JAVA DESIGN PATTERNS

REUSABLE SOLUTIONS TO COMMON PROBLEMS

ROHIT JOSHI

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Java Design Patterns

Contents

Copyright (c) Exelixis Media P.C., 2015

All rights reserved. Without limiting the rights under
copyright reserved above, no part of this publication
may be reproduced, stored or introduced into a retrieval system, or
transmitted, in any form or by any means (electronic, mechanical,
photocopying, recording or otherwise), without the prior written
permission of the copyright owner.

Preface

A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

In this book you will delve into a vast number of Design Patterns and see how those are implemented and utilized in Java. You will understand the reasons why patterns are so important and learn when and how to apply each one of them.

About the Author

Rohit Joshi works as a Software Engineer in the Consumer Product Sector. He is a Sun Certified Java Programmer. He had worked in projects related to different domains.

He is also involved in system analysis and system designing. He mainly works in Core Java and J2EE technologies but also have good experience in front-end technologies like Javascript and Jquery.

Chapter 1

Introduction to Design Patterns

1.1 Introduction

In the late 70's, an architect named Christopher Alexander started the concept of patterns. Alexander's work focused on finding patterns of solutions to particular sets of forces within particular contexts.

Christopher Alexander was a civil engineer and an architect, his patterns were related to architects of buildings, but the work done by him inspired an interest in the object-oriented (OO) community, and a number of innovators started developing patterns for software design. Kent Beck and Ward Cunningham were among the few who presented the set of design patterns for Smalltalk in an OOPSLA conference. James Coplien was another who actively promoted the tenets of patterns.

Soon, the patterns community started growing at OOPSLA, as it placed an environment for the members to share their innovations and ideas about the patterns. Another important forum for the evolution of the patterns movement was the Hillside Group, established by Kent Beck and Grady Booch.

This is what design patterns are - the distillation of expertise by an exuberant and robust community. This is crowd sourcing at its best. The patterns community that has grown over the decade-plus since the original GoF work is large and energetic. Grady Booch and Celso Gonzalez have been collecting every pattern they can find in the industry. So far, they have over 2,000 of them.

This course is all about Design Patterns. In this course, we will present to you, the most useful and famous design patterns. In this lesson, first we will see what really are the Design Patterns. What is their use? Why one should really use them, and how to use them?

Later, we will also see how patterns are organized, and categorized into different groups according to their behavior and structure.

In the next several lessons, we will discuss about the different design patterns one by one. We will go into depth and analyze each and every design pattern, and will also see how to implement them in Java.

1.2 What are Design Patterns

As an Object Oriented developer, we may think that our code contains all the benefits provided by the Object Oriented language. The code we have written is flexible enough that we can make any changes to it with less or any pain. Our code is re-usable so that we can re-use it anywhere without any trouble. We can maintain our code easily and any changes to a part of the code will not affect any other part of the code.

Unfortunately, these advantages do not come by its own. As a developer, it is our responsibility to design the code in such a way which allow our code to be flexible, maintainable, and re-usable.

Designing is an art and it comes with the experience. But there are some set of solutions already written by some of the advanced and experienced developers while facing and solving similar designing problems. These solutions are known as Design Patterns.

The Design Patterns is the experience in designing the object oriented code.

Design Patterns are general reusable solution to commonly occurring problems. These are the best practices, used by the experienced developers. Patterns are not complete code, but it can use as a template which can be applied to a problem. Patterns

are re-usable; they can be applied to similar kind of design problem regardless to any domain. In other words, we can think of patterns as a formal document which contains recurring design problems and its solutions. A pattern used in one practical context can be re-usable in other contexts also.

Christopher had said that "Each pattern describes a problem that 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".

In general, a pattern has four essential elements:

- **Pattern name**, is used to provide a single and meaningful name to the pattern which defines a design problem and a solution for it. Naming a design pattern helps itself to be referred to others easily. It also becomes easy to provide documentation for and the right vocabulary word makes it easier to think about the design.
- **The problem describes when to apply the pattern.** It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe a class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
- The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution is not the complete code, but it works as a template which can be fulfilled with code. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.
- The **results and consequences** of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

1.3 Why use them

Flexibility: Using design patterns your code becomes flexible. It helps to provide the correct level of abstraction due to which objects become loosely coupled to each other which makes your code easy to change.

Reusability: Loosely coupled and cohesive objects and classes can make your code more reusable. This kind of code becomes easy to be tested as compared to the highly coupled code.

Shared Vocabulary: Shared vocabulary makes it easy to share your code and thought with other team members. It creates more understanding between the team members related to the code.

Capture best practices: Design patterns capture solutions which have been successfully applied to problems. By learning these patterns and the related problem, an inexperienced developer learns a lot about software design.

Design patterns make it easier to reuse successful designs and architectures.

Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design "right" faster.

1.4 How to select and use one

There are a number of design patterns to choose from; to choose one, you must have good knowledge of each one of them. There are many design patterns which look very similar to one another. They solve almost a similar type of design problem and also have similar implementation. One must have a very deep understanding of them in order to implement the correct design pattern for the specific design problem.

First, you need to identify the kind of design problem you are facing. A design problem can be categorized into creational, structural, or behavioral. Based to this category you can filter the patterns and selects the appropriate one. For example:

- **There are too many instances of a class which represent only a single thing, the value in the properties of the objects are same, and they are only used as read-only**: you can select the Singleton pattern for this design problem which ensures only a single instance for the entire application. It also helps to decrease the memory size.
- **Classes are too much dependent on each other. A Change in one class affects all other dependent classes:** you can use Bridge, Mediator, or Command to solve this design problem.
- **There are two different incompatible interfaces in two different parts of the code, and your need is to convert one interface into another which is used by the client code to make the entire code work:** the Adapter pattern fits into this problem.

A design pattern can be used to solve more than one design problem, and one design problem can be solved by more than one design patterns. There could be plenty of design problems and solutions for them, but, to choose the pattern which fits exactly is depends on your knowledge and understanding about the design patterns. It also depends on the code you already have in place.

1.5 Categorization of patterns

Design patterns can be categorized in the following categories:

- Creational patterns
- Structural patterns
- Behavior patterns

1.5.1 Creational patterns

Creational design patterns are used to design the instantiation process of objects. The creational pattern uses the inheritance to vary the object creation.

There are two recurring themes in these patterns. First, they all encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together. All the system at large knows about the objects is their interfaces as defined by abstract classes. Consequently, the creational patterns give you a lot of flexibility in what gets created, who creates it, how it gets created, and when.

There can be some cases when two or more patterns looks fit as a solution to a problem. At other times, the two patterns complement each other for example; Builder can be used with other pattern to implements which components to get built.

1.5.2 Structural patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.

Rather than composing interfaces or implementations, structural object patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

1.5.3 Behavior patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself. An important issue here is how peer objects know about each other. Peers could maintain explicit references to each other, but that would increase their coupling. In the extreme, every object would know about every other. The Mediator pattern avoids this by introducing a mediator object between peers. The mediator provides the indirection needed for loose coupling.

The below tables shows the list of patterns under their respective categories:

Table 1.1: List of patterns

Creational Patterns	**Structural Patterns**	**Behavioral Patterns**
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	FaÃ§ade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

Chapter 2

Adapter Design Pattern

2.1 Adapter Pattern

A software developer, Max, has worked on an e-commerce website. The website allows users to shop and pay online. The site is integrated with a 3rd party payment gateway, through which users can pay their bills using their credit card. Everything was going well, until his manager called him for a change in the project.

The manager told him that they are planning to change the payment gateway vendor, and he has to implement that in the code.

The problem that arises here is that the site is attached to the Xpay payment gateway which takes an Xpay type of object. The new vendor, PayD, only allows the PayD type of objects to allow the process. Max doesn't want to change the whole set of 100 of classes which have reference to an object of type XPay. This also raises the risk on the project, which is already running on the production. Neither he can change the 3rd party tool of the payment gateway. The problem has occurred due to the incompatible interfaces between the two different parts of the code. In order to get the process work, Max needs to find a way to make the code compatible with the vendor's provided API.

Current Code with the Xpay's API

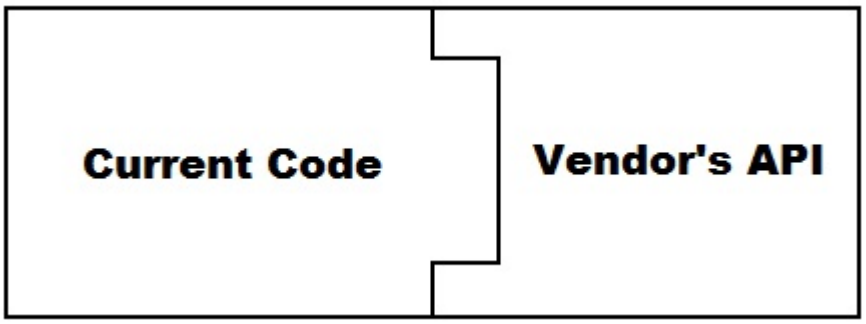

Figure 2.1: screenshot

Now, the current code interface is not compatible with the new vendor's interface.

Figure 2.2: screenshot

2.2 An Adapter to rescue

What Max needs here is an Adapter which can sit in between the code and the vendor's API, and can allow the process to flow. But before the solution, let us first see what an adapter is, and how it works.

Sometimes, there could be a scenario when two objects don't fit together, as they should in-order to get the work done. This situation could arise when we are trying to integrate a legacy code with a new code, or when changing a 3rd party API in the code. This is due to incompatible interfaces of the two objects which do not fit together.

The Adapter pattern lets you to adapt what an object or a class exposes to what another object or class expects. It converts the interface of a class into another interface the client expects. It lets classes work together that couldn't otherwise because of incompatible interfaces. It allows to fix the interface between the objects and the classes without modifying the objects and the classes directly.

You can think of an Adapter as a real world adapter which is used to connect two different pieces of equipment that cannot be connected directly. An adapter sits in-between these equipments, it gets the flow from the equipment and provides it to the other equipment in the form it wants, which otherwise, is impossible to get due to their incompatible interfaces.

An adapter uses composition to store the object it is supposed to adapt, and when the adapter's methods are called, it translates those calls into something the adapted object can understand and passes the calls on to the adapted object. The code that calls the adapter never needs to know that it's not dealing with the kind of object it thinks it is, but an adapted object instead.

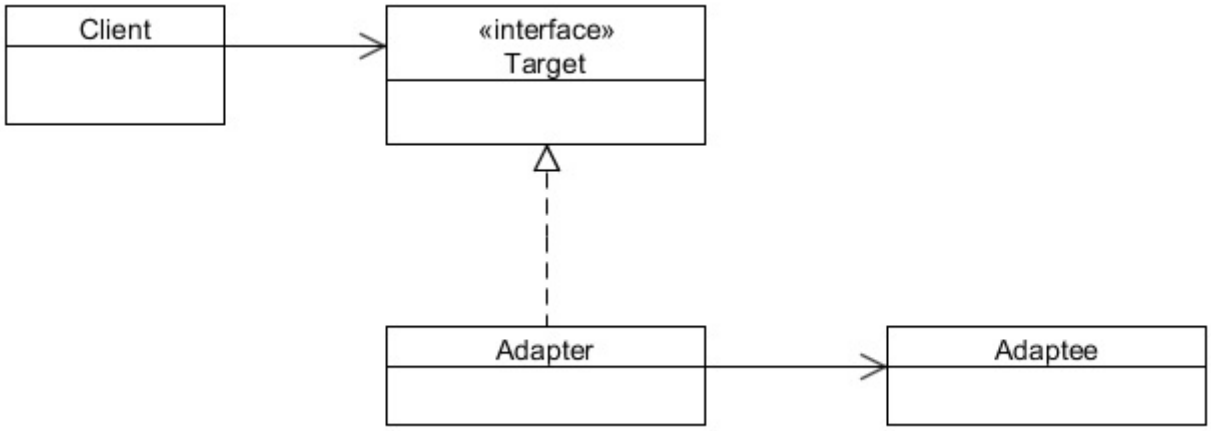

Figure 2.3: screenshot

Now, lets us see how it's going to solve the Max's problem.

2.3 Solution to the problem

Currently, the code is exposed to the Xpay interface. The interface looks something like this:

```
package com.javacodegeeks.patterns.adapterpattern.xpay;

public interface Xpay {

        public String getCreditCardNo();
        public String getCustomerName();
        public String getCardExpMonth();
        public String getCardExpYear();
        public Short getCardCVVNo();
        public Double getAmount();

        public void setCreditCardNo(String creditCardNo);
        public void setCustomerName(String customerName);
        public void setCardExpMonth(String cardExpMonth);
        public void setCardExpYear(String cardExpYear);
        public void setCardCVVNo(Short cardCVVNo);
        public void setAmount(Double amount);

}
```

It contains set of setters and getter method used to get the information about the credit card and customer name. This `Xpay` interface is implemented in the code which is used to instantiate an object of this type, and exposes the object to the vendor's API.

The following class defines the implementation to the `Xpay` interface.

```
package com.javacodegeeks.patterns.adapterpattern.site;

import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;

public class XpayImpl implements Xpay{

        private String creditCardNo;
        private String customerName;
        private String cardExpMonth;
        private String cardExpYear;
        private Short cardCVVNo;
        private Double amount;

        @Override
        public String getCreditCardNo() {
                return creditCardNo;
        }

        @Override
        public String getCustomerName() {
                return customerName;
        }

        @Override
        public String getCardExpMonth() {
                return cardExpMonth;
        }

        @Override
        public String getCardExpYear() {
                return cardExpYear;
        }
```

```
        @Override
        public Short getCardCVVNo() {
                return cardCVVNo;
        }

        @Override
        public Double getAmount() {
                return amount;
        }

        @Override
        public void setCreditCardNo(String creditCardNo) {
                this.creditCardNo = creditCardNo;
        }

        @Override
        public void setCustomerName(String customerName) {
                this.customerName = customerName;
        }

        @Override
        public void setCardExpMonth(String cardExpMonth) {
                this.cardExpMonth = cardExpMonth;
        }

        @Override
        public void setCardExpYear(String cardExpYear) {
                this.cardExpYear = cardExpYear;
        }

        @Override
        public void setCardCVVNo(Short cardCVVNo) {
                this.cardCVVNo = cardCVVNo;
        }

        @Override
        public void setAmount(Double amount) {
                this.amount = amount;
        }

}
```

New vendor's key interface looks like this:

```
package com.javacodegeeks.patterns.adapterpattern.payd;

public interface PayD {

        public String getCustCardNo();
        public String getCardOwnerName();
        public String getCardExpMonthDate();
        public Integer getCVVNo();
        public Double getTotalAmount();

        public void setCustCardNo(String custCardNo);
        public void setCardOwnerName(String cardOwnerName);
        public void setCardExpMonthDate(String cardExpMonthDate);
        public void setCVVNo(Integer cVVNo);
        public void setTotalAmount(Double totalAmount);
}
```

As you can see, this interface has a set of different methods which need to be implemented in the code. But Xpay is created by most part of the code, it's really hard and risky to change the entire set of classes.

We need some way, that's able to fulfill the vendor's requirement in order to process the payment and also make less or no change in the current code. The way is provided by the Adapter pattern.

We will create an adapter which will be of type `PayD`, and it wraps an `Xpay` object (the type it supposes to be adapted).

```
package com.javacodegeeks.patterns.adapterpattern.site;

import com.javacodegeeks.patterns.adapterpattern.payd.PayD;
import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;

public class XpayToPayDAdapter implements PayD{

        private String custCardNo;
        private String cardOwnerName;
        private String cardExpMonthDate;
        private Integer cVVNo;
        private Double totalAmount;

        private final Xpay xpay;

        public XpayToPayDAdapter(Xpay xpay){
                this.xpay = xpay;
                setProp();
        }

        @Override
        public String getCustCardNo() {
                return custCardNo;
        }

        @Override
        public String getCardOwnerName() {
                return cardOwnerName;
        }

        @Override
        public String getCardExpMonthDate() {
                return cardExpMonthDate;
        }

        @Override
        public Integer getCVVNo() {
                return cVVNo;
        }

        @Override
        public Double getTotalAmount() {
                return totalAmount;
        }

        @Override
        public void setCustCardNo(String custCardNo) {
                this.custCardNo = custCardNo;
        }

        @Override
        public void setCardOwnerName(String cardOwnerName) {
                this.cardOwnerName = cardOwnerName;
        }
```

```
        @Override
        public void setCardExpMonthDate(String cardExpMonthDate) {
                this.cardExpMonthDate = cardExpMonthDate;
        }

        @Override
        public void setCVVNo(Integer cVVNo) {
                this.cVVNo = cVVNo;
        }

        @Override
        public void setTotalAmount(Double totalAmount) {
                this.totalAmount = totalAmount;
        }

        private void setProp(){
                setCardOwnerName(this.xpay.getCustomerName());
                setCustCardNo(this.xpay.getCreditCardNo());
                setCardExpMonthDate(this.xpay.getCardExpMonth()+"/"+this.xpay. ↩
                    getCardExpYear());
                setCVVNo(this.xpay.getCardCVVNo().intValue());
                setTotalAmount(this.xpay.getAmount());
        }

}
```

In the above code, we have created an Adapter(`XpayToPayDAdapter`). The adapter implements the PayD interface, as it is required to mimic like a `PayD` type of object. The adapter uses object composition to hold the object, it's supposed to be adapting, an `Xpay` type of object. The object is passed into the adapter through its constructor.

Now, please note that we have two incompatible types of interfaces, which we need to fit together using an adapter in order to make the code work. These two interfaces have a different set of methods. But the sole purpose of these interfaces is very much similar, i.e. to provide the customer and credit card info to their specific vendors.

The `setProp()` method of the above class is used to set the xpay's properties into the payD's object. We set the methods which are similar in work in both the interfaces. However, there is only single method in PayD interface to set the month and the year of the credit card, as opposed to two methods in the `Xpay` interface. We joined the result of the two methods of the Xpay object (`this.xpay.getCardExpMonth()+"/"+this.xpay.getCardExpYear()`) and sets it into the `setCardExpMonthDate()` method.

Let us test the above code and see whether it can solve the Max's problem.

```
package com.javacodegeeks.patterns.adapterpattern.site;

import com.javacodegeeks.patterns.adapterpattern.payd.PayD;
import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;

public class RunAdapterExample {

        public static void main(String[] args) {

                // Object for Xpay
                Xpay xpay = new XpayImpl();
                xpay.setCreditCardNo("4789565874102365");
                xpay.setCustomerName("Max Warner");
                xpay.setCardExpMonth("09");
                xpay.setCardExpYear("25");
                xpay.setCardCVVNo((short)235);
                xpay.setAmount(2565.23);

                PayD payD = new XpayToPayDAdapter(xpay);
                testPayD(payD);
```

```
        }

        private static void testPayD(PayD payD){

                System.out.println(payD.getCardOwnerName());
                System.out.println(payD.getCustCardNo());
                System.out.println(payD.getCardExpMonthDate());
                System.out.println(payD.getCVVNo());
                System.out.println(payD.getTotalAmount());
        }
}
```

In the above class, first we have created an Xpay object and set its properties. Then, we created an adapter and pass it that xpay object in its constructor, and assigned it to the `PayD` interface. The `testPayD()` static method takes a `PayD` type as an argument which run and print its methods in order to test. As far as, the type passed into the `testPayD()` method is of type `PayD` the method will execute the object without any problem. Above, we passed an adapter to it, which looks like a type of `PayD`, but internally it wraps an `Xpay` type of object.

So, in the Max's project all we need to implement the vendor's API in the code and pass this adapter to the vendor's method to make the payment work. We do not need to change anything in the existing code.

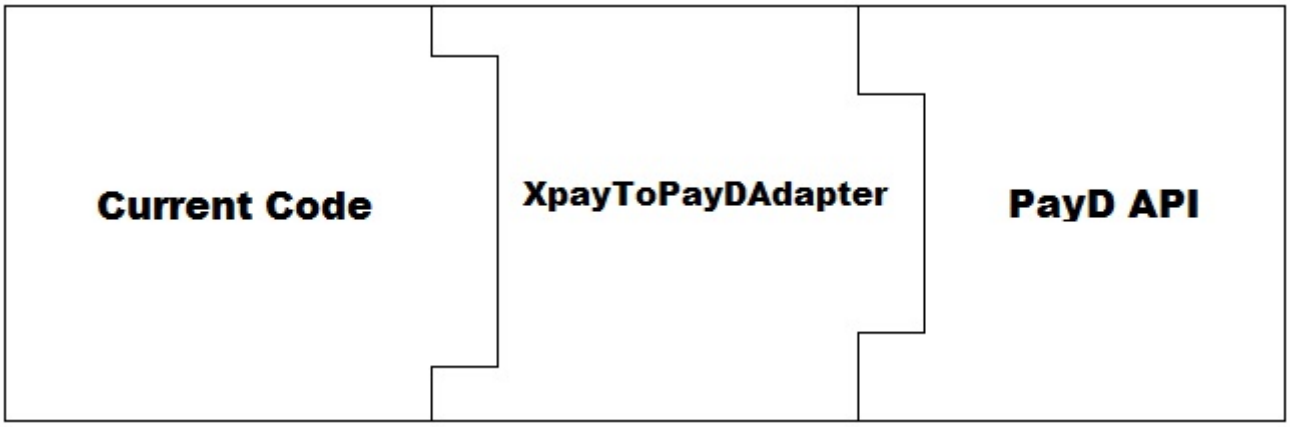

Figure 2.4: screenshot

2.4 Class Adapter

There are two types of adapters, the object adapter, and the class adapter. So far, we have seen the example of the object adapter which use object's composition, whereas, the class adapter relies on multiple inheritance to adapt one interface to another. As Java does not support multiple inheritance, we cannot show you an example of multiple inheritance, but you can keep this in mind and may implement it in one of your favorite Object Oriented Language like c++ which supports multiple inheritance.

To implement a class adapter, an adapter would inherit publicly from Target and privately from Adaptee. As the result, adapter would be a subtype of Target, but not for Adaptee.

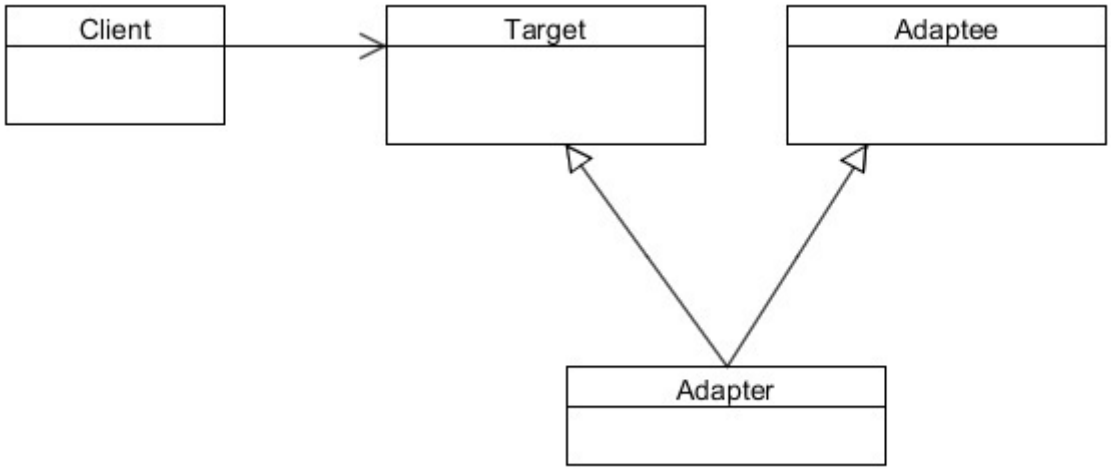

Figure 2.5: screenshot

2.5 When to use Adapter Pattern

The Adapter pattern should be used when:

- There is an existing class, and its interface does not match the one you need.
- You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- There are several existing subclasses to be use, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

2.6 Download the Source Code

This was a lesson on Adapter Pattern. You may download the source code here: **AdapterPattern-Project**

Chapter 3

Facade Design Pattern

3.1 Introduction

In this lesson, we will discuss about another structural pattern, i.e., the Facade Pattern. But before we dig into the details of it, let us discuss a problem which will be solved by this particular Pattern.

Your company is a product based company and it has launched a product in the market, named Schedule Server. It is a kind of server in itself, and it is used to manage jobs. The jobs could be any kind of jobs like sending a list of emails, sms, reading or writing files from a destination, or just simply transferring files from a source to the destination. The product is used by the developers to manage such kind of jobs and able to concentrate more towards their business goal. The server executes each job at their specified time and also manages all underline issues like concurrency issue and security by itself. As a developer, one just need to code only the relevant business requirements and a good amount of API calls is provided to schedule a job according to their needs.

Everything was going fine, until the clients started complaining about starting and stopping the process of the server. They said, although the server is working great, the initializing and the shutting down processes are very complex and they want an easy way to do that. The server has exposed a complex interface to the clients which looks a bit hectic to them.

We need to provide an easy way to start and stop the server.

A complex interface to the client is already considered as a fault in the design of the current system. But fortunately or unfortunately, we cannot start the designing and the coding from scratch. We need a way to resolve this problem and make the interface easy to access.

A Facade Pattern can help us to resolve this design problem. But before that, let us see about the Facade Pattern.

3.2 What is the Facade Pattern

The Facade Pattern makes a complex interface easier to use, using a Facade class. The Facade Pattern provides a unified interface to a set of interface in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

The Facade unifies the complex low-level interfaces of a subsystem in-order to provide a simple way to access that interface. It just provides a layer to the complex interfaces of the sub-system which makes it easier to use.

The Facade do not encapsulate the subsystem classes or interfaces; it just provides a simplified interface to their functionality. A client can access these classes directly. It still exposes the full functionality of the system for the clients who may need it.

A Facade is not just only able to simplify an interface, but it also decouples a client from a subsystem. It adheres to the Principle of Least Knowledge, which avoids tight coupling between the client and the subsystem. This provides flexibility: suppose in the above problem, the company wants to add some more steps to start or stop the Schedule Server, that have their own different interfaces. If you coded your client code to the facade rather than the subsystem, your client code doesn't need to be change, just the facade required to be changed, that's would be delivered with a new version to the client.

Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do the work of its own to translate its interface to subsystem interfaces. Clients that use the facade don't have to access its subsystem objects directly.

Please note that, a Facade same as an Adapter can wrap multiple classes, but a facade is used to an interface to simplify the use of the complex interface, whereas, an adapter is used to convert the interface to an interface the client expects.

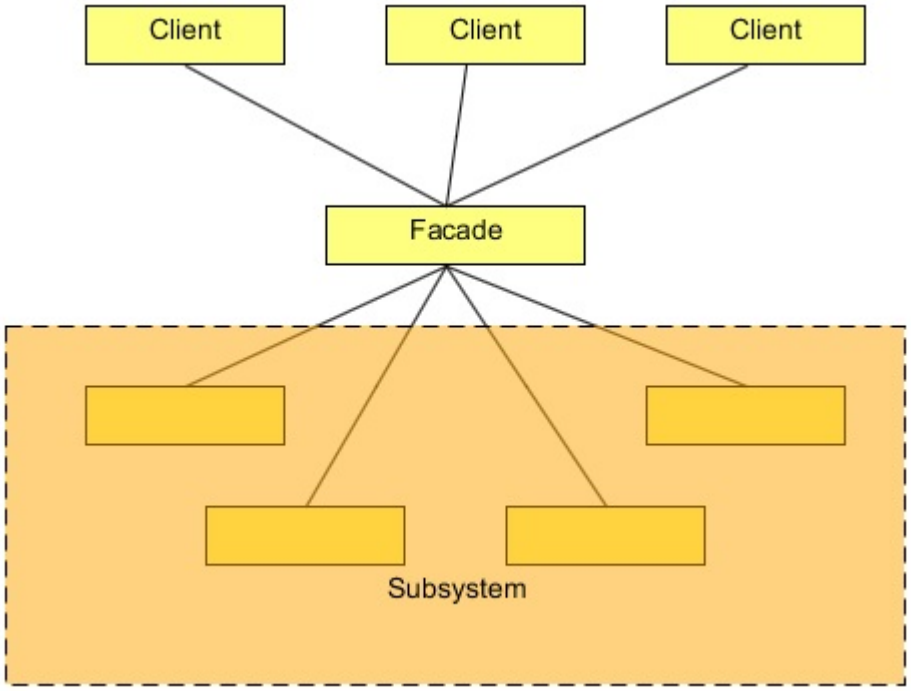

Figure 3.1: screenshot

3.3 Solution to the problem

The problem faced by the clients in using the Schedule Server is the complexity brought by the server in order to start and stop its services. The client wants a simple way to do it. The following is the code that clients required to write to start and stop the server.

```
ScheduleServer scheduleServer = new ScheduleServer();
```

To start the server, the client needs to create an object of the ScheduleServer class and then need to call the below methods in the sequence to start and initialize the server.

```
scheduleServer.startBooting();
scheduleServer.readSystemConfigFile();
scheduleServer.init();
scheduleServer.initializeContext();
scheduleServer.initializeListeners();
scheduleServer.createSystemObjects();

System.out.println("Start working.....");
System.out.println("After work done.........");
```

To stop the server, the client needs to call the following methods in the same sequence.

```
scheduleServer.releaseProcesses();
scheduleServer.destory();
```

```
scheduleServer.destroySystemObjects();
scheduleServer.destoryListeners();
scheduleServer.destoryContext();
scheduleServer.shutdown();
```

This looks a burden to them, they are not interested in doing all these stuffs, and why would they? Even though this might look interesting to some of the clients who might be interested in the low-level interface of the system, most of them disliked it.

To resolve this, we will create a facade class which will wrap a server object. This class will provide simple interfaces (methods) for the client. These interfaces internally will call the methods on the server object. Let us first see the code and then will discuss more about it.

```
package com.javacodegeeks.patterns.facadepattern;

public class ScheduleServerFacade {

        private final ScheduleServer scheduleServer;

        public ScheduleServerFacade(ScheduleServer scheduleServer){
                this.scheduleServer = scheduleServer;
        }

        public void startServer(){

                scheduleServer.startBooting();
                scheduleServer.readSystemConfigFile();
                scheduleServer.init();
                scheduleServer.initializeContext();
                scheduleServer.initializeListeners();
                scheduleServer.createSystemObjects();
        }

        public void stopServer(){

                scheduleServer.releaseProcesses();
                scheduleServer.destory();
                scheduleServer.destroySystemObjects();
                scheduleServer.destoryListeners();
                scheduleServer.destoryContext();
                scheduleServer.shutdown();
        }

}
```

The above class `ScheduleServerFacade` is the facade class, which wraps a `ScheduleServer` object, it instantiates the server object through its constructor, and has two simple methods: `startServer()` and `stopServer()`. These methods internally perform the starting and the stopping of the server. The client is just needs to call these simple methods. Now, there is no need to call all the lifecycle and destroy methods, just the simple methods and the rest of the process will be performed by the facade class.

The code below shows how facade makes a complex interface simple to use.

```
package com.javacodegeeks.patterns.facadepattern;

public class TestFacade {

        public static void main(String[] args) {

                ScheduleServer scheduleServer = new ScheduleServer();
                ScheduleServerFacade facadeServer = new ScheduleServerFacade(scheduleServer ↩
                    );
                facadeServer.startServer();
```

```
                System.out.println("Start working.....");
                System.out.println("After work done.........");

                facadeServer.stopServer();
        }

}
```

Also, please note that, although the facade class has provided a simple interface to the complex subsystem, it has not encapsulated the subsystem. A client can still access the low-level interfaces of the subsystem. So, a facade provides an extra layer, a simple interface to the complex subsystem, but it does not completely hide the direct accessibility to the low-level interfaces of the complex subsystem.

3.4 Use of the Facade Pattern

Use the Facade Pattern, when:

- You want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.
- There are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
- You can layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

3.5 Download the Source Code

This was a lesson on Facade Pattern. You may download the source code here: **FacadePattern-Project**

Chapter 4

Composite Design Pattern

4.1 Introduction

In this lesson, we will talk about a very interesting design pattern, the Composite Pattern. The English meaning of the word Composite is something that is made up of complicated and related parts. The composite means "putting together" and this is what this design pattern is all about.

There are times when you feel a need of a tree data structure in your code. There are many variations to the tree data structure, but sometimes there is a need of a tree in which both branches as well as leafs of the tree should be treated as uniformly.

The Composite Pattern allows you to compose objects into a tree structure to represent the part-whole hierarchy which means you can create a tree of objects that is made of different parts, but that can be treated as a whole one big thing. Composite lets clients to treat individual objects and compositions of objects uniformly, that's the intent of the Composite Pattern.

There can be lots of practical examples of the Composite Pattern. A file directory system, an html representation in java, an XML parser all are well managed composites and all can easily be represented using the Composite Pattern. But before digging into the details of an example, let's see some more details regarding the Composite Pattern.

4.2 What is the Composite Pattern

The formal definition of the Composite Pattern says that it allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients to treat individual objects and compositions of objects uniformly.

If you are familiar with a tree data structure, you will know a tree has parents and their children. There can be multiple children to a parent, but only one parent per child. In Composite Pattern, elements with children are called as Nodes, and elements without children are called as Leafs.

The Composite Pattern allows us to build structures of objects in the form of trees that contains both composition of objects and individual objects as nodes. Using a composite structure, we can apply the same operations over both composites and individual objects. In other words, in most cases we can ignore the differences between compositions of objects and individual objects.

The Composite Pattern has four participants:

- Component
- Leaf
- Composite
- Client

The following figure shows a typical Composite object structure. As you can see, there can be many children to a single parent i.e. Composite, but only one parent per child.

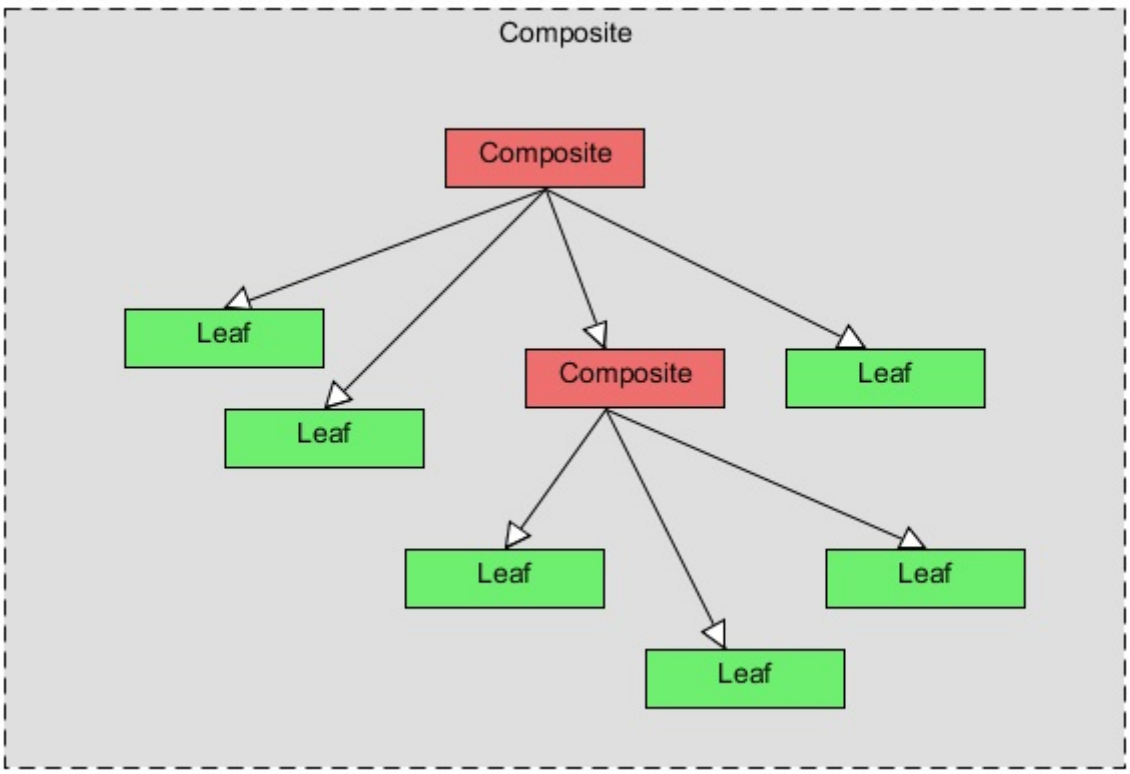

Figure 4.1: screenshot

The Component in the below class diagram, defines an interface for all objects in the composition both the composite and the leaf nodes. The Component may implement a default behavior for generic methods.

The Composite's role is to define the behavior of the components having children and to store child components. The Composite also implements the Leaf related operations. These operations may or may not take any sense; it depends on the functionality implements using the pattern.

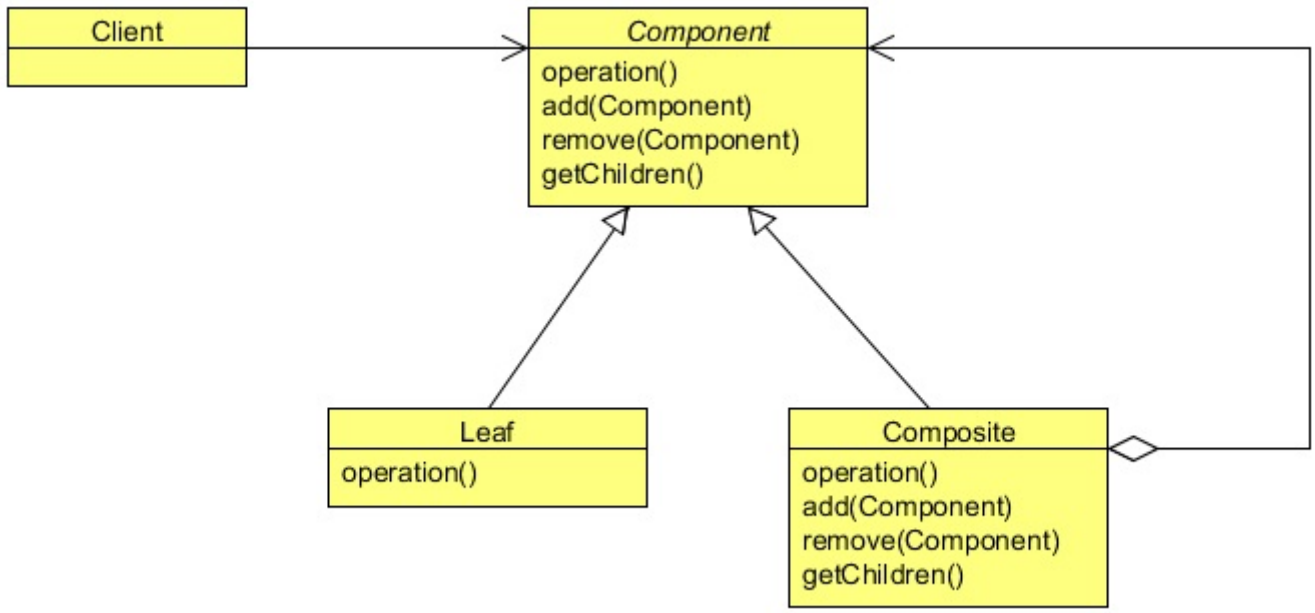

Figure 4.2: screenshot

A Leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Component supports. Leaf also inherits methods, which don't necessarily make a lot of sense for a leaf node.

The Client manipulates objects in the composition through the Component interface.

4.3 Example of Composite Pattern

The Composite pattern can be implemented anywhere you have the hierarchical nature of the system or a subsystem and you want to treat individual objects and compositions of objects uniformly. A File System, an XML, an Html, or a hierarchy of an office (starting from the president to employees) can be implemented using the Composite Pattern.

Let's see a simple example where we implement an html representation in Java using the Composite Pattern. An html is hierarchical in nature, its starts from an <html> tag which is the parent or the root tag, and it contains other tags which can be a parent or a child tag.

The Composite Pattern in Java can be implemented using the Component class as an abstract class or an interface. In this example, we will use an abstract class which contains all the important methods used in a composite class and a leaf class.

```
package com.javacodegeeks.patterns.compositepattern;

import java.util.List;


public abstract class HtmlTag {

        public abstract String getTagName();
        public abstract void setStartTag(String tag);
        public abstract void setEndTag(String tag);
        public void setTagBody(String tagBody){
                throw new UnsupportedOperationException("Current operation is not support  ↩
                    for this object");
        }
        public void addChildTag(HtmlTag htmlTag){
                throw new UnsupportedOperationException("Current operation is not support  ↩
                    for this object");
```

```
        }
        public void removeChildTag(HtmlTag htmlTag){
                throw new UnsupportedOperationException("Current operation is not support ↩
                    for this object");
        }
        public List<HtmlTag>getChildren(){
                throw new UnsupportedOperationException("Current operation is not support ↩
                    for this object");
        }
        public abstract void generateHtml();

}
```

The `HtmlTag` class is the component class which defines all the methods used by the composite and the leaf class. There are some methods which should be common in both the extended classes; hence these methods are kept abstract in the above class, to enforce their implementation in the child classes.

The `getTagName()`, just returns the tag name and should be used by both child classes, i.e., the composite class and the leaf class.

Every html element should have a start tag and an end tag, the methods `setStartTag` and `setEndTag` are used to set the start and end tag of an html element and should be implemented by both the child classes, so they are kept abstract in the above class.

There are methods which are useful only to the composite class and are useless to the leaf class. Just provide the default implementation to these methods, throwing an exception is a good implementation of these methods to avoid any accidental call to these methods by the object which should not support them.

The `generatHtml()` method is the operation which should support by both the extended classes. For the simplicity, it just prints the tag to the console.

Now, let's have a look at the Composite class.

```
package com.javacodegeeks.patterns.compositepattern;

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

public class HtmlParentElement extends HtmlTag {

        private String tagName;
        private String startTag;
        private String endTag;
        private List<HtmlTag>childrenTag;

        public HtmlParentElement(String tagName){
                this.tagName = tagName;
                this.startTag = "";
                this.endTag = "";
                this.childrenTag = new ArrayList<>();
        }

        @Override
        public String getTagName() {
                return tagName;
        }

        @Override
        public void setStartTag(String tag) {
                this.startTag = tag;
        }

        @Override
```

```
        public void setEndTag(String tag) {
                this.endTag = tag;
        }

        @Override
        public void addChildTag(HtmlTag htmlTag){
                childrenTag.add(htmlTag);
        }

        @Override
        public void removeChildTag(HtmlTag htmlTag){
                childrenTag.remove(htmlTag);
        }

        @Override
        public List<HtmlTag>getChildren(){
                return childrenTag;
        }

        @Override
        public void generateHtml() {
                System.out.println(startTag);
                for(HtmlTag tag : childrenTag){
                        tag.generateHtml();
                }
                System.out.println(endTag);

        }

}
```

The `HtmlParentElement` class is the composite class which implements methods like `addChildTag`, `removeChildTag`, `getChildren` which must be implemented by a class to become the composite of the structure. The operation method here is the `generateHtml`, which prints the tag of the current class, and also iterates through its children and calls their `generateHtml` method too.

```
package com.javacodegeeks.patterns.compositepattern;

public class HtmlElement extends HtmlTag{

        private String tagName;
        private String startTag;
        private String endTag;
        private String tagBody;

        public HtmlElement(String tagName){
                this.tagName = tagName;
                this.tagBody = "";
                this.startTag = "";
                this.endTag = "";
        }

        @Override
        public String getTagName() {
                return tagName;
        }

        @Override
        public void setStartTag(String tag) {
                this.startTag = tag;
        }
```

```
        @Override
        public void setEndTag(String tag) {
                this.endTag = tag;
        }

        @Override
        public void setTagBody(String tagBody){
                this.tagBody = tagBody;
        }

        @Override
        public void generateHtml() {
                System.out.println(startTag+""+tagBody+""+endTag);
        }

}
```

The `HtmlElement` is the leaf class, and its main job is to implement the operation method, which in this example is the `generateHtml` method. It prints the `startTag`, optionally tagBody if have, and the `endTag` of the child element.

Let's test this example.

```
package com.javacodegeeks.patterns.compositepattern;

public class TestCompositePattern {

        public static void main(String[] args) {
                HtmlTag parentTag = new HtmlParentElement("<html>");
                parentTag.setStartTag("<html>");
                parentTag.setEndTag("</html>");

                HtmlTag p1 = new HtmlParentElement("<body>");
                p1.setStartTag("<body>");
                p1.setEndTag("</body>");

                parentTag.addChildTag(p1);

                HtmlTag child1 = new HtmlElement("<P>");
                child1.setStartTag("<P>");
                child1.setEndTag("</P>");
                child1.setTagBody("Testing html tag library");
                p1.addChildTag(child1);

                child1 = new HtmlElement("<P>");
                child1.setStartTag("<P>");
                child1.setEndTag("</P>");
                child1.setTagBody("Paragraph 2");
                p1.addChildTag(child1);

                parentTag.generateHtml();

        }

}
```

The above code will result to the following output:

```
<html>
<body>
<P>Testing html tag library</P>
<P>Paragraph 2</P>
</body>
```

```
</html>
```

In the above example, first we have created a parent tag (<html>) then we add a child to it, which is another of composite type (<body>), and this object contains two children (<P>).

Please note that, the above structure represents as a part-whole hierarchy and the call to `generateHtml()` method on the parent tag allows the client to treat the compositions of objects uniformly. As it generates the html of the object and of all its children.

4.4 When to use Composite Pattern

- When you want to represent part-whole hierarchies of objects.
- When you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

4.5 Download the Source Code

This was a lesson on Composite Pattern. You may download the source code here: **CompositePattern-Project**

Chapter 5

Bridge Design Pattern

5.1 Introduction

Sec Security System is a security and electronic company which produces and assembles products for cars. It delivers any car electronic or security system you want, from air bags to GPS tracking system, reverse parking system etc. Big car companies use its products in their cars. The company uses a well defined object oriented approach to keep track of their products using software which is developed and maintained by them only. They get the car, produce the system for it and assemble it into the car.

Recently, they got new orders from BigWheel (a car company) to produce central locking and gear lock system for their new xz model. To maintain this, they are creating a new software system. They started by creating a new abstract class CarProductSecurity, in which they kept some car specific methods and some of the features which they thought are common to all security products. Then they extended the class and created two different sub classes named them BigWheelXZCentralLocking, and BigWheelXZGearLocking. The class diagram looks like this:

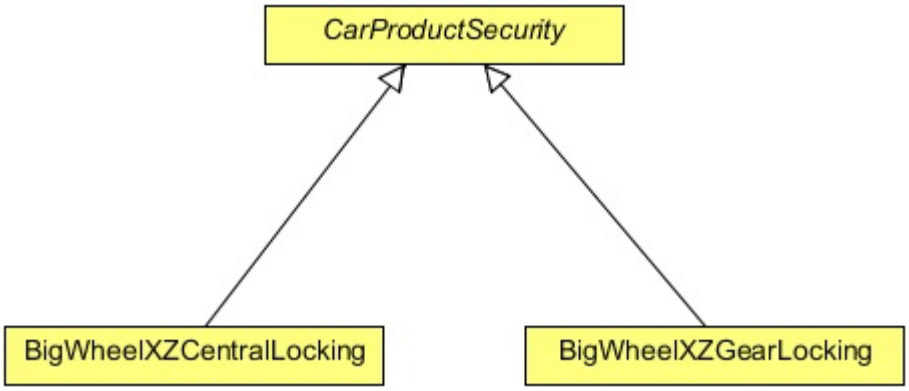

Figure 5.1: screenshot

After a while, another car company Motoren asked them to produce a new system of central locking and gear lock for their lm model. Since, the same security system cannot be used in both models of different cars, the Sec Security System has produced the new system for them, and also has created to new classes MotorenLMCentralLocking, and MotorenLMGearLocking which also extend the CarProductSecurity class.

Now the new class diagram looks like this:

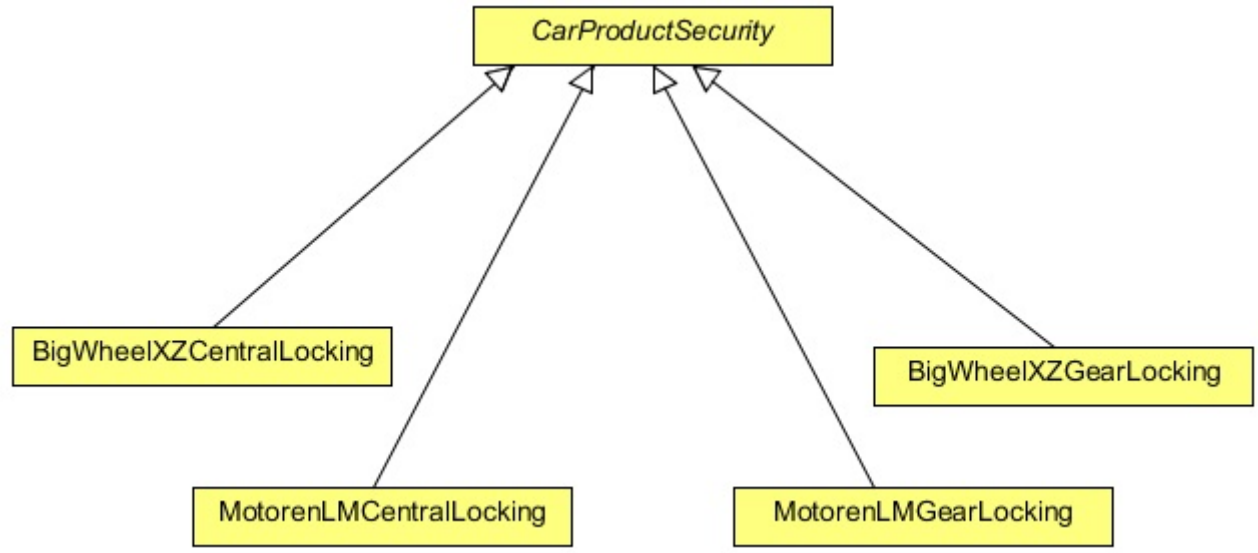

Figure 5.2: screenshot

So far so good, but what happens if another car company demands another new system of central locking and gear lock? One needs to create another two new classes for it. This design will create one class per system, or worse, if the reverse parking system is produced for each of these two car companies, two more new classes will be created for each of them.

A design with too many subclasses is not flexible and is hard to maintain. An Inheritance also binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse the abstraction and implementation independently.

Please note that, the car and the product should vary independently in order to make the software system easy to extend and reusable.

The Bridge design pattern can resolve this problem, but before that, let us first have some details about the Bridge Pattern.

5.2 What is Bridge Pattern

The Bridge Pattern's intent is to decouple an abstraction from its implementation so that the two can vary independently. It puts the abstraction and implementation into two different class hierarchies so that both can be extend independently.

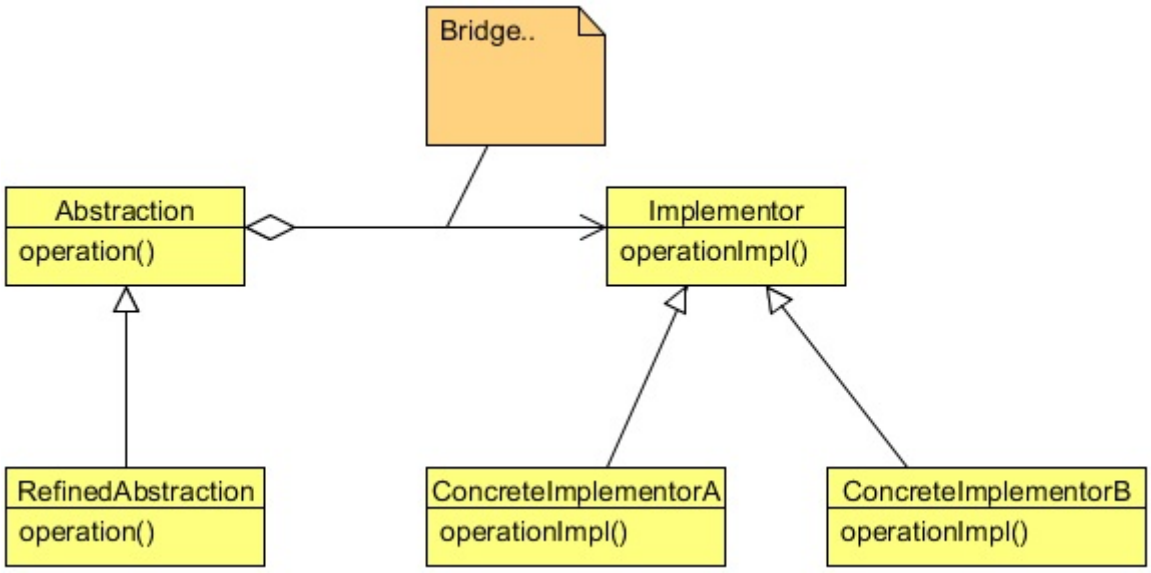

Figure 5.3: screenshot

The components of the Bridge Pattern comprise of an abstraction, refined abstraction, an implementer, and concrete implementer.

An abstraction defines the abstraction's interface and also maintains a reference to an object of type implementer, and the link between the abstraction and the implementer is called a Bridge.

Refined Abstraction extends the interface defined by the abstraction.

The Implementer provides the interface for implementation classes (concrete implementers).

And the Concrete Implementer implements the Implementer interface and defines its concrete implementation.

The Bridge Pattern decouples the interface and the implementation. As a result, an implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It also eliminates compile-time dependencies on the implementation. Changing an implementation class doesn't required recompiling the abstraction class and its clients. The Client only needs to know about the abstraction and you can hide the implementation from them.

5.3 Solution to the Problem

Instead of creating a subclass for each product per car model in the above discussed problem, we can separate the design into two different hierarchies. One interface is for the product which will be used as an implementer and the other will be an abstraction of car type. The implementer will be implemented by the concrete implementers and provides an implementation for it. On the other side, the abstraction will be extended by more refined abstraction.

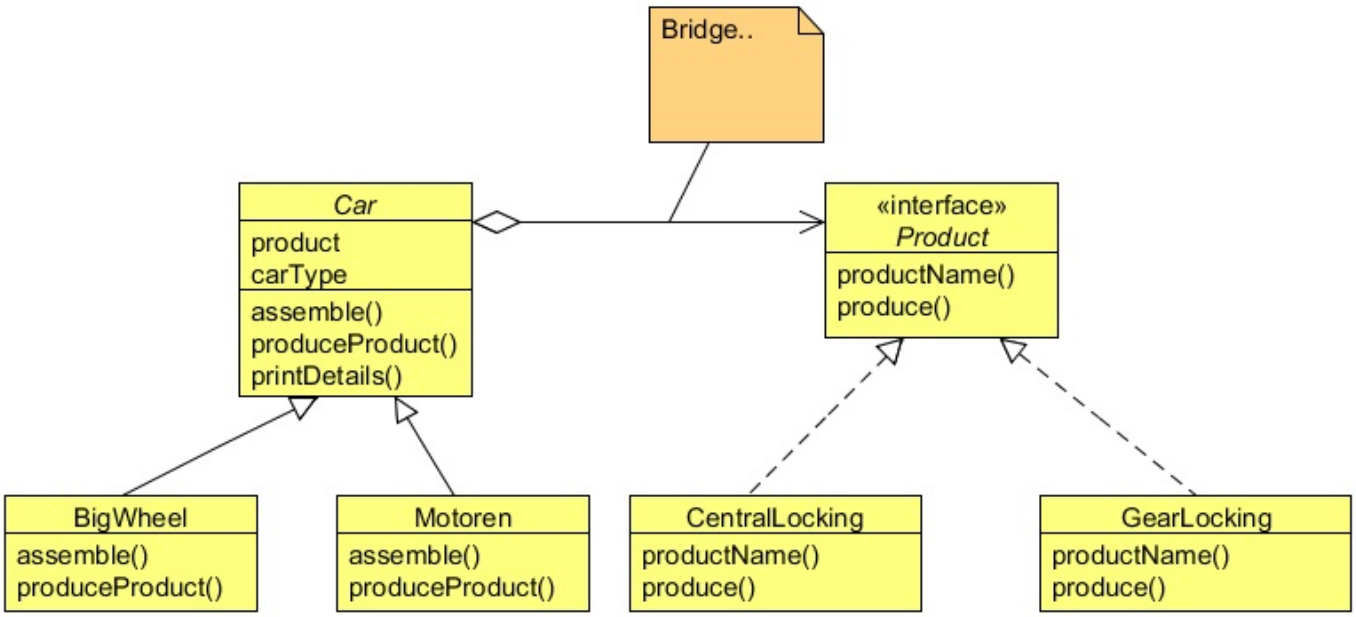

Figure 5.4: screenshot

```
package com.javacodegeeks.patterns.bridgepattern;

public interface Product {

        public String productName();
        public void produce();
}
```

The implementer `Product` has a method `produce()` which will be used by the concrete implementers to provide concrete functionality to it. The method will produce the base model of the product which can be used with any car model after some modifications specific to that car model.

```
package com.javacodegeeks.patterns.bridgepattern;

public class CentralLocking implements Product{

        private final String productName;

        public CentralLocking(String productName){
                this.productName = productName;
        }

        @Override
        public String productName() {
                return productName;
        }

        @Override
        public void produce() {
                System.out.println("Producing Central Locking System");
        }

}

package com.javacodegeeks.patterns.bridgepattern;
```

```
public class GearLocking implements Product{

        private final String productName;

        public GearLocking(String productName){
                this.productName = productName;
        }

        @Override
        public String productName() {
                return productName;
        }

        @Override
        public void produce() {
                System.out.println("Producing Gear Locking System");
        }

}
```

The two different concrete implementers provide implementation to the `Product` implementer.

Now the abstraction, the `Car` class which holds a reference of a product type and provides two abstract methods `produceProduct()` and `assemble()`.

```
package com.javacodegeeks.patterns.bridgepattern;

public abstract class Car {

        private final Product product;
        private final String carType;

        public Car(Product product,String carType){
                this.product = product;
                this.carType = carType;
        }

        public abstract void assemble();
        public abstract void produceProduct();

        public void printDetails(){
                System.out.println("Car: "+carType+", Product:"+product.productName());
        }
}
```

The subclasses of the `Car` will provide the concrete and specific implementation to the methods `assemble()` and `produceProduct()`.

```
package com.javacodegeeks.patterns.bridgepattern;

public class BigWheel extends Car{

        private final Product product;
        private final String carType;

        public BigWheel(Product product, String carType) {
                super(product, carType);
                this.product = product;
                this.carType = carType;
        }

        @Override
```

```
        public void assemble() {
                System.out.println("Assembling "+product.productName()+" for "+carType);
        }

        @Override
        public void produceProduct() {
                product.produce();
                System.out.println("Modifing product "+product.productName()+" according to ↩
                     "+carType);
        }

}

package com.javacodegeeks.patterns.bridgepattern;

public class Motoren extends Car{

        private final Product product;
        private final String carType;

        public Motoren(Product product, String carType) {
                super(product, carType);
                this.product = product;
                this.carType = carType;
        }

        @Override
        public void assemble() {
                System.out.println("Assembling "+product.productName()+" for "+carType);
        }

        @Override
        public void produceProduct() {
                product.produce();
                System.out.println("Modifing product "+product.productName()+" according to ↩
                     "+carType);
        }

}
```

Now, let's test the example.

```
package com.javacodegeeks.patterns.bridgepattern;

public class TestBridgePattern {

        public static void main(String[] args) {
                Product product = new CentralLocking("Central Locking System");
                Product product2 = new GearLocking("Gear Locking System");
                Car car = new BigWheel(product , "BigWheel xz model");
                car.produceProduct();
                car.assemble();
                car.printDetails();

                System.out.println();

                car = new BigWheel(product2 , "BigWheel xz model");
                car.produceProduct();
                car.assemble();
                car.printDetails();

                car = new Motoren(product, "Motoren lm model");
```

```
                car.produceProduct();
                car.assemble();
                car.printDetails();

                System.out.println();

                car = new Motoren(product2, "Motoren lm model");
                car.produceProduct();
                car.assemble();
                car.printDetails();

        }

}
```

The above example will produce the following output:

```
Producing Central Locking System
Modifing product Central Locking System according to BigWheel xz model
Assembling Central Locking System for BigWheel xz model
Car: BigWheel xz model, Product:Central Locking System

Producing Gear Locking System
Modifing product Gear Locking System according to BigWheel xz model
Assembling Gear Locking System for BigWheel xz model
Car: BigWheel xz model, Product:Gear Locking System

Producing Central Locking System
Modifing product Central Locking System according to Motoren lm model
Assembling Central Locking System for Motoren lm model
Car: Motoren lm model, Product:Central Locking System

Producing Gear Locking System
Modifing product Gear Locking System according to Motoren lm model
Assembling Gear Locking System for Motoren lm model
Car: Motoren lm model, Product:Gear Locking System
```

5.4 Use of Bridge Pattern

You should use the Bridge Pattern when:

- You want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
- Both the abstractions and their implementations should be extensible by sub-classing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
- Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
- You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client.

5.5 Download the Source Code

This was a lesson on Bridge Pattern. You may download the source code here: **BridgePattern-Project**

Chapter 6

Singleton Design Pattern

6.1 Singleton Pattern

Sometimes it's important for some classes to have exactly one instance. There are many objects we only need one instance of them and if we, instantiate more than one, we'll run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results.

You may require only one object of a class, for example, when you are a creating the context of an application, or a thread manageable pool, registry settings, a driver to connect to the input or output console etc. More than one object of that type clearly will cause inconsistency to your program.

The Singleton Pattern ensures that a class has only one instance, and provides a global point of access to it. However, although the Singleton is the simplest in terms of its class diagram because there is only one single class, its implementation is a bit trickier.

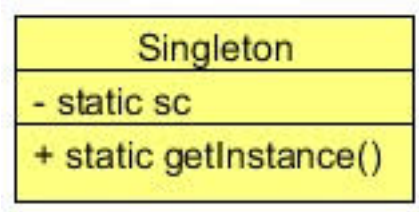

Figure 6.1: screenshot

In this lesson, we will try different ways to create only a single object of the class and will also see how one way is better than the other.

6.2 How to create a class using the Singleton Pattern

There could be many ways to create such type of class, but still, we will see how one way is better than the other.

Let's start with a simple way.

What if, we provide a global variable that makes an object accessible? For example:

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEager {
        public static SingletonEager sc = new SingletonEager();

}
```

As we know, there is only one copy of the static variables of a class, we can apply this. As far as, the client code is using this `sc` static variable its fine. But, if the client uses a new operator there would be a new instance of this class.

To stop the class to get instantiated outside the class, let's make the constructor of the class as private.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEager {
        public static SingletonEager sc = new SingletonEager();
        private SingletonEager(){}

}
```

Is this going to work? I think yes. By keeping the constructor private, no other class can instantiate this class. The only way to get the object of this class is using the `sc` static variable which ensures only one object is there.

But as we know, providing a direct access to a class member is not a good idea. We will provide a method through which the sc variable will get access, not directly.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEager {
        private static SingletonEager sc = new SingletonEager();
        private SingletonEager(){}
        public static SingletonEager getInstance(){
                return sc;
        }
}
```

So, this is our singleton class which makes sure that only one object of the class gets created and even if there are several requests, only the same instantiated object will be returned.

The one problem with this approach is that, the object would get created as soon as the class gets loaded into the JVM. If the object is never requested, there would be an object useless inside the memory.

It's always a good approach that an object should get created when it is required. So, we will create an object on the first call and then will return the same object on other successive calls.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonLazy {

        private static SingletonLazy sc = null;
        private SingletonLazy(){}
        public static SingletonLazy getInstance(){
                if(sc==null){
                        sc = new SingletonLazy();
                }
                return sc;
        }
}
```

In the `getInstance()` method, we check if the static variable `sc` is null, then instantiate the object and return it. So, on the first call when sc would be null the object gets created and on the next successive calls it will return the same object.

This surely looks good, doesn't it? But this code will fail in a multi-threaded environment. Imagine two threads concurrently accessing the class, thread t1 gives the first call to the `getInstance()` method, it checks if the static variable sc is null and then gets interrupted due to some reason. Another thread t2 calls the `getInstance()` method successfully passes the if check and instantiates the object. Then, thread t1 gets awake and it also creates the object. At this time, there would be two objects of this class.

To avoid this, we will use the `synchronized` keyword to the `getInstance()` method. With this way, we force every thread to wait its turn before it can enter the method. So, no two threads will enter the method at the same time. The synchronized

comes with a price, it will decrease the performance, but if the call to the `getInstance()` method isn't causing a substantial overhead for your application, forget about it. The other workaround is to move to eager instantiation approach as shown in the previous example.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonLazyMultithreaded {

        private static SingletonLazyMultithreaded sc = null;
        private SingletonLazyMultithreaded(){}
        public static synchronized SingletonLazyMultithreaded getInstance(){
                if(sc==null){
                        sc = new SingletonLazyMultithreaded();
                }
                return sc;
        }
}
```

But if you want to use synchronization, there is another technique known as "double-checked locking" to reduce the use of synchronization. With the double-checked locking, we first check to see if an instance is created, and if not, then we synchronize. This way, we only synchronize the first time.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonLazyDoubleCheck {

        private volatile static SingletonLazyDoubleCheck sc = null;
        private SingletonLazyDoubleCheck(){}
        public static SingletonLazyDoubleCheck getInstance(){
                if(sc==null){
                        synchronized(SingletonLazyDoubleCheck.class){
                                if(sc==null){
                                        sc = new SingletonLazyDoubleCheck();
                                }
                        }
                }
                return sc;
        }
}
```

Apart from this, there are some other ways to break the singleton pattern.

- If the class is Serializable.
- If it's Clonable.
- It can be break by Reflection.
- And also if, the class is loaded by multiple class loaders.

The following example shows how you can protect your class from getting instantiated more than once.

```
package com.javacodegeeks.patterns.singletonpattern;

import java.io.ObjectStreamException;
import java.io.Serializable;

public class Singleton implements Serializable{

        private static final long serialVersionUID = -1093810940935189395L;
        private static Singleton sc = new Singleton();
        private Singleton(){
```

```
                if(sc!=null){
                        throw new IllegalStateException("Already created.");
                }
        }
        public static Singleton getInstance(){
                return sc;
        }

        private Object readResolve() throws ObjectStreamException{
                return sc;
        }

        private Object writeReplace() throws ObjectStreamException{
                return sc;
        }

        public Object clone() throws CloneNotSupportedException{
                throw new CloneNotSupportedException("Singleton, cannot be clonned");
        }

        private static Class getClass(String classname) throws ClassNotFoundException {
                ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
                if(classLoader == null)
                        classLoader = Singleton.class.getClassLoader();
                return (classLoader.loadClass(classname));
        }

}
```

- Implement the `readResolve()` and `writeReplace()` methods in your singleton class and return the same object through them.
- You should also implement the `clone()` method and throw an exception so that the singleton cannot be cloned.
- An "if condition" inside the constructor can prevent the singleton from getting instantiated more than once using reflection.
- To prevent the singleton getting instantiated from different class loaders, you can implement the `getClass()` method. The above `getClass()` method associates the classloader with the current thread; if that classloader is null, the method uses the same classloader that loaded the singleton class.

Although we can use all these techniques, there is one simple and easier way of creating a singleton class. As of JDK 1.5, you can create a singleton class using enums. The Enum constants are implicitly static and final and you cannot change their values once created.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletoneEnum {

        public enum SingleEnum{
                SINGLETON_ENUM;
        }
}
```

You will get a compile time error when you attempt to explicitly instantiate an Enum object. As Enum gets loaded statically, it is thread-safe. The clone method in Enum is final which ensures that enum constants never get cloned. Enum is inherently serializable, the serialization mechanism ensures that duplicate instances are never created as a result of deserialization. Instantiation using reflection is also prohibited. These things ensure that no instance of an enum exists beyond the one defined by the enum constants.

6.3 When to use Singleton

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.

6.4 Download the Source Code

This was a lesson on Singleton Pattern. You may download the source code here: **SingletonPattern-Project**

Chapter 7

Observer Design Pattern

7.1 Observer Pattern

Sports Lobby is a fantastic sports site for sport lovers. They cover almost all kinds of sports and provide the latest news, information, matches scheduled dates, information about a particular player or a team. Now, they are planning to provide live commentary or scores of matches as an SMS service, but only for their premium users. Their aim is to SMS the live score, match situation, and important events after short intervals. As a user, you need to subscribe to the package and when there is a live match you will get an SMS to the live commentary. The site also provides an option to unsubscribe from the package whenever you want to.

As a developer, the Sport Lobby asked you to provide this new feature for them. The reporters of the Sport Lobby will sit in the commentary box in the match, and they will update live commentary to a commentary object. As a developer your job is to provide the commentary to the registered users by fetching it from the commentary object when it's available. When there is an update, the system should update the subscribed users by sending them the SMS.

This situation clearly shows one-to-many mapping between the match and the users, as there could be many users to subscribe to a single match. The Observer Design Pattern is best suited to this situation, let's see about this pattern and then create the feature for Sport Lobby.

7.2 What is the Observer Pattern

The Observer Pattern is a kind of behavior pattern which is concerned with the assignment of responsibilities between objects. The behavior patterns characterize complex control flows that are difficult to follow at run-time. They shift your focus away from the flow of control to let you concentrate just on the way objects are interconnected.

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The Observer pattern describes these dependencies. The key objects in this pattern are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in its state. In response, each observer will query the subject to synchronize its state with the subject state.

The other way to understand the Observer Pattern is the way Publisher-Subscriber relationship works. Let's assume for example that you subscribe to a magazine for your favorite sports or fashion magazine. Whenever a new issue is published, it gets delivered to you. If you unsubscribe from it when you don't want the magazine anymore, it will not get delivered to you. But the publisher continues to work as before, since there are other people who are also subscribed to that particular magazine.

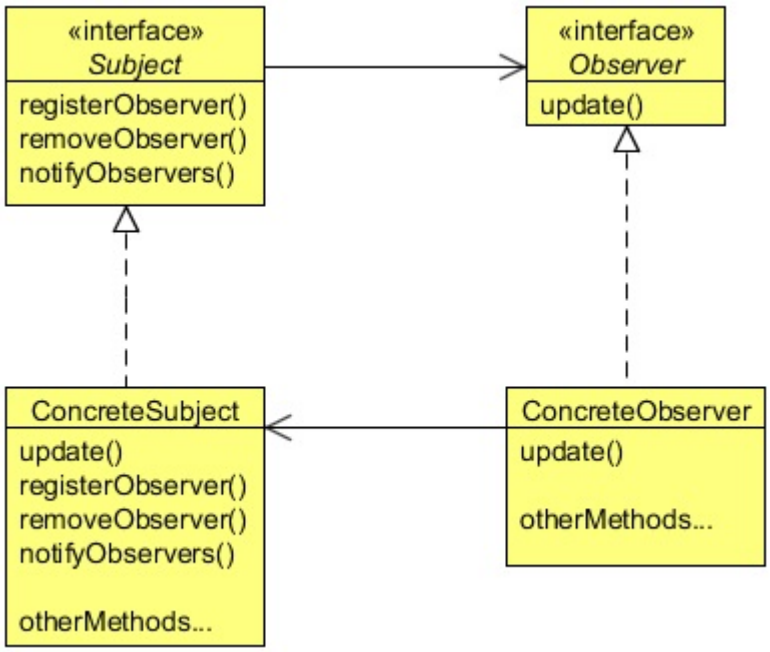

Figure 7.1: screenshot

There are four participants in the Observer pattern:

- Subject, which is used to register observers. Objects use this interface to register as observers and also to remove themselves from being observers.
- Observer, defines an updating interface for objects that should be notified of changes in a subject. All observers need to implement the Observer interface. This interface has a method `update()`, which gets called when the Subject's state changes.
- ConcreteSubject, stores the state of interest to ConcreteObserver objects. It sends a notification to its observers when its state changes. A concrete subject always implements the Subject interface. The `notifyObservers()` method is used to update all the current observers whenever the state changes.
- ConcreateObserver, maintains a reference to a ConcreteSubject object and implements the Observer interface. Each observer registers with a concrete subject to receive updates.

7.3 Implementing Observer Pattern

Let's see how to use the Observer Pattern in developing the feature for the Sport Lobby. Someone will update the concrete subject's object and your job is to update the state of the object registered with the concrete subject object. So, whenever there is a change in the state of the concrete subject's object all its dependent objects should get notified and then updated.

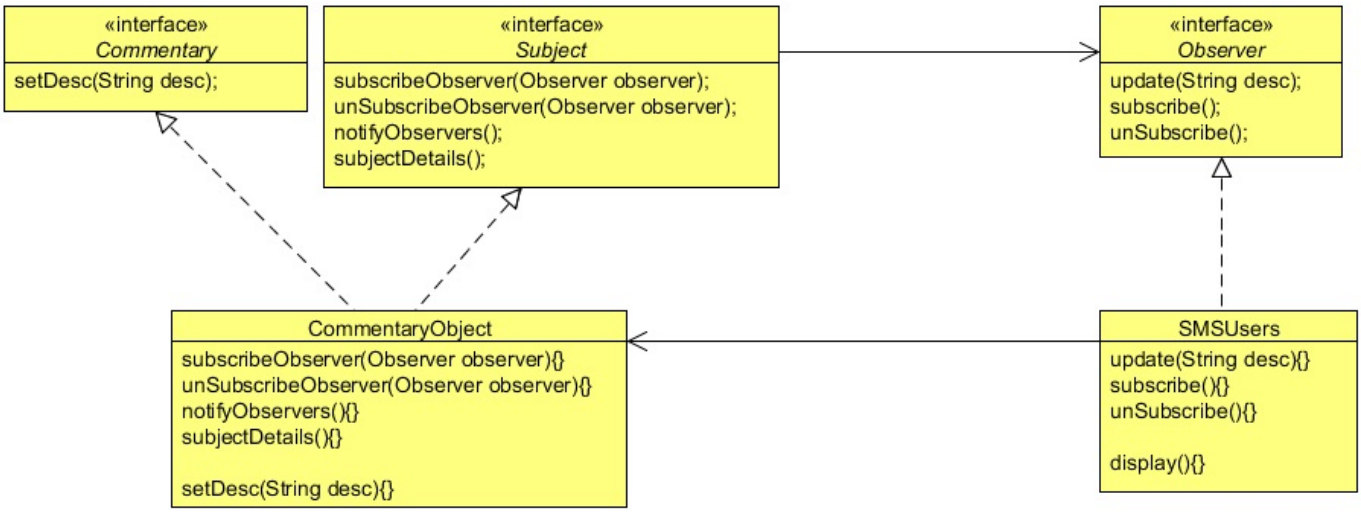

Figure 7.2: screenshot

Based on this, let's first create a `Subject` interface. In the Subject interface there are three key methods and optionally, if required, you can add some more methods according to your need.

```
package com.javacodegeeks.patterns.observerpattern;

public interface Subject {

        public void subscribeObserver(Observer observer);
        public void unSubscribeObserver(Observer observer);
        public void notifyObservers();
        public String subjectDetails();
}
```

The three key methods in the `Subject` interface are:

- `subscribeObserver`, which is used to subscribe observers or we can say register the observers so that if there is a change in the state of the subject, all these observers should get notified.
- `unSubscribeObserver`, which is used to unsubscribe observers so that if there is a change in the state of the subject, this unsubscribed observer should not get notified.
- `notifyObservers`, this method notifies the registered observers when there is a change in the state of the subject.

And optionally there is one more method `subjectDetails()`, it is a trivial method and is according to your need. Here, its job is to return the details of the subject.

Now, let's see the `Observer` interface.

```
package com.javacodegeeks.patterns.observerpattern;

public interface Observer {

        public void update(String desc);
        public void subscribe();
        public void unSubscribe();
}
```

- `update(String desc)`, method is called by the subject on the observer in order to notify it, when there is a change in the state of the subject.

- `subscribe()`, method is used to subscribe itself with the subject.
- `unsubscribe()`, method is used to unsubscribe itself with the subject.

```
package com.javacodegeeks.patterns.observerpattern;

public interface Commentary {

        public void setDesc(String desc);
}
```

The above interface is used by the reporters to update the live commentary on the commentary object. It's an optional interface just to follow the **code to interface principle**, not related to the Observer pattern. You should apply oops principles along with the design patterns wherever applicable. The interface contains only one method which is used to change the state of the concrete subject object.

```
package com.javacodegeeks.patterns.observerpattern;

import java.util.List;

public class CommentaryObject implements Subject,Commentary{

        private final List<Observer>observers;
        private String desc;
        private final String subjectDetails;

        public CommentaryObject(List<Observer>observers,String subjectDetails){
                this.observers = observers;
                this.subjectDetails = subjectDetails;
        }
        @Override
        public void subscribeObserver(Observer observer) {
                observers.add(observer);
        }

        @Override
        public void unSubscribeObserver(Observer observer) {
                int index = observers.indexOf(observer);
                observers.remove(index);

        }

        @Override
        public void notifyObservers() {
                System.out.println();
                for(Observer observer : observers){
                        observer.update(desc);
                }

        }

        @Override
        public void setDesc(String desc) {
                this.desc = desc;
                notifyObservers();
        }
        @Override
        public String subjectDetails() {
                return subjectDetails;
        }

}
```

The above class works as a concrete subject which implements the Subject interface and provides implementation of it. It also stores the reference to the observers registered to it.

```
package com.javacodegeeks.patterns.observerpattern;

public class SMSUsers implements Observer{

        private final Subject subject;
        private String desc;
        private String userInfo;

        public SMSUsers(Subject subject,String userInfo){
                if(subject==null){
                        throw new IllegalArgumentException("No Publisher found.");
                }
                this.subject = subject;
                this.userInfo = userInfo;
        }

        @Override
        public void update(String desc) {
                this.desc = desc;
                display();
        }

        private void display(){
                System.out.println("["+userInfo+"]: "+desc);
        }

        @Override
        public void subscribe() {
                System.out.println("Subscribing "+userInfo+" to "+subject.subjectDetails()+ ↩
                    " ...");
                this.subject.subscribeObserver(this);
                System.out.println("Subscribed successfully.");
        }

        @Override
        public void unSubscribe() {
                System.out.println("Unsubscribing "+userInfo+" to "+subject.subjectDetails ↩
                    ()+" ...");
                this.subject.unSubscribeObserver(this);
                System.out.println("Unsubscribed successfully.");
        }

}
```

The above class is the concrete observer class which implements the `Observer` interface. It also stores the reference to the subject it subscribed and optionally a `userInfo` variable which is used to display the user information.

Now, let's test the example.

```
package com.javacodegeeks.patterns.observerpattern;

import java.util.ArrayList;

public class TestObserver {

        public static void main(String[] args) {
                Subject subject = new CommentaryObject(new ArrayList<Observer>(), "Soccer  ↩
                    Match [2014AUG24]");
                Observer observer = new SMSUsers(subject, "Adam Warner [New York]");
                observer.subscribe();
```

```
                System.out.println();

                Observer observer2 = new SMSUsers(subject, "Tim Ronney [London]");
                observer2.subscribe();

                Commentary cObject = ((Commentary)subject);
                cObject.setDesc("Welcome to live Soccer match");
                cObject.setDesc("Current score 0-0");

                System.out.println();

                observer2.unSubscribe();

                System.out.println();

                cObject.setDesc("It's a goal!!");
                cObject.setDesc("Current score 1-0");

                System.out.println();

                Observer observer3 = new SMSUsers(subject, "Marrie [Paris]");
                observer3.subscribe();

                System.out.println();

                cObject.setDesc("It's another goal!!");
                cObject.setDesc("Half-time score 2-0");

        }

}
```

The above example will produce the following output:

```
Subscribing Adam Warner [New York] to Soccer Match [2014AUG24] ...
Subscribed successfully.

Subscribing Tim Ronney [London] to Soccer Match [2014AUG24] ...
Subscribed successfully.

[Adam Warner [New York]]: Welcome to live Soccer match
[Tim Ronney [London]]: Welcome to live Soccer match

[Adam Warner [New York]]: Current score 0-0
[Tim Ronney [London]]: Current score 0-0

Unsubscribing Tim Ronney [London] to Soccer Match [2014AUG24] ...
Unsubscribed successfully.

[Adam Warner [New York]]: It's a goal!!

[Adam Warner [New York]]: Current score 1-0

Subscribing Marrie [Paris] to Soccer Match [2014AUG24] ...
Subscribed successfully.

[Adam Warner [New York]]: It's another goal!!
[Marrie [Paris]]: It's another goal!!

[Adam Warner [New York]]: Half-time score 2-0
[Marrie [Paris]]: Half-time score 2-0
```

As you can see, at first two users subscribed themselves for the soccer match and started receiving the commentary. But later one user unsubscribed it, so the user did not receive the commentary again. Then, another user subscribed and starts getting the commentary.

All this happens dynamically without changing the existing code and not only this, suppose if, the company wants to broadcast the commentary on emails or any other firm wants to do collaboration with this company to broadcast the commentary. All you need to do is to create two new classes like `UserEmail` and `ColCompany` and make them observer of the subject by implementing the `Observer` interface. As far as the `Subject` knows it's an observer, it will provide the update.

7.4 Java's built-in Observer Pattern

Java has built-in support for the `Observer` Pattern. The most general is the Observer interface and the Observable class in the `java.util` package. These are quite similar to our Subject and Observer interface, but give you a lot of functionality out of the box.

Let's try to implement the above example using the Java's built-in Observer pattern.

```
package com.javacodegeeks.patterns.observerpattern;

import java.util.Observable;

public class CommentaryObjectObservable extends Observable implements Commentary {
        private String desc;
        private final String subjectDetails;

        public CommentaryObjectObservable(String subjectDetails){
                this.subjectDetails = subjectDetails;
        }

        @Override
        public void setDesc(String desc) {
                this.desc = desc;
                setChanged();
                notifyObservers(desc);
        }

        public String subjectDetails() {
                return subjectDetails;
        }
}
```

This time we extends the `Observable` class to make our class as a subject and please note that the above class does not hold any reference to the observers, it is handled by the parent class, that's is, the `Observable` class. However, we declared the `setDesc` method to change the state of the object, as done in the previous example. The `setChanged` method is the method from the upper class which is used to set the changed flag to true. The `notifyObservers` method notifies all of its observers and then calls the `clearChanged` method to indicate that this object has no longer changed. Each observer has its update method called with two arguments: an `observable` object and the `arg` argument.

```
package com.javacodegeeks.patterns.observerpattern;

import java.util.Observable;

public class SMSUsersObserver implements java.util.Observer{

        private String desc;
        private final String userInfo;
        private final Observable observable;

        public SMSUsersObserver(Observable observable,String userInfo){
                this.observable = observable;
```

```
                this.userInfo = userInfo;
        }

        public void subscribe() {
                System.out.println("Subscribing "+userInfo+" to "+(( ↩
                    CommentaryObjectObservable)(observable)).subjectDetails()+" ...");
                this.observable.addObserver(this);
                System.out.println("Subscribed successfully.");
        }

        public void unSubscribe() {
                System.out.println("Unsubscribing "+userInfo+" to "+(( ↩
                    CommentaryObjectObservable)(observable)).subjectDetails()+" ...");
                this.observable.deleteObserver(this);
                System.out.println("Unsubscribed successfully.");
        }

        @Override
        public void update(Observable o, Object arg) {
                        desc = (String)arg;
                        display();
        }

        private void display(){
                System.out.println("["+userInfo+"]: "+desc);
        }

}
```

Let's discuss some of the key methods.

The above class implements the `Observer` interface which has one key method `update`, which is called when the subject calls the `notifyObservers` method. The `update` method takes an `Observable` object and an `arg` as parameters.

The `addObserver` method is used to register a observer to the subject, and the `deleteObserver` method is used to remove the observer from the subject's list.

Let's test this example.

```
package com.javacodegeeks.patterns.observerpattern;

public class Test {

        public static void main(String[] args) {
                CommentaryObjectObservable obj = new CommentaryObjectObservable("Soccer  ↩
                    Match [2014AUG24]");
                SMSUsersObserver observer = new SMSUsersObserver(obj, "Adam Warner [New  ↩
                    York]");
                SMSUsersObserver observer2 = new SMSUsersObserver(obj,"Tim Ronney [London]" ↩
                    );
                observer.subscribe();
                observer2.subscribe();
                obj.setDesc("Welcome to live Soccer match");
                obj.setDesc("Current score 0-0");

                observer.unSubscribe();

                obj.setDesc("It's a goal!!");
                obj.setDesc("Current score 1-0");
        }
}
```

The above example will produce the following output:

```
Subscribing Adam Warner [New York] to Soccer Match [2014AUG24] ...
Subscribed successfully.
Subscribing Tim Ronney [London] to Soccer Match [2014AUG24] ...
Subscribed successfully.

[Tim Ronney [London]]: Welcome to live Soccer match
[Adam Warner [New York]]: Welcome to live Soccer match
[Tim Ronney [London]]: Current score 0-0
[Adam Warner [New York]]: Current score 0-0
Unsubscribing Adam Warner [New York] to Soccer Match [2014AUG24] ...
Unsubscribed successfully.
[Tim Ronney [London]]: It's a goal!!
[Tim Ronney [London]]: Current score 1-0
```

The above class creates a subject and two observers. The `subscribe` method of the `observer` adds itself to the subject observers list. Then `setDesc` changes the state of the subject which call the `setChanged` method to set the change flag to true, and notifies the observers. As a result, observer's `update` method is called which internally class the `display` method to display the result. Later, one of the observers gets `unsubscribe`d, i.e. it is removed from the observer's list. Due to which later commentaries were not updated to it.

Java provides built-in facility for the Observer Pattern, but it comes with its own drawbacks. The `Observable` is a class, you have to subclass it. That means you can't add on the Observable behavior to an existing class that already extends another superclass. This limits the reuse potential. You can't even create your own implementation that plays well with Java's built-in Observer API. Some of the methods in the Observable API are protected. This means you can't call the methods like `setChange` unless you've subclassed `Observable`. And you can't even create an instance of the `Observable` class and compose it with your own objects, you have to subclass. This design violates the **"favor composition over inheritance"** design principle.

7.5 When to use the Observer Pattern

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

7.6 Download the Source Code

This was a lesson on Observer Pattern. You may download the source code here: **ObserverPattern - Lesson 7**

Chapter 8

Mediator Design Pattern

8.1 Introduction

Today's world runs on software. Software runs in almost everything now, it is not only used in computers. Smart television, mobile phones, wrist watches, washing machines, etc. are having embedded software which operates the machine.

A big electronic company has asked you to develop a piece of software to operate its new fully automatic washing machine. The company has provided you with the hardware specification and the working knowledge of the machine. In the specification, they have provided you the different washing programs the machine supports. They want to produce a fully automatic washing machine that will require almost 0% of human interaction. The user should only need to connect the machine with a tap to supply water, load the clothes to wash, set the type of clothes in the machine like cotton, silk, or denims etc, provide detergent and softener to their respective trays, and press the start button.

The machine should be smart enough to fill the water in the drum, as much as required. It should adjust the wash temperature by itself by turning the heater on, according to the type of clothes in it. It should start the motor and spin the drum as much required, rinse according to the clothes needs, use soil removal if required, and softener too.

As an Object Oriented developer, you started analyzing and classifying objects, classes, and their relationships. Let's check some of the important classes and parts of the system. First of all, a Machine class, which has a drum. So a Drum class, but also a heater, a sensor to check the temperature, a motor. Additionally, the machine also has a valve to control the water supply, a soil removal, detergent, and a softener.

These classes have a very complex relationship with each other, and the relationship also varies. Please note that currently we are taking only about the high level abstraction of the machine. If we try to design it without keeping much of OOP principles and patterns in mind, then the initial design would be very tightly coupled and hard to maintain. This is because the above classes should contact with each other to get the job done. Like for example, the Machine class should ask the Valve class to open the valve, or the Motor should spin the Drum at certain rpm according to the wash program set (which is set by the type of clothes in the machine). Some type of clothes require softener or soil removal while others don't, or the temperature should be set according to the type of clothes.

If we allow classes to contact each other directly, that is, by providing a reference, the design will become very tightly coupled and hard to maintain. It would become very difficult to change one class without affecting the other. Even worse, the relationship between the classes varies, according to the different wash programs, like different temperature for different type of clothes. So these classes won't able to be reused. Even worse, in order to support all the wash programs we need to put control statements like if-else in the code which would make the code even more complex and difficult to maintain.

To decouple these objects from each other we need a mediator, which will contact the object on behalf of the other object, hence providing loose coupling between them. The object only needs to know about the mediator, and perform operations on it. The mediator will perform operations on the required underlying object in order to get the work done.

The Mediator Pattern is best suited for this, but before implementing it to solve our problem. Let's first know more about the Mediator Design Pattern.

8.2 What is the Mediator Pattern

The Mediator Pattern defines 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.

Rather than interacting directly with each other, objects ask the Mediator to interact on their behalf which results in reusability and loose coupling. It encapsulates the interaction between the objects and makes them independent from each other. This allows them to vary their interaction with other objects in a totally different way by implementing a different mediator. The Mediator helps to reduce the complexity of the classes. Each object no longer has to know in detail about how to interact with the other objects. The coupling between objects goes from tight and brittle to loose and agile.

Before Mediator the interaction between the classes might look like this, containing references of each other.

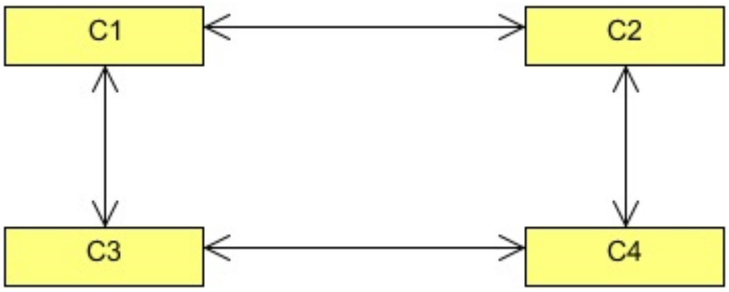

Figure 8.1: screenshot

Now, after implementing a Mediator the interaction between the classes looks like this, only containing a reference to the mediator.

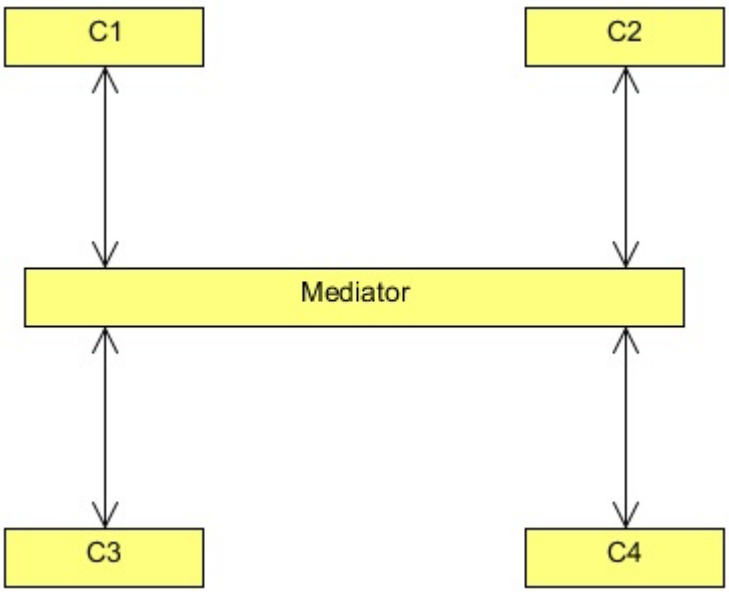

Figure 8.2: screenshot

The Mediator design pattern should be your first choice any time you have a set of objects that are tightly coupled. If every one of a series of objects has to know the internal details of the other objects, and maintaining those relationships becomes a problem, think of the Mediator. Using a Mediator means the interaction code has to reside in only one place, and that makes it easier to maintain. Using a mediator can hide a more serious problem: If you have multiple objects that are too tightly coupled, your encapsulation may be faulty. It might be time to rethink how you've broken your program into objects.

Let's look a more formal structure of the Mediator Pattern.

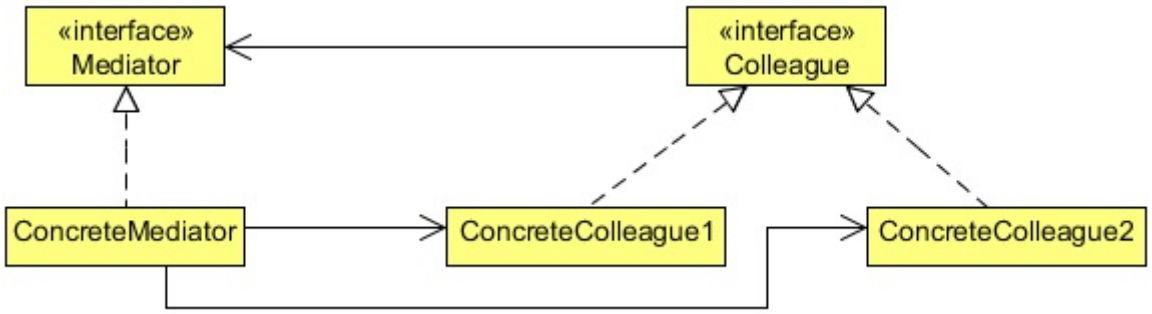

Figure 8.3: screenshot

The classess which hold reference of the mediator are called colleagues. The major participants of the Mediator Pattern are:

- Mediator: Defines an interface for communicating with Colleague objects.
- ConcreteMediator: Implements cooperative behavior by coordinating Colleague objects. It also knows and maintains its colleagues.
- Colleague Classes: Each Colleague class knows its Mediator object. Each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

8.3 Implementing the Mediator Pattern

Now, we will see how the Mediator Pattern will make the washing machine design better, reusable, maintainable and loosely coupled.

```
package com.javacodegeeks.patterns.mediatorpattern;

public interface MachineMediator {

        public void start();
        public void wash();
        public void open();
        public void closed();
        public void on();
        public void off();
        public boolean checkTemperature(int temp);

}
```

The `MachineMediator` is an interface which acts as a generic mediator. The interface contains operations call by one object to another.

```
package com.javacodegeeks.patterns.mediatorpattern;

public interface Colleague {

        public void setMediator(MachineMediator mediator);

}
```

The `Colleague` interface has one method to set the mediator for the concrete colleague's class.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class Button implements Colleague {

        private MachineMediator mediator;

    @Override
    public void setMediator(MachineMediator mediator){
                this.mediator = mediator;
        }

        public void press(){
                System.out.println("Button pressed.");
                mediator.start();
        }

}
```

The above `Button` class is a colleague class which holds a reference to a mediator. The user press the button which calls the `press()` method of this class which in turn, calls the `start()` method of the concrete mediator class. This `start()` method of the mediator calls the `start()` method of machine class on behalf of the `Button` class.

Later, we will see the structure of the mediator class. But now let's us first see the rest of the colleague classes.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class Machine implements Colleague {

        private MachineMediator mediator;

        @Override
        public void setMediator(MachineMediator mediator){
                this.mediator = mediator;
        }

        public void start(){
                mediator.open();
        }

        public void wash(){
                mediator.wash();
        }
}
```

The above `Machine` class which hold a reference to the mediator has the `start()` method which is called on the press of the button by the mediator class as discussed above. The method has an `open()` method of the mediator which in turn calls the `open()` method of the `Valve` class in order to open the valve of the machine.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class Valve implements Colleague {

        private MachineMediator mediator;

        @Override
        public void setMediator(MachineMediator mediator){
                this.mediator = mediator;
        }

        public void open(){
                System.out.println("Valve is opened...");
```

```
                System.out.println("Filling water...");
                mediator.closed();
        }

        public void closed(){
                System.out.println("Valve is closed...");
                mediator.on();
        }
}
```

The `Valve` class has two methods, an `open()` method which is called to open the valve and when the water is filled it called the `closed()` method. But please note that it is not calling the `closed()` method directly, it calls the `closed()` method of the mediator which invokes the method of this class.

On closing the valve it turns the heater on but again by invoking the mediator's method instead of directly calling the heater's method.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class Heater implements Colleague {

        private MachineMediator mediator;

        @Override
        public void setMediator(MachineMediator mediator){
                this.mediator = mediator;
        }
        public void on(int temp){
                System.out.println("Heater is on...");
                if(mediator.checkTemperature(temp)){
                        System.out.println("Temperature is set to "+temp);
                        mediator.off();
                }
        }

        public void off(){
                System.out.println("Heater is off...");
                mediator.wash();
        }
}
```

The heater's `on()` method switch on the heater and set the temperature as required. It also checks if temperature is reached as required, it turns `off()` the method. The checking of the temperature and switching off the heater is done through the mediator.

After switching off, it calls the `wash()` method of the Machine class through the mediator to start washing.

As stated by the company, the washing machine has a set of wash programs and the software should support all these programs. The below mediator is actually one of the washing programs for the machine. The below mediator is set as a washing program for cottons, so parameters such as temperature, drum spinning speed, level of soil removal etc., are set accordingly. We can have different mediators for different washing programs without changing the existing colleague classes and thus providing loose coupling and reusability. All these colleague classes can be reused with others washing programs of the machine.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class CottonMediator implements MachineMediator{

        private final Machine machine;
        private final Heater heater;
        private final Motor motor;
        private final Sensor sensor;
        private final SoilRemoval soilRemoval;
        private final Valve valve;
```

```
        public CottonMediator(Machine machine,Heater heater,Motor motor,Sensor sensor, ↩
            SoilRemoval soilRemoval,Valve valve){
                this.machine = machine;
                this.heater = heater;
                this.motor = motor;
                this.sensor = sensor;
                this.soilRemoval = soilRemoval;
                this.valve = valve;

                System.out.println("Setting up for COTTON program");
        }
        @Override
        public void start() {
                machine.start();
        }

        @Override
        public void wash() {
                motor.startMotor();
                motor.rotateDrum(700);
                System.out.println("Adding detergent");
                soilRemoval.low();
                System.out.println("Adding softener");
        }

        @Override
        public void open() {
                valve.open();
        }

        @Override
        public void closed() {
                valve.closed();
        }

        @Override
        public void on() {
                heater.on(40);
        }

        @Override
        public void off() {
                heater.off();
        }

        @Override
        public boolean checkTemperature(int temp) {
                return sensor.checkTemperature(temp);
        }

}
```

The `CottonMediator` class implements the `MachineMediator` interface and provides the required methods. These methods are the operations that are performed by the colleague objects in order to get the work done. The above mediator class just calls the method of a colleague object on behalf of another colleague object in order to achieve this.

There are also some other supporting classes:

```
package com.javacodegeeks.patterns.mediatorpattern;

public class Sensor {
```

```
        public boolean checkTemperature(int temp){
                System.out.println("Temperature reached "+temp+" *C");
                return true;
        }

}
```

`Sensor` class is used by the `Heater` to check the temperature.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class SoilRemoval {

        public void low(){
                System.out.println("Setting Soil Removal to low");
        }

        public void medium(){
                System.out.println("Setting Soil Removal to medium");
        }

        public void high(){
                System.out.println("Setting Soil Removal to high");
        }
}
```

`SoilRemoval` class is used by the `Machine` class.

In order to feel the advantages and power of the Mediator Pattern, let's take another mediator that is used as a washing program for denims. Now we just need to create a new mediator and set the rules in it to wash denims: rules like temperature, and the speed at which drum will spin, whether softener is required or not, the level of the soil removal, etc. We don't need to change anything in the existing structure. No conditional statements like "if-else" are required, something that would increase the complexity.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class DenimMediator implements MachineMediator{

        private final Machine machine;
        private final Heater heater;
        private final Motor motor;
        private final Sensor sensor;
        private final SoilRemoval soilRemoval;
        private final Valve valve;

        public DenimMediator(Machine machine,Heater heater,Motor motor,Sensor sensor, ↩
            SoilRemoval soilRemoval,Valve valve){
                this.machine = machine;
                this.heater = heater;
                this.motor = motor;
                this.sensor = sensor;
                this.soilRemoval = soilRemoval;
                this.valve = valve;

                System.out.println("Setting up for DENIM program");
        }
        @Override
        public void start() {
                machine.start();
        }

        @Override
```

```
        public void wash() {
                motor.startMotor();
                motor.rotateDrum(1400);
                System.out.println("Adding detergent");
                soilRemoval.medium();
                System.out.println("Adding softener");
        }

        @Override
        public void open() {
                valve.open();
        }

        @Override
        public void closed() {
                valve.closed();
        }

        @Override
        public void on() {
                heater.on(30);
        }

        @Override
        public void off() {
                heater.off();
        }

        @Override
        public boolean checkTemperature(int temp) {
                return sensor.checkTemperature(temp);
        }

}
```

You can clearly see the differences between the two mediator classes. There is different temperature, spinning speed is also different and no softener is required for the denim wash.

Now test these mediators.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class TestMediator {

        public static void main(String[] args) {
                MachineMediator mediator = null;
                Sensor sensor = new Sensor();
                SoilRemoval soilRemoval = new SoilRemoval();
                Motor motor = new Motor();
                Machine machine = new Machine();
                Heater heater = new Heater();
                Valve valve = new Valve();
                Button button = new Button();

                mediator = new CottonMediator(machine, heater, motor, sensor, soilRemoval,  ↩
                    valve);

                button.setMediator(mediator);
                machine.setMediator(mediator);
                heater.setMediator(mediator);
                valve.setMediator(mediator);
```

```
                button.press();

                mediator = new DenimMediator(machine, heater, motor, sensor, soilRemoval,  ↩
                    valve);

                button.setMediator(mediator);
                machine.setMediator(mediator);
                heater.setMediator(mediator);
                valve.setMediator(mediator);

                button.press();
        }

}
```

The above program will have as a result the following output:

```
Setting up for COTTON program
Button pressed.
Valve is opened...
Filling water...
Valve is closed...
Heater is on...
Temperature reached 40 C
Temperature is set to 40
Heater is off...
Start motor...
Rotating drum at 700 rpm.
Adding detergent
Setting Soil Removal to low
Adding softener

Setting up for DENIM program
Button pressed.
Valve is opened...
Filling water...
Valve is closed...
Heater is on...
Temperature reached 30 C
Temperature is set to 30
Heater is off...
Start motor...
Rotating drum at 1400 rpm.
Adding detergent
Setting Soil Removal to medium
No softener is required
```

In the above class, we created the objects required, mediators (or we can say different wash programs), then we set the wash programs to the colleagues and vice-versa, and the we called the `start()` method on the button object to start the machine. The rest is done automatically without any human interaction.

Please note that just to work with a different wash program, a different mediator is set and the rest remains the same. You can clearly see the differences from the output.

8.4 When to use the Mediator Pattern

- A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- Reusing an object is difficult because it refers to and communicates with many other objects.

- A behavior that's distributed between several classes should be customizable without a lot of sub-classing.

8.5 Mediator Pattern in JDK

Design Patterns are used almost everywhere in JDK. The following are the usages of the Mediator Pattern in JDK.

- `java.util.concurrent.ScheduledExecutorService` (all `scheduleXXX()` methods)
- `java.util.concurrent.ExecutorService` (the `invokeXXX()` and `submit()` methods)
- `java.util.concurrent.Executor#execute()`
- `java.util.Timer` (all `scheduleXXX()` methods)
- `java.lang.reflect.Method#invoke()`

8.6 Download the Source Code

This was a lesson on Mediator Pattern. You may download the source code here: **MediatorPattern-Project**

Chapter 9

Proxy Design Pattern

9.1 Introduction

In this lesson we will discuss about a Structural Pattern, the Proxy Pattern. The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.

The Proxy Pattern comes up with many different variations. Some of the important variations are, Remote Proxy, Virtual Proxy, and Protection Proxy. In this lesson, we will know more about these variations and we will implement each of them in Java. But before we do that, let's get to know more about the Proxy Pattern in general.

9.2 What is the Proxy Pattern

The Proxy Pattern is used to create a representative object that controls access to another object, which may be remote, expensive to create or in need of being secured.

One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. Another reason could be to act as a local representative for an object that lives in a different JVM. The Proxy can be very useful in controlling the access to the original object, especially when objects should have different access rights.

In the Proxy Pattern, a client does not directly talk to the original object, it delegates it calls to the proxy object which calls the methods of the original object. The important point is that the client does not know about the proxy, the proxy acts as an original object for the client. But there are many variations to this approach which we will see soon.

Let us see the structure of the Proxy Pattern and its important participants.

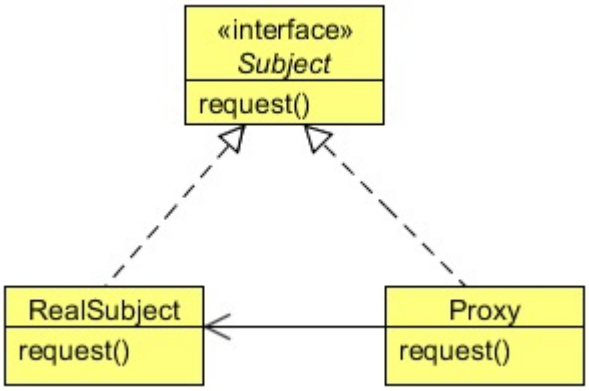

Figure 9.1: screenshot

- **Proxy**: 1a. Maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same. 1b. Provides an interface identical to Subject's so that a proxy can be substituted for the real subject. 1c. Controls access to the real subject and may be responsible for creating and deleting it.
- **Subject**: 2a. Defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject**: 3a. Defines the real object that the proxy represents.

There are three main variations to the Proxy Pattern:

- A remote proxy provides a local representative for an object in a different address space.
- A virtual proxy creates expensive objects on demand.
- A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

We will discuss these proxies one by one in next sections.

9.3 Remote Proxy

There is a Pizza Company, which has its outlets at various locations. The owner of the company gets a daily report by the staff members of the company from various outlets. The current application supported by the Pizza Company is a desktop application, not a web application. So, the owner has to ask his employees to generate the report and send it to him. But now the owner wants to generate and check the report by his own, so that he can generate it whenever he wants without anyone's help. The owner wants you to develop an application for him.

The problem here is that all applications are running at their respective JVMs and the Report Checker application (which we will design soon) should run in the owner's local system. The object required to generate the report does not exist in the owner's system JVM and you cannot directly call on the remote object.

Remote Proxy is used to solve this problem. We know that the report is generated by the users, so there is an object which is required to generate the report. All we need is to contact that object which resides in a remote location in order to get the result that we want. The Remote Proxy acts as a local representative of a remote object. A remote object is an object that lives in the heap of different JVM. You call methods to the local object which forward that calls on to the remote object.

Your client object acts like its making remote method calls. But it is calling methods on a heap-local proxy object that handles all the low-level details of network communication.

Java supports the communication between the two objects residing at two different locations (or two different JVMs) using RMI. RMI is Remote Method Invocation which is used to build the client and service helper objects, right down to creating a client helper object with the same methods as the remote service. Using RMI you don't have to write any of the networking or I/O code yourself. With your client, you call remote methods just like normal method calls on objects running in the client's local JVM.

RMI also provides the running infrastructure to make it all work, including a lookup service that the client can use to find and access the remote objects. There is one difference between RMI calls and local method calls. The client helper send the method call across the network, so there is networking and I/O which involved in the RMI calls.

Now let's take a look at the code. We have an interface `ReportGenerator` and its concrete implementation `ReportGeneratorImpl` already running at JVMs of different locations. First to create a remote service we need to change the codes.

The `ReportGenerator` interface will now looks like this:

```
package com.javacodegeeks.patterns.proxypattern.remoteproxy;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ReportGenerator extends Remote{
```

```
        public String generateDailyReport() throws RemoteException;

}
```

This is a remote interface which defines the methods that a client can call remotely. It's what the client will use as the class type for your service. Both the Stub and actual service will implement this. The method in the interface returns a `String` object. You can return any object from the method; this object is going to be shipped over the wire from the server back to the client, so it must be `Serializable`. Please note that all the methods in this interface should throw `RemoteException`.

```
package com.javacodegeeks.patterns.proxypattern.remoteproxy;

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class ReportGeneratorImpl extends UnicastRemoteObject implements ReportGenerator{

        private static final long serialVersionUID = 3107413009881629428L;

        protected ReportGeneratorImpl() throws RemoteException {
        }

        @Override
        public String generateDailyReport() throws RemoteException {
                StringBuilder sb = new StringBuilder();
                sb.append("********************Location X Daily Report********************" ↩
                    );
                sb.append("\\n Location ID: 012");
                sb.append("\\n Today's Date: "+new Date());
                sb.append("\\n Total Pizza's Sell: 112");
                sb.append("\\n Total Price: $2534");
                sb.append("\\n Net Profit: $1985");
                sb.append("\\n  ↩
                    ************************************************************");

                return sb.toString();
        }

        public static void main(String[] args) {

                try {
                        ReportGenerator reportGenerator = new ReportGeneratorImpl();
                        Naming.rebind("PizzaCoRemoteGenerator", reportGenerator);
                } catch (Exception e) {
                        e.printStackTrace();
                }

        }

}
```

The above class is the remote implementation which does the real work. It's the object that the client wants to call methods on. The class extends `UnicastRemoteObject`, in order to work as a remote service object, your object needs some functionality related to being remote. The simplest way is to extend `UnicastRemoteObject` from the `java.rmi.server` package and let that class do the work for you.

The `UnicastRemoteObject` class constructor throws a `RemoteException`, so you need to write a no-arg constructor that declares a `RemoteException`.

To make the remote service available to the clients you need to register the service with the RMI registry. You do this by instantiating it and putting it into the RMI registry. When you register the implementation object, the RMI system actually puts

the stub in the registry, since that's what a client needs. The `Naming.rebind` method is used to register the object. It has two parameters, first a string to name the service and the other parameter takes object which will be fetched by the clients to use the service.

Now, to create the stub you need to run **rmic** on the remote implementation class. The rmic tool comes with the Java software development kid, takes a service implementation and creates a new stub. You should open your command prompt (cmd) and run rmic from the directory where your remote implementation is located.

The next step is to run the rmiregistry, bring up a terminal and start the registry, just type the command **rmiregistry**. But be sure you start it from a directory that has access to your classes.

The final step is to start the service that is, run your concrete implementation of remote class, in this case the class is `ReportGeneratorImpl`.

So far, we have created and run a service. Now, let's see how the client will use it. The report application for the owner of the pizza company will use this service in order to generate and check the report. We need to provide the interface `ReportGenerator` and the stub to the clients which will use the service. You can simply hand-deliver the stub and any other classes or interfaces required in the service.

```
package com.javacodegeeks.patterns.proxypattern.remoteproxy;

import java.rmi.Naming;

public class ReportGeneratorClient {

        public static void main(String[] args) {
                new ReportGeneratorClient().generateReport();
        }

        public void generateReport(){
                try {
                        ReportGenerator reportGenerator = (ReportGenerator)Naming.lookup(" ↩
                            rmi://127.0.0.1/PizzaCoRemoteGenerator");
                        System.out.println(reportGenerator.generateDailyReport());
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

}
```

The above program will have as a result the following output:

```
********************Location X Daily Report********************
 Location ID: 012
 Today's Date: Sun Sep 14 00:11:23 IST 2014
 Total Pizza Sell: 112
 Total Sale: $2534
 Net Profit: $1985
 ************************************************************
```

The above class does a naming lookup and retrieves the object which is used to generate the daily report. You need to provide the IP of the hostname and the name used to bind the service. The rest of it just looks like a regular old method call.

In conclusion, the Remote Proxy acts as a local representative for an object that lives in a different JVM. A method call on the proxy results in the call being transferred over the wire, invoked remotely, and the result being returned back to the proxy and then to the client.

9.4 Virtual Proxy

The Virtual Proxy pattern is a memory saving technique that recommends postponing an object creation until it is needed; it is used when creating an object the is expensive in terms of memory usage or processing involved. In a typical application,

different objects make up different parts of the functionality. When an application is started, it may not need all of its objects to be available immediately. In such cases, the Virtual Proxy pattern suggests deferring objects creation until it is needed by the application. The object that is created the first time is referenced in the application and the same instance is reused from that point onwards. The advantage of this approach is a faster application start-up time, as it is not required to created and load all of the application objects.

Suppose there is a `Company` object in your application and this object contains a list of employees of the company in a `ContactList` object. There could be thousands of employees in a company. Loading the `Company` object from the database along with the list of all its employees in the `ContactList` object could be very time consuming. In some cases you don't even require the list of the employees, but you are forced to wait until the company and its list of employees loaded into the memory.

One way to save time and memory is to avoid loading of the employee objects until required, and this is done using the Virtual Proxy. This technique is also known as Lazy Loading where you are fetching the data only when it is required.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

public class Company {

        private String companyName;
        private String companyAddress;
        private String companyContactNo;
        private ContactList contactList ;

        public Company(String companyName,String companyAddress, String companyContactNo, ↩
            ContactList contactList){
                this.companyName = companyName;
                this.companyAddress = companyAddress;
                this.companyContactNo = companyContactNo;
                this.contactList = contactList;

                System.out.println("Company object created...");
        }

        public String getCompanyName() {
                return companyName;
        }

        public String getCompanyAddress() {
                return companyAddress;
        }

        public String getCompanyContactNo() {
                return companyContactNo;
        }

        public ContactList getContactList(){
                return contactList;
        }

}
```

The above `Company` class has a reference to `ContactList` interface whose real object will be load only when requested to call of `getContactList()` method.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

import java.util.List;

public interface ContactList {

        public List<Employee> getEmployeeList();
}
```

The `ContactList` interface only contains one method `getEmployeeList()` which is used to get the employee list of the company.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

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

public class ContactListImpl implements ContactList{

        @Override
        public List<Employee> getEmployeeList() {
                return getEmpList();
        }

        private static List<Employee>getEmpList(){
                List<Employee> empList = new ArrayList<Employee>(5);
                empList.add(new Employee("Employee A", 2565.55, "SE"));
                empList.add(new Employee("Employee B", 22574, "Manager"));
                empList.add(new Employee("Employee C", 3256.77, "SSE"));
                empList.add(new Employee("Employee D", 4875.54, "SSE"));
                empList.add(new Employee("Employee E", 2847.01, "SE"));
                return empList;
        }

}
```

The above class will create a real `ContactList` object which will return the list of employees of the company. The object will be loaded on demand, only when required.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

import java.util.List;

public class ContactListProxyImpl implements ContactList{

        private ContactList contactList;

        @Override
        public List<Employee> getEmployeeList() {
                if(contactList == null){
                        System.out.println("Creating contact list and fetching list of  ↩
                            employees...");
                        contactList = new ContactListImpl();
                }
                return contactList.getEmployeeList();
        }

}
```

The `ContactListProxyImpl` is the proxy class which also implements `ContactList` and holds a reference to the real `ContactList` object. On the implementation of the method `getEmployeeList()` it will check if the `contactList` reference is null, then it will create a real `ContactList` object and then will invoke the `getEmployeeList()` method on it to get the list of the employees.

The Employee class looks like this.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

public class Employee {

        private String employeeName;
```

```
        private double employeeSalary;
        private String employeeDesignation;

        public Employee(String employeeName,double employeeSalary,String  ↩
            employeeDesignation){
                this.employeeName = employeeName;
                this.employeeSalary = employeeSalary;
                this.employeeDesignation = employeeDesignation;
        }

        public String getEmployeeName() {
                return employeeName;
        }

        public double getEmployeeSalary() {
                return employeeSalary;
        }

        public String getEmployeeDesignation() {
                return employeeDesignation;
        }

        public String toString(){
                return "Employee Name: "+employeeName+", EmployeeDesignation: "+ ↩
                    employeeDesignation+", Employee Salary: "+employeeSalary;
        }

}

package com.javacodegeeks.patterns.proxypattern.virtualproxy;

import java.util.List;

public class TestVirtualProxy {

        public static void main(String[] args) {
                ContactList contactList = new ContactListProxyImpl();
                Company company = new Company("ABC Company", "India", "+91-011-28458965",  ↩
                    contactList);

                System.out.println("Company Name: "+company.getCompanyName());
                System.out.println("Company Address: "+company.getCompanyAddress());
                System.out.println("Company Contact No.: "+company.getCompanyContactNo());

                System.out.println("Requesting for contact list");

                contactList = company.getContactList();

                List<Employee>empList = contactList.getEmployeeList();
                for(Employee emp : empList){
                        System.out.println(emp);
                }
        }

}
```

The above program will have as a result the following output:

```
Company object created...
Company Name: ABC Company
Company Address: India
Company Contact No.: +91-011-28458965
```

```
Requesting for contact list
Creating contact list and fetching list of employees...
Employee Name: Employee A, EmployeeDesignation: SE, Employee Salary: 2565.55
Employee Name: Employee B, EmployeeDesignation: Manager, Employee Salary: 22574.0
Employee Name: Employee C, EmployeeDesignation: SSE, Employee Salary: 3256.77
Employee Name: Employee D, EmployeeDesignation: SSE, Employee Salary: 4875.54
Employee Name: Employee E, EmployeeDesignation: SE, Employee Salary: 2847.01
```

As you can see in the output generated by the `TestVirtualProxy`, first we have created a `Company` object with a proxy `ContactList` object. At this time, the `Company` object holds a proxy reference, not the real `ContactList` object's reference, so there no employee list loaded into the memory. We made some calls on the company object, and then asked for the employee list from the contact list proxy object using the `getEmployeeList()` method. On call of this method, the proxy object creates a real `ContactList` object and provides the list of employees.

9.5 Protection Proxy

In general, objects in an application interact with each other to implement the overall application functionality. Most application object are generally accessible to all other objects in the application. At times, it may be necessary to restrict the accessibility of an object only to a limited set of client objects based on their access rights. When a client object tries to access such an object, the client is given access to the services provided by the object only if the client can furnish proper authentication credentials. In such cases, a separate object can be designated with the responsibility of verifying the access privileges of different client objects when they access the actual object. In other words, every client must successfully authenticate with this designated object to get access to the actual object functionality. Such an object with which a client needs to authenticate to get access to the actual object can be referred as an object authenticator which is implemented using the Protection Proxy.

Returning back to the `ReportGenerator` application that we developed for the pizza company, the owner now requires that only he can generate the daily report. No other employee should be able to do so.

To implement this security feature, we used Protection Proxy which checks if the object which is trying to generate the report is the owner; in this case, the report gets generated, otherwise it is not.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public interface Staff {

        public boolean isOwner();
        public void setReportGenerator(ReportGeneratorProxy reportGenerator);
}
```

The `Staff` interface is used by the `Owner` and the `Employee` classes and the interface has two methods: `isOwner()` returns a `boolean` to check whether the calling object is the owner or not. The other method is used to set the `ReportGeneratorProxy` which is a protection proxy used to generate the report is `isOwner()` method return true.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public class Employee implements Staff{

        private ReportGeneratorProxy reportGenerator;

        @Override
        public void setReportGenerator(ReportGeneratorProxy reportGenerator){
                this.reportGenerator = reportGenerator;
        }

        @Override
        public boolean isOwner() {
                return false;
        }
```

```
        public String generateDailyReport(){
                try {
                        return reportGenerator.generateDailyReport();
                } catch (Exception e) {
                        e.printStackTrace();
                }
                return "";
        }

}
```

The `Employee` class implements the `Staff` interface, since it's an employee `isOwner()` method return `false`. The `generateDailyReport()` method ask `ReportGenerator` to generate the daily report.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public class Owner implements Staff {

        private boolean isOwner=true;
        private ReportGeneratorProxy reportGenerator;

        @Override
        public void setReportGenerator(ReportGeneratorProxy reportGenerator){
                this.reportGenerator = reportGenerator;
        }

        @Override
        public boolean isOwner(){
                return isOwner;
        }

        public String generateDailyReport(){
                try {
                        return reportGenerator.generateDailyReport();
                } catch (Exception e) {
                        e.printStackTrace();
                }
                return "";
        }

}
```

The `Owner` class looks almost same as the `Employee` class, the only difference is that the `isOwner()` method returns `true`.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public interface ReportGeneratorProxy {

        public String generateDailyReport();
}
```

The `ReportGeneratorProxy` acts as a Protection Proxy which has only one method `generateDailyReport()` that is used to generate the report.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

import java.rmi.Naming;

import com.javacodegeeks.patterns.proxypattern.remoteproxy.ReportGenerator;

public class ReportGeneratorProtectionProxy implements ReportGeneratorProxy{
```

```
        ReportGenerator reportGenerator;
        Staff staff;

        public ReportGeneratorProtectionProxy(Staff staff){
                this.staff = staff;
        }

        @Override
        public String generateDailyReport() {
                if(staff.isOwner()){
                        ReportGenerator reportGenerator = null;
                        try {
                                reportGenerator = (ReportGenerator)Naming.lookup("rmi ↩
                                    ://127.0.0.1/PizzaCoRemoteGenerator");
                                return reportGenerator.generateDailyReport();
                        } catch (Exception e) {
                                e.printStackTrace();
                        }
                        return "";
                }
                else{
                        return "Not Authorized to view report.";
                }
        }
}
```

The above class is the concrete implementation of the `ReportGeneratorProxy` which holds a reference to the `ReportGenerator` interface which is the remote proxy. In the `generateDailyReport()` method, it checks if the staff is referring to the owner, then asks the remote proxy to generate the report, otherwise it returns a string with "Not Authorized to view report" as a message.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public class TestProtectionProxy {

        public static void main(String[] args) {

                Owner owner = new Owner();
                ReportGeneratorProxy reportGenerator = new ReportGeneratorProtectionProxy( ↩
                    owner);
                owner.setReportGenerator(reportGenerator);

                Employee employee = new Employee();
                reportGenerator = new ReportGeneratorProtectionProxy(employee);
                employee.setReportGenerator(reportGenerator);
                System.out.println("For owner:");
                System.out.println(owner.generateDailyReport());
                System.out.println("For employee:");
                System.out.println(employee.generateDailyReport());

        }

}
```

The above program will have as a result the following output:

```
For owner:
********************Location X Daily Report********************
 Location ID: 012
 Today's Date: Sun Sep 14 13:28:12 IST 2014
 Total Pizza Sell: 112
 Total Sale: $2534
```

```
 Net Profit: $1985
 ************************************************************
For employee:
Not Authorized to view report.
```

The above output clearly shows that the owner can generate the report, whereas, the employee does not. The Protection Proxy protects the access of generating the report and only allows the authorized objects to generate the report.

9.6 When to use the Proxy Pattern

Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

- A remote proxy provides a local representative for an object in a different address space.
- A virtual proxy creates expensive objects on demand.
- A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

9.7 Other Proxies

Besides the above discussed three main proxies, there are some other kinds of proxies.

- **Cache Proxy/Server Proxy**: To provide the functionality required to store the results of most frequently used target operations. The proxy object stores these results in some kind of a repository. When a client object requests the same operation, the proxy returns the operation results from the storage area without actually accessing the target object.
- **Firewall Proxy**: The primary use of a firewall proxy is to protect target objects from bad clients. A firewall proxy can also be used to provide the functionality required to prevent clients from accessing harmful targets.
- **Synchronization Proxy**: To provide the required functionality to allow safe concurrent accesses to a target object by different client objects.
- **Smart Reference Proxy**: To provide the functionality to prevent the accidental disposal/deletion of the target object when there are clients currently with references to it. To accomplish this, the proxy keeps a count of the number of references to the target object. The proxy deletes the target object if and when there are no references to it.
- **Counting Proxy**: To provide some kind of audit mechanism before executing a method on the target object.

9.8 Proxy Pattern in JDK

The following cases are examples of usage of the Proxy Pattern in the JDK.

- `java.lang.reflect.Proxy`
- `java.rmi.*` `(whole package)`

9.9 Download the Source Code

This was a lesson on Proxy Pattern. You may download the source code here: **ProxyPattern-Project.zip**

Chapter 10

Chain of Responsibility Design Pattern

10.1 Chain of Responsibility Pattern

The Chain of Responsibility pattern is a behavior pattern in which a group of objects is chained together in a sequence and a responsibility (a request) is provided in order to be handled by the group. If an object in the group can process the particular request, it does so and returns the corresponding response. Otherwise, it forwards the request to the subsequent object in the group.

For a real life scenario, in order to understand this pattern, suppose you got a problem to solve. If you are able to handle it on your own, you will do so, otherwise you will tell your friend to solve it. If he'll able to solve he will do that, or he will also forward it to some other friend. The problem would be forwarded until it gets solved by one of your friends or all your friends have seen the problem, but no one is able to solve it, in which case the problem stays unresolved.

Let's address a real life scenario. Your company has got a contract to provide an analytical application to a health company. The application would tell the user about the particular health problem, its history, its treatment, medicines, interview of the person suffering from it etc, everything that is needed to know about it. For this, your company receives a huge amount of data. The data could be in any format, it could text files, doc files, excels, audio, images, videos, anything that you can think of would be there.

Now, your job is to save this data in the company's database. Users will provide the data in any format and you should provide them a single interface to upload the data into the database. The user is not interested, not even aware, to know that how you are saving the different unstructured data?

The problem here is that you need to develop different handlers to save the various formats of data. For example, a text file save handler does not know how to save an mp3 file.

To solve this problem you can use the Chain of Responsibility design pattern. You can create different objects which process different formats of data and chain them together. When a request comes to a single object, it will check whether it can process and handle the specfic file format. If it can, it will process it; otherwise, it will forward it to the next object chained to it. This design pattern also decouples the user from the object that is serving the request; the user is not aware which object is actually serving its request.

Before solving the problem, let's first know more about the Chain of Responsibility design pattern.

10.2 What is the Chain of Responsibility Pattern

The intent of this pattern is to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. We chain the receiving objects and pass the request along the chain until an object handles it.

This pattern is all about connecting objects in a chain of notification; as a notification travels down the chain, it's handled by the first object that is set up to deal with the particular notification.

When there is more than one objects that can handle or fulfill a client request, the pattern recommends giving each of these objects a chance to process the request in some sequential order. Applying the pattern in such a case, each of these potential

handlers can be arranged in the form of a chain, with each object having a reference to the next object in the chain. The first object in the chain receives the request and decides either to handle the request or to pass it on to the next object in the chain. The request flows through all objects in the chain one after the other until the request is handled by one of the handlers in the chain or the request reaches the end of the chain without getting processed.

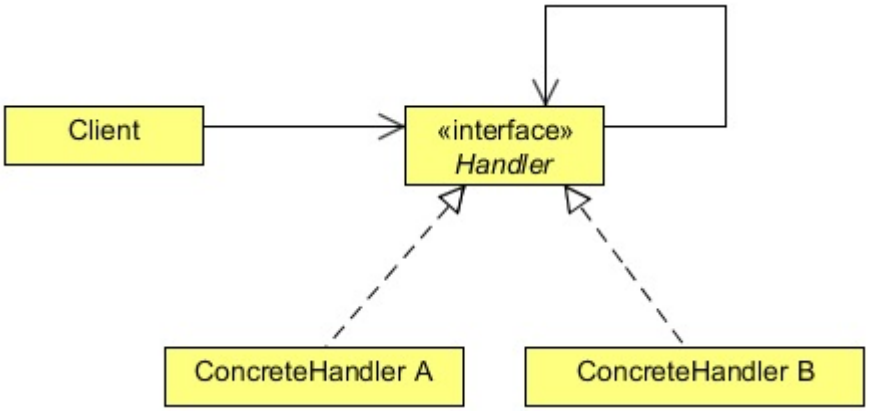

Figure 10.1: screenshot

Handler

- Defines an interface for handling requests.
- (Optionally) Implements the successor link.

ConcreteHandler

- Handles requests it is responsible for.
- Can access its successor.
- If the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.

Client

- Initiates the request to a ConcreteHandler object on the chain.

When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

10.3 Implementing Chain of Responsibility

To implement the Chain of Responsibility in order to solve the above problem, we will create an interface, Handler.

```
package com.javacodegeeks.patterns.chainofresponsibility;

public interface Handler {

        public void setHandler(Handler handler);
        public void process(File file);
        public String getHandlerName();
}
```

The above interface contains two main methods, the `setHandler` and the process methods. The `setHandler` is used to set the next handler in the chain, whereas; the process method is used to process the request, only if the handler can able process the request. Optionally, we have the `getHandlerName` method which is used to return the handler's name.

The handlers are designed to process files which contain data. The concrete handler checks if it's able to handle the file by checking the file type, otherwise forwards to the next handler in the chain.

The `File` class looks like this.

```
package com.javacodegeeks.patterns.chainofresponsibility;

public class File {

        private final String fileName;
        private final String fileType;
        private final String filePath;

        public File(String fileName, String fileType, String filePath){
                this.fileName = fileName;
                this.fileType = fileType;
                this.filePath = filePath;
        }

        public String getFileName() {
                return fileName;
        }

        public String getFileType() {
                return fileType;
        }

        public String getFilePath() {
                return filePath;
        }

}
```

The `File` class creates simple file objects which contain the file name, file type, and the file path. The file type would be used by the handler to check if the file can be handled by them or not. If a handler can, it will process and save it, or it will forward it to the next handler in the chain.

Let's see some concrete handlers now.

```
package com.javacodegeeks.patterns.chainofresponsibility;

public class TextFileHandler implements Handler {

        private Handler handler;
        private String handlerName;

        public TextFileHandler(String handlerName){
                this.handlerName = handlerName;
        }

        @Override
        public void setHandler(Handler handler) {
                this.handler = handler;
        }

        @Override
        public void process(File file) {

                if(file.getFileType().equals("text")){
```

```
                        System.out.println("Process and saving text file... by "+ ↩
                            handlerName);
                }else if(handler!=null){
                        System.out.println(handlerName+" fowards request to "+handler. ↩
                            getHandlerName());
                        handler.process(file);
                }else{
                        System.out.println("File not supported");
                }

        }

        @Override
        public String getHandlerName() {
                return handlerName;
        }
}
```

The `TextFileHandler` is used to handle text files. It implements the `Handler` interface and overrides its three methods. It holds a reference to the next handler in the chain. In the `process` method, it checks the file type if the file type is text, it processes it or it forwards it to the next handler.

The other handlers are similar to the above handler.

```
package com.javacodegeeks.patterns.chainofresponsibility;

public class DocFileHandler implements Handler{

        private Handler handler;
        private String handlerName;

        public DocFileHandler(String handlerName){
                this.handlerName = handlerName;
        }

        @Override
        public void setHandler(Handler handler) {
                this.handler = handler;
        }

        @Override
        public void process(File file) {

                if(file.getFileType().equals("doc")){
                        System.out.println("Process and saving doc file... by "+handlerName ↩
                            );
                }else if(handler!=null){
                        System.out.println(handlerName+" fowards request to "+handler. ↩
                            getHandlerName());
                        handler.process(file);
                }else{
                        System.out.println("File not supported");
                }

        }

        @Override
        public String getHandlerName() {
                return handlerName;
        }

}
```

```
package com.javacodegeeks.patterns.chainofresponsibility;

public class AudioFileHandler implements Handler {

        private Handler handler;
        private String handlerName;

        public AudioFileHandler(String handlerName){
                this.handlerName = handlerName;
        }

        @Override
        public void setHandler(Handler handler) {
                this.handler = handler;
        }

        @Override
        public void process(File file) {

                if(file.getFileType().equals("audio")){
                        System.out.println("Process and saving audio file... by "+ ↩
                            handlerName);
                }else if(handler!=null){
                        System.out.println(handlerName+" fowards request to "+handler. ↩
                            getHandlerName());
                        handler.process(file);
                }else{
                        System.out.println("File not supported");
                }

        }

        @Override
        public String getHandlerName() {
                return handlerName;
        }

}

package com.javacodegeeks.patterns.chainofresponsibility;

public class ExcelFileHandler implements Handler{

        private Handler handler;
        private String handlerName;

        public ExcelFileHandler(String handlerName){
                this.handlerName = handlerName;
        }

        @Override
        public void setHandler(Handler handler) {
                this.handler = handler;
        }

        @Override
        public void process(File file) {

                if(file.getFileType().equals("excel")){
                        System.out.println("Process and saving excel file... by "+ ↩
                            handlerName);
```

```
                }else if(handler!=null){
                        System.out.println(handlerName+" fowards request to "+handler. ↩
                            getHandlerName());
                        handler.process(file);
                }else{
                        System.out.println("File not supported");
                }

        }

        @Override
        public String getHandlerName() {
                return handlerName;
        }
}

package com.javacodegeeks.patterns.chainofresponsibility;

public class ImageFileHandler implements Handler {

        private Handler handler;
        private String handlerName;

        public ImageFileHandler(String handlerName){
                this.handlerName = handlerName;
        }

        @Override
        public void setHandler(Handler handler) {
                this.handler = handler;
        }

        @Override
        public void process(File file) {

                if(file.getFileType().equals("image")){
                        System.out.println("Process and saving image file... by "+ ↩
                            handlerName);
                }else if(handler!=null){
                        System.out.println(handlerName+" fowards request to "+handler. ↩
                            getHandlerName());
                        handler.process(file);
                }else{
                        System.out.println("File not supported");
                }

        }

        @Override
        public String getHandlerName() {
                return handlerName;
        }

}

package com.javacodegeeks.patterns.chainofresponsibility;

public class VideoFileHandler implements Handler {

        private Handler handler;
        private String handlerName;
```

```
        public VideoFileHandler(String handlerName){
                this.handlerName = handlerName;
        }

        @Override
        public void setHandler(Handler handler) {
                this.handler = handler;
        }

        @Override
        public void process(File file) {

                if(file.getFileType().equals("video")){
                        System.out.println("Process and saving video file... by "+ ↩
                            handlerName);
                }else if(handler!=null){
                        System.out.println(handlerName+" fowards request to "+handler. ↩
                            getHandlerName());
                        handler.process(file);
                }else{
                        System.out.println("File not supported");
                }

        }

        @Override
        public String getHandlerName() {
                return handlerName;
        }
}
```

Now, let's test the code above.

```
package com.javacodegeeks.patterns.chainofresponsibility;

public class TestChainofResponsibility {

        public static void main(String[] args) {
                File file = null;
                Handler textHandler = new TextFileHandler("Text Handler");
                Handler docHandler = new DocFileHandler("Doc Handler");
                Handler excelHandler = new ExcelFileHandler("Excel Handler");
                Handler audioHandler = new AudioFileHandler("Audio Handler");
                Handler videoHandler = new VideoFileHandler("Video Handler");
                Handler imageHandler = new ImageFileHandler("Image Handler");

                textHandler.setHandler(docHandler);
                docHandler.setHandler(excelHandler);
                excelHandler.setHandler(audioHandler);
                audioHandler.setHandler(videoHandler);
                videoHandler.setHandler(imageHandler);

                file = new File("Abc.mp3", "audio", "C:");
                textHandler.process(file);

                file = new File("Abc.jpg", "video", "C:");
                textHandler.process(file);

                file = new File("Abc.doc", "doc", "C:");
                textHandler.process(file);

                file = new File("Abc.bat", "bat", "C:");
```

```
                textHandler.process(file);
        }

}
```

The above program will have the following output.

```
Text Handler fowards request to Doc Handler
Doc Handler fowards request to Excel Handler
Excel Handler fowards request to Audio Handler
Process and saving audio file... by Audio Handler

Text Handler fowards request to Doc Handler
Doc Handler fowards request to Excel Handler
Excel Handler fowards request to Audio Handler
Audio Handler fowards request to Video Handler
Process and saving video file... by Video Handler

Text Handler fowards request to Doc Handler
Process and saving doc file... by Doc Handler

Text Handler fowards request to Doc Handler
Doc Handler fowards request to Excel Handler
Excel Handler fowards request to Audio Handler
Audio Handler fowards request to Video Handler
Video Handler fowards request to Image Handler
File not supported
```

In the example above, first we created different handlers and chained them. The chain starts from the text handler, which is used to process text files, to the doc handler and so on, till the last handler, the image handler.

Then we created different file objects and passed it to the text handler. If the file can be processed by the text handler it does that, otherwise it forwards the file to the next chained handler. You can see in the output how the requested file was forwarded by the chained objects until it reached the appropriate handler.

Also, please note down, we have not created a handler to process a bat file. So, it passes through all the handlers and results in the output - "File not supported".

The client code is decoupled from the served object. It only sends the request, and the request gets served by any one of the handlers in the chain or does not get processed in case there is support for it.

10.4 When to use the Chain of Responsibility Pattern

Use Chain of Responsibility when

- More than one objects may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

10.5 Chain of Responsibility in JDK

The following are the usages of the Chain of Responsibility Pattern in Java.

- `java.util.logging.Logger#log()`
- `javax.servlet.Filter#doFilter()`

10.6 Download the Source Code

This was a lesson on the Chain of Responsibility Pattern. You may download the source code here: **ChainofResponsibility-Project**

Chapter 11

Flyweight Design Pattern

11.1 Flyweight Pattern

Object Oriented programming has made programming easy and interesting. It makes a programmer's job easier by modeling real world entities into the programming world. A programmer creates a class and instantiates it by creating an object of it. This object models a real world entity and objects inside an application coordinate with each other in order to accomplish the required work.

But sometimes too many objects can slow things down. Too many objects might consume a large piece of memory and can slow down the application or even cause out of memory problems. As a good programmer, one should keep track of instantiated objects and control the object creation in an application. This is especially true, when we have a lot of similar objects and two objects from the pool don't have much differences between them.

Sometimes the objects in an application might have great similarities and be of a similar kind (a similar kind here means that most of their properties have similar values and only a few of them vary in value). In case they are also heavy objects to create, they should be controlled by the application developer. Otherwise, they might consume much of the memory and eventually slow down the whole application.

The Flyweight Pattern is designed to control such kind of object creation and provides you with a basic caching mechanism. It allows you to create one object per type (the type here differs by a property of that object), and if you ask for an object with the same property (already created), it will return you the same object instead of creating a new one.

Before digging into the details of the Flyweight pattern, let's consider the following scenario: a site which allows users to create and execute programs online. We will discuss the scenario now and we will later try to solve the problem using the Flyweight pattern.

The X-programming site allows users to create and execute programs using their favorite programming language. It provides you with plenty of programming language options. You choose one, write a program with it and execute it to see the result.

But now the site has started losing its users, the reason being the slowness of the site. The users are not interested in it any more. The site is very popular and sometimes there could be more than thousands of programmers using it. Because of that, the site is crawling. But the heavy usage is not the real problem behind the slowness of the site. Let us see the core programming of the site which allows users to run and execute their program, and the true issue will be revealed there.

```
package com.javacodegeeks.patterns.flyweightpattern;

public class Code {

        private String code;

        public String getCode() {
                return code;
        }

        public void setCode(String code) {
```

```
                this.code = code;
        }

}
```

The above class is used to set the code done by the programmer in order to get it executed. The `Code` object is a lightweight simple object having a property `code` along with its setter and getter.

```
package com.javacodegeeks.patterns.flyweightpattern;

public interface Platform {

        public void execute(Code code);
}
```

The `Platform` interface is implemented by the language specific platform in order to execute the code. It has one method, `executes`, which takes the `Code` object as its parameter.

```
package com.javacodegeeks.patterns.flyweightpattern;

public class JavaPlatform implements Platform {

        public JavaPlatform(){
                System.out.println("JavaPlatform object created");
        }

        @Override
        public void execute(Code code) {
                System.out.println("Compiling and executing Java code.");
        }

}
```

The above class implements the `Platform` interface and provides an implementation for the `execute` method, to execute the code in Java.

To execute the code, a `Code` object which contains the code and a `Platform` object to execute the code get created. The code looks like this:

```
Platform platform = new JavaPlatform();
platform.execute(code);
```

Now suppose, there are around 2k users online and executing their code which results in 2k `Code` objects and 2k `Platform` objects. The `Code` object is a light weighted object and there should also be one `Code` object per user code. But, the `Platform` is a heavy object which is used to set the execution environment. Creating too many `Platform` objects is time consuming, and a heavy task. We need to control the creation of the `Platform` object which can be done using the Flyweight Pattern, but before that, let's look at the detail of the Flyweight Pattern.

11.2 What is the Flyweight Pattern

The intent of the Flyweight Pattern is to use shared objects to support large numbers of fine-grained objects efficiently. 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. The key concept here is the distinction between intrinsic and extrinsic state. **Intrinsic state** is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable. The **extrinsic state** depends on and varies with the flyweight's context and therefore can't be shared. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.

Consider an application scenario that involves creating a large number of objects that are unique only in terms of a few parameters. In other words, these objects contain some intrinsic, invariant data that are common among all objects. This intrinsic data needs to be created and maintained as part of every object that is being created. The overall creation and maintenance of a large group of such objects can be very expensive in terms of memory-usage and performance. The Flyweight pattern can be used in such scenarios to design a more efficient way of creating objects.

Here is the class diagram for the Flyweight design pattern:

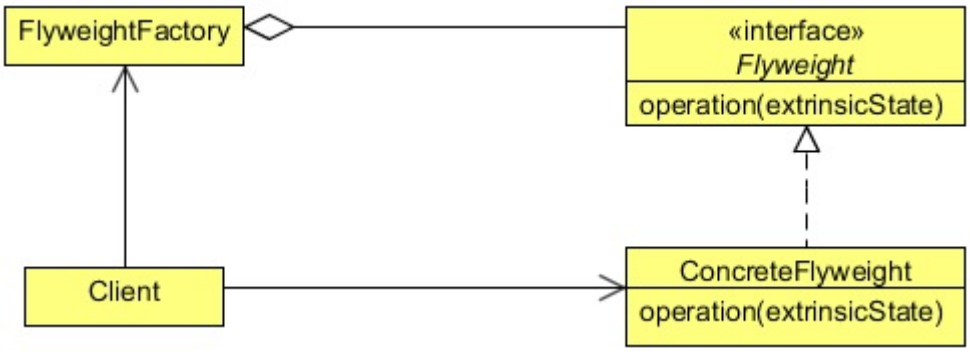

Figure 11.1: screenshot

Flyweight

- Declares an interface through which flyweights can receive and act on extrinsic state.

ConcreteFlyweight

- Implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.

FlyweightFactory

- Creates and manages flyweight objects.
- Ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.

Client

- Maintains a reference to flyweight(s).
- Computes or stores the extrinsic state of flyweight(s).

11.3 Solution to the Problem

To solve the above problem, we will provide a platform factory class which will control the creation of the Platform objects.

```
package com.javacodegeeks.patterns.flyweightpattern;

import java.util.HashMap;
import java.util.Map;

public final class PlatformFactory {
```

```
        private static Map<String, Platform> map = new HashMap<>();
        private PlatformFactory(){
                throw new AssertionError("Cannot instantiate the class");
        }

        public static synchronized Platform getPlatformInstance(String platformType){
                Platform platform = map.get(platformType);
                if(platform==null){
                        switch(platformType){
                                case "C" : platform = new CPlatform();
                                                break;
                                case "CPP" : platform = new CPPPlatform();
                                                break;
                                case "JAVA" : platform = new JavaPlatform();
                                                break;
                                case "RUBY" : platform = new RubyPlatform();
                                                break;
                        }
                        map.put(platformType, platform);
                }
                return platform;
        }

}
```

The above class contains a static map which holds a `String` object as key and a `Platform` type object as its value. We don't want to create the instance of this class so just kept its constructor private and throw an `AssertionError` just to avoid any accidental creation of the object even within the class.

The main and the only method of this class is the `getPlatformInstance` method. This is a static method which has a `platformType` as its parameter. This `platformType` is used as the key in the map, it first checks the map whether a platform object having the key is already exists or not. If no object found, the appropriate platform object gets created, it is put into the map and then the method returns the object. Next time, when the same platform type object is requested, the same existing object is returned, instead of a new object.

Also, please note that the `getPlatformInstance` method is `synchronized` in order to provide the thread safety while checking and creating the instance of the object. In the above example, there isn't any intrinsic property of the object that is shared, but only the extrinsic property which is the code object provided by the client code.

Now, let's test the code.

```
package com.javacodegeeks.patterns.flyweightpattern;

public class TestFlyweight {

        public static void main(String[] args) {

                Code code = new Code();
                code.setCode("C Code...");
                Platform platform = PlatformFactory.getPlatformInstance("C");
                platform.execute(code);
                System.out.println("*************************");
                code = new Code();
                code.setCode("C Code2...");
                platform = PlatformFactory.getPlatformInstance("C");
                platform.execute(code);
                System.out.println("*************************");
                code = new Code();
                code.setCode("JAVA Code...");
                platform = PlatformFactory.getPlatformInstance("JAVA");
                platform.execute(code);
                System.out.println("*************************");
```

```
                code = new Code();
                code.setCode("JAVA Code2...");
                platform = PlatformFactory.getPlatformInstance("JAVA");
                platform.execute(code);
                System.out.println("************************");
                code = new Code();
                code.setCode("RUBY Code...");
                platform = PlatformFactory.getPlatformInstance("RUBY");
                platform.execute(code);
                System.out.println("************************");
                code = new Code();
                code.setCode("RUBY Code2...");
                platform = PlatformFactory.getPlatformInstance("RUBY");
                platform.execute(code);
        }

}
```

The above code will result into the following output:

```
CPlatform object created
Compiling and executing C code.
************************
Compiling and executing C code.
************************
JavaPlatform object created
Compiling and executing Java code.
************************
Compiling and executing Java code.
************************
RubyPlatform object created
Compiling and executing Ruby code.
************************
Compiling and executing Ruby code.
```

In the above class, we have first created a `Code` object and set the C code in it. Then, we asked the `PlatformFactory` to provide C platform to execute the code. Later, we called the `execute` method on the object returned, bypassing the `Code` object to it.

We performed the same procedure, i.e. creating and setting the `Code` object, and then asking for a platform object, specific to the code. The output clearly shows that the platform objects created only the first time they are requested for; on the next attempts the same object gets returned.

The other platform specific classes are similar to the `JavaPlatform` class, already shown.

```
package com.javacodegeeks.patterns.flyweightpattern;

public class CPlatform implements Platform {

        public CPlatform(){
                System.out.println("CPlatform object created");
        }

        @Override
        public void execute(Code code) {
                System.out.println("Compiling and executing C code.");
        }

}

package com.javacodegeeks.patterns.flyweightpattern;
```

```
public class CPPPlatform implements Platform{

        public CPPPlatform(){
                System.out.println("CPPPlatform object created");
        }

        @Override
        public void execute(Code code) {
                System.out.println("Compiling and executing CPP code.");
        }

}

package com.javacodegeeks.patterns.flyweightpattern;

public class RubyPlatform implements Platform{

        public RubyPlatform(){
                System.out.println("RubyPlatform object created");
        }

        @Override
        public void execute(Code code) {
                System.out.println("Compiling and executing Ruby code.");
        }

}
```

If we reconsider that 2k users concurrently use the site, exactly 2k light weight `Code` object get created, and only 4 heavy weight platform objects are instantiated. Please note that, we are saying 4 platform objects by considering that there would be at least one user per language. If, for example, let say no user have coded using the Ruby, only 3 platform objects get created in that scenario.

11.4 When to use the Flyweight Pattern

The Flyweight pattern's effectiveness depends heavily on how and where it's used. Apply the Flyweight pattern when all of the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

11.5 Flyweight in the JDK

The following are the usage(s) of the Flyweight Pattern in JDK.

- `java.lang.Integer#valueOf(int)` (also on Boolean, Byte, Character, Short and Long)

11.6 Download the Source Code

This was a lesson on the Flyweight Pattern. You may download the source code here: **Flyweight Pattern Project**

Chapter 12

Builder Design Pattern

12.1 Builder Pattern

In general, the details of object construction, such as instantiating and initializing the components that make up the object, are kept within the object, often as part of its constructor. This type of design closely ties the object construction process with the components that make up the object. This approach is suitable as long as the object under construction is simple and the object construction process is definite and always produces the same representation of the object.

However, this design may not be effective when the object being created is complex and the series of steps constituting the object creation process can be implemented in different ways, thus producing different representations of the object. Because the different implementations of the construction process are all kept within the object, the object can become bulky (construction bloat) and less modular. Subsequently, adding a new implementation or making changes to an existing implementation requires changes to the existing code.

Using the Builder pattern, the process of constructing such an object can be designed more effectively. The Builder pattern suggests moving the construction logic out of the object class to a separate class referred to as a builder class. There can be more than one such builder classes, each with different implementations for the series of steps to construct the object. Each builder implementation results in a different representation of the object.

To illustrate the use of the Builder Pattern, let's help a Car company which shows its different cars using a graphical model to its customers. The company has a graphical tool which displays the car on the screen. The requirement of the tool is to provide a car object to it. The car object should contain the car's specifications. The graphical tool uses these specifications to display the car. The company has classified its cars into different classifications like Sedan or Sports Car. There is only one car object, and our job is to create the car object according to the classification. For example, for a Sedan car, a car object according to the sedan specification should be built or, if a sports car is required, then a car object according to the sports car specification should be built. Currently, the Company wants only these two types of cars, but it may require other types of cars also in the future.

We will create two different builders, one of each classification, i.e., for sedan and sports cars. These two builders will help us in building the car object according to its specification. But before that, let's have discuss some details of the Builder Pattern.

12.2 What is the Builder Pattern

The intent of the Builder Pattern is to separate the construction of a complex object from its representation, so that the same construction process can create different representations. This type of separation reduces the object size. The design turns out to be more modular with each implementation contained in a different builder object. Adding a new implementation (i.e., adding a new builder) becomes easier. The object construction process becomes independent of the components that make up the object. This provides more control over the object construction process.

In terms of implementation, each of the different steps in the construction process can be declared as methods of a common interface to be implemented by different concrete builders.

A client object can create an instance of a concrete builder and invoke the set of methods required to construct different parts of the final object. This approach requires every client object to be aware of the construction logic. Whenever the construction logic undergoes a change, all client objects need to be modified accordingly.

The Builder pattern introduces another level of separation that addresses this problem. Instead of having client objects invoke different builder methods directly, the Builder pattern suggests using a dedicated object referred to as a Director, which is responsible for invoking different builder methods required for the construction of the final object. Different client objects can make use of the Director object to create the required object. Once the object is constructed, the client object can directly request from the builder the fully constructed object. To facilitate this process, a new method `getObject` can be declared in the common Builder interface to be implemented by different concrete builders.

The new design eliminates the need for a client object to deal with the methods constituting the object construction process and encapsulates the details of how the object is constructed from the client.

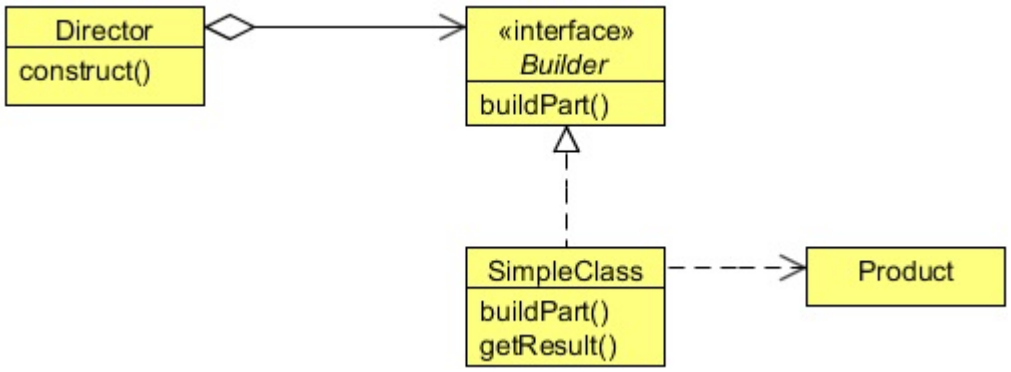

Figure 12.1: screenshot

Builder

- Specifies an abstract interface for creating parts of a Product object.

ConcreteBuilder

- Constructs and assembles parts of the product by implementing the Builder interface.
- Defines and keeps track of the representation it creates.
- Provides an interface for retrieving the product.

Director

- Constructs an object using the Builder interface.

Product

- Represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
- Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

12.3 Implementing the Builder Pattern

Blow is the `Car` class (Product) which contains some of the important components of the car that are required in order to construct the complete `car` object.

```
package com.javacodegeeks.patterns.builderpattern;

public class Car {

	private String bodyStyle;
	private String power;
	private String engine;
	private String breaks;
	private String seats;
	private String windows;
	private String fuelType;
	private String carType;

	public Car (String carType){
		this.carType = carType;
	}

	public String getBodyStyle() {
		return bodyStyle;
	}
	public void setBodyStyle(String bodyStyle) {
		this.bodyStyle = bodyStyle;
	}
	public String getPower() {
		return power;
	}
	public void setPower(String power) {
		this.power = power;
	}
	public String getEngine() {
		return engine;
	}
	public void setEngine(String engine) {
		this.engine = engine;
	}
	public String getBreaks() {
		return breaks;
	}
	public void setBreaks(String breaks) {
		this.breaks = breaks;
	}
	public String getSeats() {
		return seats;
	}
	public void setSeats(String seats) {
		this.seats = seats;
	}
	public String getWindows() {
		return windows;
	}
	public void setWindows(String windows) {
		this.windows = windows;
	}
	public String getFuelType() {
		return fuelType;
	}
```

```
        public void setFuelType(String fuelType) {
                this.fuelType = fuelType;
        }

        @Override
        public String toString(){
                StringBuilder sb = new StringBuilder();
                sb.append("--------------"+carType+"---------------------- \\n");
                sb.append(" Body: ");
                sb.append(bodyStyle);
                sb.append("\\n Power: ");
                sb.append(power);
                sb.append("\\n Engine: ");
                sb.append(engine);
                sb.append("\\n Breaks: ");
                sb.append(breaks);
                sb.append("\\n Seats: ");
                sb.append(seats);
                sb.append("\\n Windows: ");
                sb.append(windows);
                sb.append("\\n Fuel Type: ");
                sb.append(fuelType);

                return sb.toString();
        }
}
```

The `CarBuilder` is the builder interface contains set of common methods used to build the `car` object and its components.

```
package com.javacodegeeks.patterns.builderpattern;

public interface CarBuilder {

        public void buildBodyStyle();
        public void buildPower();
        public void buildEngine();
        public void buildBreaks();
        public void buildSeats();
        public void buildWindows();
        public void buildFuelType();
        public Car getCar();
}
```

The `getCar` method is used to return the final car object to the client after its construction.

Let's see two implementations of the `CarBuilder` interface, one for each type of car, i.e., for sedan and sports car.

```
package com.javacodegeeks.patterns.builderpattern;

public class SedanCarBuilder implements CarBuilder{

        private final Car car = new Car("SEDAN");

        @Override
        public void buildBodyStyle() {
                car.setBodyStyle("External dimensions: overall length (inches): 202.9, " +
                                "overall width (inches): 76.2, overall height (inches):  ↩
                                    60.7, wheelbase (inches): 112.9," +
                                " front track (inches): 65.3, rear track (inches): 65.5 and ↩
                                     curb to curb turning circle (feet): 39.5");
        }
```

```
        @Override
        public void buildPower(){
                car.setPower("285 hp @ 6,500 rpm; 253 ft lb of torque @ 4,000 rpm");
        }

        @Override
        public void buildEngine() {
                car.setEngine("3.5L Duramax V 6 DOHC");
        }

        @Override
        public void buildBreaks() {
                car.setBreaks("Four-wheel disc brakes: two ventilated. Electronic brake ↩
                    distribution");
        }

        @Override
        public void buildSeats() {
                car.setSeats("Front seat center armrest.Rear seat center armrest.Split- ↩
                    folding rear seats");
        }

        @Override
        public void buildWindows() {
                car.setWindows("Laminated side windows.Fixed rear window with defroster");

        }

        @Override
        public void buildFuelType() {
                car.setFuelType("Gasoline 19 MPG city, 29 MPG highway, 23 MPG combined and ↩
                    437 mi. range");

        }

        @Override
        public Car getCar(){
                return car;
        }

}
```

```
package com.javacodegeeks.patterns.builderpattern;

public class SportsCarBuilder implements CarBuilder{

        private final Car car = new Car("SPORTS");

        @Override
        public void buildBodyStyle() {
                car.setBodyStyle("External dimensions: overall length (inches): 192.3," +
                                " overall width (inches): 75.5, overall height (inches): ↩
                                    54.2, wheelbase (inches): 112.3," +
                                " front track (inches): 63.7, rear track (inches): 64.1 and ↩
                                    curb to curb turning circle (feet): 37.7");
        }

        @Override
        public void buildPower(){
                car.setPower("323 hp @ 6,800 rpm; 278 ft lb of torque @ 4,800 rpm");
        }
```

```
        @Override
        public void buildEngine() {
                car.setEngine("3.6L V 6 DOHC and variable valve timing");
        }

        @Override
        public void buildBreaks() {
                car.setBreaks("Four-wheel disc brakes: two ventilated. Electronic brake ↩
                    distribution. StabiliTrak stability control");

        }

        @Override
        public void buildSeats() {
                car.setSeats("Driver sports front seat with one power adjustments manual ↩
                    height, front passenger seat sports front seat with one power ↩
                    adjustments");
        }

        @Override
        public void buildWindows() {
                car.setWindows("Front windows with one-touch on two windows");

        }

        @Override
        public void buildFuelType() {
                car.setFuelType("Gasoline 17 MPG city, 28 MPG highway, 20 MPG combined and ↩
                    380 mi. range");
        }

        @Override
        public Car getCar(){
                return car;
        }

}
```

The above two builders create and construct the product, i.e. the `car` object according to the specification required.

Now, let's test the Builder.

```
package com.javacodegeeks.patterns.builderpattern;

public class TestBuilderPattern {

        public static void main(String[] args) {
                CarBuilder carBuilder = new SedanCarBuilder();
                CarDirector director = new CarDirector(carBuilder);
                director.build();
                Car car = carBuilder.getCar();
                System.out.println(car);

                carBuilder = new SportsCarBuilder();
                director = new CarDirector(carBuilder);
                director.build();
                car = carBuilder.getCar();
                System.out.println(car);
        }

}
```

The above code will result to the following output.

```
-------------SEDAN---------------------
 Body: External dimensions: overall length (inches): 202.9, overall width (inches): 76.2,  ↩
     overall height (inches): 60.7, wheelbase (inches): 112.9, front track (inches): 65.3,  ↩
     rear track (inches): 65.5 and curb to curb turning circle (feet): 39.5
 Power: 285 hp @ 6,500 rpm; 253 ft lb of torque @ 4,000 rpm
 Engine: 3.5L Duramax V 6 DOHC
 Breaks: Four-wheel disc brakes: two ventilated. Electronic brake distribution
 Seats: Front seat center armrest.Rear seat center armrest.Split-folding rear seats
 Windows: Laminated side windows.Fixed rear window with defroster
 Fuel Type: Gasoline 19 MPG city, 29 MPG highway, 23 MPG combined and 437 mi. range
-------------SPORTS---------------------
 Body: External dimensions: overall length (inches): 192.3, overall width (inches): 75.5,  ↩
     overall height (inches): 54.2, wheelbase (inches): 112.3, front track (inches): 63.7,  ↩
     rear track (inches): 64.1 and curb to curb turning circle (feet): 37.7
 Power: 323 hp @ 6,800 rpm; 278 ft lb of torque @ 4,800 rpm
 Engine: 3.6L V 6 DOHC and variable valve timing
 Breaks: Four-wheel disc brakes: two ventilated. Electronic brake distribution. StabiliTrak ↩
      stability control
 Seats: Driver sports front seat with one power adjustments manual height, front passenger  ↩
     seat sports front seat with one power adjustments
 Windows: Front windows with one-touch on two windows
 Fuel Type: Gasoline 17 MPG city, 28 MPG highway, 20 MPG combined and 380 mi. range
```

In the above class, we have first created a `SedanCarBuilder` and a `CarDirector`. Then we ask the `CarDirector` to build the `car` for us according to the `builder` passed to it. Then finally, we directly ask the `builder` to provide us the created `car` object.

We did the same for the `SportsCarBuilder` which returns the `car` object according to the sports car specification.

The approach to use the Builder Pattern is flexible enough to add any new type of a car in the future without changing any of the existing code. All we need is to create a new builder according to the specification of the new car and provide it to the Director to build.

12.4 Another form of the Builder Pattern

There is another form of the Builder Pattern other than what we have seen so far. Sometimes there is an object with a long list of properties, and most of these properties are optional. Consider an online form which needs to be filled in order to become a member of a site. You need to fill all the mandatory fields but you can skip the optional fields or sometimes it may look valuable to fill some of the optional fields.

Please check the below `Form` class which contains long list of properties and some of the properties are optional. It is mandatory to have `firstName`, `lastName`, `userName`, and `password` in the form but all others are optional fields.

```
package com.javacodegeeks.patterns.builderpattern;

import java.util.Date;

public class Form {

        private String firstName;
        private String lastName;
        private String userName;
        private String password;
        private String address;
        private Date dob;
        private String email;
        private String backupEmail;
        private String spouseName;
        private String city;
```

```
        private String state;
        private String country;
        private String language;
        private String passwordHint;
        private String secuirtyQuestion;
        private String securityAnswer;

}
```

The question is, what sort of constructor should we write for such a class? Well writing a constructor with long list of parameters is not a good choice, this could frustrate the client especially if the important fields are only a few. This could increase the scope of error; the client may provide a value accidentally to a wrong field.

Another way is to use telescoping constructors, in which you provide a constructor with only the required parameters, another with a single optional parameter, a third with two optional parameters, and so on, culminating in a constructor with all the optional parameters.

Telescoping constructor might looks like this.

```
public Form(String firstName,String lastName){
        this(firstName,lastName,null,null);
}

public Form(String firstName,String lastName,String userName,String password){
        this(firstName,lastName,userName,password,null,null);
}

public Form(String firstName,String lastName,String userName,String password,String address ↩
    ,Date dob){
        this(firstName,lastName,userName,password,address,dob,null,null);
}

public Form(String firstName,String lastName,String userName,String password,String address ↩
    ,Date dob,String email,String backupEmail){
        ...
}
```

When you want to create an instance, you use the constructor with the shortest parameter list containing all the parameters you want to set.

The telescoping constructor works, but it is hard to write client code when there are many parameters, and it is even harder to read it. The reader is left wondering what all those values mean and must carefully count parameters to find out. Long sequences of identically typed parameters can cause subtle bugs. If the client accidentally reverses two such parameters, the compiler won't complain, but the program will misbehave at runtime.

A second alternative when you are faced with many constructor parameters is the JavaBeans pattern, in which you call a parameter less constructor to create the object and then call setter methods to set each required parameter and each optional parameter of interest.

Unfortunately, the JavaBeans pattern has serious disadvantages of its own. Because construction is split across multiple calls, a JavaBean may be in an inconsistent state partway through its construction. The class does not have the option of enforcing consistency merely by checking the validity of the constructor parameters. Attempting to use an object when it's in an inconsistent state may cause failures that are far removed from the code containing the bug, hence difficult to debug.

There is a third alternative that combines the safety of the telescoping constructor pattern with the readability of the JavaBeans pattern. It is a form of the Builder pattern. Instead of making the desired object directly, the client calls a constructor with all of the required parameters and gets a builder object. Then the client calls setter-like methods on the builder object to set each optional parameter of interest. Finally, the client calls a parameter less build method to generate the object.

```
package com.javacodegeeks.patterns.builderpattern;

import java.util.Date;
```

```
public class Form {

        private String firstName;
        private String lastName;
        private String userName;
        private String password;
        private String address;
        private Date dob;
        private String email;
        private String backupEmail;
        private String spouseName;
        private String city;
        private String state;
        private String country;
        private String language;
        private String passwordHint;
        private String securityQuestion;
        private String securityAnswer;

        public static class FormBuilder {

                private String firstName;
                private String lastName;
                private String userName;
                private String password;
                private String address;
                private Date dob;
                private String email;
                private String backupEmail;
                private String spouseName;
                private String city;
                private String state;
                private String country;
                private String language;
                private String passwordHint;
                private String securityQuestion;
                private String securityAnswer;

                public FormBuilder(String firstName, String lastName, String userName,  ↩
                    String password){
                        this.firstName = firstName;
                        this.lastName = lastName;
                        this.userName = userName;
                        this.password = password;
                }

                public FormBuilder address(String address){
                        this.address = address;
                        return this;
                }

                public FormBuilder dob(Date dob){
                        this.dob = dob;
                        return this;
                }

                public FormBuilder email(String email){
                        this.email = email;
                        return this;
                }

                public FormBuilder backupEmail(String backupEmail){
```

```
                this.backupEmail = backupEmail;
                return this;
        }

        public FormBuilder spouseName(String spouseName){
                this.spouseName = spouseName;
                return this;
        }

        public FormBuilder city(String city){
                this.city = city;
                return this;
        }

        public FormBuilder state(String state){
                this.state = state;
                return this;
        }

        public FormBuilder country(String country){
                this.country = country;
                return this;
        }

        public FormBuilder language(String language){
                this.language = language;
                return this;
        }

        public FormBuilder passwordHint(String passwordHint){
                this.passwordHint = passwordHint;
                return this;
        }

        public FormBuilder securityQuestion(String securityQuestion){
                this.securityQuestion = securityQuestion;
                return this;
        }

        public FormBuilder securityAnswer(String securityAnswer){
                this.securityAnswer = securityAnswer;
                return this;
        }

        public Form build(){
                return new Form(this);
        }
}

private Form(FormBuilder formBuilder){

        this.firstName = formBuilder.firstName;
        this.lastName = formBuilder.lastName;
        this.userName = formBuilder.userName;
        this.password = formBuilder.password;
        this.address = formBuilder.address;
        this.dob = formBuilder.dob;
        this.email = formBuilder.email;
        this.backupEmail = formBuilder.backupEmail;
        this.spouseName = formBuilder.spouseName;
        this.city = formBuilder.city;
        this.state = formBuilder.state;
```

```
                this.country =  formBuilder.country;
                this.language = formBuilder.language;
                this.passwordHint = formBuilder.passwordHint;
                this.securityQuestion = formBuilder.securityQuestion;
                this.securityAnswer = formBuilder.securityAnswer;
        }

        @Override
        public String toString(){
                StringBuilder sb = new StringBuilder();
                sb.append(" First Name: ");
                sb.append(firstName);
                sb.append("\\n Last Name: ");
                sb.append(lastName);
                sb.append("\\n User Name: ");
                sb.append(userName);
                sb.append("\\n Password: ");
                sb.append(password);

                if(this.address!=null){
                        sb.append("\\n Address: ");
                        sb.append(address);
                }
                if(this.dob!=null){
                        sb.append("\\n DOB: ");
                        sb.append(dob);
                }
                if(this.email!=null){
                        sb.append("\\n Email: ");
                        sb.append(email);
                }
                if(this.backupEmail!=null){
                        sb.append("\\n Backup Email: ");
                        sb.append(backupEmail);
                }
                if(this.spouseName!=null){
                        sb.append("\\n Spouse Name: ");
                        sb.append(spouseName);
                }
                if(this.city!=null){
                        sb.append("\\n City: ");
                        sb.append(city);
                }
                if(this.state!=null){
                        sb.append("\\n State: ");
                        sb.append(state);
                }
                if(this.country!=null){
                        sb.append("\\n Country: ");
                        sb.append(country);
                }
                if(this.language!=null){
                        sb.append("\\n Language: ");
                        sb.append(language);
                }
                if(this.passwordHint!=null){
                        sb.append("\\n Password Hint: ");
                        sb.append(passwordHint);
                }
                if(this.securityQuestion!=null){
                        sb.append("\\n Security Question: ");
                        sb.append(securityQuestion);
```

```
                }
                if(this.securityAnswer!=null){
                        sb.append("\\n Security Answer: ");
                        sb.append(securityAnswer);
                }

                return sb.toString();
        }

        public static void main(String[] args) {
                Form form = new Form.FormBuilder("Dave", "Carter", "DavCarter", "DAvCaEr123 ↩
                    ").passwordHint("MyName").city("NY").language("English").build();
                System.out.println(form);
        }

}
```

The above code will produce the following output:

```
First Name: Dave
Last Name: Carter
User Name: DavCarter
Password: DAvCaEr123
City: NY
Language: English
Password Hint: MyName
```

As you can clearly see, now a client only needs to provide the mandatory fields and the fields which are important to him. To create the form object now, we need invoke the `FormBuilder` constructor which takes the mandatory fields and then we need to call the set of required methods on it and finally the `build` method to get the form object.

12.5 When to use the Builder Pattern

Use the Builder pattern when

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- The construction process must allow different representations for the object that's constructed.

12.6 Builder Pattern in JDK

- `java.lang.StringBuilder#append()` (unsynchronized)
- `java.lang.StringBuffer#append()` (synchronized)
- `java.nio.ByteBuffer#put()` (also on CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)
- `javax.swing.GroupLayout.Group#addComponent()`
- All implementations of `java.lang.Appendable`

12.7 Download the Source Code

This was a lesson on the Builder Pattern. You may download the source code here: **Builder Pattern Project**

Chapter 13

Factory Method Design Pattern

13.1 Introduction

In today's modern world, everyone is using software to facilitate their jobs. Recently, a product company has shifted the way they used to take orders from their clients. The company is now looking to use an application to take orders from them. They receive orders, errors in orders, feedback for the previous order, and responses to the order in an XML format. The company has asked you to develop an application to parse the XML and display the result to them.

The main challenge for you is to parse an XML and display its content to the user. There are different XML formats depending on the different types of messages the company receives from its clients. Like, for example, an order type XML has different sets of xml tags as compared to the response or error XML. But the core job is the same; that is, to display to the user the message being carried in these XMLs.

Although the core job is the same, the object that would be used varies according to the kind of XML the application gets from the user. So, an application object may only know that it needs to access a class from within the class hierarchy (hierarchy of different parsers), but does not know exactly which class from among the set of subclasses of the parent class is to be selected.

In this case, it is better to provide a factory, i.e. a factory to create parsers, and at runtime a parser gets instantiated to do the job, according to the kind of XML the application receives from the user.

The Factory Method Pattern, suited for this situation, defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Let us see some more details about the Factory Method Pattern and then we will use it to implement the XML parser for the application.

13.2 What is the Factory Method Pattern

The Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types. The Factory Method pattern encapsulates the functionality required to select and instantiate an appropriate class, inside a designated method referred to as a factory method. The Factory Method selects an appropriate class from a class hierarchy based on the application context and other influencing factors. It then instantiates the selected class and returns it as an instance of the parent class type.

The advantage of this approach is that the application objects can make use of the factory method to get access to the appropriate class instance. This eliminates the need for an application object to deal with the varying class selection criteria.

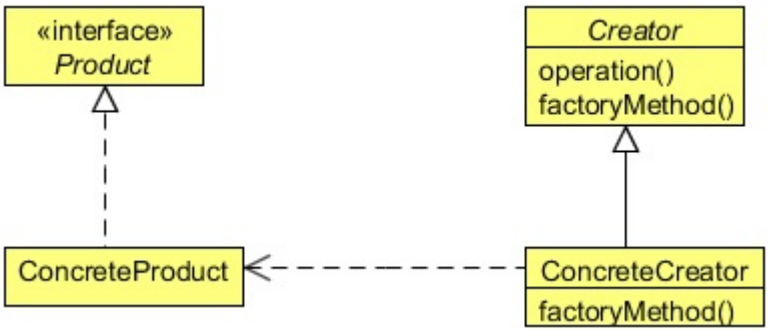

Figure 13.1: screenshot

Product

- Defines the interface of objects the factory method creates.

ConcreteProduct

- Implements the Product interface.

Creator

- Declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
- May call the factory method to create a Product object.

ConcreteCreator

- Overrides the factory method to return an instance of a ConcreteProduct.

Factory methods eliminate the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct classes.

13.3 Implementing Factory Method Pattern

To implement the solution for the application as discussed above, let's first check the product we have.

```
package com.javacodegeeks.patterns.factorymethodpattern;

public interface XMLParser {

        public String parse();

}
```

The above interface will be used by the different XML parsers.

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class ErrorXMLParser implements XMLParser{

        @Override
        public String parse() {
```

```
                System.out.println("Parsing error XML...");
                return "Error XML Message";
        }

}
```

The `ErrorXMLParser` implements the `XMLParser` and is used to parse the error message XMLs.

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class FeedbackXML implements XMLParser{

        @Override
        public String parse() {
                System.out.println("Parsing feedback XML...");
                return "Feedback XML Message";
        }

}
```

The above class is used to parse the feedback message XMLs.

The other XML parsers are:

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class OrderXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("Parsing order XML...");
                return "Order XML Message";
        }

}

package com.javacodegeeks.patterns.factorymethodpattern;

public class ResponseXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("Parsing response XML...");
                return "Response XML Message";
        }

}
```

To display the parsed messages from the parsers, an abstract service class is created which will be extends by service specific, i.e. parser specific, display classes.

```
package com.javacodegeeks.patterns.factorymethodpattern;

public abstract class DisplayService {

        public void display(){
                XMLParser parser = getParser();
                String msg = parser.parse();
                System.out.println(msg);
        }

        protected abstract XMLParser getParser();
```

```
}
```

The above class is used to display the message fetched by the XML parser to the user. The above class is an abstract class that contains two important methods. The `display` method is used to display the message to the user. The `getParser` method is the factory method which is implemented by the subclasses to instantiate the parser object and the method is used by the `display` method in order to parse the XML and gets the message to display.

Below are the subclasses of the `DisplayService` which implement the `getParser` method.

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class ErrorXMLDisplayService extends DisplayService{

        @Override
        public XMLParser getParser() {
                return new ErrorXMLParser();
        }

}
```

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class FeedbackXMLDisplayService extends DisplayService{

        @Override
        public XMLParser getParser() {
                return new FeedbackXML();
        }

}
```

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class OrderXMLDisplayService extends DisplayService{

        @Override
        public XMLParser getParser() {
                return new OrderXMLParser();
        }

}
```

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class ResponseXMLDisplayService extends DisplayService{

        @Override
        public XMLParser getParser() {
                return new ResponseXMLParser();
        }

}
```

Now, let's test the factory method.

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class TestFactoryMethodPattern {

        public static void main(String[] args) {
```

```
                DisplayService service = new FeedbackXMLDisplayService();
                service.display();

                service = new ErrorXMLDisplayService();
                service.display();

                service = new OrderXMLDisplayService();
                service.display();

                service = new ResponseXMLDisplayService();
                service.display();

        }

}
```

The above class results to the following output:

```
Parsing feedback XML...
Feedback XML Message
Parsing error XML...
Error XML Message
Parsing order XML...
Order XML Message
Parsing response XML...
Response XML Message
```

In the above class, you can clearly see that the by letting the subclasses to implement the factory method creates the different instances of parsers which can be used at runtime according to the need.

13.4 When to use the Factory Method Pattern

Use the Factory Method pattern when

- A class can't anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

13.5 Factory Method Pattern in JDK

The following are the usage(s) of the Factory Method Pattern in JDK.

- `java.util.Calendar#getInstance()`
- `java.util.ResourceBundle#getBundle()`
- `java.text.NumberFormat#getInstance()`
- `java.nio.charset.Charset#forName()`
- `java.net.URLStreamHandlerFactory#createURLStreamHandler(String)` (Returns singleton object per protocol)

13.6 Download the Source Code

This was a lesson on the Factory Method Pattern. You may download the source code here: **FactoryMethodPattern-Project**

Chapter 14

Abstract Factory Method Design Pattern

14.1 Introduction

In the previous lesson, we had developed an application for a product company to parse XMLs and display results to them. We did this by creating different parsers for the different types of communication between the company and its clients. We used the Factory Method Design Pattern to solve their problem.

The application is working fine for them. But now the clients don't want to follow the company's specifics XML rules. The clients want to use their own XML rules to communicate with the product company. This means that for every client, the company should have client specific XML parsers. For example, for the NY client there should be four specific types of XML parsers, i.e. NYErrorXMLParser, NYFeedbackXML, NYOrderXMLParser, NYResponseXMLParser, and four different parsers for the TW client.

The company has asked you to change the application according to the new requirement. To develop the parser application we have used the Factory Method Design Pattern in which the exact object to use is decided by the subclasses according to the type of parser. Now, to implement this new requirement, we will use a factory of factories i.e. an Abstract Factory.

This time we need parsers according to client specific XMLs, so we will create different factories for different clients which will provide us the client specific XML to parse. We will do this by creating an Abstract Factory and then implement the factory to provide client specific XML factory. Then we will use that factory to get the desired client specific XML parser object.

Abstract Factory is the design pattern of our choice and before implementing it to solve our problem, lets us know more about it.

14.2 What is the Abstract Factory Design Pattern

The Abstract Factory (A.K.A. Kit) is a design pattern which provides an interface for creating families of related or dependent objects without specifying their concrete classes. The Abstract Factory pattern takes the concept of the Factory Method Pattern to the next level. An abstract factory is a class that provides an interface to produce a family of objects. In Java, it can be implemented using an interface or an abstract class.

The Abstract Factory pattern is useful when a client object wants to create an instance of one of a suite of related, dependent classes without having to know which specific concrete class is to be instantiated. Different concrete factories implement the abstract factory interface. Client objects make use of these concrete factories to create objects and, therefore, do not need to know which concrete class is actually instantiated.

The abstract factory is useful for plugging in a different group of objects to alter the behavior of the system. For each group or family, a concrete factory is implemented that manages the creation of the objects and the inter-dependencies and consistency requirements between them. Each concrete factory implements the interface of the abstract factory

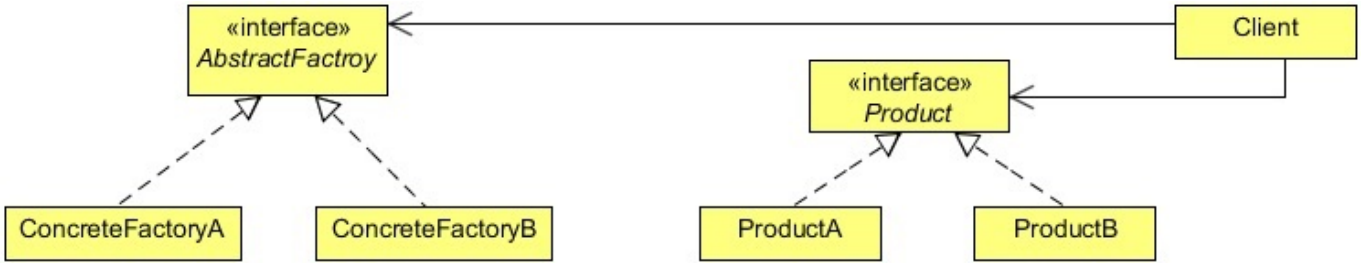

Figure 14.1: screenshot

AbstractFactory

- Declares an interface for operations that create abstract product objects.

ConcreteFactory

- Implements the operations to create concrete product objects.

AbstractProduct

- Declares an interface for a type of product object.

ConcreteProduct

- Defines a product object to be created by the corresponding concrete factory.
- Implements the AbstractProduct interface.

Client

- Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

14.3 Implementing the Abstract Factory Design Pattern

To implement the Abstract Factory Design Pattern will we first create an interface that will be implemented by all the concrete factories.

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public interface AbstractParserFactory {

        public XMLParser getParserInstance(String parserType);
}
```

The above interface is implemented by the client specific concrete factories which will provide the XML parser object to the client object. The `getParserInstance` method takes the `parserType` as an argument which is used to get the message specific (error parser, order parser etc) parser object.

The two client specific concrete parser factories are:

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYParserFactory implements AbstractParserFactory {

        @Override
        public XMLParser getParserInstance(String parserType) {

                switch(parserType){
                        case "NYERROR": return new NYErrorXMLParser();
                        case "NYFEEDBACK": return new NYFeedbackXMLParser ();
                        case "NYORDER": return new NYOrderXMLParser();
                        case "NYRESPONSE": return new NYResponseXMLParser();
                }

                return null;
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWParserFactory implements AbstractParserFactory {

        @Override
        public XMLParser getParserInstance(String parserType) {

                switch(parserType){
                        case "TWERROR": return new TWErrorXMLParser();
                        case "TWFEEDBACK": return new TWFeedbackXMLParser ();
                        case "TWORDER": return new TWOrderXMLParser();
                        case "TWRESPONSE": return new TWResponseXMLParser();
                }

                return null;
        }

}
```

The above two factories implement the `AbstractParserFactory` interface and overrides the `getParserInstance` method. It returns the client specific parser object, according to the parser type requested in the argument.

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public interface XMLParser {

        public String parse();

}
```

The above interface is implemented by the concrete parser classes to parse the XMLs and returns the string message.

There are two clients and four different type of messages exchange between the company and its client. So, there should be six different types of concrete XML parsers that are specific to the client.

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYErrorXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("NY Parsing error XML...");
```

```
                return "NY Error XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYFeedbackXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("NY Parsing feedback XML...");
                return "NY Feedback XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYOrderXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("NY Parsing order XML...");
                return "NY Order XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYResponseXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("NY Parsing response XML...");
                return "NY Response XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWErrorXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("TW Parsing error XML...");
                return "TW Error XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWFeedbackXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("TW Parsing feedback XML...");
```

```
                return "TW Feedback XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWOrderXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("TW Parsing order XML...");
                return "TW Order XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWResponseXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("TW Parsing response XML...");
                return "TW Response XML Message";
        }

}
```

To avoid a dependency between the client code and the factories, optionally we implemented a factory-producer which has a static method and is responsible to provide a desired factory object to the client object.

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public final class ParserFactoryProducer {

        private ParserFactoryProducer(){
                throw new AssertionError();
        }

        public static AbstractParserFactory getFactory(String factoryType){

                switch(factoryType)
                {
                        case "NYFactory": return new NYParserFactory();
                        case "TWFactory": return new TWParserFactory();
                }

                return null;
        }

}
```

Now, let's test the code.

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TestAbstractFactoryPattern {

        public static void main(String[] args) {
```

```
                AbstractParserFactory parserFactory = ParserFactoryProducer.getFactory(" ↩
                    NYFactory");
                XMLParser parser = parserFactory.getParserInstance("NYORDER");
                String msg="";
                msg = parser.parse();
                System.out.println(msg);

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

                parserFactory = ParserFactoryProducer.getFactory("TWFactory");
                parser = parserFactory.getParserInstance("TWFEEDBACK");
                msg = parser.parse();
                System.out.println(msg);
        }

}
```

The above code will result to the following output:

```
NY Parsing order XML...
NY Order XML Message
**********************************
TW Parsing feedback XML...
TW Feedback XML Message
```

In the above class, we first got the NY factory from the factory producer, and then the Order XML parser from the NY parser factory. Then, we called the `parse` method on the parser object and displayed the return message. We did same for the TW client as clearly shown in the output.

14.4 When to use the Abstract Factory Design Pattern

Use the Abstract Factory pattern when

- A system should be independent of how its products are created, composed, and represented.
- A system should be configured with one of multiple families of products.
- A family of related product objects is designed to be used together, and you need to enforce this constraint.
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

14.5 Abstract Factory Pattern in JDK

- `java.util.Calendar#getInstance()`
- `java.util.Arrays#asList()`
- `java.util.ResourceBundle#getBundle()`
- `java.sql.DriverManager#getConnection()`
- `java.sql.Connection#createStatement()`
- `java.sql.Statement#executeQuery()`
- `java.text.NumberFormat#getInstance()`
- `javax.xml.transform.TransformerFactory#newInstance()`

14.6 Download the Source Code

This was a lesson on the Abstract Factory Design Pattern. You may download the source code here: **AbstractFactoryPattern-Project**

Chapter 15

Prototype Design Pattern

15.1 Introduction

In Object Oriented Programming, you need objects to work with; objects interact with each other to get the job done. But sometimes, creating a heavy object could become costly, and if your application needs too many of that kind of objects (containing almost similar properties), it might create some performance issues.

Let us consider a scenario where an application requires some access control. The features of the applications can be used by the users according to the access rights provided to them. For example, some users have access to the reports generated by the application, while some don't. Some of them even can modify the reports, while some can only read it. Some users also have administrative rights to add or even remove other users.

Every user object has an access control object, which is used to provide or restrict the controls of the application. This access control object is a bulky, heavy object and its creation is very costly since it requires data to be fetched from some external resources, like databases or some property files etc.

We also cannot share the same access control object with users of the same level, because the rights can be changed at runtime by the administrator and a different user with the same level could have a different access control. One user object should have one access control object.

We can use the Prototype Design Pattern to resolve this problem by creating the access control objects on all levels at once, and then provide a copy of the object to the user whenever required. In this case, data fetching from the external resources happens only once. Next time, the access control object is created by copying the existing one. The access control object is not created from scratch every time the request is sent; this approach will certainly reduce object creation time.

Before digging into the solution, let us know more about the Prototype Design Pattern.

15.2 What is the Prototype Design Pattern

The Prototype design pattern is used to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations. The existing object acts as a prototype and contains the state of the object. The newly copied object may change same properties only if required. This approach saves costly resources and time, especially when the object creation is a heavy process.

In Java, there are certain ways to copy an object in order to create a new one. One way to achieve this is using the `Cloneable` interface. Java provides the `clone` method, which an object inherits from the `Object` class. You need to implement the `Cloneable` interface and override this `clone`method according to your needs.

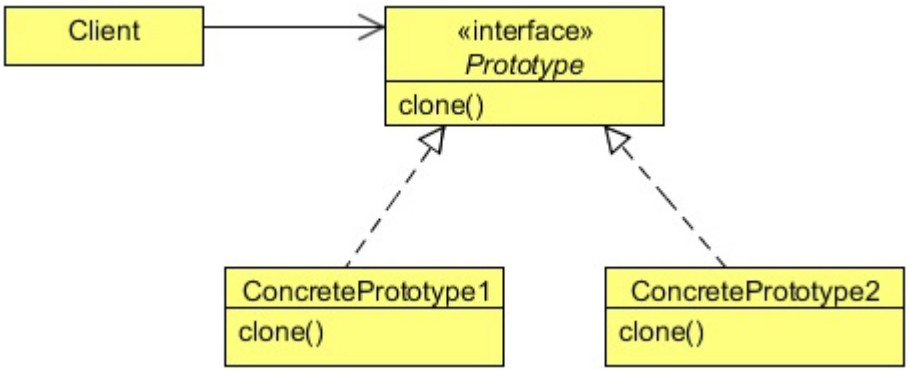

Figure 15.1: screenshot

Prototype

- Declares an interface for cloning itself.

ConcretePrototype

- Implements an operation for cloning itself.

Client

- Creates a new object by asking a prototype to clone itself.

Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client.

15.3 Solution to the Problem

In this solution, we will use the clone method to solve the above problem.

```
package com.javacodegeeks.patterns.prototypepattern;

public interface Prototype extends Cloneable {

        public AccessControl clone() throws CloneNotSupportedException;

}
```

The above interface extends the `Cloneable` interface and contains a method `clone`. This interface is implemented by classes which want to create a prototype object.

```
package com.javacodegeeks.patterns.prototypepattern;

public class AccessControl implements Prototype{

        private final String controlLevel;
        private String access;

        public AccessControl(String controlLevel,String access){
                this.controlLevel = controlLevel;
                this.access = access;
        }
```

```
        @Override
        public AccessControl clone(){
                try {
                        return (AccessControl) super.clone();
                } catch (CloneNotSupportedException e) {
                        e.printStackTrace();
                }
                return null;
        }

        public String getControlLevel(){
                return controlLevel;
        }

        public String getAccess() {
                return access;
        }

        public void setAccess(String access) {
                this.access = access;
        }

}
```

The `AccessControl` class implements the `Prototype` interface and overrides the `clone` method. The method calls the `clone` method of the super class and returns the object after down-casting it to the `AccessControl` type. The `clone` method throws `CloneNotSupportedException` which is caught within the method itself.

The class also contains two properties; the `controlLevel` is used to specific the level of control this object contains. The level depends upon the type of user going to use it, for example, USER, ADMIN, MANAGER etc.

The other property is the `access`; it contains the access right for the user. Please note that, for simplicity, we have used access as a `String` type attribute. This could be of type `Map` which can contain key value pairs of long access rights assigned to the user.

```
package com.javacodegeeks.patterns.prototypepattern;

public class User {

        private String userName;
        private String level;
        private AccessControl accessControl;

        public User(String userName,String level, AccessControl accessControl){
                this.userName = userName;
                this.level = level;
                this.accessControl = accessControl;
        }

        public String getUserName() {
                return userName;
        }
        public void setUserName(String userName) {
                this.userName = userName;
        }
        public String getLevel() {
                return level;
        }
        public void setLevel(String level) {
                this.level = level;
        }
```

```
        public AccessControl getAccessControl() {
                return accessControl;
        }
        public void setAccessControl(AccessControl accessControl) {
                this.accessControl = accessControl;
        }

        @Override
        public String toString(){
                return "Name: "+userName+", Level: "+level+", Access Control Level:"+ ↩
                    accessControl.getControlLevel()+", Access: "+accessControl.getAccess();
        }

}
```

The `User` class has a `userName`, `level` and a reference to the `AccessControl` assigned to it.

We have used an `AccessControlProvider` class that creates and stores the possible `AccessControl` objects in advance. And when the there's a request to an `AccessControl` object, it returns a new object created by copying the stored prototypes.

```
package com.javacodegeeks.patterns.prototypepattern;

import java.util.HashMap;
import java.util.Map;

public class AccessControlProvider {

        private static Map<String, AccessControl>map = new HashMap<String, AccessControl>() ↩
            ;

        static{

                System.out.println("Fetching data from external resources and creating  ↩
                    access control objects...");
                map.put("USER", new AccessControl("USER","DO_WORK"));
                map.put("ADMIN", new AccessControl("ADMIN","ADD/REMOVE USERS"));
                map.put("MANAGER", new AccessControl("MANAGER","GENERATE/READ REPORTS"));
                map.put("VP", new AccessControl("VP","MODIFY REPORTS"));
        }

        public static AccessControl getAccessControlObject(String controlLevel){
                AccessControl ac = null;
                ac = map.get(controlLevel);
                if(ac!=null){
                        return ac.clone();
                }
                return null;
        }
}
```

The `getAccessControlObject` method fetches a stored prototype object according to the `controlLevel` passed to it, from the map and returns a newly created cloned object to the client code.

Now, let's test the code.

```
package com.javacodegeeks.patterns.prototypepattern;

public class TestPrototypePattern {

        public static void main(String[] args) {
                AccessControl userAccessControl = AccessControlProvider. ↩
                    getAccessControlObject("USER");
                User user = new User("User A", "USER Level", userAccessControl);
```

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

                userAccessControl = AccessControlProvider.getAccessControlObject("USER");
                user = new User("User B", "USER Level", userAccessControl);
                System.out.println("Changing access control of: "+user.getUserName());
                user.getAccessControl().setAccess("READ REPORTS");
                System.out.println(user);

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

                AccessControl managerAccessControl = AccessControlProvider. ↩
                    getAccessControlObject("MANAGER");
                user = new User("User C", "MANAGER Level", managerAccessControl);
                System.out.println(user);
        }
}
```

The above code will produce the following output:

```
Fetching data from external resources and creating access control objects...
************************************
Name: User A, Level: USER Level, Access Control Level:USER, Access: DO_WORK
Changing access of: User B
Name: User B, Level: USER Level, Access Control Level:USER, Access: READ REPORTS
************************************
Name: User C, Level: MANAGER Level, Access Control Level:MANAGER, Access: GENERATE/READ  ↩
    REPORTS
```

In the above code, we have created an `AccessControl` object at USER level and assigned it to User A. Then, again another `AccessControl` object to User B, but this time we have changed the access right of User B. And in the end, MANAGER level access control to User C.

The `getAccessControlObject` is used to get the new copy of the `AccessControl` object, and this can be clearly seen when we change the access right for User B, the access right for User A is not changed (just print the User A object again). This confirms that the `clone` method is working fine, as it returns the new copy of the object not a reference which points to the same object.

15.4 When to use the Prototype Design Pattern

Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

- When the classes to instantiate are specified at run-time, for example, by dynamic loading; or
- To avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
- When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

15.5 Prototype Pattern in JDK

- `java.lang.Object#clone()`
- `java.lang.Cloneable`

15.6 Download the Source Code

This was a lesson on the Prototype Design Pattern. You may download the source code here: **PrototypePattern-Project**

Chapter 16

Memento Design Pattern

16.1 Introduction

Sometimes it's necessary to record the internal state of an object. This is required when implementing checkpoints and "undo" mechanisms that let users back out of tentative operations or recover from errors. You must save state information somewhere, so that you can restore objects to their previous conditions. But objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally. Exposing this state would violate encapsulation, which can compromise the application's reliability and extensibility.

The Memento pattern can be used to accomplish this without exposing the object's internal structure. The object whose state needs to be captured is referred to as the originator.

To illustrate the use of the Memento Pattern, let's see an example. We will create a class that will contain two double type fields and we will run some mathematical operations on it. We will provide users with the undo operation. If the results after some operations are not satisfied to a user, the user can call the undo operation which will restore the state of the object to the last saved point.

The example also includes a save point mechanism which is used by the user to save the state of the object. We will also provide a variety of undo operations. A simple undo would restore the object state to the previous save point. An undo with the specified save point will restore that particular state of the object and undo all will delete all the saved state of the object and restore the object in the initialized state, when the object was created.

Before implementing the pattern, let's know more about the Memento Design Pattern.

16.2 What is the Memento Design Pattern

The Memento Pattern's intent is, without violating encapsulation, to capture and externalize an object's internal state so that the object can be restored to this state later.

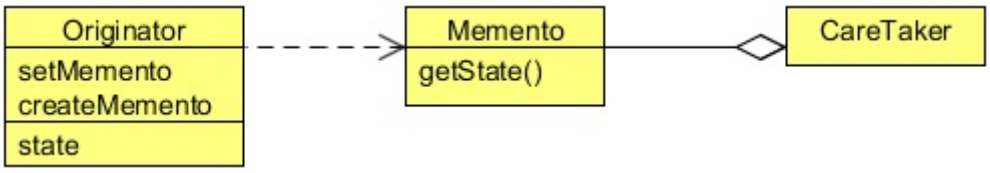

Figure 16.1: screenshot

Memento

- Stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
- Protects against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento-it can only pass the memento to other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.

Originator

- Creates a memento containing a snapshot of its current internal state.
- Uses the memento to restore its internal state.

Caretaker

- Is responsible for the memento's safekeeping.
- Never operates on or examines the contents of a memento.

When a client wants to save the state of the originator, it requests the current state from the originator. The originator stores all those attributes that are required for restoring its state in a separate object referred to as a Memento and returns it to the client. Thus a Memento can be viewed as an object that contains the internal state of another object, at a given point of time. A Memento object must hide the originator variable values from all objects except the originator. In other words, it should protect its internal state against access by objects other than the originator. Towards this end, a Memento should be designed to provide restricted access to other objects while the originator is allowed to access its internal state.

When the client wants to restore the originator back to its previous state, it simply passes the memento back to the originator. The originator uses the state information contained in the memento and puts itself back to the state stored in the Memento object.

16.3 Implementing the Memento Design Pattern

```
package com.javacodegeeks.patterns.mementopattern;

public class Originator {

        private double x;
        private double y;

        private String lastUndoSavepoint;
        CareTaker careTaker;

        public Originator(double x, double y,CareTaker careTaker){
                this.x = x;
                this.y = y;

                this.careTaker = careTaker;

                createSavepoint("INITIAL");
        }

        public double getX(){
                return x;
        }

        public double getY(){
                return y;
        }
```

```
        public void setX(double x) {
                this.x = x;
        }

        public void setY(double y) {
                this.y = y;
        }

        public void createSavepoint(String savepointName){
                careTaker.saveMemento(new Memento(this.x, this.y), savepointName);
                lastUndoSavepoint = savepointName;
        }

        public void undo(){
                setOriginatorState(lastUndoSavepoint);
        }

        public void undo(String savepointName){
                setOriginatorState(savepointName);
        }

        public void undoAll(){
                setOriginatorState("INITIAL");
                careTaker.clearSavepoints();
        }

        private void setOriginatorState(String savepointName){
                Memento mem = careTaker.getMemento(savepointName);
                this.x = mem.getX();
                this.y = mem.getY();
        }

        @Override
        public String toString(){
                return "X: "+x+", Y: "+y;
        }

}
```

The above is the `Originator` class whose object state should be saved in a memento. The class contains two double type's fields `x` and `y`, and also takes a reference of a `CareTaker`. The `CareTaker` is used to save and retrieve the memento objects which represent the state of the `Originator` object.

In the constructor, we have saved the initial state of the object using the `createSavepoint` method. This method creates a `memento` object and requests the caretaker to take care of the object. We have used a `lastUndoSavepoint` variable which is used to store the key name of last saved memento in order to implement the `undo` operation.

The class provides three types of `undo` operations. The `undo` method without any parameter restores the last saved state, the `undo` with the savepoint name as a parameter restores the state saved with that particular savepoint name. The `undoAll` method asks the care taker to clear all the savepoints and set it to the initial state (the state at the time of the creation of the object).

```
package com.javacodegeeks.patterns.mementopattern;

public class Memento {

        private double x;
        private double y;

        public Memento(double x, double y){
                this.x = x;
                this.y = y;
```

```
        }

        public double getX(){
                return x;
        }

        public double getY(){
                return y;
        }
}
```

The `Memento` class is used to store the state of the `Originator` and stored by the care taker. The class does not have any setter methods, it is only used to get the state of the object.

```
package com.javacodegeeks.patterns.mementopattern;

import java.util.HashMap;
import java.util.Map;

public class CareTaker {

        private final Map<String, Memento>savepointStorage = new HashMap<String, Memento>() ↩
            ;

        public void saveMemento(Memento memento,String savepointName){
                System.out.println("Saving state..."+savepointName);
                savepointStorage.put(savepointName, memento);
        }

        public Memento getMemento(String savepointName){
                System.out.println("Undo at ..."+savepointName);
                return savepointStorage.get(savepointName);
        }

        public void clearSavepoints(){
                System.out.println("Clearing all save points...");
                savepointStorage.clear();
        }

}
```

The above class is the care taker class used to store and provide the requested memento object. The class contains the `saveMemento` method is used to save the memento object, the `getMemento` is used to return the request memento object and the `clearSavepoints` method which is used to clear all the savepoints and it deletes all the saved memento objects.

Now, let us test the example.

```
package com.javacodegeeks.patterns.mementopattern;

public class TestMementoPattern {

        public static void main(String[] args) {

                CareTaker careTaker = new CareTaker();
                Originator originator = new Originator(5, 10, careTaker);

                System.out.println("Default State: "+originator);

                originator.setX(originator.getY()*51);

                System.out.println("State: "+originator);
                originator.createSavepoint("SAVE1");
```

```
                originator.setY(originator.getX()/22);
                System.out.println("State: "+originator);

                originator.undo();
                System.out.println("State after undo: "+originator);

                originator.setX(Math.pow(originator.getX(),3));
                originator.createSavepoint("SAVE2");
                System.out.println("State: "+originator);
                originator.setY(originator.getX()-30);
                originator.createSavepoint("SAVE3");
                System.out.println("State: "+originator);
                originator.setY(originator.getX()/22);
                originator.createSavepoint("SAVE4");
                System.out.println("State: "+originator);

                originator.undo("SAVE2");
                System.out.println("Retrieving at: "+originator);

                originator.undoAll();
                System.out.println("State after undo all: "+originator);
        }

}
```

The above code will result to the following output.

```
Saving state...INITIAL
Default State: X: 5.0, Y: 10.0
State: X: 510.0, Y: 10.0
Saving state...SAVE1
State: X: 510.0, Y: 23.181818181818183
Undo at ...SAVE1
State after undo: X: 510.0, Y: 10.0
Saving state...SAVE2
State: X: 1.32651E8, Y: 10.0
Saving state...SAVE3
State: X: 1.32651E8, Y: 1.3265097E8
Saving state...SAVE4
State: X: 1.32651E8, Y: 6029590.909090909
Undo at ...SAVE2
Retrieving at: X: 1.32651E8, Y: 10.0
Undo at ...INITIAL
Clearing all save points...
State after undo all: X: 5.0, Y: 10.0
```

In the above code, we have created a `CareTaker` object and then assigned it to an `Originator` object. Then we have set the value of `x` and `y` to 5 and 10. Then, we applied some operation on the `x` and saved the state of the object as “SAVE1”.

After some more operations, we called the `undo` method to restore the last state of the object which is clearly shows in the output. Then we applied some operations and again saved the states of the object as “SAVE2, SAVE3, and SAVE4”.

Then, we asked the `Originator` to restore the SAVE2 state and the call the `undoAll` method which set the initial state of the object and deleted all the savepoints.

Please note that in above example, the `Originator` is responsible to provide its memento to the care taker. The reason is that we don’t want to give this responsibility to the user. In our example, the user should only require to request for the savepoint and restoration of the state of the object. In many cases, a care taker operates outside the originator by some other class (as shown in the class diagram above).

16.4 When to use the Memento Pattern

Use the Memento Pattern in the following cases:

- A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and
- A direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

16.5 Memento Pattern in JDK

- `java.util.Date`
- `java.io.Serializable`

16.6 Download the Source Code

This was a lesson on the Memento Design Pattern. You may download the source code here: **MementoDesignPattern-Project**

Chapter 17

Template Design Pattern

17.1 Introduction

The Template Design Pattern is a behavior pattern and, as the name suggests, it provides a template or a structure of an algorithm which is used by users. A user provides its own implementation without changing the algorithm's structure.

It is easier to understand this pattern with the help of a problem. We will understand the scenario in this section and will implement the solution using the Template pattern in a later section.

Have you ever connected to a relation database using your Java application? Let's recall some important steps which are required to connect and insert data into the database. First, we need a driver according to the database we want to connect with. Then, we pass some credentials to the database, then, we prepare a statement, set data into the insert statement and insert it using the insert command. Later, we close all the connections, and optionally destroy all the connection objects.

You need to write all these steps regardless of any vendor's relational database. Consider a problem where you need to insert some data into the different databases. You need to fetch some data from a CSV file and have to insert it into a MySQL database. Some data comes from a text file and which should be insert into an Oracle database. The only difference is the driver and the data, the rest of the steps should be the same, as JDBC provides a common set of interfaces to communicate to any vendor's specific relation database.

We can create a template, which will perform some steps for the client, and we will leave some steps to let the client to implement them in its own specific way. Optionally, a client can override the default behavior of some already defined steps.

Now, before implementing the code, let's know more about the Template Design Pattern.

17.2 What is the Template Design Pattern

The Template Pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses to redefine certain steps of an algorithm without changing the algorithm's structure.

The Template Method pattern can be used in situations when there is an algorithm, some steps of which could be implemented in multiple different ways. In such scenarios, the Template Method pattern suggests keeping the outline of the algorithm in a separate method referred to as a template method inside a class, which may be referred to as a template class, leaving out the specific implementations of the variant portions (steps that can be implemented in multiple different ways) of the algorithm to different subclasses of this class.

The Template class does not necessarily have to leave the implementation to subclasses in its entirety. Instead, as part of providing the outline of the algorithm, the Template class can also provide some amount of implementation that can be considered as invariant across different implementations. It can even provide default implementation for the variant parts, if appropriate. Only specific details will be implemented inside different subclasses. This type of implementation eliminates the need for duplicate code, which means a minimum amount of code to be written.

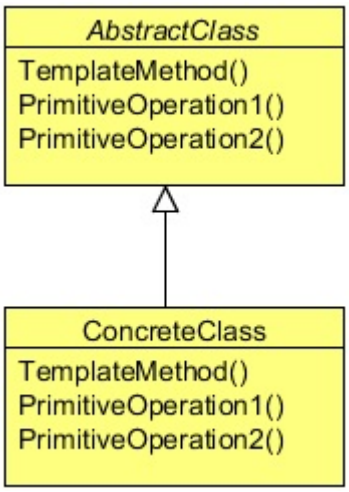

Figure 17.1: screenshot

AbstractClass

- Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
- Implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in `AbstractClass` or those of other objects. `ConcreteClass`
- Implements the primitive operations to carry.

17.3 Implementing the Template Design Pattern

Below we can see the connection template class used to provide a template for clients to connect and communicate with the various databases.

```
package com.javacodegeeks.patterns.templatepattern;

public abstract class ConnectionTemplate {

        public final void run() {
                setDBDriver();
                setCredentials();
                connect();
                prepareStatement();
                setData();
                insert();
                close();
                destroy();
        }

        public abstract void setDBDriver();

        public abstract void setCredentials();

        public void connect() {
                System.out.println("Setting connection...");
        }

        public void prepareStatement() {
                System.out.println("Preparing insert statement...");
```

```
        }

        public abstract void setData();

        public void insert() {
                System.out.println("Inserting data...");
        }

        public void close() {
                System.out.println("Closing connections...");
        }

        public void destroy() {
                System.out.println("Destroying connection objects...");
        }
}
```

The abstract class provides steps to connect, communicate and later to close the connections. All these steps are required in order to get the work done. The class provides default implementation to some common steps and leaves the specific steps as abstract which force the client to provide an implementation to them.

The `setDBDriver` method should be implementing by the user to provide database specific drivers. The credentials could be different for different databases; therefore, `setCredentials` is also left as abstract to let the user to implement it.

Similarly, connecting to the database using JDBC API and preparing a statement is common. But, data would be specific so user will provide it, and rest of other steps like running an insert statement, closing connections and destroying object are similar to any database, therefore their implementations are kept common inside the template.

The key method of the above class is the `run` method. The `run` method is used to run these steps in order. The method is set as final because the steps should be kept safe and should not be change by any user.

The below two classes extend the template class and provide specific implementation to some methods.

```
package com.javacodegeeks.patterns.templatepattern;

public class MySqLCSVCon extends ConnectionTemplate {

        @Override
        public void setDBDriver() {
                System.out.println("Setting MySQL DB drivers...");
        }

        @Override
        public void setCredentials() {
                System.out.println("Setting credentials for MySQL DB...");
        }

        @Override
        public void setData() {
                System.out.println("Setting up data from csv file....");
        }
}
```

The above class is used to connect to a MySQL database and provides data by reading a CSV file.

```
package com.javacodegeeks.patterns.templatepattern;

public class OracleTxtCon extends ConnectionTemplate {

        @Override
        public void setDBDriver() {
                System.out.println("Setting Oracle DB drivers...");
        }
```

```
        @Override
        public void setCredentials() {
                System.out.println("Setting credentials for Oracle DB...");
        }

        @Override
        public void setData() {
                System.out.println("Setting up data from txt file....");
        }
}
```

The above class is used to connect to an Oracle database and provides data by reading a text file.

Now, let's test the code.

```
package com.javacodegeeks.patterns.templatepattern;

public class TestTemplatePattern {

        public static void main(String[] args) {
                System.out.println("For MYSQL....");
                ConnectionTemplate template = new MySqLCSVCon();
                template.run();
                System.out.println("For Oracle...");
                template = new OracleTxtCon();
                template.run();
        }
}
```

The above code will result to the following output:

```
For MYSQL....
Setting MySQL DB drivers...
Setting credentials for MySQL DB...
Setting connection...
Preparing insert statement...
Setting up data from csv file....
Inserting data...
Closing connections...
Destroying connection objects...

For Oracle...
Setting Oracle DB drivers...
Setting credentials for Oracle DB...
Setting connection...
Preparing insert statement...
Setting up data from txt file....
Inserting data...
Closing connections...
Destroying connection objects...
```

The above output clearly shows how the template pattern works to connect and communicate with the different databases using the similar way. The pattern keeps the common code under one class and promotes code reusability. It sets a framework and controls it for the users and allows the users to extends the template in order to provide their specific implementation to some of the steps.

Now, if we enhance the above example by adding a logging mechanism. But some of the users of the code do not want to add this facility, to implement this we can use a hook. A hook is a simple method inside a template class with a default behavior; this behavior can be used to alter some optional steps. A user should implement this method which can hook inside the template class to alter the optional steps of the algorithm.

17.4 Introducing a hook inside the template

Let's enhance the above example with a hook.

```
package com.javacodegeeks.patterns.templatepattern;

import java.util.Date;

public abstract class ConnectionTemplate {

        private boolean isLoggingEnable = true;

        public ConnectionTemplate() {
                isLoggingEnable = disableLogging();
        }

        public final void run() {
                setDBDriver();
                logging("Drivers set [" + new Date() + "]");
                setCredentials();
                logging("Credentails set [" + new Date() + "]");
                connect();
                logging("Conencted");
                prepareStatement();
                logging("Statement prepared [" + new Date() + "]");
                setData();
                logging("Data set [" + new Date() + "]");
                insert();
                logging("Inserted [" + new Date() + "]");
                close();
                logging("Conenctions closed [" + new Date() + "]");
                destroy();
                logging("Object destoryed [" + new Date() + "]");
        }

        public abstract void setDBDriver();

        public abstract void setCredentials();

        public void connect() {
                System.out.println("Setting connection...");
        }

        public void prepareStatement() {
                System.out.println("Preparing insert statement...");
        }

        public abstract void setData();

        public void insert() {
                System.out.println("Inserting data...");
        }

        public void close() {
                System.out.println("Closing connections...");
        }

        public void destroy() {
                System.out.println("Destroying connection objects...");
        }

        public boolean disableLogging() {
```

```
                return true;
        }

        private void logging(String msg) {
                if (isLoggingEnable) {
                        System.out.println("Logging....: " + msg);
                }
        }
}
```

We introduced two new methods inside the above template class. The `disableLogging` is the hook which returns a `boolean`. By default, the boolean `isLoggingEnable`, which enables the logging, is true. A user can override this method if logging should be disabled for his code. The other is a private method used to log the messages.

The below class implements the hook method and returns false, something that switches off the logging mechanism for this specific work.

```
package com.javacodegeeks.patterns.templatepattern;

public class MySqLCSVCon extends ConnectionTemplate {

        @Override
        public void setDBDriver() {
                System.out.println("Setting MySQL DB drivers...");
        }

        @Override
        public void setCredentials() {
                System.out.println("Setting credentials for MySQL DB...");
        }

        @Override
        public void setData() {
                System.out.println("Setting up data from csv file....");
        }

        @Override
        public boolean disableLogging() {
                return false;
        }
}
```

Let's test this code.

```
package com.javacodegeeks.patterns.templatepattern;

public class TestTemplatePattern {
        public static void main(String[] args) {
                System.out.println("For MYSQL....");
                ConnectionTemplate template = new MySqLCSVCon();
                template.run();
                System.out.println("For Oracle...");
                template = new OracleTxtCon();
                template.run();
        }
}
```

The above class will result to the following output:

```
For MYSQL....
Setting MySQL DB drivers...
Setting credentials for MySQL DB...
```

```
Setting connection...
Preparing insert statement...
Setting up data from csv file....
Inserting data...
Closing connections...
Destroying connection objects...

For Oracle...
Setting Oracle DB drivers...
Logging....: Drivers set [Sat Nov 08 23:53:47 IST 2014]
Setting credentials for Oracle DB...
Logging....: Credentails set [Sat Nov 08 23:53:47 IST 2014]
Setting connection...
Logging....: Conencted
Preparing insert statement...
Logging....: Statement prepared [Sat Nov 08 23:53:47 IST 2014]
Setting up data from txt file....
Logging....: Data set [Sat Nov 08 23:53:47 IST 2014]
Inserting data...
Logging....: Inserted [Sat Nov 08 23:53:47 IST 2014]
Closing connections...
Logging....: Conenctions closed [Sat Nov 08 23:53:47 IST 2014]
Destroying connection objects...
Logging....: Object destoryed [Sat Nov 08 23:53:47 IST 2014]
```

You can clearly see in the output, that logging is off for the MySQL implementation, whereas, on for the Oracle implementation.

17.5 When to use the Template Design Pattern

The Template Method pattern should be used in the following cases:

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication. You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.
- To control subclasses extensions. You can define a template method that calls "hook" operations (see Consequences) at specific points, thereby permitting extensions only at those points.

17.6 Template Pattern in JDK

- `java.util.Collections#sort()`
- `java.io.InputStream#skip()`
- `java.io.InputStream#read()`
- `java.util.AbstractList#indexOf()`

17.7 Download the Source Code

This was a lesson on the Template Design Pattern. You may download the source code here: **TemplatePattern-Project**

Chapter 18

State Design Pattern

18.1 Introduction

To illustrate the use of the State Design Pattern, let us help a company which is looking to build a robot for cooking. The company wants a simple robot that can simply walk and cook. A user can operate a robot using a set of commands via remote control. Currently, a robot can do three things, it can walk, cook, or can be switched off.

The company has set protocols to define the functionality of the robot. If a robot is in "on" state you can command it to walk. If asked to cook, the state would change to "cook" or if set to "off", it will be switched off.

Similarly, when in "cook" state it can walk or cook, but cannot be switched off. And finally, when in "off" state it will automatically get on and walk when the user commands it to walk but cannot cook in off state.

This might look like an easy implementation: a robot class with a set of methods like walk, cook, off, and states like on, cook, and off. We can use if-else branches or switch to implement the protocols set by the company. But too much if-else or switch statements will create a maintenance nightmare as complexity might increase in the future.

You might think that we can implement this without issues using if-else statements, but as a change comes the code would become more complex. The requirement clearly shows that the behavior of an object is truly based on the state of that object. We can use the State Design Pattern which encapsulates the states of the object into another individual class and keeps the context class independent of any state change.

Let's first know about the State Design Pattern and then we will implement it to solve the problem above.

18.2 What is the State Design Pattern

The State Design Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

The state of an object can be defined as its exact condition at any given point of time, depending on the values of its properties or attributes. The set of methods implemented by a class constitutes the behavior of its instances. Whenever there is a change in the values of its attributes, we say that the state of an object has changed.

The State pattern is useful in designing an efficient structure for a class, a typical instance of which can exist in many different states and exhibit different behavior depending on the state it is in. In other words, in the case of an object of such a class, some or all of its behavior is completely influenced by its current state. In the State design pattern terminology, such a class is referred to as a `Context` class. A `Context` object can alter its behavior when there is a change in its internal state and is also referred as a Stateful object.

The State pattern suggests moving the state-specific behavior out of the `Context` class into a set of separate classes referred to as `State` classes. Each of the many different states that a `Context` object can exist in can be mapped into a separate `State` class. The implementation of a State class contains the context behavior that is specific to a given state, not the overall behavior

of the context itself. The context acts as a client to the set of State objects in the sense that it makes use of different State objects to offer the necessary state-specific behavior to an application object that uses the context in a seamless manner.

By encapsulating the state-specific behavior in separate classes, the context implementation becomes simpler to read: free of too many conditional statements such as if-else or switch-case constructs. When a `Context` object is first created, it initializes itself with its initial State object. This State object becomes the current State object for the context. By replacing the current State object with a new State object, the context transitions to a new state.

The client application using the context is not responsible for specifying the current State object for the context, but instead, each of the State classes representing specific states are expected to provide the necessary implementation to transition the context into other states. When an application object makes a call to a `Context` method (behavior),it forwards the method call to its current State object.

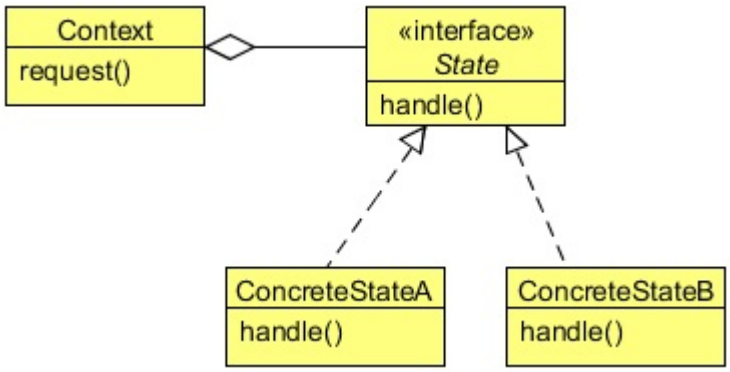

Figure 18.1: - Class Diagram

Context

- Defines the interface of interest to clients.
- Maintains an instance of a `ConcreteState` subclass that defines the current state.

State

- Defines an interface for encapsulating the behavior associated with a particular state of the Context.

ConcreteState subclasses

- Each subclass implements a behavior associated with a state of the `Context`.

18.3 Implementing the State Design Pattern

The following is the `RoboticState` interface which contains the behavior of a robot.

```
package com.javacodegeeks.patterns.statepattern;

public interface RoboticState {

        public void walk();
        public void cook();
        public void off();

}
```

The `Robot` class is a concrete class implements the `RoboticState` interface. The class contains the set of all possible states a robot can be in.

```
package com.javacodegeeks.patterns.statepattern;

public class Robot implements RoboticState{

	private RoboticState roboticOn;
	private RoboticState roboticCook;
	private RoboticState roboticOff;

	private RoboticState state;

	public Robot(){
		this.roboticOn = new RoboticOn(this);
		this.roboticCook = new RoboticCook(this);
		this.roboticOff = new RoboticOff(this);

		this.state = roboticOn;
	}

	public void setRoboticState(RoboticState state){
		this.state = state;
	}

	@Override
	public void walk() {
		state.walk();
	}

	@Override
	public void cook() {
		state.cook();

	}

	@Override
	public void off() {
		state.off();
	}

	public RoboticState getRoboticOn() {
		return roboticOn;
	}

	public void setRoboticOn(RoboticState roboticOn) {
		this.roboticOn = roboticOn;
	}

	public RoboticState getRoboticCook() {
		return roboticCook;
	}

	public void setRoboticCook(RoboticState roboticCook) {
		this.roboticCook = roboticCook;
	}

	public RoboticState getRoboticOff() {
		return roboticOff;
	}

	public void setRoboticOff(RoboticState roboticOff) {
```

```
                this.roboticOff = roboticOff;
        }

        public RoboticState getState() {
                return state;
        }

        public void setState(RoboticState state) {
                this.state = state;
        }

}
```

The class initializes all the states and sets the current state as on.

Now, we will see all the concrete states of a robot. A robot will be in any of these states at any time.

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticOn implements RoboticState{

        private final Robot robot;

        public RoboticOn(Robot robot){
                this.robot = robot;
        }

        @Override
        public void walk() {
                System.out.println("Walking...");
        }

        @Override
        public void cook() {
                System.out.println("Cooking...");
                robot.setRoboticState(robot.getRoboticCook());
        }

        @Override
        public void off() {
                robot.setState(robot.getRoboticOff());
                System.out.println("Robot is switched off");

        }

}
```

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticCook implements RoboticState{

        private final Robot robot;

        public RoboticCook(Robot robot){
                this.robot = robot;
        }

        @Override
        public void walk() {
                System.out.println("Walking...");
                robot.setRoboticState(robot.getRoboticOn());
        }
```

```
        @Override
        public void cook() {
                System.out.println("Cooking...");
        }

        @Override
        public void off() {
                System.out.println("Cannot switched off while cooking...");
        }
}
```

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticOff implements RoboticState{

        private final Robot robot;

        public RoboticOff(Robot robot){
                this.robot = robot;
        }

        @Override
        public void walk() {
                System.out.println("Walking...");
                robot.setRoboticState(robot.getRoboticOn());
        }

        @Override
        public void cook() {
                System.out.println("Cannot cook at Off state.");
        }

        @Override
        public void off() {
                System.out.println("Already switched off...");
        }
}
```

Now, let's test the code.

```
package com.javacodegeeks.patterns.statepattern;

public class TestStatePattern {

        public static void main(String[] args) {
                Robot robot = new Robot();
                robot.walk();
                robot.cook();
                robot.walk();
                robot.off();

                robot.walk();
                robot.off();
                robot.cook();

        }

}
```

The above code will result to the following output:

```
Walking...
Cooking...
Walking...
Robot is switched off
Walking...
Robot is switched off
Cannot cook at Off state.
```

In the above example, we have seen that by encapsulating the states of an object into different classes makes the code manageable and flexible.

Any change in a state will only affect that particular class and we can include a new state without changing much in the existing code. Let's for example, we include a stand-by state. After a walk or cook the robot goes into the stand-by mode to save power and we again walk, cook or switch off from the stand-by mode.

To implement this all we need to introduce a new state class and include that state in the `Robot` class. The following are the changes.

```
package com.javacodegeeks.patterns.statepattern;

public class Robot implements RoboticState{

	private RoboticState roboticOn;
	private RoboticState roboticCook;
	private RoboticState roboticOff;
	private RoboticState roboticStandby;

	private RoboticState state;

	public Robot(){
		this.roboticOn = new RoboticOn(this);
		this.roboticCook = new RoboticCook(this);
		this.roboticOff = new RoboticOff(this);
		this.roboticStandby = new RoboticStandby(this);

		this.state = roboticOn;
	}

	public void setRoboticState(RoboticState state){
		this.state = state;
	}

	@Override
	public void walk() {
		state.walk();
		setState(getRoboticStandby());
	}

	@Override
	public void cook() {
		state.cook();
		setState(getRoboticStandby());
	}

	@Override
	public void off() {
		state.off();
	}

	public RoboticState getRoboticOn() {
		return roboticOn;
	}
```

```
        public void setRoboticOn(RoboticState roboticOn) {
                this.roboticOn = roboticOn;
        }

        public RoboticState getRoboticCook() {
                return roboticCook;
        }

        public void setRoboticCook(RoboticState roboticCook) {
                this.roboticCook = roboticCook;
        }

        public RoboticState getRoboticOff() {
                return roboticOff;
        }

        public void setRoboticOff(RoboticState roboticOff) {
                this.roboticOff = roboticOff;
        }

        public RoboticState getState() {
                return state;
        }

        public void setState(RoboticState state) {
                this.state = state;
        }

        public RoboticState getRoboticStandby() {
                return roboticStandby;
        }

        public void setRoboticStandby(RoboticState roboticStandby) {
                this.roboticStandby = roboticStandby;
        }

}
```

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticStandby implements RoboticState{

private final Robot robot;

        public RoboticStandby(Robot robot){
                this.robot = robot;
        }

        @Override
        public void walk() {
                System.out.println("In standby state...");
                robot.setState(robot.getRoboticOn());
                System.out.println("Walking...");
        }

        @Override
        public void cook() {
                System.out.println("In standby state...");
                robot.setRoboticState(robot.getRoboticCook());
                System.out.println("Cooking...");
        }
```

```
        @Override
        public void off() {
                System.out.println("In standby state...");
                robot.setState(robot.getRoboticOff());
                System.out.println("Robot is switched off");

        }

}
```

Now, the above code change will result to the following output:

```
Walking...
In standby state...
Cooking...
In standby state...
Walking...
In standby state...
Robot is switched off
Walking...
In standby state...
Robot is switched off
Cannot cook at Off state.
```

18.4 When to use the State Design Pattern

Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

18.5 State Design Pattern in Java

- `javax.faces.lifecycle.LifeCycle#execute()`

18.6 Download the Source Code

This was a lesson on the State Design Pattern. You may download the source code here: **StatePattern-Project**

Chapter 19

Strategy Design Pattern

19.1 Introduction

The Strategy Design Pattern seems to be the simplest of all design patterns, yet it provides great flexibility to your code. This pattern is used almost everywhere, even in conjunction with the other design patterns. The patterns we have discussed so far have a relation with this pattern, either directly or indirectly. After this lesson, you will get an idea on how important this pattern is.

To understand the Strategy Design Pattern, let us create a text formatter for a text editor. Everyone should be aware of a text editor. A text editor can have different text formatters to format text. We can create different text formatters and then pass the required one to the text editor, so that the editor will able to format the text as required.

The text editor will hold a reference to a common interface for the text formatter and the editor's job will be to pass the text to the formatter in order to format the text.

Let's implement this using the Strategy Design Pattern which will make the code very flexible and maintainable. But before that, let us more about the Strategy Design Pattern.

19.2 2. What is the Strategy Pattern

The Strategy Design Pattern defines a family of algorithms, encapsulating each one, and making them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

The Strategy pattern is useful when there is a set of related algorithms and a client object needs to be able to dynamically pick and choose an algorithm from this set that suits its current need. The Strategy pattern suggests keeping the implementation of each of the algorithms in a separate class. Each such algorithm encapsulated in a separate class is referred to as a `strategy`. An object that uses a `Strategy` object is often referred to as a `context` object.

With different `Strategy` objects in place, changing the behavior of a `Context` object is simply a matter of changing its `Strategy` object to the one that implements the required algorithm. To enable a `Context` object to access different `Strategy` objects in a seamless manner, all `Strategy` objects must be designed to offer the same interface. In the Java programming language, this can be accomplished by designing each `Strategy` object either as an implementer of a common interface or as a subclass of a common abstract class that declares the required common interface.

Once the group of related algorithms is encapsulated in a set of `Strategy` classes in a class hierarchy, a client can choose from among these algorithms by selecting and instantiating an appropriate `Strategy` class. To alter the behavior of the `context`, a client object needs to configure the `context` with the selected `strategy` instance. This type of arrangement completely separates the implementation of an algorithm from the `context` that uses it. As a result, when an existing algorithm implementation is changed or a new algorithm is added to the group, both the `context` and the client object (that uses the `context`) remain unaffected.

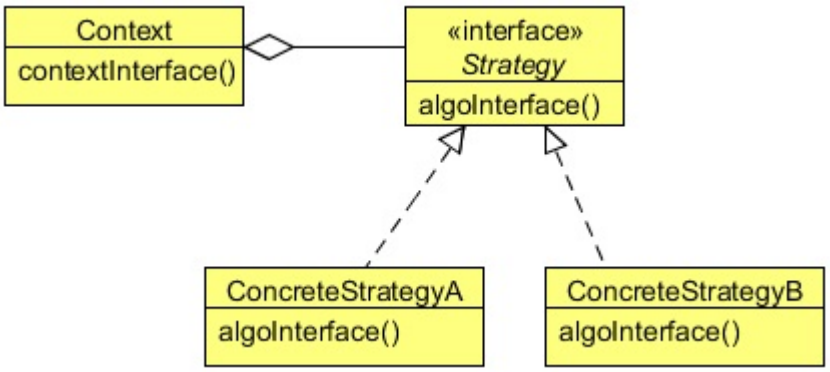

Figure 19.1: - Strategy class diagram

Strategy

- Declares an interface common to all supported algorithms. `Context` uses this interface to call the algorithm defined by a `ConcreteStrategy`.

ConcreteStrategy

- Implements the algorithm using the `Strategy` interface.

Context

- Is configured with a `ConcreteStrategy` object.
- Maintains a reference to a `Strategy` object.
- May define an interface that lets `Strategy` access its data.

19.3 Implementing the Strategy Design Pattern

Below is the `TextFormatter` interface implement by all the concrete formatters.

```
package com.javacodegeeks.patterns.strategypattern;

public interface TextFormatter {

        public void format(String text);

}
```

The above interface contains only one method, `format`, used to format the text.

```
package com.javacodegeeks.patterns.strategypattern;

public class CapTextFormatter implements TextFormatter{

        @Override
        public void format(String text) {
                System.out.println("[CapTextFormatter]: "+text.toUpperCase());
        }

}
```

The above class, `CapTextFormatter`, is a concrete text formatter that implements the `TextFormatter` interface and the class is used to change the text into capital case.

```
package com.javacodegeeks.patterns.strategypattern;

public class LowerTextFormatter implements TextFormatter{

	@Override
	public void format(String text) {
		System.out.println("[LowerTextFormatter]: "+text.toLowerCase());
	}

}
```

The `LowerTextFormatter` is a concrete text formatter that implements the `TextFormatter` interface and the class is used to change the text into small case.

```
package com.javacodegeeks.patterns.strategypattern;

public class TextEditor {

	private final TextFormatter textFormatter;

	public TextEditor(TextFormatter textFormatter){
		this.textFormatter = textFormatter;
	}

	public void publishText(String text){
		textFormatter.format(text);
	}

}
```

The above class is the `TextEditor` class which holds a reference to the `TextFormatter` interface. The class contains the `publishText` method which forwards the text to the formatter in order to publish the text in desired format.

Now, let us test the code above.

```
package com.javacodegeeks.patterns.strategypattern;

public class TestStrategyPattern {

	public static void main(String[] args) {
		TextFormatter formatter = new CapTextFormatter();
		TextEditor editor = new TextEditor(formatter);
		editor.publishText("Testing text in caps formatter");

		formatter = new LowerTextFormatter();
		editor = new TextEditor(formatter);
		editor.publishText("Testing text in lower formatter");

	}

}
```

The above code will result to the following output:

```
[CapTextFormatter]: TESTING TEXT IN CAPS FORMATTER
[LowerTextFormatter]: testing text in lower formatter
```

In the above class, we have first created a `CapTextFormatter` and assigned it to the `TextEditor` instance. Then we called the `publishText` method and passed some input text to it.

Again, we did the same thing, but this time, the `LowerTextFormatter` is passed to the `TextEditor`.

The output clearly shows the different text format produced by the different text editors due to the different text formatter used by it.

The main advantage of the Strategy Design Pattern is that we can enhance the code without much trouble. We can add new text formatters without disturbing the current code. This would make our code maintainable and flexible. This design pattern also promotes the "code to interface" design principle.

19.4 When to use the Strategy Design Pattern

Use the Strategy pattern when:

- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
- An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own `Strategy` class.

19.5 Strategy Pattern in JDK

- `java.util.Comparator#compare()`
- `javax.servlet.http.HttpServlet`
- `javax.servlet.Filter#doFilter()`

19.6 Download the Source Code

This was a lesson on the Strategy Design Pattern. You may download the source code here: **StrategyPattern-Project**

Chapter 20

Command Design Pattern

20.1 Introduction

The Command Design Pattern is a behavioral design pattern and helps to decouples the invoker from the receiver of a request.

To understand the Command Design Pattern let's create an example to execute different types of jobs. A job can be anything in a system, for example, sending emails, SMS, logging, and performing some IO functions.

The Command pattern would help to decouple the invoker from a receiver and helps to execute any type of job without knowing its implementation. Let us make this example more interesting by creating threads which will help to execute these jobs concurrently. As these jobs are independent of each other, so the sequence of execution of these jobs is not really important. We will create a thread pool to limit the number of threads to execute jobs. A command object will encapsulates jobs and will hand it over to a thread from the pool that will execute the job.

Before implementing the example, let's know more about the Command Design Pattern.

20.2 What is the Command Design Pattern

The intent of the Command Design Pattern is to encapsulate a request as an object, thereby letting the developer to parameterize clients with different requests, queue or log requests, and support undoable operations.

In general, an object-oriented application consists of a set of interacting objects each offering limited, focused functionality. In response to user interaction, the application carries out some kind of processing. For this purpose, the application makes use of the services of different objects for the processing requirement.

In terms of implementation, the application may depend on a designated object that invokes methods on these objects by passing the required data as arguments. This designated object can be referred to as an invoker as it invokes operations on different objects. The invoker may be treated as part of the client application. The set of objects that actually contain the implementation to offer the services required for the request processing can be referred to as `Receiver` objects.

Using the Command pattern, the invoker that issues a request on behalf of the client and the set of service-rendering `Receiver` objects can be decoupled. The Command pattern suggests creating an abstraction for the processing to be carried out or the action to be taken in response to client requests. This abstraction can be designed to declare a common interface to be implemented by different concrete implementers referred to as `Command` objects. Each `Command` object represents a different type of client request and the corresponding processing.

A given `Command` object is responsible for offering the functionality required to process the request it represents, but it does not contain the actual implementation of the functionality. `Command` objects make use of `Receiver` objects in offering this functionality.

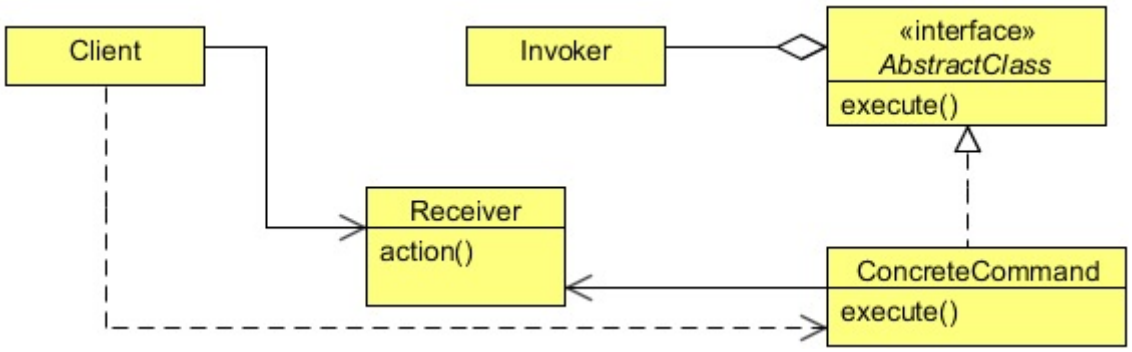

Figure 20.1: - Command pattern Class diagram

Command

- Declares an interface for executing an operation.

ConcreteCommand

- Defines a binding between a `Receiver` object and an action.
- Implements `Execute` by invoking the corresponding operation(s) on `Receiver`.

Client

- Creates a `ConcreteCommand` object and sets its receiver.

Invoker

- Asks the command to carry out the request.

Receiver

- Knows how to perform the operations associated with carrying out a request. Any class may serve as a `Receiver`.

20.3 Implementing the Command Design Pattern

We will implement the example using a command object. The command object will be referenced by a common interface and will contain a method that will be used to execute the requests. The concrete command classes will override that method and will provide their own specific implementation to execute the request.

```
package com.javacodegeeks.patterns.commandpattern;

public interface Job {

        public void run();
}
```

The `Job` interface is the command interface, contains a single method `run`, which is executed by a thread. Our command's `execute` method is the run method which will be used to execute by a thread in order to get the work done.

There would be a different type of jobs that can be executed. The following are the different concrete classes whose instances will be executed by the different command objects.

```
package com.javacodegeeks.patterns.commandpattern;

public class Email {

        public void sendEmail(){
                System.out.println("Sending email.......");
        }
}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class FileIO {

        public void execute(){
                System.out.println("Executing File IO operations...");
        }
}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class Logging {

        public void log(){
                System.out.println("Logging...");
        }
}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class Sms {

        public void sendSms(){
                System.out.println("Sending SMS...");
        }
}
```

The following are the different command classes that encapsulate the above classes and implement the `Job` interface.

```
package com.javacodegeeks.patterns.commandpattern;

public class EmailJob implements Job{

        private Email email;

        public void setEmail(Email email){
                this.email = email;
        }

        @Override
        public void run() {
                System.out.println("Job ID: "+Thread.currentThread().getId()+" executing ↩
                    email jobs.");
                if(email!=null){
                        email.sendEmail();
                }

                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
```

```
                }

        }

}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class FileIOJob implements Job{

        private FileIO fileIO;

        public void setFileIO(FileIO fileIO){
                this.fileIO = fileIO;
        }

        @Override
        public void run() {
                System.out.println("Job ID: "+Thread.currentThread().getId()+" executing as
                    fileIO jobs.");
                if(fileIO!=null){
                        fileIO.execute();
                }

                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                }

        }
}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class LoggingJob implements Job{

        private Logging logging;

        public void setLogging(Logging logging){
                this.logging = logging;
        }

        @Override
        public void run() {
                System.out.println("Job ID: "+Thread.currentThread().getId()+" executing as
                    logging jobs.");
                if(logging!=null){
                        logging.log();
                }

                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                }

        }
}
```

```
package com.javacodegeeks.patterns.commandpattern;
```

```
public class SmsJob implements Job{

        private Sms sms;

        public void setSms(Sms sms) {
                this.sms = sms;
        }


        @Override
        public void run() {
                System.out.println("Job ID: "+Thread.currentThread().getId()+" executing ↩
                    sms jobs.");
                if(sms!=null){
                        sms.sendSms();
                }

                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                }

        }

}
```

The above classes hold a reference to their respective classes that will be used to get the job done. The classes override the `run` method and do the work requested. For example, the `SmsJob` class is used to send sms, its run method calls the `sendSms` method of the `Sms` object in order to get the job done.

You can set different objects one by one to the same `command` object.

The below is the `ThreadPool` class used to create pool of threads and allow a thread to fetch and execute the job from the job queue.

```
package com.javacodegeeks.patterns.commandpattern;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class ThreadPool {

        private final BlockingQueue<Job> jobQueue;
        private final Thread[] jobThreads;
        private volatile boolean shutdown;

        public ThreadPool(int n)
        {
                jobQueue = new LinkedBlockingQueue<>();
                jobThreads = new Thread[n];

                for (int i = 0; i < n; i++) {
                        jobThreads[i] = new Worker("Pool Thread " + i);
                        jobThreads[i].start();
                }
        }

        public void addJob(Job r)
        {
                try {
                        jobQueue.put(r);
```

```
                } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                }
        }

        public void shutdownPool()
        {
                while (!jobQueue.isEmpty()) {
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
                shutdown = true;
                for (Thread workerThread : jobThreads) {
                        workerThread.interrupt();
                }
        }

        private class Worker extends Thread
        {
                public Worker(String name)
                {
                        super(name);
                }

                public void run()
                {
                        while (!shutdown) {
                                try {
                                        Job r = jobQueue.take();
                                        r.run();
                                } catch (InterruptedException e) {
                                }
                        }
                }
        }

}
```

The above class is used to create n threads (worker threads). Each worker thread will wait for a job in a queue and then execute the job and will go back to waiting state. The class contains a job queue; when a new job will be added into the queue, a worker thread from the pool will execute the job.

We also include a `shutdownPool` method which will used to shut down the pool by interrupting all the worker threads only when the job queue is empty. The `addJob` method is used to add jobs to the queues.

Now, let's test the code.

```
package com.javacodegeeks.patterns.commandpattern;


public class TestCommandPattern {
        public static void main(String[] args)
    {
        init();
    }

    private static void init()
    {
        ThreadPool pool = new ThreadPool(10);
```

```
        Email email = null;
        EmailJob  emailJob = new EmailJob();

        Sms sms = null;
        SmsJob smsJob = new SmsJob();

        FileIO fileIO = null;;
        FileIOJob fileIOJob = new FileIOJob();

        Logging logging = null;
        LoggingJob logJob = new LoggingJob();

        for (int i = 0; i < 5; i++) {
                email = new Email();
                emailJob.setEmail(email);

                sms = new Sms();
                smsJob.setSms(sms);

                fileIO = new FileIO();
                fileIOJob.setFileIO(fileIO);

                logging = new Logging();
                logJob.setLogging(logging);

            pool.addJob(emailJob);
            pool.addJob(smsJob);
            pool.addJob(fileIOJob);
            pool.addJob(logJob);
        }
        pool.shutdownPool();
    }

}
```

The above code will result to the following output:

```
Job ID: 9 executing email jobs.
Sending email.......
Job ID: 12 executing logging jobs.
Job ID: 17 executing email jobs.
Sending email.......
Job ID: 13 executing email jobs.
Sending email.......
Job ID: 10 executing sms jobs.
Sending SMS...
Job ID: 11 executing fileIO jobs.
Executing File IO operations...
Job ID: 18 executing sms jobs.
Sending SMS...
Logging...
Job ID: 16 executing logging jobs.
Logging...
Job ID: 15 executing fileIO jobs.
Executing File IO operations...
Job ID: 14 executing sms jobs.
Sending SMS...
Job ID: 12 executing fileIO jobs.
Executing File IO operations...
Job ID: 10 executing logging jobs.
Logging...
Job ID: 18 executing email jobs.
```

```
Sending email.......
Job ID: 16 executing sms jobs.
Sending SMS...
Job ID: 14 executing fileIO jobs.
Executing File IO operations...
Job ID: 9 executing logging jobs.
Logging...
Job ID: 17 executing email jobs.
Sending email.......
Job ID: 13 executing sms jobs.
Sending SMS...
Job ID: 15 executing fileIO jobs.
Executing File IO operations...
Job ID: 11 executing logging jobs.
Logging...
```

Please note that the output may differ on subsequent executions.

In the above class, we created a thread pool with 10 threads. Then, we set different command objects with different jobs and add these jobs to the queue using the `addJob` method of the `ThreadPool` class. As soon as the job is inserted into the queue, a thread executes the job and removes it from the queue.

We have set different type of jobs, but by using the Command Design Pattern, we decouple the job from the invoker thread. The thread will execute any kind of object that implements the `Job` interface. The different command objects encapsulate the different object and executed the requested operations on these objects.

The output shows the different threads executing the different job. By watching the job id in the output, you can clearly see that a single thread is executing more than one job. This is because after executing a job the thread sends back to the pool.

The advantage of the Command Design Pattern is that you can add more different kind of jobs without changing the existing classes. The leads to more flexibility, and maintainability and also reduce the chances of bugs in the code.

20.4 When to use the Command Design Pattern

Use the Command pattern when you want to:

- Parameterize objects by an action to perform.
- Specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there.
- Support undo. The Command's `Execute` operation can store state for reversing its effects in the command itself. The `Command` interface must have an added `Un-execute` operation that reverses the effects of a previous call to Execute. Executed commands are stored in a history list. Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling `Un-execute` and `Execute`, respectively.
- Support logging changes so that they can be reapplied in case of a system crash. By augmenting the Command interface with load and store operations, you can keep a persistent log of changes. Recovering from a crash involves reloading logged commands from disk and re-executing them with the Execute operation.
- Structure a system around high-level operations built on primitives operations. Such a structure is common in information systems that support transactions. A transaction encapsulates a set of changes to data. The Command pattern offers a way to model transactions. Commands have a common interface, letting you invoke all transactions the same way. The pattern also makes it easy to extend the system with new transactions.

20.5 Command Design Pattern in JDK

- `java.lang.Runnable`
- `javax.swing.Action`

20.6 Download the Source Code

This was a lesson on the Command Design Pattern. You may download the source code here: **CommandPattern-Project**

Chapter 21

Interpreter Design Pattern

21.1 Introduction

The Interpreter Design Pattern is a heavy-duty pattern. It's all about putting together your own programming language, or handling an existing one, by creating an interpreter for that language. To use this pattern, you have to know a fair bit about formal grammars to put together a language. As you can imagine, this is one of those patterns that developers don't really use every day, because creating your own language is not something many people do.

For example, defining an expression in your new language might look something like the following snippet in terms of formal grammars:

```
expression ::=<command> | <repetition> | <sequence>
```

Each expression in your new language, then, might be made up of commands, repetitions of commands, and sequences expressions. Each item might be represented as an object with an interpret method to translate your new language into something you can run in Java.

To illustrate the use of Interpreter Design Pattern let's create an example to solve simple mathematical expressions, but before that, let's discuss some details about the Interpreter Design Pattern in the section below.

21.2 What is the Interpreter Design Pattern

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

In general, languages are made up of a set of grammar rules. Different sentences can be constructed by following these grammar rules. Sometimes an application may need to process repeated occurrences of similar requests that are a combination of a set of grammar rules. These requests are distinct but are similar in the sense that they are all composed using the same set of rules.

A simple example of this would be the set of different arithmetic expressions submitted to a calculator program. Though each such expression is different, they are all constructed using the basic rules that make up the grammar for the language of arithmetic expressions.

In such cases, instead of treating every distinct combination of rules as a separate case, it may be beneficial for the application to have the ability to interpret a generic combination of rules. The Interpreter pattern can be used to design this ability in an application so that other applications and users can specify operations using a simple language defined by a set of grammar rules.

A class hierarchy can be designed to represent the set of grammar rules with every class in the hierarchy representing a separate grammar rule. An Interpreter module can be designed to interpret the sentences constructed using the class hierarchy designed above and carries out the necessary operations.

Because a different class represents every grammar rule, the number of classes increases with the number of grammar rules. A language with extensive, complex grammar rules requires a large number of classes. The Interpreter pattern works best when the

grammar is simple. Having a simple grammar avoids the need to have many classes corresponding to the complex set of rules involved, which are hard to manage and maintain.

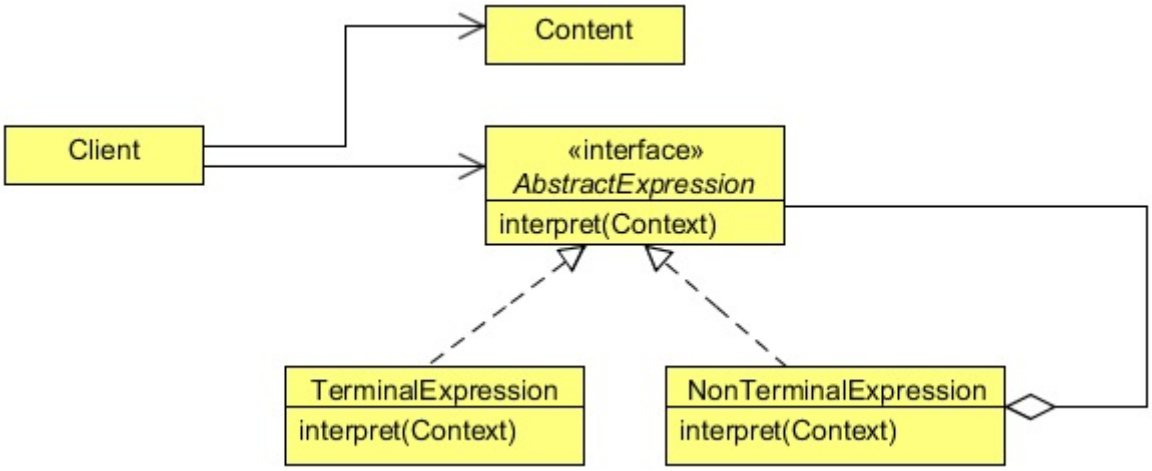

Figure 21.1: - Class Diagram

AbstractExpression

- Declares an abstract `Interpret` operation that is common to all nodes in the abstract syntax tree.

TerminalExpression

- Implements an `Interpret` operation associated with terminal symbols in the grammar.
- An instance is required for every terminal symbol in a sentence.

NonterminalExpression

- One such class is required for every rule `R ::=R1 R2 ...Rn` in the grammar.
- Maintains instance variables of type `AbstractExpression` for each of the symbols `R1` through `Rn`.
- Implements an `Interpret` operation for non terminal symbols in the grammar. `Interpret` typically calls itself recursively on the variables representing `R1` through `Rn`.

Context

- Contains information that's global to the interpreter.

Client

- Builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the `NonterminalExpression` and `TerminalExpression` classes.
- Invokes the `Interpret` operation.

21.3 Implementing the Interpreter Design Pattern

```
package com.javacodegeeks.patterns.interpreterpattern;

public interface Expression {
        public int interpret();
}
```

The above interface is used by all different concrete expressions and overrides the interpret method to define their specific operation on the expression.

The following are the operation specific expression classes.

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Add implements Expression{

        private final Expression leftExpression;
        private final Expression rightExpression;

        public Add(Expression leftExpression,Expression rightExpression ){
                this.leftExpression = leftExpression;
                this.rightExpression = rightExpression;
        }
        @Override
        public int interpret() {
                return leftExpression.interpret() + rightExpression.interpret();
        }

}
```

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Product implements Expression{

        private final Expression leftExpression;
        private final Expression rightExpression;

        public Product(Expression leftExpression,Expression rightExpression ){
                this.leftExpression = leftExpression;
                this.rightExpression = rightExpression;
        }
        @Override
        public int interpret() {
                return leftExpression.interpret() * rightExpression.interpret();
        }
}
```

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Substract implements Expression{

        private final Expression leftExpression;
        private final Expression rightExpression;

        public Substract(Expression leftExpression,Expression rightExpression ){
                this.leftExpression = leftExpression;
                this.rightExpression = rightExpression;
        }
        @Override
        public int interpret() {
```

```
                return leftExpression.interpret() - rightExpression.interpret();
        }

}
```

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Number implements Expression{

        private final int n;

        public Number(int n){
                this.n = n;
        }
        @Override
        public int interpret() {
                return n;
        }

}
```

Below is the optional utility class that contains different utility methods used to execute the expression.

```
package com.javacodegeeks.patterns.interpreterpattern;

public class ExpressionUtils {

        public static boolean isOperator(String s) {
                if (s.equals("+") || s.equals("-") || s.equals("*"))
                        return true;
                else
                        return false;
        }

        public static Expression getOperator(String s, Expression left, Expression right) {
                switch (s) {
                case "+":
                        return new Add(left, right);
                case "-":
                        return new Substract(left, right);
                case "*":
                        return new Product(left, right);
                }
                return null;
        }

}
```

Now, let's test the example.

```
package com.javacodegeeks.patterns.interpreterpattern;

import java.util.Stack;

public class TestInterpreterPattern {

        public static void main(String[] args) {

                String tokenString = "7 3 - 2 1 + *";
                Stack<Expression> stack = new Stack<>();
                String[] tokenArray = tokenString.split(" ");
                for (String s : tokenArray) {
```

```
                        if (ExpressionUtils.isOperator(s)) {
                                Expression rightExpression = stack.pop();
                                Expression leftExpression = stack.pop();
                                Expression operator = ExpressionUtils.getOperator(s,  ↩
                                    leftExpression,rightExpression);
                                int result = operator.interpret();
                                stack.push(new Number(result));
                        } else {
                                Expression i = new Number(Integer.parseInt(s));
                                stack.push(i);
                        }
                }
                System.out.println("( "+tokenString+" ): "+stack.pop().interpret());

        }


}
```

The code above will provide the following output:

```
( 7 3 -2 1 + * ):12
```

Please note that we have used a postfix expression to solve it.

If you don't know about postfix, here is a brief introduction about it. There are three notations to a mathematical expression i.e. infix, postfix, and prefix.

- **Infix** notation is the common arithmetic and logical formula notation, in which operators are written infix-style between the operands they act on e.g. `3+4`.
- A **postfix** a.k.a. Reverse Polish notation (RPN) is mathematical notation in which every operator follows all of its operands e.g. `34+`.
- **Prefix** (Polish notation) is a form of notation for logic, arithmetic, and algebra in which operators to the left of their operands e.g. `+34`.

The Infix notation is a normally used in mathematical expression. The other two notations are used as syntax for mathematical expressions by interpreters of programming languages.

In the above class, we declared a postfix of an expression in `tokenString` variable. Then we split the `tokenString` and assigned it into an array, the `tokenArray`. While iterating tokens one by one, first we have checked whether the token is an operator or an operand. If the token is an operand we pushed it into the stack, but if it is an operator we popped the first two operands from the stack. The `getOperation` method from `ExpressionUtils` returns the appropriate expression class according to the operator passed to it.

Then, we interpret the result and pushed it back to the stack. After iterating the complete `tokenList` we got the final result.

21.4 When to use the Interpreter Design Pattern

Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. The Interpreter pattern works best when

- The grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable. Tools such as parser generators are a better alternative in such cases. They can interpret expressions without building abstract syntax trees, which can save space and possibly time.
- Efficiency is not a critical concern. The most efficient interpreters are usually not implemented by interpreting parse trees directly but by first translating them into another form. For example, regular expressions are often transformed into state machines. But even then, the translator can be implemented by the Interpreter pattern, so the pattern is still applicable.

21.5 Interpreter Design Pattern in JDK

- `java.util.Pattern`
- `java.text.Normalizer`
- `java.text.Format`

21.6 Download the Source Code

This was a lesson on the Interpreter Design Pattern. You may download the source code here: **InterpreterPattern-Project**

Chapter 22

Decorator Design Pattern

22.1 Introduction

To understand the Decorator Design Pattern, let's help a pizza company make an extra topping calculator. A user can ask to add extra topping to a pizza and our job is to add toppings and increase its price using the system.

This is something like adding an extra responsibility to our pizza object at runtime and the Decorator Design Pattern is suitable for this type of requirement. But before that, let us know more about this beautiful pattern.

22.2 What is the Decorator Design Pattern

The intent of the Decorator Design Pattern is to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

The Decorator Pattern is used to extend the functionality of an object dynamically without having to change the original class source or using inheritance. This is accomplished by creating an object wrapper referred to as a `Decorator` around the actual object.

The `Decorator` object is designed to have the same interface as the underlying object. This allows a client object to interact with the `Decorator` object in exactly the same manner as it would with the underlying actual object. The `Decorator` object contains a reference to the actual object. The `Decorator` object receives all requests (calls) from a client. In turn, it forwards these calls to the underlying object. The `Decorator` object adds some additional functionality before or after forwarding requests to the underlying object. This ensures that the additional functionality can be added to a given object externally at runtime without modifying its structure.

Decorator prevents the proliferation of subclasses leading to less complexity and confusion. It is easy to add any combination of capabilities. The same capability can even be added twice. It becomes possible to have different decorator objects for a given object simultaneously. A client can choose what capabilities it wants by sending messages to an appropriate decorator.

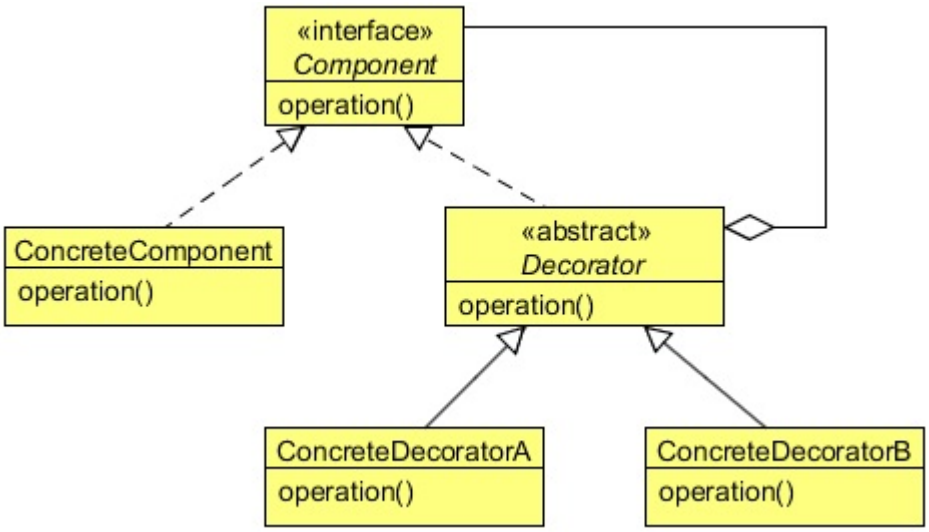

Figure 22.1: - Class Diagram

Component

- Defines the interface for objects that can have responsibilities added to them dynamically.

ConcreteComponent

- Defines an object to which additional responsibilities can be attached.

Decorator

- Maintains a reference to a `Component` object and defines an interface that conforms to Component's interface.

ConcreteDecorator

- Adds responsibilities to the component.

22.3 Implementing the Decorator Design Pattern

For simplicity, let's create a simple `Pizza` interface which contains only two methods.

```
package com.javacodegeeks.patterns.decoratorpattern;

public interface Pizza {

        public String getDesc();
        public double getPrice();
}
```

The `getDesc` method is used to get the pizza's description whereas the `getPrice` is used to get the price.

Below are the two concrete `Pizza` classes:

```
package com.javacodegeeks.patterns.decoratorpattern;

public class SimplyVegPizza implements Pizza{
```

```
        @Override
        public String getDesc() {
                return "SimplyVegPizza (230)";
        }

        @Override
        public double getPrice() {
                return 230;
        }


}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class SimplyNonVegPizza implements Pizza{

        @Override
        public String getDesc() {
                return "SimplyNonVegPizza (350)";
        }

        @Override
        public double getPrice() {
                return 350;
        }

}
```

The decorator wraps the object which functionality needs to be increased, so it needs to implement the same interface. Below is an abstract decorator class which will be extended by all the concrete decorators.

```
package com.javacodegeeks.patterns.decoratorpattern;

public abstract class PizzaDecorator implements Pizza {

        @Override
        public String getDesc() {
                return "Toppings";
        }

}
```

The following are the concrete decorator classes.

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Broccoli extends PizzaDecorator{

        private final Pizza pizza;

        public Broccoli(Pizza pizza){
                this.pizza = pizza;
        }

        @Override
        public String getDesc() {
                return pizza.getDesc()+", Broccoli (9.25)";
        }
```

```
        @Override
        public double getPrice() {
                return pizza.getPrice()+9.25;
        }

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Cheese extends PizzaDecorator{

        private final Pizza pizza;

        public Cheese(Pizza pizza){
                this.pizza = pizza;
        }

        @Override
        public String getDesc() {
                return pizza.getDesc()+", Cheese (20.72)";
        }


        @Override
        public double getPrice() {
                return pizza.getPrice()+20.72;
        }

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Chicken extends PizzaDecorator{

        private final Pizza pizza;

        public Chicken(Pizza pizza){
                this.pizza = pizza;
        }

        @Override
        public String getDesc() {
                return pizza.getDesc()+", Chicken (12.75)";
        }


        @Override
        public double getPrice() {
                return pizza.getPrice()+12.75;
        }

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class FetaCheese extends PizzaDecorator{

        private final Pizza pizza;

        public FetaCheese(Pizza pizza){
```

```
                this.pizza = pizza;
        }

        @Override
        public String getDesc() {
                return pizza.getDesc()+", Feta Cheese (25.88)";
        }


        @Override
        public double getPrice() {
                return pizza.getPrice()+25.88;
        }

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class GreenOlives extends PizzaDecorator{

        private final Pizza pizza;

        public GreenOlives(Pizza pizza){
                this.pizza = pizza;
        }

        @Override
        public String getDesc() {
                return pizza.getDesc()+", Green Olives (5.47)";
        }


        @Override
        public double getPrice() {
                return pizza.getPrice()+5.47;
        }

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Ham extends PizzaDecorator{

        private final Pizza pizza;

        public Ham(Pizza pizza){
                this.pizza = pizza;
        }

        @Override
        public String getDesc() {
                return pizza.getDesc()+", Ham (18.12)";
        }


        @Override
        public double getPrice() {
                return pizza.getPrice()+18.12;
        }

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Meat extends PizzaDecorator{

	private final Pizza pizza;

	public Meat(Pizza pizza){
		this.pizza = pizza;
	}

	@Override
	public String getDesc() {
		return pizza.getDesc()+", Meat (14.25)";
	}


	@Override
	public double getPrice() {
		return pizza.getPrice()+14.25;
	}

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class RedOnions extends PizzaDecorator{

	private final Pizza pizza;

	public RedOnions(Pizza pizza){
		this.pizza = pizza;
	}

	@Override
	public String getDesc() {
		return pizza.getDesc()+", Red Onions (3.75)";
	}


	@Override
	public double getPrice() {
		return pizza.getPrice()+3.75;
	}

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class RomaTomatoes extends PizzaDecorator{

	private final Pizza pizza;

	public RomaTomatoes(Pizza pizza){
		this.pizza = pizza;
	}

	@Override
	public String getDesc() {
		return pizza.getDesc()+", Roma Tomatoes (5.20)";
	}
```

```
        @Override
        public double getPrice() {
                return pizza.getPrice()+5.20;
        }

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Spinach extends PizzaDecorator{

        private final Pizza pizza;

        public Spinach(Pizza pizza){
                this.pizza = pizza;
        }

        @Override
        public String getDesc() {
                return pizza.getDesc()+", Spinach (7.92)";
        }


        @Override
        public double getPrice() {
                return pizza.getPrice()+7.92;
        }

}
```

We need to decorate our pizza object with these toppings. The above classes contain a reference to a pizza object that needs to be decorated. The decorator object adds its functionality to the decorator after calling the decorator's function.

```
package com.javacodegeeks.patterns.decoratorpattern;

import java.text.DecimalFormat;

public class TestDecoratorPattern {

        public static void main(String[] args) {

                DecimalFormat dformat = new DecimalFormat("#.##");
                Pizza pizza = new SimplyVegPizza();

                pizza = new RomaTomatoes(pizza);
                pizza = new GreenOlives(pizza);
                pizza = new Spinach(pizza);

                System.out.println("Desc: "+pizza.getDesc());
                System.out.println("Price: "+dformat.format(pizza.getPrice()));

                pizza = new SimplyNonVegPizza();

                pizza = new Meat(pizza);
                pizza = new Cheese(pizza);
                pizza = new Cheese(pizza);
                pizza = new Ham(pizza);

                System.out.println("Desc: "+pizza.getDesc());
                System.out.println("Price: "+dformat.format(pizza.getPrice()));
        }
```

```
}
```

The above code will results the following output:

```
Desc: SimplyVegPizza (230), Roma Tomatoes (5.20), Green Olives (5.47), Spinach (7.92)
Price: 248.59
Desc: SimplyNonVegPizza (350), Meat (14.25), Cheese (20.72), Cheese (20.72), Ham (18.12)
Price: 423.81
```

In the above class, first we have created a `SimplyVegPizza` and then decorated it with `RomaTomatoes`, `GreenOlives`, and `Spinach`. The `desc` in the output shows the toppings added in the `SimplyVegPizza` and the price are the sum of all.

We did the same thing for the `SimplyNonVegPizza` and added different topping on it. Please note that you can decorate the same thing more than once for an object. In the above example, we added `cheese` twice; it got added twice in the price too, which can be seen in the output.

The Decorator Design Pattern looks good when you need to add extra functionality to an object with modifying it, at runtime. But this results in lots of little objects. A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.

22.4 When to use the Decorator Design Pattern

Use the Decorator pattern in the following cases:

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- For responsibilities that can be withdrawn.
- When extension by sub-classing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for sub-classing.

22.5 Decorator Design Pattern in Java

- `java.io.BufferedInputStream(InputStream)`
- `java.io.DataInputStream(InputStream)`
- `java.io.BufferedOutputStream(OutputStream)`
- `java.util.zip.ZipOutputStream(OutputStream)`
- `java.util.Collections#checked[List|Map|Set|SortedSet|SortedMap]()`

22.6 Download the Source Code

This was a lesson on the Decorator Design Pattern.

You may download the relevant source code here: **DecoratorPattern-Project**

Chapter 23

Iterator Design Pattern

23.1 Introduction

An aggregate object, such as a list, should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the `List` interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list.

The Iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object. The `Iterator` class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

23.2 What is the Iterator Design Pattern

The intent of the Iterator Design Pattern is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The Iterator pattern allows a client object to access the contents of a container in a sequential manner, without having any knowledge about the internal representation of its contents. The term container, used above, can simply be defined as a collection of data or objects. The objects within the container could in turn be collections, making it a collection of collections.

The Iterator pattern enables a client object to traverse through this collection of objects (or the container) without having the container to reveal how the data is stored internally. To accomplish this, the Iterator pattern suggests that a Container object should be designed to provide a public interface in the form of an Iterator object for different client objects to access its contents. An Iterator object contains public methods to allow a client object to navigate through the list of objects within the container.

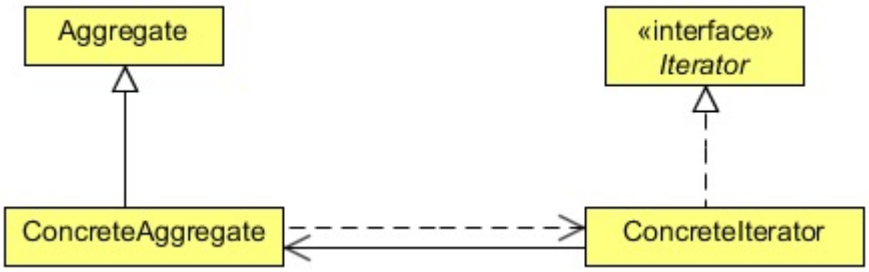

Figure 23.1: - Class diagram

Iterator

- Defines an interface for accessing and traversing elements.

ConcreteIterator

- Implements the Iterator interface.
- Keeps track of the current position in the traversal of the aggregate.

Aggregate

- Defines an interface for creating an Iterator object.

ConcreteAggregate

- Implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

23.3 Implementing the Iterator Design Pattern

Let us implement the Iterator Design Pattern using a `Shape` class. We will store and iterate the `Shape` objects using an iterator.

```
package com.javacodegeeks.patterns.iteratorpattern;

public class Shape {

        private int id;
        private String name;

        public Shape(int id, String name){
                this.id = id;
                this.name = name;
        }

        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;
        }

        @Override
        public String toString(){
                return "ID: "+id+" Shape: "+name;
        }

}
```

The simple `Shape` class has an `id` and `name` as its attributes.

```
package com.javacodegeeks.patterns.iteratorpattern;

public class ShapeStorage {

        private Shape []shapes = new Shape[5];
```

```
        private int index;

        public void addShape(String name){
                int i = index++;
                shapes[i] = new Shape(i,name);
        }

        public Shape[] getShapes(){
                return shapes;
        }
}
```

The above class is used to store the `Shape` objects. The class contains an array of `Shape` type, for simplicity we have initialized that array up to 5. The `addShape` method is used to add a `Shape` object to the array and increment the index by one. The `getShapes` method returns the array of `Shape` type.

```
package com.javacodegeeks.patterns.iteratorpattern;

import java.util.Iterator;

public class ShapeIterator implements Iterator<Shape>{

        private Shape [] shapes;
        int pos;

        public ShapeIterator(Shape []shapes){
                this.shapes = shapes;
        }
        @Override
        public boolean hasNext() {
                if(pos >= shapes.length || shapes[pos] == null)
                        return false;
                return true;
        }

        @Override
        public Shape next() {
                return shapes[pos++];
        }

        @Override
        public void remove() {
                if(pos <=0 )
                        throw new IllegalStateException("Illegal position");
                if(shapes[pos-1] !=null){
                        for (int i= pos-1; i<(shapes.length-1);i++){
                                shapes[i] = shapes[i+1];
                        }
                        shapes[shapes.length-1] = null;
                }
        }
}
```

The above class is an `Iterator` to the `Shape` class. The class implements the `Iterator` interface and defines all the methods of the `Iterator` interface.

The `hasNext` method returns a `boolean` if there's an item left. The next method returns the next item from the collection and the remove method remove the current item from the collection.

```
package com.javacodegeeks.patterns.iteratorpattern;

public class TestIteratorPattern {
```

```
        public static void main(String[] args) {
                ShapeStorage storage = new ShapeStorage();
                storage.addShape("Polygon");
                storage.addShape("Hexagon");
                storage.addShape("Circle");
                storage.addShape("Rectangle");
                storage.addShape("Square");

                ShapeIterator iterator = new ShapeIterator(storage.getShapes());
                while(iterator.hasNext()){
                        System.out.println(iterator.next());
                }
                System.out.println("Apply removing while iterating...");
                iterator = new ShapeIterator(storage.getShapes());
                while(iterator.hasNext()){
                        System.out.println(iterator.next());
                        iterator.remove();
                }
        }

}
```

The above code will result to the following output:

```
ID: 0 Shape: Polygon
ID: 1 Shape: Hexagon
ID: 2 Shape: Circle
ID: 3 Shape: Rectangle
ID: 4 Shape: Square
Apply removing while iterating...
ID: 0 Shape: Polygon
ID: 2 Shape: Circle
ID: 4 Shape: Square
```

In the above class, we have created a `ShapeStorage` object and stores the `Shape` objects in it. Next, we created a `ShapeIterator` object and assigned it the shapes. We iterated twice, first without calling the remove method and then with the remove method.

The output shows you the impact of the remove method. At first iteration, the iterator prints all the shapes but when the remove method is called, it first prints the current shape and moved to the next shape. Then, we called the remove method on it which removes the current shape and then the iterator points to the next available shape object.

That's why, the output after "Apply removing while iterating..." shows only 0, 2, and 4 shapes object.

23.4 Internal and External Iterators

An iterator can be designed either as an internal iterator or as an external iterator.

23.4.1 Internal Iterators

- The collection itself offers methods to allow a client to visit different objects within the collection. For example, the `java.util.ResultSet` class contains the data and also offers methods such as next to navigate through the item list.
- There can be only one iterator on a collection at any given time.
- The collection has to maintain or save the state of iteration.

23.4.2 External Iterators

- The iteration functionality is separated from the collection and kept inside a different object referred to as an iterator. Usually, the collection itself returns an appropriate iterator object to the client depending on the client input. For example, the `java.util.Vector` class has its iterator defined in the form of a separate object of type `Enumeration`. This object is returned to a client object in response to the `elements()` method call.
- There can be multiple iterators on a given collection at any given time.
- The overhead involved in storing the state of iteration is not associated with the collection. It lies with the exclusive Iterator object.

23.5 When to use the Iterator Design Pattern

Use the Iterator pattern:

- To access an aggregate object's contents without exposing its internal representation.
- To support multiple traversals of aggregate objects.
- To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

23.6 Iterator Pattern in JDK

- `java.util.Iterator`
- `java.util.Enumeration`

23.7 Download the Source Code

This was a lesson on the Iterator Design Pattern. You may download the source code here: **IteratorPattern-Project**

Chapter 24

Visitor Design Pattern

24.1 Introduction

To understand the Visitor Design Pattern, let us revisit the Composite Design Pattern. The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies.

In the Composite Pattern example, we had created an html structure composed of different types of objects. Now suppose that we need to add a `css` class to the html tags. One way to do this is by adding the class when adding a start tag using the `setStartTag` method. But this hard coded setting will create inflexibility to our code.

Another way of doing this is by adding a new method like `addClass` in the parent abstract `HtmlTag` class. All the child classes will override this method and will provide the `css` class. One major drawback of this approach is that, if there are many child classes (will be in large html page), it will become very expensive and hectic to implement this method in all the child classes. And suppose, later we need to add another style element in the tags, we again need to do the same thing.

The Visitor Design Pattern provides you with a way to add new operations on the objects without changing the classes of the elements, especially when the operations change quite often.

24.2 What is the Visitor Design Pattern

The intent of the Visitor Design Pattern is to 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.

The Visitor pattern is useful when designing an operation across a heterogeneous collection of objects of a class hierarchy. The Visitor pattern allows the operation to be defined without changing the class of any of the objects in the collection. To accomplish this, the Visitor pattern suggests defining the operation in a separate class referred to as a visitor class. This separates the operation from the object collection that it operates on. For every new operation to be defined, a new visitor class is created. Since the operation is to be performed across a set of objects, the visitor needs a way of accessing the public members of these objects. This requirement can be addressed by implementing the following two design ideas.

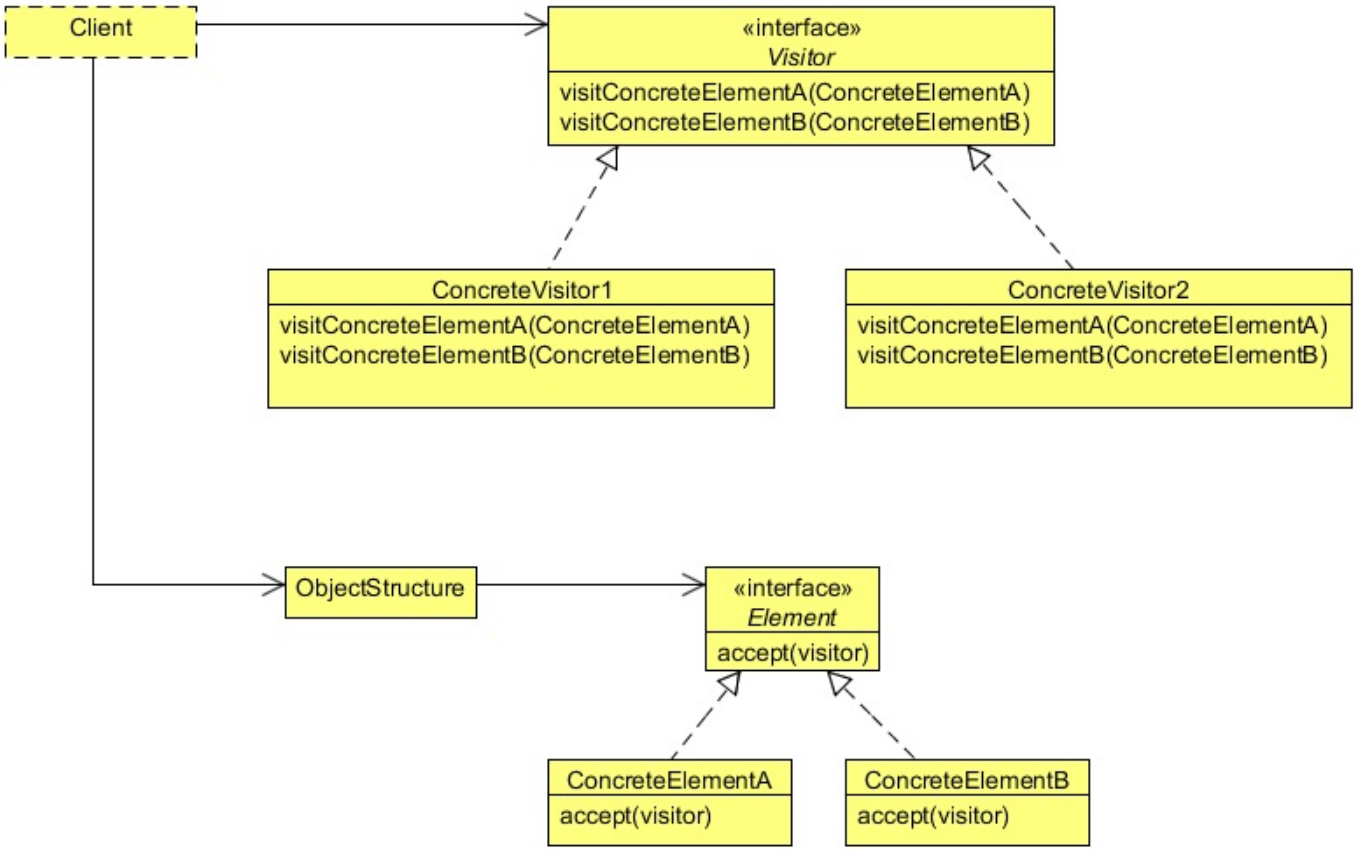

Figure 24.1: - Visitor design pattern Class diagram

Visitor

- Declares a `Visit` operation for each class of `ConcreteElement` in the object structure. The operation's name and signature identifies the class that sends the `Visit` request to the `visitor`. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.

ConcreteVisitor

- Implements each operation declared by `Visitor`. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. `ConcreteVisitor` provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.

Element

- Defines an `Accept` operation that takes a `visitor` as an argument.

ConcreteElement

- Implements an `Accept` operation that takes a `visitor` as an argument.

ObjectStructure

- Can enumerate its elements.
- May provide a high-level interface to allow the `visitor` to visit its elements.
- May either be a composite or a collection such as a list or a set.

24.3 Implement the Visitor Design Pattern

To implement the Visitor Design Pattern, we will use the same Composite Pattern code and will introduce some new interfaces, classes and methods to it.

Implementing Visitor Pattern requires two important interfaces, an `Element` interface which will contain an `accept` method with an argument of type `Visitor`. This interface will be implemented by all the classes that need to allow visitors to visit them. In our case, the `HtmlTag` will implement the `Element` interface, as the `HtmlTag` is the parent abstract class of all the concrete html classes, the concrete classes will inherit and will override the `accept` method of the `Element` interface.

The other important interface is the `Visitor` interface; this interface will contain visit methods with an argument of a class that implements the `Element` interface. Please also note that we have added two new methods in our `HtmlTag` class, the `getStartTag` and the `getEndTag` as opposed to the example shown in the Composite Design Pattern lesson.

```
package com.javacodegeeks.patterns.visitorpattern;

public interface Element {

        public void accept(Visitor visitor);
}

package com.javacodegeeks.patterns.visitorpattern;

public interface Visitor {
        public void visit(HtmlElement element);
        public void visit(HtmlParentElement parentElement);
}
```

The code below is from the Composite Pattern example with a few changes.

```
package com.javacodegeeks.patterns.visitorpattern;

import java.util.List;


public abstract class HtmlTag implements Element{

        public abstract String getTagName();
        public abstract void setStartTag(String tag);
        public abstract String getStartTag();
        public abstract String getEndTag();
        public abstract void setEndTag(String tag);
        public void setTagBody(String tagBody){
                throw new UnsupportedOperationException("Current operation is not support  ↩
                    for this object");
        }
        public void addChildTag(HtmlTag htmlTag){
                throw new UnsupportedOperationException("Current operation is not support  ↩
                    for this object");
        }
        public void removeChildTag(HtmlTag htmlTag){
                throw new UnsupportedOperationException("Current operation is not support  ↩
                    for this object");
        }
        public List<HtmlTag>getChildren(){
                throw new UnsupportedOperationException("Current operation is not support  ↩
                    for this object");
        }
        public abstract void generateHtml();

}
```

The abstract `HtmlTag` class implements the `Element` interface. The below concrete classes will override the accept method of the `Element` interface and will call the `visit` method, and will pass `this` operator as an argument. This will allow the `visitor` method to get all the public members of the object, to add new operations on it.

```
package com.javacodegeeks.patterns.visitorpattern;

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

public class HtmlParentElement extends HtmlTag {

        private String tagName;
        private String startTag;
        private String endTag;
        private List<HtmlTag>childrenTag;

        public HtmlParentElement(String tagName){
                this.tagName = tagName;
                this.startTag = "";
                this.endTag = "";
                this.childrenTag = new ArrayList<>();
        }

        @Override
        public String getTagName() {
                return tagName;
        }

        @Override
        public void setStartTag(String tag) {
                this.startTag = tag;
        }

        @Override
        public void setEndTag(String tag) {
                this.endTag = tag;
        }

        @Override
        public String getStartTag() {
                return startTag;
        }

        @Override
        public String getEndTag() {
                return endTag;
        }

        @Override
        public void addChildTag(HtmlTag htmlTag){
                childrenTag.add(htmlTag);
        }

        @Override
        public void removeChildTag(HtmlTag htmlTag){
                childrenTag.remove(htmlTag);
        }

        @Override
        public List<HtmlTag>getChildren(){
                return childrenTag;
        }
```

```
        @Override
        public void generateHtml() {
                System.out.println(startTag);
                for(HtmlTag tag : childrenTag){
                        tag.generateHtml();
                }
                System.out.println(endTag);

        }

        @Override
        public void accept(Visitor visitor) {
                visitor.visit(this);
        }

}
```

```
package com.javacodegeeks.patterns.visitorpattern;

public class HtmlElement extends HtmlTag{

        private String tagName;
        private String startTag;
        private String endTag;
        private String tagBody;

        public HtmlElement(String tagName){
                this.tagName = tagName;
                this.tagBody = "";
                this.startTag = "";
                this.endTag = "";
        }

        @Override
        public String getTagName() {
                return tagName;
        }

        @Override
        public void setStartTag(String tag) {
                this.startTag = tag;
        }

        @Override
        public void setEndTag(String tag) {
                this.endTag = tag;
        }

        @Override
        public String getStartTag() {
                return startTag;
        }

        @Override
        public String getEndTag() {
                return endTag;
        }

        @Override
        public void setTagBody(String tagBody){
                this.tagBody = tagBody;
```

```
        }

        @Override
        public void generateHtml() {
                System.out.println(startTag+""+tagBody+""+endTag);
        }

        @Override
        public void accept(Visitor visitor) {
                visitor.visit(this);
        }

}
```

Now, the concrete visitor classes: we have created two concrete classes, one will add a `css` class `visitor` to all html tags and the other one will change the width of the tag using the style attribute of the html tag.

```
package com.javacodegeeks.patterns.visitorpattern;

public class CssClassVisitor implements Visitor{

        @Override
        public void visit(HtmlElement element) {
                element.setStartTag(element.getStartTag().replace(">", " class='visitor'>") ↩
                    );

        }

        @Override
        public void visit(HtmlParentElement parentElement) {
                parentElement.setStartTag(parentElement.getStartTag().replace(">", " class ↩
                    ='visitor'>"));
        }

}
```

```
package com.javacodegeeks.patterns.visitorpattern;

public class StyleVisitor implements Visitor {

        @Override
        public void visit(HtmlElement element) {
                element.setStartTag(element.getStartTag().replace(">", " style='width:46px ↩
                    ;'>"));

        }

        @Override
        public void visit(HtmlParentElement parentElement) {
                parentElement.setStartTag(parentElement.getStartTag().replace(">", " style ↩
                    ='width:58px;'>"));
        }

}
```

Now, let"s test the above example.

```
package com.javacodegeeks.patterns.visitorpattern;

public class TestVisitorPattern {
```

```
        public static void main(String[] args) {

                System.out.println("Before visitor..... \\n");

                HtmlTag parentTag = new HtmlParentElement("<html>");
                parentTag.setStartTag("<html>");
                parentTag.setEndTag("</html>");

                HtmlTag p1 = new HtmlParentElement("<body>");
                p1.setStartTag("<body>");
                p1.setEndTag("</body>");

                parentTag.addChildTag(p1);

                HtmlTag child1 = new HtmlElement("<P>");
                child1.setStartTag("<P>");
                child1.setEndTag("</P>");
                child1.setTagBody("Testing html tag library");
                p1.addChildTag(child1);

                child1 = new HtmlElement("<P>");
                child1.setStartTag("<P>");
                child1.setEndTag("</P>");
                child1.setTagBody("Paragraph 2");
                p1.addChildTag(child1);

                parentTag.generateHtml();

                System.out.println("\\nAfter visitor..... \\n");

                Visitor cssClass = new CssClassVisitor();
                Visitor style = new StyleVisitor();

                parentTag = new HtmlParentElement("<html>");
                parentTag.setStartTag("<html>");
                parentTag.setEndTag("</html>");
                parentTag.accept(style);
                parentTag.accept(cssClass);

                p1 = new HtmlParentElement("<body>");
                p1.setStartTag("<body>");
                p1.setEndTag("</body>");
                p1.accept(style);
                p1.accept(cssClass);

                parentTag.addChildTag(p1);

                child1 = new HtmlElement("<P>");
                child1.setStartTag("<P>");
                child1.setEndTag("</P>");
                child1.setTagBody("Testing html tag library");
                child1.accept(style);
                child1.accept(cssClass);

                p1.addChildTag(child1);

                child1 = new HtmlElement("<P>");
                child1.setStartTag("<P>");
                child1.setEndTag("</P>");
                child1.setTagBody("Paragraph 2");
                child1.accept(style);
                child1.accept(cssClass);
```

```
                p1.addChildTag(child1);

                parentTag.generateHtml();
        }

}
```

The above code will result to the following output:

```
Before visitor.........

<html>
<body>
<P>Testing html tag library</P>
<P>Paragraph 2</P>
</body>
</html>

After visitor.......

<html style='width:58px;' class='visitor'>
<body style='width:58px;' class='visitor'>
<p style='width:46px;' class='visitor'>Testing html tag library</P>
<p style='width:46px;' class='visitor'>Paragraph 2</P>
</body>
</html>
```

The output after "Before Visitor..." is the same as it results in the Composite Pattern lesson. Later, we created two concrete visitors and then added them to the concrete html objects using the `accept` method. The output "After visitor..." shows you the result, in which `css` class and style elements are added into the html tags.

Please note that the advantage of the Visitor Pattern is that we can add new operations to the objects without changing its classes. For example, we can add some javascript functions like `onclick` or some angularjs ng tags without modifying the classes.

24.4 When to use the Visitor Design Pattern

Use the Visitor pattern when:

- An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
- The classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes.

24.5 Visitor Design Pattern in JDK

- `javax.lang.model.element.Element` and `javax.lang.model.element.ElementVisitor`
- `javax.lang.model.type.TypeMirror` and `javax.lang.model.type.TypeVisitor`

24.6 Download the Source Code

This was a lesson on the Visitor Design Pattern. You may download the relevant source code here: **VisitorPattern-Project**