catch/finally` block and update the log information accordingly.
    - The null object pattern basically helps us to implement a default behavior when you unconsciously want to deal with an object that is not present at all. But this approach may not suite every possible object in a system.
    - Incorrect implementation of a null object pattern can suppress true bags that may appear as normal in your program execution.
    - Creating a proper null object in every possible scenario may not be easy. In some classes, this may cause a change that influences the class methods.

5. **Null objects work like proxies. Is this correct?**

    No. In general, proxies act on real objects at some point of time and they may also provide some behavior. But a null object should not do any such thing.

6. **The null object pattern is always associated with NullPointerException. Is this correct?**

    The concept is same, but the exception name can be different or language specific. For example, in Java, you are using it to guard java.lang.NullPointerException but in a language like C#, you may use this pattern to guard System.NullReferenceException.

CHAPTER 26

MVC Pattern

Model-View-Controller (MVC) is an architectural pattern.

The use of this pattern is commonly seen in web applications or when we develop powerful user interfaces. But it is important to note that Trygve Reenskaug first described MVC in 1979 in a paper titled, "Applications Programming in Smalltalk-80TM: How to Use Model-View-Controller," which was before the World Wide Web era. At that time, there was no concept of web applications. But modern-day applications can be seen as an adaptation of that original concept. It is important to note that some developers believe that it is not a true design pattern, instead, they prefer to call it "MVC architecture."

Here you separate the user interface logic from the business logic and decouple the major components in such a way that they can be reused efficiently. This approach also promotes parallel development. One of the best rubrics for MVC is "We need SMART models, THIN controllers, and DUMB views." (`http://wiki.c2.com/?ModelViewController`)

Concept

From this introduction, it is apparent that the pattern consists of the three major components: Model, View, and Controller. Controller is placed between View and Model in such a way that Model and View can communicate to each other only through Controller. Here you separate the mechanism of how data is displayed from the mechanism of how the data is manipulated. Figure 26-1 shows a typical MVC architecture.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_26

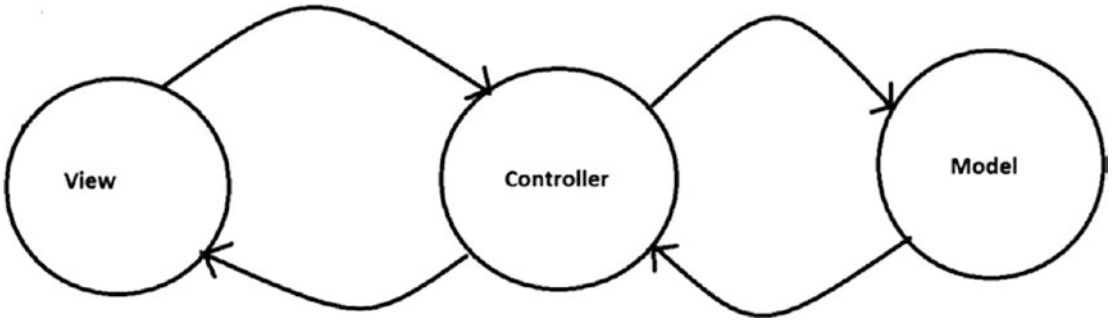


Figure 26-1. *A typical MVC architecture*

Key Points to Remember

The following are brief descriptions of the key components in this pattern.

- View represents the output. It is a presentation layer. Think of it as a user interface/GUI. You can design it with various technologies. For example, in a .NET application, you can use HTML, CSS, WPF, and so forth, and for a Java application, you can use AWT, Swing, JSF, JavaFX, and so forth.
- Model is the brain of your application. It manages the data and the business logic. It knows how to store and manage (or manipulate) the data, and how to handle the requests that come from controller. But this component is separated from the View component. A typical example is a database, a file system, or similar kinds of storage. It can be designed with JavaBeans, Oracle, SQL Server, DB2, Hadoop, MySQL, and so forth.
- Controller is the intermediary that accepts users input from the View component and passes the request to the model. When it gets a response from the model, it passes the data to the view. It can be designed with C#.NET, ASP.NET, VB.NET, Core Java, JSP, servlets, PHP, Ruby, Python, and so forth.

There are various implementations of this architecture in different applications. Some of them are as follows:

- You can have multiple views.
- Views can pass runtime values (e.g., using JavaScript) to controllers.
- Your controller can validate the user's input.

- Your controller can receive input in various ways. For example, it can get input from a web request via a URL, or you can pass the input by pressing a Submit button on a form.
- In some applications, Model components can update the View component.

Basically, you need to use this pattern to support your own needs. Figure 26-2, Figure 26-3, and Figure 26-4 show some of the known variations of an MVC architecture.

Variation 1

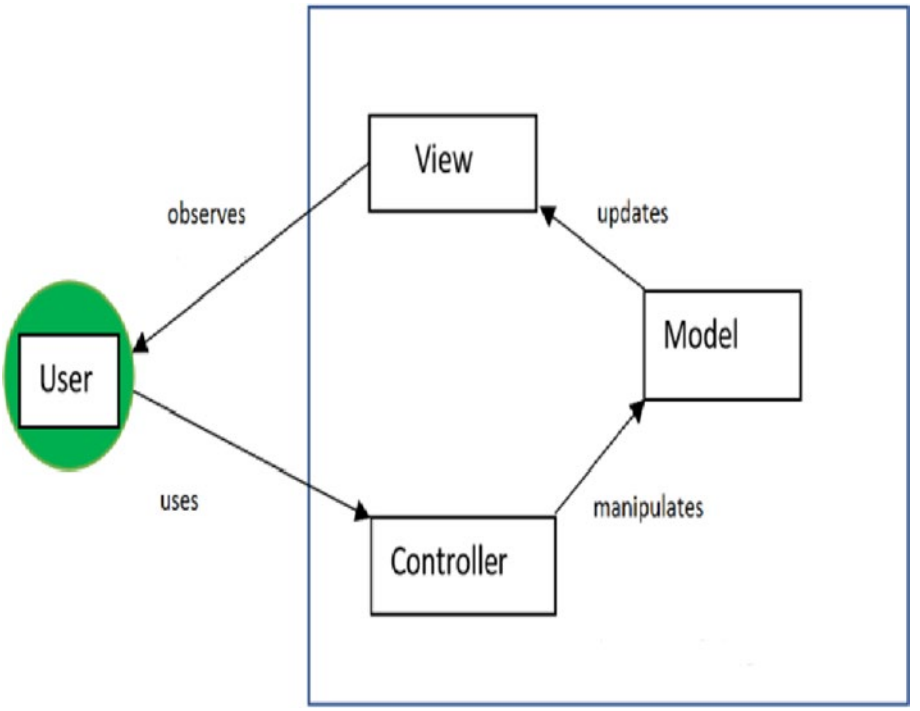


Figure 26-2. *A typical MVC framework*

Variation 2

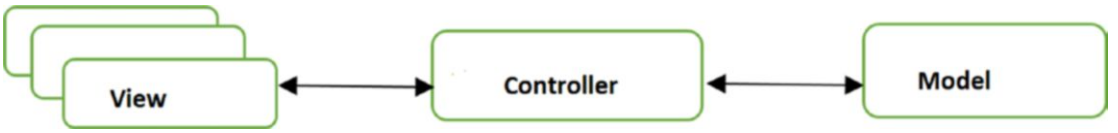


Figure 26-3. *A MVC framework with multiple views*

Variation 3

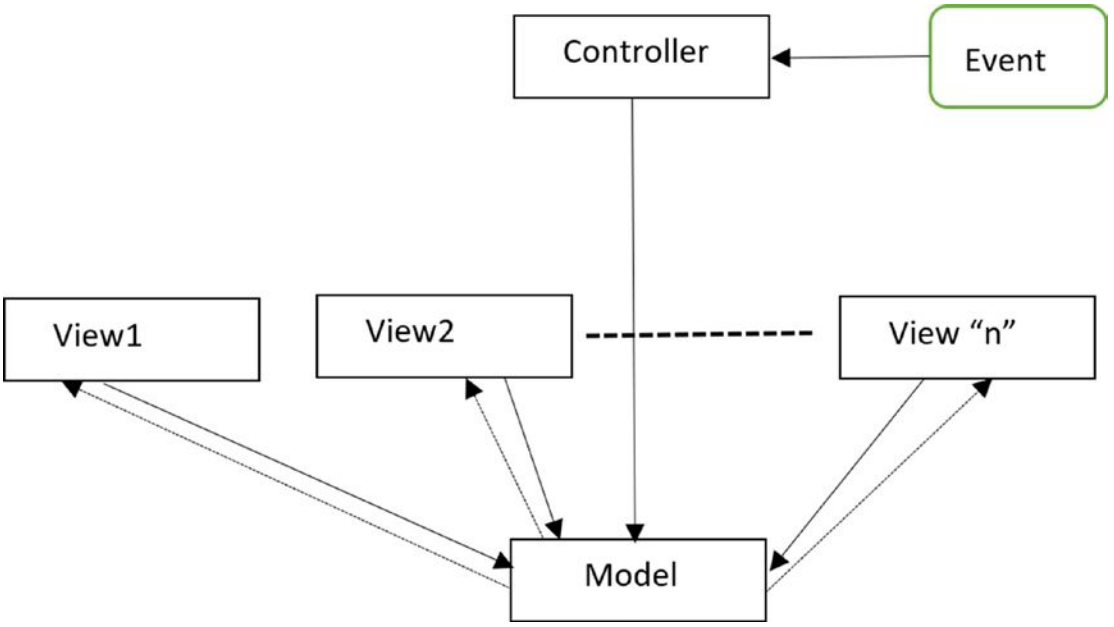


Figure 26-4. *An MVC pattern implemented with an observer pattern/ event-based mechanism*

My favorite description of MVC comes from Connelly Barnes, who states, "An easy way to understand MVC: the model is the data, the view is the window on the screen, and the controller is the glue between the two." (http://wiki.c2.com/?ModelViewController)

Real-World Example

Let's revisit our template method pattern's real-life example. But this time you interpret it differently. I said that in a restaurant, based on customer input, a chef can vary the taste and make the final products. The customers do not place their orders directly to the chef. The customers see the menu card (view), may consult with the waiter/waitress, and place their order. The waiter/waitress passes the order slip to the chef, who gathers the required materials from the restaurant's kitchen (similar to storehouses/computer databases). Once prepared, the waiter/waitress carries the plate to the customer's table. So, you can consider the role of the waiter/waitress as the controller, and the chef with their kitchen as the model (and the food preparation materials as data).

Computer-World Example

Many web programming frameworks uses the concept of MVC framework. Some of the typical example include Django, Ruby on Rails, ASP.NET, and so forth. For example, a typical ASP.NET MVC project has the structure shown in Figure 26-5.

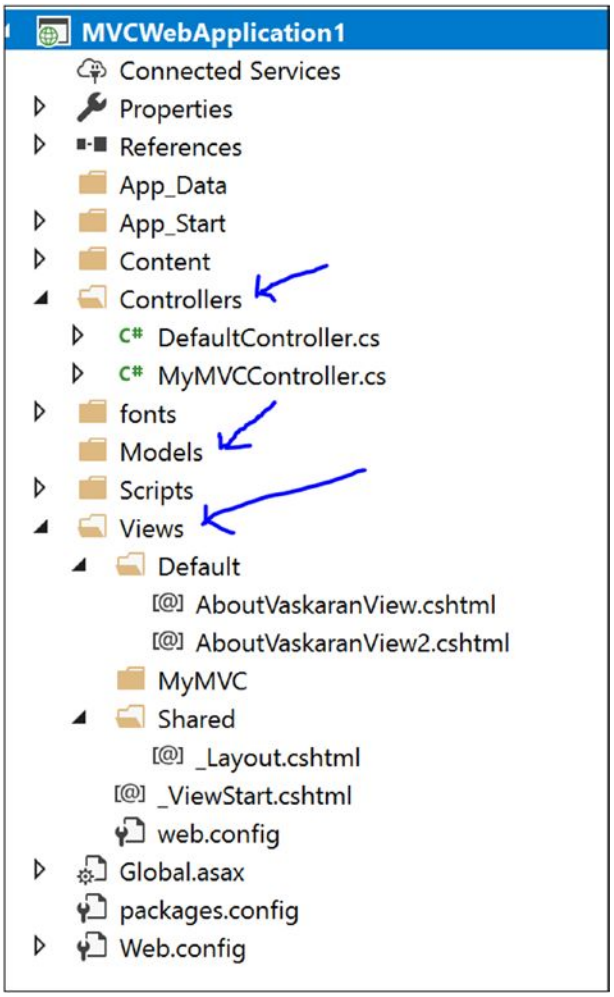


Figure 26-5. *A typical MVC structure in a ASP.NET project*

But it should be noted that different technologies can follow different structure and so, it is not necessary to get a folder structure with the strict naming convention like this. In the Java world, in a MVC architecture, you may notice the use of Java servlets as controllers and JavaBeans as models, whereas JSPs create different views.

Illustration

Most of the time, you want to use the concept of MVC with technologies that can give you built-in support and that can do a lot of ground work for you. In that case, you may need to learn new terminologies. In Java applications, you may want to use Swing or JavaFX, and so forth, for a better GUI.

Throughout this book, I used a console window to show output from different design pattern implementations. So, let's continue to use the console window as a view in the upcoming implementation because the focus here is on the MVC structure, not new technologies.

For simplicity and to match our theory, I divided the upcoming implementation into three basic parts: Model, View, and Controller. Once you look at the Package Explorer view, you see that separate packages are created to accomplish this task. Here are some important points.

- In this application, the requirement is very simple. There are employees who need to register themselves in an application/system. Initially, the application starts with three different registered employees: Amit, Jon, and Sam. At any time, you should be able to see the enrolled employees in the system.
- You can add a new employee or delete an employee from the registered employees list.
- A simple check is added in the Employee class to ensure that you are not adding an employee repeatedly in the application.
- To delete an employee from the registered list, you need to pass the employee ID in the client code, but the application will do nothing if an employee ID is not found in the registered list.

Now go through the implementation and consider the comments for your immediate reference.

Class Diagram

Figure 26-6 shows the class diagram. I omitted the client code dependencies to emphasize the core architecture.

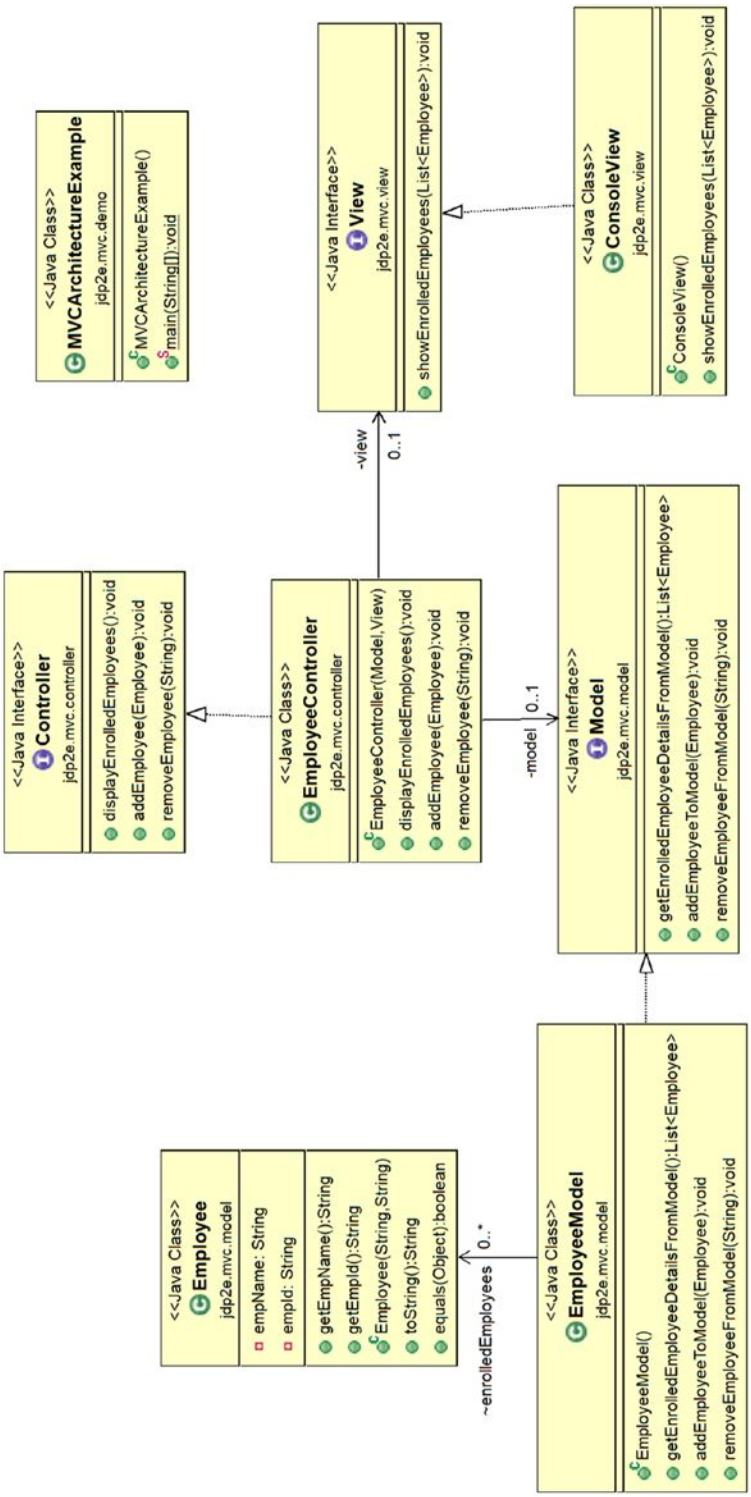


Figure 26-6. *Class diagram*

Package Explorer View

Figure 26-7 shows the high-level structure of the program.

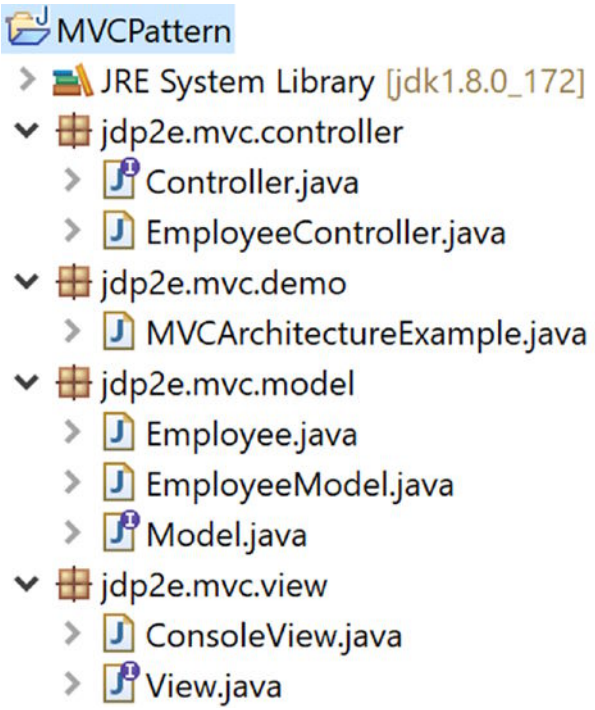


Figure 26-7. *Package Explorer view*

Implementation

Here is the implementation.

//Employee.java

```
package jdp2e.mvc.model;

//The key "data" in this application
public class Employee
{
    private String empName;
    private String empId;
```

```
    public String getEmpName() {
        return empName;
    }
    public String getEmpId() {
        return empId;
    }
    public Employee(String empName, String empId)
    {
        this.empName=empName;
        this.empId=empId;
    }
    @Override
    public String toString()
    {
        return empName + "'s employee id is: "+ empId ;
    }
    @Override
    //To check uniqueness.
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Employee)) return false;

        Employee empObject = (Employee) o;

        if (!empName.equals(empObject.empName)) return false;
        //cannot use the following for an int
        if (!empId.equals(empObject.empId)) return false;
        return true;
    }
}
```

//**Model.java**

```
package jdp2e.mvc.model;

import java.util.List;

//Model interface
public interface Model
{

    List<Employee> getEnrolledEmployeeDetailsFromModel();
    void addEmployeeToModel(Employee employeee);
    void removeEmployeeFromModel(String employeeId);
}
```

//**EmployeeModel.java**

```
package jdp2e.mvc.model;

import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

//EmployeeModel class

public class EmployeeModel implements Model
{
    List<Employee> enrolledEmployees;

    public EmployeeModel()
    {
        //Adding 3 employees at the beginning.
        enrolledEmployees = new ArrayList<Employee>();
        enrolledEmployees.add(new Employee("Amit","E1"));
        enrolledEmployees.add(new Employee("John","E2"));
        enrolledEmployees.add(new Employee("Sam","E3"));
    }

    public List<Employee> getEnrolledEmployeeDetailsFromModel()
    {
        return enrolledEmployees;
    }
```

```
//Adding an employee to the model(student list)
@Override
public void addEmployeeToModel(Employee employee)
{
    System.out.println("\nTrying to add an employee to the registered
    list.");
    if( !enrolledEmployees.contains(employee))
    {
        enrolledEmployees.add(employee);
        System.out.println(employee+" [added recently.]");
    }
    else
    {
        System.out.println(employee+" is already added in the
        system.");
    }
}
//Removing an employee from model(student list)
@Override
public void removeEmployeeFromModel(String employeeId)
{
    boolean flag=false;
    ListIterator<Employee> employeeIterator=enrolledEmployees.
    listIterator();
    System.out.println("\nTrying to remove an employee from the
    registered list.");
    while(employeeIterator.hasNext())
    {
        Employee removableEmployee=((Employee)employeeIterator.next());
        if(removableEmployee.getEmpId().equals(employeeId))
        {
            //To avoid ConcurrentModificationException,try to
            //remember to invoke remove() on the iterator but not on
            //the list.
            employeeIterator.remove();
```

```
                System.out.println("Employee " + removableEmployee.
                getEmpName()+ " with id "+ employeeId+" is removed now.");
                flag=true;
            }
        }
        if(flag==false)
        {
            System.out.println("###Employee Id " + employeeId +" Not
            found.###");
        }

    }

}
```

//View.java

```
package jdp2e.mvc.view;
import java.util.List;
import jdp2e.mvc.model.Employee;

public interface View
{
    void showEnrolledEmployees(List<Employee> enrolledEmployees);
}
```

//ConsoleView.java

```
package jdp2e.mvc.view;

import java.util.List;
import jdp2e.mvc.model.Employee;

//ConsoleView class

public class ConsoleView implements View
{
    @Override
    public void showEnrolledEmployees(List<Employee> enrolledEmployees)
```

```
    {
        System.out.println("\n ***This is a console view of currently
        enrolled employees.*** ");
        for( Employee employee : enrolledEmployees)
        {
            System.out.println(employee);
        }
        System.out.println("---------------------");
    }
}
```

//**Controller.java**

```
package jdp2e.mvc.controller;

import jdp2e.mvc.model.Employee;

//Controller
public interface Controller
{
    void displayEnrolledEmployees();
    void addEmployee(Employee employee);
    void removeEmployee(String employeeId);
}
```

//**EmployeeController.java**

```
package jdp2e.mvc.controller;

import java.util.List;

import jdp2e.mvc.model.*;
import jdp2e.mvc.view.*;

public class EmployeeController implements Controller
{
    private Model model;
    private View  view;
```

```
    public EmployeeController(Model model, View view)
    {
        this.model = model;
        this.view = view;
    }
    @Override
    public void displayEnrolledEmployees()
    {
        //Get data from Model
        List<Employee> enrolledEmployees = model.
        getEnrolledEmployeeDetailsFromModel();
        //Connect to View
        view.showEnrolledEmployees(enrolledEmployees);
    }

    //Sending a request to model to add an employee to the list.
    @Override
    public void addEmployee(Employee employee)
    {
        model.addEmployeeToModel(employee);
    }
    //Sending a request to model to remove an employee from the list.
    @Override
    public void removeEmployee(String employeeId)
    {
        model.removeEmployeeFromModel(employeeId);
    }
}
```

//**Client code**
//**MVCArchitectureExample.java**

```
package jdp2e.mvc.demo;
import jdp2e.mvc.model.*;
import jdp2e.mvc.view.*;
import jdp2e.mvc.controller.*;
```

```
public class MVCArchitectureExample {

    public static void main(String[] args) {
        System.out.println("***MVC architecture Demo***\n");
        //Model
        Model model  = new EmployeeModel();

        //View
        View view = new ConsoleView();

        //Controller
        Controller controller = new EmployeeController(model, view);
        controller.displayEnrolledEmployees();

        //Add an employee
        controller.addEmployee(new Employee("Kevin","E4"));
        controller.displayEnrolledEmployees();

        //Remove an existing employee using the employee id.
        controller.removeEmployee("E2");
        controller.displayEnrolledEmployees();

        //Cannot remove an  employee who does not belong to the list.
        controller.removeEmployee("E5");
        controller.displayEnrolledEmployees();

        //Avoiding duplicate entry
        controller.addEmployee(new Employee("Kevin","E4"));
    }
}
```

Output

Here is the output.

```
***MVC architecture Demo***

 ***This is a console view of currently enrolled employees.***
Amit's employee id is: E1
John's employee id is: E2
Sam's employee id is: E3
---------------------

Trying to add an employee to the registered list.
Kevin's employee id is: E4 [added recently.]

 ***This is a console view of currently enrolled employees.***
Amit's employee id is: E1
John's employee id is: E2
Sam's employee id is: E3
Kevin's employee id is: E4
---------------------

Trying to remove an employee from the registered list.
Employee John with id E2 is removed now.

 ***This is a console view of currently enrolled employees.***
Amit's employee id is: E1
Sam's employee id is: E3
Kevin's employee id is: E4
---------------------

Trying to remove an employee from the registered list.
###Employee Id E5 Not found.###
 ***This is a console view of currently enrolled employees.***
Amit's employee id is: E1
Sam's employee id is: E3
Kevin's employee id is: E4
---------------------

Trying to add an employee to the registered list.
Kevin's employee id is: E4 is already added in the system.
```

Q&A Session

1. **Suppose you have a programmer, a DBA, and a graphic designer. Can you guess their roles in a MVC architecture?**

   The graphic designer designs the view layer. The DBA makes the model and programmer works to make an intelligent controller.

2. **What are the key advantages of using MVC design patterns?**

   - "High cohesion and low coupling" is the slogan of MVC. Tight coupling between model and view is easily removed in this pattern. So, it can be easily extendable and reusable.
   - It supports parallel development.
   - You can also provide multiple runtime views.

3. **What are the challenges associated with MVC patterns?**

   - Requires highly skilled personnel.
   - It may not be suitable for a tiny application.
   - Developers need to be familiar with multiple languages/platforms/technologies.
   - Multiartifact consistency is a big concern because you are separating the overall project into three different parts.

4. **Can you provide multiple views in this implementation?**

   Sure. Let's add a new view called "Mobile view" in the application. Let's add this class inside the jdp2e.mvc.view package as follows.

```
package jdp2e.mvc.view;
import java.util.List;
import jdp2e.mvc.model.Employee;

//This class is added to discuss a question in "Q&A Session"

//MobileView class
```

```
public class MobileView implements View
{
    @Override
    public void showEnrolledEmployees(List<Employee>
    enrolledEmployees)
    {
        System.out.println("\n ***This is a mobile view of
        currently enrolled employees.*** ");
        System.out.println("Employee Id"+ "\t"+ " Employee Name");
        System.out.println("_____________________");
        for( Employee employee : enrolledEmployees)
        {
            System.out.println(employee.getEmpId() + "\t"+
            employee.getEmpName());
        }
        System.out.println("--------------------");
    }
}
```

The modified Package Explorer view is similar to Figure 26-8.

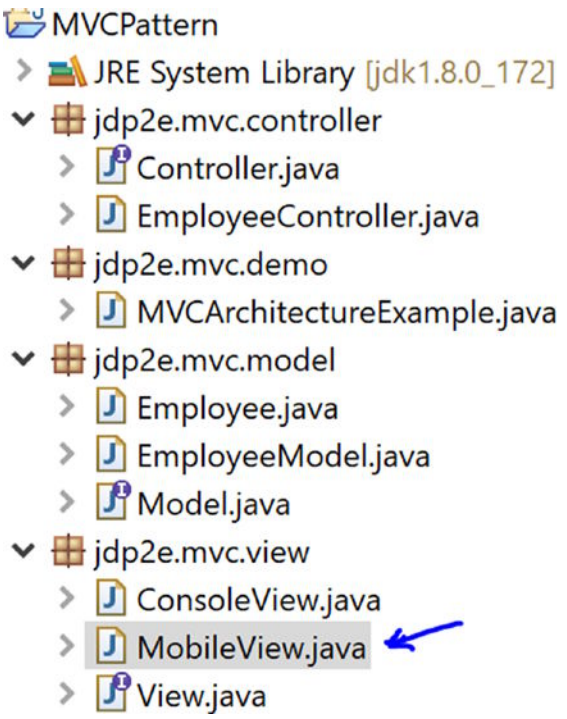


Figure 26-8. *Modified Package Explorer view*

Add the following segment of code at the end of your client code.

```
//This segment is addeed to discuss a question in "Q&A Session"
view = new MobileView();
controller = new EmployeeController(model, view);
controller.displayEnrolledEmployees();
```

Now if you run the application, you see the modified output.

Modified Output

Here is the modified output. The last part of your output shows the effect of your new changes. These changes are shown in bold.

```
***MVC architecture Demo***

 ***This is a console view of currently enrolled employees.***
Amit's employee id is: E1
```

```
John's employee id is: E2
Sam's employee id is: E3
--------------------

Trying to add an employee to the registered list.
Kevin's employee id is: E4 [added recently.]

 ***This is a console view of currently enrolled employees.***
Amit's employee id is: E1
John's employee id is: E2
Sam's employee id is: E3
Kevin's employee id is: E4
--------------------

Trying to remove an employee from the registered list.
Employee John with id E2 is removed now.

 ***This is a console view of currently enrolled employees.***
Amit's employee id is: E1
Sam's employee id is: E3
Kevin's employee id is: E4
--------------------

Trying to remove an employee from the registered list.
###Employee Id E5 Not found.###

 ***This is a console view of currently enrolled employees.***
Amit's employee id is: E1
Sam's employee id is: E3
Kevin's employee id is: E4
--------------------

Trying to add an employee to the registered list.
Kevin's employee id is: E4 is already added in the system.
```

```
 ***This is a mobile view of currently enrolled employees.***
Employee Id   Employee Name

_____________________

E1    Amit
E3    Sam
E4    Kevin
---------------------
```

PART III

Final Discussions on Design Patterns

CHAPTER 27

Criticisms of Design Patterns

In this chapter, I present some of the criticisms of design patterns. Reading about the criticisms can offer real value. If you think critically about patterns before you design your software, you can predict your "return on investment" to some degree. Design patterns basically help you benefit from another people's experience. This is often called *experience reuse*. You learn how they solved challenges, how they tried to adapt new behaviors in their systems, and so on. A pattern may not perfectly fit into your work, but if you concentrate on the best practices as well as the problems of a pattern at the beginning, you are more likely to make a better application.

The following are some of the criticisms of patterns.

- Christopher Alexander considered the domain that did not change a lot over the years (compared to software industry). On the contrary, software industry is always changing and the changes in software development are much faster than any other domain. So, you cannot start from the domain (of buildings and towns) that Christopher Alexander considered.
- The way you write program in today's world is different and the facilities that you have nowadays are much more compared to old days of programming. So, when you extract patterns based on some old practices, you basically show additional respect to them.
- Many of the patterns are close to each other. And there are always pros and cons associated with each of the patterns (I discussed about them in the "Q&A Sessions" at the end of each chapter.)The pitfall in one case can be a real virtue in a different case.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_27

- The pattern that is giving you the satisfactory results today, can be a big burden to you in the near future due to the "continuous changes" in the software industry.
- It is very unlikely that an infinite number of requirements can be well designed with a finite number of design patterns.
- Designing a software is basically an art. And there is no definition or criteria for best art.
- Design patterns give you the idea but not the implementations (like libraries or frameworks). Each human mind is unique. So, each engineer may have his/her own preferences for implementing a similar concept, and that can create chaos in a team.
- Consider a simple example. Patterns encourage people to code to a super type (abstract class/ interface). But for a simple application where you know that there are no upcoming changes, or the application is created for a demo purposes only, this rule may not make much sense.
- In a similar way, in some small applications, you may find that enforcing the rules of design patterns are increasing your code size and maintenance costs.
- Erasing the old and adapting the new is not always easy. For example, when you first learned about inheritance, you were excited. You probably wanted to use it in many ways and were seeing only the benefits from the concept. But later when you started experimenting with design patterns, you started learning that in many cases, compositions are preferred over inheritance. This shifting of gears is not easy.
- Design patterns are based on some of the key principles, and one of them is to *identify the code that may vary and then separate it from rest of the code*. It sounds very good from theoretical perspective. But in real world implementations, who guarantees you that your judgment is perfect? Software industry always changes, and it needs to adapt with new requirements/demands.

- Many patterns are already integrated with modern day languages. Instead of implementing the pattern from the scratch, you can use the built-in support in the language constructs. For example, you may notice that each of the patterns has JDK implementations in some context.
- Inappropriate use of patterns can lead to antipatterns (e.g., an inappropriate use of mediator pattern can lead to a "God Class" antipattern). I give a brief overview of antipatterns in Chapter 28.
- Many people believe that the concepts of design patterns simply indicate that a programming language may need additional features. So, patterns have less significance with the increasing capabilities of modern-day programming languages. Wikipedia says that computer scientist Peter Norvig believes that 16 out of the 23 patterns in the GoF's design patterns are simplified or eliminated via direct language support in Lisp or Dylan. You can see some similar thoughts at https://en.wikipedia.org/wiki/Software_design_pattern.
- At the end, design patterns basically help you to get benefit from others experience. You are getting their thoughts, you come to know how they encountered the challenges, how they tried to adapt new behaviors in their systems, and so forth. But you start with the assumption that a beginner or relatively less-experienced person cannot solve a problem better than his/her seniors. In some specific occasions, a relatively less experienced person can have a better vision than his seniors, and he can prove himself more effective in the future.

Q&A Session

1. **Is there a catalog for these patterns?**

   I started with the GoF's 23 design patterns and then discussed three more patterns in this book. The GoF's catalog is considered the most fundamental pattern catalog.

But there are definitely many other catalogs that focus on particular domains.

The Portland Patterns Repository and The Hillside Group's website are well-known in this context. You can get valuable insights and thoughts from these resources at http://wiki.c2.com/?WelcomeVisitors and https://hillside.net/patterns/patterns-catalog.

The Hillside Group's website also notes its various conferences and workshops.

Note At the time of writing, the URLs in the book worked fine but some of these links and the policies to access the links may change in the future.

---

2. **Why are you not covering other patterns?**

   These are my personal beliefs:

   - Computer science keeps growing, and you keep getting new patterns.
   - If you are not familiar with the fundamental patterns, you cannot evaluate the true needs of the remaining or upcoming patterns. For example, if you know MVC well, you can see how it is different than Model-View-Presenter (MVP) and understand why MVP is needed.
   - The book is already fat. The detailed discussion of each pattern would need many more pages, which would make the size of the book too big to digest.

     So, in this book, I focused on fundamental patterns that are still relevant in today's programming world.

3. **I often see the term "force" with the description of design patterns. What does it mean?**

   It is the criteria based on which developers justify their developments. Broadly, your target and current constraints are two important parts of your force. Therefore, when you develop your application, you can justify your development with these parts.

4. **In various forums, I have seen people fighting about the pattern definition and say something like, "A pattern is a proven solution to a problem in a context." What does it mean?**

   This is a simple and easy-to-remember definition of what a pattern is. But simply breaking it down into three parts (problem, context, and solution) is not enough.

   As an example, suppose you are visiting to Airport and you are in a hurry. Suddenly, you discover that you have left your boarding pass at home. Let's analyze the situation:

   Problem: You need to reach airport on time.

   Context: Left the boarding pass at home.

   The Solution that may come to mind: Turn back, go at a high speed and rush toward home to get the boarding pass.

   This solution may work one time, but can you apply the same procedure repeatedly? You know the answer. It is not an intelligent solution because it depends on how much time you have to collect the pass from home and go back to the airport. It also depends on the current traffic on the road and many other factors. So, even if you can get the success for one time, you may want to prepare yourself for a better solution for a similar situation in future.

   So, try to understand the meaning, intent, context, and so on, to understand a pattern clearly.

5. **Sometimes I am confused to see similar UML diagrams for two different patterns. Also, I am further confused with the classification of the patterns in many cases. How can I overcome this?**

   This is perfectly natural. The more you read and analyze the implementations and the more you try to understand the intent behind the designs, the distinctions among them will be clearer to you.

6. **When should I consider writing a new pattern?**

   Writing a new pattern is not easy. You need to study a lot and evaluate the available patterns before you put your effort. But if you do not find any existing pattern to serve your domain-specific need, you may need to write your own pattern. It would be very good if your solution passes the "rule of three" (which basically says that to get the tag "pattern," a solution needs to be successfully applied in a real-world solution at least three times). Once you have done this, you can let others know about it, participate in discussion forums and take feedbacks from others. This activity can help both you and the development community.

CHAPTER 28

AntiPatterns: Avoid the Common Mistakes

The discussion of design patterns cannot be completed without antipatterns. This chapter briefly overviews antipatterns. Let's start.

What Is an Antipattern?

In real-world application development, you may follow approaches that are very attractive at first, but in the long run, they cause problems. For example, you try to do a quick fix to meet a delivery deadline, but if you are not aware of the potential pitfalls, you may pay a big price.

Antipatterns alert you about common mistakes that lead to a bad solution. Knowing them helps you take precautionary measures. The proverb "prevention is better than cure" very much fits in this context.

Note Antipatterns alert you to common mistakes by describing how attractive approaches can make your life difficult in future. At the same time, they suggest alternate solutions that may seem tough or ugly at the beginning but ultimately help you build a better solution. In short, antipatterns identify problems with established practices and they can map general situations to a specific class of highly productive solutions. They can also provide better plans to reverse some bad practices and make healthy solutions.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_28

Brief History of Antipatterns

The original idea of design patterns came from building architect Christopher Alexander. He shared his ideas for the construction of buildings within well-planned towns. Gradually, these concepts entered into software development and they gained popularity through the leading-edge software developers like Ward Cunningham and Kent Beck. In 1994, the idea of design patterns entered mainstream object-oriented software development through an industry conference on design patterns, known as Pattern Languages of Program Design (PLoP). It was hosted by the Hillside Group. Jim Coplien's paper "A Development Process Generative Pattern Language" is a famous one in this context. And with the launch of the classic text *Design Patterns: Elements of Reusable Object–Oriented Software* by the Gang of Four, design patterns became extremely popular.

Undoubtedly, these great design patterns helped (and are still helping) programmers to develop the high-quality software. But people started noticing the negative impacts also. A common example is that many developers wanted to show their expertise without the true evaluation or the consequences of these patterns in their specific domains. As an obvious side effect, patterns were implanted in the wrong context, which produced low-quality software, and ultimately caused big penalties to the developers and their companies.

So, the software industry needed to focus on the negative consequences of these mistakes, and eventually, the idea of antipatterns evolved. Many experts started contributing to this field, but the first well-formed model came through Michael Akroyd's presentation, "AntiPatterns: Vaccinations against Object Misuse." It was the antithesis of the GoF's design patterns.

The term *antipattern* became popular with the authors (Brown, Malveau, McCormickIII, Mowbray) in their book *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (Wiley, 1998). Later, Scott Thomas joined their group. They said,

> *Because AntiPatterns have had so many contributors, it would be unfair to assign the original idea for AntiPatterns to a single source. Rather, AntiPatterns are a natural step in complementing the work of the design pattern movement and extending the design pattern model.*

Examples of Antipatterns

The following are some examples of the antipatterns and the concepts/mindsets behind them.

- *Over Use of Patterns*: Developers may try to use patterns at any cost, regardless of whether it is appropriate or not.
- *God Class*: A big object that tries to control almost everything with many unrelated methods. An inappropriate use of the mediator pattern may end up with this antipattern.
- *Not Invented Here*: I am a big company and I want to build everything from scratch. Although there is already a library available that was developed by a smaller company, I'll not use that. I will make everything of my own and once it is developed, I'll use my brand value to announce, "Hey Guys. The ultimate library is launched for you."
- *Zero Means Null*: A common example includes developers who think that no one wants to be at latitude zero, longitude zero. Another common variation is when a programmer uses :1, 999 or anything like that to represent an inappropriate integer value. Another erroneous use case is observed when a user treats "09/09/9999" as a null date in an application. So, in these cases, if the user needs to have the numbers :1,999 or the date "09/09/9999", he is unable to get them.
- *Golden Hammer*: Mr. X believes that technology T is always best. So, if he needs to develop a new system (that demands new learning), he still prefers T, even if it is inappropriate. He thinks, "I do not need to learn any more technology if I can somehow manage it with T."
- *Management by Numbers*: The greater the number of commits, the greater the number of lines of code, or the greater the number of defects fixed are the signs of a great developer. Bill Gates said, "Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

- *Shoot the Messenger*: You are already under pressure and the program deadline is approaching. There is a smart tester who always finds typical defects that are hard to fix. So, at this stage, you do not want to involve him because he will find more defects and the deadline may be missed.
- *Swiss Army Knife*: Demand a product that can serve the customer's every need. Or make a drug that cures all illnesses. Or design software that serves a wide range of customers with varying needs. It does not matter how complex the interface is.
- *Copy and Paste Programming*: I need to solve a problem but I already have a piece of code to deal with a similar situation. So, I can copy the old code that is currently working and start modifying it if necessary. But when you start from an existing copy, you essentially inherit all the potential bugs associated with it. Also, if the original code needs to be modified in the future, you need to implement the modification in multiple places. This approach also violates the *Don't Repeat Yourself (DRY)* principle.
- *Architects Don't Code*: I am an architect. My time is valuable. I'll only show paths or give a great lecture on coding. There are enough implementers who should implement my idea. *Architects Play Golf* is a sister of this antipattern.
- *Hide and Hover*: Do not expose all edits or delete links until he/she hovers the element.
- *Disguised Links and Ads*: Earn revenue when users click a link or an advertisement, but they cannot get what they want.

Note Nowadays, you can learn about various antipatterns from different websites/sources. For example, a Wikipedia page talks about various antipatterns (see `https://en.wikipedia.org/wiki/Antipattern`). You can also get a detailed list of the antipattern catalog at `http://wiki.c2.com/?AntiPatternsCatalog` to learn more. You may also notice that the concept of antipatterns is not limited to object-oriented programming.

Types of Antipatterns

Antipatterns can belong in more than one category. Even a typical antipattern can belong in more than one category.

The following are some common classifications.

- *Architectural antipatterns*: The Swiss Army Knife antipattern is an example in this category.
- *Development antipatterns*: The God Class and Over Use of Patterns are examples in this category.
- *Management antipatterns*: The Shoot the Messenger antipattern falls in this category.
- *Organizational antipatterns*: Architects Don't Code and Architects Play Golf are examples in this category.
- *User Interface antipatterns*: Examples include Disguised Links and Ads.

Note Disguised Links and Ads are also called as *dark patterns*.

Q&A Session

1. **How are antipatterns related to design patterns?**

   With design patterns, you reuse the experiences of others who came before you. When you start blindly using those concepts for the sake of use only, you fall into the traps of *reuse of recurring solutions*. This can lead you to a bad situation. And then you learn that your return on investment keeps decreasing but maintenance costs keep increasing. The apparently easy and standard solutions (or patterns) may cause more problems for you in the future.

2. **A design pattern may turn into an antipattern. Is this correct?**

   Yes. If you apply a design pattern in the wrong context, that can cause more trouble than the problem it solves. Eventually it will turn into an antipattern. So, understanding the nature and context of the problem is very important.

3. **Antipatterns are related to software developers only. Is this correct?**

   No. The usefulness of an antipattern is not limited to developers; it may be applicable to others also; for example, antipatterns are useful to managers and technical architects also.

4. **Even if you do not get much benefit from antipatterns now, they can help you adapt new features easily with fewer maintenance costs in future. Is this correct?**

   Yes.

5. **What are the probable causes of antipatterns?**

   It can come from various sources/mindsets. The following are a few common examples.

   - "We need to deliver the product as soon as possible."
   - "We do not need to analyze the impact right now."
   - "I am an expert of reuse. I know design patterns very well."
   - "We will use the latest technologies and features to impress our customers. We do not need to care about legacy systems."
   - "More complicated code will reflect my expertise on the subject."

6. **Discuss some symptoms of antipatterns.**

   In object-oriented programming, the most common symptom is your system cannot easily adapt a new feature. Also, maintenance costs are keep increasing. You may also notice that you have lost the power of key object-oriented features like inheritance, polymorphism, and so forth.

Apart from these, you may notice some/all of the following symptoms.

- Use of global variables
- Code duplication
- Limited/no reuse of code
- One big class (God Class)
- A large number of parameterless methods, etc.

7. **What is the remedy if you detect an antipattern?**

   You may need to follow a refactored and better solution. For example, the following are some solutions for avoiding antipatterns.

   Golden Hammer: You may try to educate Mr. X through proper training.

   Zero Means Null: You can use an additional boolean variable to indicate the null value properly.

   Management by Numbers: Numbers are good if you use them wisely. You cannot judge the ability of a programmer by the number of defects he/she fixes per week. The quality is also important. A typical example includes fixing a simple UI layout is much easy compared to fix a critical memory leak in the system. Consider another example. "More number of tests are passing" does not indicate that your system is more stable unless the tests exercise different code paths/branches.

   Shoot the Messenger: Welcome the tester and involve him immediately. He can find typical defects early, and you can avoid last-moment surprises.

   Copy and Paste Programming: Instead for searching a quick solution, you can refactor your code. You can also make a common place to maintain the frequently used methods to avoid duplicates and provide easier maintenance.

Architects Don't Code: Involve architects in parts of the implementation phase. This can help both the organization and the architects. This activity can give them a clearer picture about the true functionalities of the product.

8. **What do you mean by refactoring?**

   In the coding world, the term *refactoring* means improving the design of existing code without changing the external behavior of the system/application. This process helps you get more readable code. At the same time, the code should be more adaptable to new requirements (or change requests) and they should be more maintainable.

CHAPTER 29

FAQs

This chapter is a subset of the "Q&A Session" sections in all the chapters in this book. Many of these questions were not discussed in certain chapters because the related patterns were not yet covered. So, it is highly recommended that in addition to the following Q&As, you go through all the "Q&A Session" sections in the book for a better understanding of all the patterns.

1. **Which design pattern do you like the most?**

   It depends on many factors, such as the context, situation, demand, constraints, and so on. If you know about all the patterns, you have more options to choose from.

2. **Why should developers use design patterns?**

   They are reusable solutions for software design problems that appear repeatedly in real-world software development.

3. **What is the difference between the command and the memento patterns?**

   All actions are stored for the command pattern, but the memento pattern saves the state only on request. Additionally, the command pattern has undo and redo operations for every action, but the memento pattern does not need that.

4. **What is the difference between the facade pattern and the builder pattern?**

   The aim of the facade pattern is to make a specific portion of code easier to use. It abstracts details away from the developer.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_29

The builder pattern separates the construction of an object from its representation. In Chapter 3, the director is calling the same `construct()` method to create different types of vehicles. In other words, you can use the same construction process to create multiple types.

5. **What is the difference between the builder pattern and the strategy pattern? They have similar UML representations.**

   You need to understand intent first. The builder pattern falls into the category of creational patterns, and the strategy pattern falls into the category of behavioral patterns. Their areas of focus are different. With the builder pattern, you can use the same construction process to create multiple types, and with the strategy pattern, you have the freedom to select an algorithm at runtime.

6. **What is the difference between the command pattern and the interpreter pattern?**

   In the command pattern, the commands are basically objects. In the interpreter pattern, the commands are sentences. Sometimes the interpreter pattern may look convenient, but you must keep in mind the cost of building an interpreter.

7. **What is the difference between the chain-of-responsibility pattern and the observer pattern?**

   In observer patterns, all registered users are notified/get request (for the change in subject) in parallel, but in the chain-of-responsibility pattern, you may not reach the end of chain, so all users need not handle the same scenario. The request can be processed much earlier by a user who is placed at the beginning of the chain.

8. **What is the difference between the chain-of-responsibility pattern and the decorator pattern?**

   They are not same at all but you may feel that they are similar in the structures. Similar to the previous differences, in the chain-of-responsibility pattern, only one class handles the request, but

in the decorator pattern, all classes handle the request. You must remember that decorators are effective in the context of adding and removing responsibilities only, and if you can combine the decorator pattern with the single responsibility principle, you can add/remove a single responsibility at runtime.

9. **What is the difference between the mediator pattern and the observer pattern?**

   The GoF says, "These are competing patterns. The difference between them is that Observer distributes communication by introducing observer and subject objects, whereas a mediator object encapsulates the communication between other objects." I suggest you consider the mediator pattern example in Chapter 21. In this example, two workers are always getting messages from their boss. It doesn't matter whether they like those messages. But if they are simple observers, then they should have the option to unregister their boss's control of them, effectively saying "I do not want to see messages from the boss/Raghu."

   The GoF also found that you may face fewer challenges when you make reusable observers and subjects compared to when you make reusable mediators. But regarding the flow of communication, the mediator pattern scores higher than the observer pattern.

10. **Which do you prefer—a singleton class or a static class?**

    The first thing to remember is that Java does not support a top-level static class. You can create objects of a singleton class, which is not possible with a static class. So, the concepts of inheritance and polymorphism can be implemented with a singleton class. Now let's consider a language that supports a full-phased static class(like C#). In that case, some developers believe that mocking a static class (e.g., consider unit testing scenarios) in a real-world application is challenging.

11. **How can you distinguish between proxies and adapters?**

    Proxies work on similar interfaces as their subjects but adapters work on different interfaces (to the objects they adapt).

12. **How are proxies different from decorators?**

    There are different types of proxies, and they vary by their implementations. So, it may appear that some implementations are close to decorators. For example, a protection proxy might be implemented like a decorator. But you must remember that decorators focus on adding responsibilities, while proxies focus on controlling the access to an object.

13. **How are mediators different from facades?**

    In general, both simplify a complex system. In a mediator pattern, a two-way connection exists between a mediator and the internal subsystems, whereas in a facade pattern, you provide a one-way connection (the subsystems do not know about the facades).

14. **Is there any relation between flyweight patterns and state patterns?**

    The GoF says that the flyweight pattern can help you to decide *when and how* to share the state objects.

15. **What are the similarities among simple factory, factory method, and abstract factory design patterns?**

    All of them encapsulate object creation. They suggest you code to the abstraction (interface) but not to the concrete classes. Each of these factories promotes loose coupling by reducing the dependencies on concrete classes.

16. **What are the differences among simple factory, factory method and abstract factory design patterns?**

    This is an important question that you may face in various job interviews. I suggest you clearly understand it. So, refer to the answer of question 3 in the "Q&A Session" section in Chapter 5.

17. **How can you distinguish the singleton pattern from the factory method pattern?**

    The singleton pattern ensures that you get a unique instance each time. It also restricts the creation of additional instances.

    But the factory method pattern does not say that you will get a unique instance only. Most often, this pattern is used to create as many instances as you want, and these instances are not necessarily unique. These newly typed instances may implement a common base class. (Do you remember that the *factory method lets a class defer instantiation to subclasses* from the GoF definition ?)

18. **How can you distinguish the builder pattern from the prototype pattern?**

    In the prototype pattern, you are using the cloning/ copying mechanism. So, at the end, you may want to override the original implementation (note the word *@override* in our implementation of the Ford class and Nano class). But changing the legacy (or original) code is not always easy.

    Apart from this point, when you are using cloning mechanisms, you do not need to think about the constructors with different parameters.

    But the use of constructors with different parameters is very common in a builder pattern implementation.

19. **How can you distinguish the visitor pattern from the strategy pattern?**

    In a strategy pattern, each subclass uses different algorithms to solve a common problem. But in a visitor design pattern, each visitor subclass may provide different functionalities.

20. **How are null objects different from proxies?**

    In general, proxies act on real objects at some point and they may also provide behaviors. But a null object does not do any such operation.

21. **How can you distinguish the interpreter pattern from the visitor pattern?**

    In an interpreter pattern, you represent simple grammar as an object structure, but in a visitor pattern, you define specific operations that you want to use on an object structure. In addition to this, an interpreter has direct access to the properties that are needed, but in a visitor pattern, you need special functionalities (similar to an observer) to access them.

22. **How can you distinguish the flyweight pattern from the object pool pattern?**

    I did not discuss the object pool pattern in this book. But if you already know about the object pool pattern, you notice that in the flyweight pattern, flyweights have intrinsic and extrinsic states. So, if a flyweight has both states, the states are divided and the client needs to pass some part of the state to it. Also in general, the client does not change the intrinsic state because it is shared.

    The object pool pattern does not store any part of state outside; all state information is stored/encapsulated inside the pooled object. Also, clients can change the state of a pooled object.

23. **How are libraries (or frameworks) similar/different from design patterns?**

    They are not design patterns. They provide the implementations that you can use directly in your application. But they can use the concept of the patterns in those implementations.

APPENDIX A

A Brief Overview of GoF Design Patterns

We all have unique thought processes. So, in the early days of software development, engineers faced a common problem—there was no standard to instruct them how to design their applications. Each team followed their own style, and when a new member (experienced or unexperienced) joined an existing team, understanding the architecture was a gigantic task. Senior or experienced members of the team would need to explain the advantages of the existing architecture and why alternative designs were not considered.

The experienced developer also knows how to reduce future efforts by simply reusing the concepts already in place. Design patterns address this kind of issue and provide a common platform for all developers. You can think of them as the recorded experience of experts in the field. Patterns were intended to be applied in object-oriented designs with the intention of reuse.

In 1994, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides published the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley). In this book, they introduced the concept of design patterns in software development. *These authors became known as the Gang of Four. I refer to them as the "GoF" throughout this book*. The GoF described 23 patterns that were developed by the common experiences of software developers over a period of time. Nowadays, when a new member joins a development team, the developer is expected to know about the design patterns, and then the developer learns about the existing architecture. This approach allows a developer to actively participate in the development process within a short period of time.

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6

The first concept of a real-life design pattern came from the building architect Christopher Alexander. During his lifetime, he discovered that many of the problems he faced were similar in nature. So, he tried to address those issues with similar types of solutions.

> *Each pattern describes a problem, which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*
>
> —Christopher Alexander

The software engineering community started believing that although these patterns were described for buildings and towns, the same concepts could be applied to patterns in object-oriented design. They felt that we could substitute the original concepts of walls and doors with objects and interfaces. The common thing in both fields is that, at their cores, patterns are solutions to common problems.

Lastly, it is important to note that the GoF discussed the original concepts of design patterns in the context of C++. But Sun Microsystems released its first public implementation of Java 1.0 in 1995, and then it went through various changes. In 1995, Java was totally new to the programming world. But it grew rapidly and secured its rank in the world's top programming languages within a short period of time, and in today's market, it is always in high demand. (You may know that later Oracle Corporation acquired Sun Microsystems and the acquisition process was finished on January 27, 2010.)

On the other hand, the concepts of design patterns are universal. So, when you exercise the fundamental concepts of design patterns with Java, you will be a better programmer, and you'll remake yourself in the programming community.

Key Points

- A design pattern describes a general reusable solution to software design problems. While developing software, you may encounter these problems frequently. The basic idea is that you can solve similar kinds of problems with similar kinds of solutions. And these solutions were tested over a long period of time.

- Patterns provide a template of how to solve a problem. They can be used in many different situations. At the same time, they help you get the best possible design much faster.
- Patterns are descriptions of how to create objects and classes, and customize them to solve a general design problem in a particular context.
- The GoF discussed 23 design patterns. Each of these patterns focuses on a particular object-oriented design. Each pattern can also describe the consequences and trade-offs of use. The GoF categorized these 23 patterns based on their purposes, as shown in the following sections.

A. Creational Patterns

Creational patterns abstract the instantiation process. You make the systems independent from the way that their objects are composed, created and represented. In these patterns, you are concerned about "Where should I place the "new" keyword in my application?" This decision can determine the degree of coupling in your classes. The following five patterns belong in this category.

- Singleton pattern
- Prototype pattern
- Factory method pattern
- Builder pattern
- Abstract factory pattern

B. Structural Patterns

Structural patterns focus on how classes and objects can be composed to form a relatively large structure. They generally use inheritance or composition to group different interfaces or implementations. Your choice of composition over inheritance (and vice versa) can affect the flexibility of your software. The following seven patterns fall into this category.

- Proxy pattern
- Flyweight pattern
- Composite pattern
- Bridge pattern
- Facade pattern
- Decorator pattern
- Adapter pattern

C. Behavioral Patterns

Behavioral patterns concentrate on algorithms and the assignment of responsibilities among objects. They focus on communication between them and how objects are interconnected. The following 11 patterns fall into this category.

- Observer pattern
- Strategy pattern
- Template method pattern
- Command pattern
- Iterator pattern
- Memento pattern
- State pattern
- Mediator pattern
- Chain of Responsibility pattern
- Visitor pattern
- Interpreter pattern

The GoF made another classification based on scope, namely whether the pattern primary focuses on the classes or its objects. Class patterns deal with classes and subclasses. They use inheritance mechanisms, so they are static and fixed at compile time. Object patterns deal with objects that can change at runtime. So, object patterns are dynamic.

For a quick reference, you can refer to the following table, which was introduced by the GoF.

			Purpose	
		Creational	**Structural**	**Behavioral**
Scope	**Class**	1.Factory Method	1.1.Adapter(class)	1.Interpreter 2.Template Method
	Object	2.Singleton 3.Prototype 4.Builder 5.Abstract Factory	1.2.Adapter(object) 2.Proxy 3.Flyweight 4.Composite 5.Bridge 6.Facade 7.Decorator	3.Observer 4.Strategy 5.Command 6.Iterator 7.Memento 8.State 9.Mediator 10.Visitor 11.Chain of Responsibility

---

Note In this book, each chapter is self-contained. You can start with any pattern you like, following the guidelines given at the beginning of the book. I have chosen simple examples so that you can pick up basic ideas quickly. But you must keep reading and practice. Try to link problems and then keep coding. This process helps you master the subject quickly.

Q&A Session

1. **What are the differences between class patterns and object patterns?**

   In general, class patterns focus on static relationship but object patterns can focus on dynamic relationships. As name suggests, class patterns focus on classes and its subclasses and object patterns focus on the objects relationships.

   As per GoF, these patterns can be further differentiated in Table A-1.

   Table A-1. *Class Patterns vs Object Patterns*

	Class Patterns	Object Patterns
Creational	Defers object creation to its subclasses.	Defers object creation to another object.
Structural	Focuses on the composition of classes (primarily uses the concept of inheritance).	Focuses on the different ways of composition of objects.
Behavioral	Describes the algorithms and execution flows.	Describes how different objects can work together and complete a task.

2. **Can I combine two or more patterns in an application?**

   Yes. In real-world scenarios, this type of activity is common.

3. **Do these patterns depend on a particular programming language?**

   Programming languages can play an important role. But the basic ideas are same, patterns are just like templates and they give you some idea in advance of how you can solve a particular problem. In this book, I primarily focused on object-oriented programming with the concept of reuse. But instead of any object-oriented programming language, suppose you have chosen some other language like C. In that case, you may need to think about the core object-oriented principles such as inheritance, polymorphism,

encapsulation, abstraction, and so on, and how to implement them. So, the choice of a particular language is always important because it may have specialized features that can make your life easier.

4. **Should I consider common data structures like arrays and linked lists as design patterns?**

   The GoF clearly excludes those saying that *they are not complex, domain-specific designs for an entire application or subsystem*. They can be encoded in classes and reused as is. So, they are not your concern in this book.

5. **If no particular pattern is 100% suitable for my problem, how should I proceed?**

   An infinite number of problems cannot be solved with a finite number of patterns, for sure. But if you know these common patterns and their trade-offs, you can pick a close match. No one prevents you from using your own pattern for your own problem, but you have to tackle the risk.

6. **Do you suggest any general advice before I jump into the topics?**

   I always follow the footsteps of my seniors and teachers who are experts in this field. And the following are general suggestions from them.

   - Program to a supertype(Abstract class/Interface), not an implementation.
   - Prefer composition over inheritance.
   - Try to make a loosely coupled system.
   - Segregate the code that is likely to vary from the rest of your code.
   - Encapsulate what varies.

7. **How can I use this book effectively?**

   This book focuses on commonly used design patterns. Most likely, you face them very often in your everyday life. But the world is always changing, and new patterns are keep evolving. To understand the necessity of a new pattern, you may also need to understand why an old/existing pattern is not enough to fulfil the requirement. You may consider this book as an attempt to make a solid foundation with design patterns, so that, you can move smoothly in your professional life and you can adapt the upcoming changes easily.

APPENDIX B

Winning Notes and the Road Ahead

Congratulations. You have reached the end of the journey. Anyone can start a journey but only few can complete it with care. So, you are among the minority who possess the extraordinary capability to cover the distance successfully. I believe that you have enjoyed your learning experience and this experience can help you learn and experiment further. If you continue to think about the discussions, examples, implementations, and the Q&A sessions from the book, you will have more clarity and you will be confident about what you learned, and you can remake yourself in the programming world.

Truly, an in-depth discussion of any particular design pattern would require many more pages, and the size of the book would be too gigantic to digest.

So, what is next? You should not forget the basic principle: *learning is a continuous process*. This book encourages you to learn the core concepts so that you can continue learning in more depth.

I believe that learning and thinking on your own is not enough. So, I suggest you participate in open forums and join discussion groups to get more clarity on this subject. This process will not only help you, it will help others also.

I have a request. You can always point out areas for improvement in this book, but at the same time, please let me know what you liked about it. In general, it is always easy to criticize but an artistic view and open mind is required to discover the true efforts that are associated with any kind of work.

Thank you and happy coding!

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6

APPENDIX C

Bibliography

This appendix lists some useful resources.

The following are helpful books.

- Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- Freeman, Eric. *Head First Design Patterns*. O'Reilly, 2016.
- Bevis, Tony. *Java Design Pattern Essentials*. Ability First, 2012.
- Brown, William J., and Raphael Malveau. *Anti-Patterns: Refactoring Software, Architectures and Projects in Crisis*. Wiley, 1998.
- Sarcar, Vaskaran. *Design Patterns in C#*. Apress, 2018.

The following are helpful online resources/websites.

- https://en.wikipedia.org/wiki/Design_pattern
- https://sourcemaking.com/design_patterns
- www.tutorialspoint.com/design_pattern
- www.dotnetexamples.com
- https://java.dzone.com
- http://wiki.c2.com/?AntiPatternsCatalog
- https://hillside.net
- www.youtube.com/watch?v=ffQZIGTTM48&list=PL8C53D99ABAD3F4C8

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6

- www.dofactory.com
- www.c-sharpcorner.com
- www.dotnet-tricks.com
- www.codeproject.com

Index

A

B

© Vaskaran Sarcar 2019
V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6

C

D

E

F

G

H

I

J, K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y, Z