THE EXPERT'S VOICE® IN JAVA

Java Design Patterns

A Tour with 23 Gang of Four Design Patterns in Java

Vaskaran Sarcar

Apress®

Java Design Patterns

A Tour of 23 Gang of Four Design Patterns in Java

Vaskaran Sarcar

Apress®

Java Design Patterns: A tour of 23 gang of four design patterns in Java

ISBN-13 (pbk): 978-1-4842-1801-3

ISBN-13 (electronic): 978-1-4842-1802-0

Managing Director: Welmoed Spahr
Lead Editor: Pramila Balan
Technical Reviewers: Anupam Chakraborty and Shekar Maravi
Editorial Board: Steve Anglin, Pramila Balan, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing
Coordinating Editor: Prachi Mehta
Copy Editor: Karen Jameson
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

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

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

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com/9781484218013. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Dedicated to

Almighty GOD

My Family

and

The Gang of Four

You are my inspiration.

Contents at a Glance

Contents

About the Author

Vaskaran Sarcar, ME (Software Engineering), MCA, B Sc. (Math) is a Senior Software Engineer at Hewlett Packard India Software Operation Pvt. Ltd. He has been working in the HP India PPS R&D Hub since August 2009. He is an alumnus of prestigious institutions like Jadavpur University, Kolkata, WB, Vidyasagar University, Midnapore, WB and Presidency College, Kolkata, WB, Jodhpur Park Boys School, Kolkata, WB and Ramakrishna Mission Vidyalaya, Narendrapur, Kolkata, WB. He is also the author of the following books: *Design Patterns in C#, Operating System: Computer Science Interview Series, C# Basics: Test Your Skill*, and *Easy Manifestation*. He devoted his early years (2005–2007) to teaching in various engineering colleges. Later he received the MHRD-GATE Scholarship (India). Reading and learning new things are passions for him. You can reach him at: vaskaran@rediffmail.com.

Acknowledgments

I offer sincere thanks to my family, my friends, my great teachers, my publisher and to everyone who supported this project directly or indirectly. Though it is my book, I believe that it was only with the help of these extraordinary people that I was able to complete this work. Again, thanks to those who helped me to fulfill my desire to help people.

Preface

Welcome to the journey. It is my privilege to present *Java Design Patterns*. Before jumping into the topic, I want to highlight a few points about the topic and contents:

#1. You are an intelligent person. You have chosen a topic that can assist you throughout your career. If you are a developer/programmer, you need these concepts. If you are an architect at a software organization, you need these concepts. If you are a college student, you need these concepts—not only to achieve good grades but also to enter into the corporate world. Even if you are a tester who needs to take care of the white box testing or simply to know about the code paths of the product, these concepts will help you a lot.

#2. This book is written in Java, but the focus is not on the language construct of Java. I has made the examples simple in such a way that if you are familiar with any other popular language (C#, C++, etc.) you can still easily grasp the concept. *Except in a few special places, I have made his best effort to follow the naming conventions used in Java.*

#3. There are many books on this topic or a related topic. Then why was I interested in adding a new one in the same area? The true and simple answer for this question is: I found that the materials on this topic are scattered. In most cases, many of those examples are unnecessarily big and complex. I always like simple examples. He believes that anyone can grasp an idea with simple examples first. And when the concept is clear, readers are motivated to delve deeper. I believe that this book scores high in this area. The examples used here are simple. I didn't want to make the book fat, but he did want to make it concise and simple. He wants to grow interest about this topic in your mind—so that, you can also motivate yourself to continue the journey and to dig further with the concepts in your professional fields.

#4. Each of the topics is divided into seven parts—the definition, the core concept, a real-life example, a computer/coding world example, a UML class diagram, a sample program with a high-level view in Package Explorer and output of the program. So, even before you enter into the coding parts, you will be able to form some impression in your mind. These examples are chosen in such a way that you will be able to get back to the core concepts with these examples whenever you need to do so.

#5. *Please remember that you have just started the journey. You have to consistently think about these concepts and try to write codes, and only then will you master this area. I was also involved (and am still involved!) in the journey. The sources/references are only a small part of this journey. I went through a large number of materials and ultimately he picked up those which made his concept clearer. So, he is grateful to the authors of those sources because those resources helped him ultimately to clear his doubts and increase his interest in the topic. I am also grateful to his family and friends for their continuous help and support.*

#6. Though the patterns described here are broadly classified into three categories—creational, structural, and behavioral—all similar patterns are not grouped together here, *so before you start with the examples, concepts, and definition, you can test your understanding—which category is your pattern falling under. Also, you can start with any pattern you like.*

#7. No book can be completed without the reader's feedback and support. So, please share your comments and feedback to truly complete this book and enhance my work in the future.

#8. Always check for the most updated version available in the market. I have decided to highlight the key changes at the beginning of the book. So that you can also update your copy accordingly whenever a new update appears in this book.

—Vaskaran Sarcar

Key Points

#1. We need to know design patterns to find solutions for frequently occurring problems. And we want to reuse these solutions whenever we face a similar situation in the future.

#2. These are one kind of template to address solutions in many different situations.

#3. In other words, these are the descriptions of how different objects and their respective classes solve a design problem in a specific context.

#4. Here we have discussed 23 design patterns as outlined by the Gang of Four. These patterns can be classified into three major categories:

A. Creational Patterns:

These patterns mainly deal with the instantiation process. Here we make the systems independent from how their objects are created, collected, or represented. The following five patterns fall into this category:

Singleton Pattern

Abstract Factory Pattern

Prototype Pattern

Factory Method Pattern

Builder Pattern

B. Structural Patterns:

Here we focus on how objects and classes are associated or can be composed to make relatively large structures. Inheritance mechanisms are mostly used to combine interfaces or implementations. The following seven patterns fall into this category:

Proxy Pattern

Flyweight Pattern

Bridge Pattern

Facade Pattern

Decorator Pattern

Adapter Pattern

Composite Pattern

C. Behavioral Patterns:

Here our concentration is on algorithms and the assignment of the critical responsibilities among the objects. We also need to focus on the communication between them. We need to take a closer look at the way those objects are interconnected. The following 11 patterns fall into this category.

Observer Pattern

Template Method Pattern

Command Pattern

Iterator Pattern

State Pattern

Mediator Pattern

Strategy Pattern

Chain of Responsibility Pattern

Visitor Pattern

Interpreter Pattern

Memento Pattern

#5. *Here you can start with any pattern you like. I have chosen the simplest examples for your easy understanding. But you must think of the topic, practice more, try to link with other problems, and then ultimately keep doing the code. This process will help you, ultimately, to master the subject.*

CHAPTER 1

Introduction

Over a period of time, software engineers were facing a common problem during the development of various software programs. There were no standards to instruct them how to design and proceed. The issue became significant when a new member (experienced or unexperienced; it does not matter) joined the team and was assigned to do something from scratch or to modify something in the existing product. As already mentioned, since there were no standards, it took a lot of effort to become familiar with the existing product. *Design Patterns* simply addresses this issue and makes a common platform for all developers. *We shall remember that these patterns were intended to be applied in object-oriented designs with the intention of reuse.*

In 1994–95, four Is—Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides—published *Design Patterns—Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995) in which they initiated the concept of design patterns for software development. These Is became known as the Gang of Four (GoF). They introduced 23 patterns which were developed by experienced software engineers over a very long period of time. As a result, now if any new member joins a development team and he knows that the new system is following some specific design patterns, he can actively participate in the development process with the other members of the team within a very short period of time.

The first concept of real-life design pattern came from the building architect Christopher Alexander. In his experience he came to understand some common problems. Then he tried to address those issues with related solutions (for building design) in a uniform manner. People believe that the software industry grasped those concepts because software engineers can also relate their product applications with these building applications.

> *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*
>
> *—Christopher Alexander*

GoF assures us that though the patterns were described for buildings and towns, the same concepts can be applied for the patterns in object-oriented design. We can substitute the original concepts of walls and doors with objects and interfaces. The common thing in both is that at the core, both types of patterns are solution to problems in some context.

In 1995 the original concepts were discussed with C++. Sun Microsystems released its first public implementation—Java 1.0—in 1995 and then it went through various changes. So, the key point is: Java was relatively new at that time. In this book, we'll try to examine these core concepts with Java. The book is written in Java, but still, if you are familiar with any other popular programming languages (C#, C++ etc.), you can also grasp the concept very easily because I has made as his main focus the design patterns and how we can implement the concept with the basic Java language construct. Again: he has chosen simple, easy-to-remember examples to help you to develop these concepts easily.

CHAPTER 2

Observer Patterns

GoF Definition: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Concept

In this pattern, there are many observers (objects) which are observing a particular subject (object). Observers are basically interested and want to be notified when there is a change made inside that subject. So, they register themselves to that subject. When they lose interest in the subject they simply unregister from the subject. Sometimes this model is also referred to as the Publisher-Subscriber model.

Real-Life Example

We can think about a celebrity who has many fans. Each of these fans wants to get all the latest updates of his/her favorite celebrity. So, he/she can follow the celebrity as long as his/her interest persists. When he loses interest, he simply stops following that celebrity. Here we can think of the fan as an observer and the celebrity as a subject.

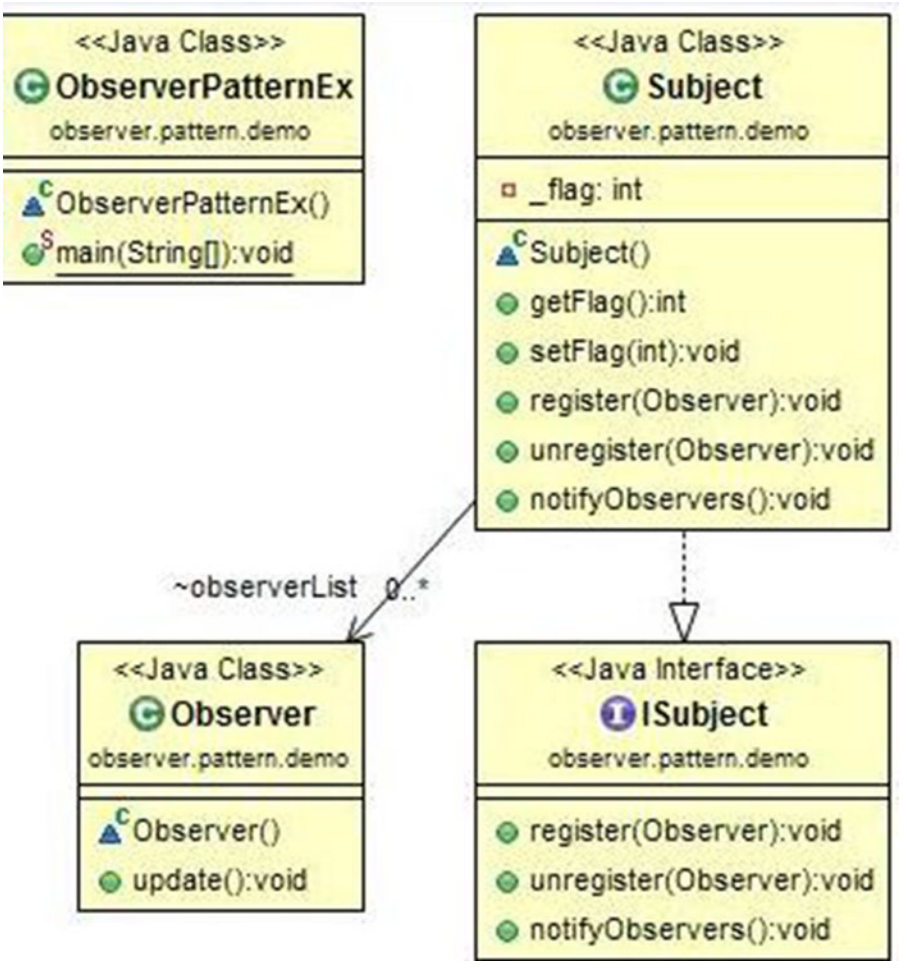
Computer World Example

In the world of computer science, consider a simple UI-based example, where this UI is connected with some database (or business logic). A user can execute some query through that UI and after searching the database, the result is reflected back in the UI. In most of the cases we segregate the UI with the database. If a change occurs in the database, the UI should be notified so that it can update its display according to the change.

Illustration

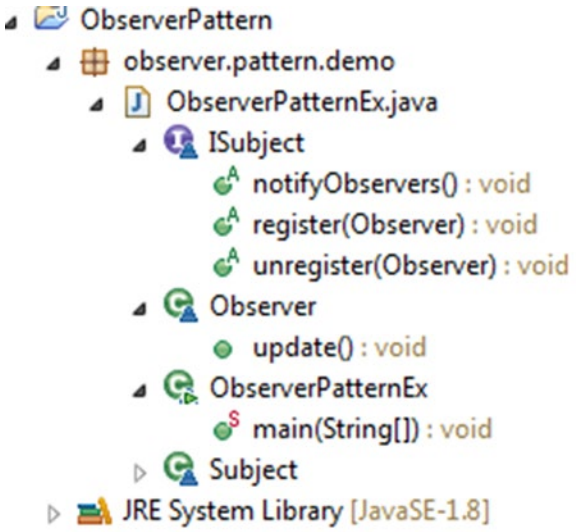
Now let us directly enter into our simple example. Here I have created one observer (though you can create more) and one subject. The subject maintains a list for all of its observers (though here we have only one for simplicity). Our observer here wants to be notified when the flag value changes in the subject. With the output, you will discover that the observer is getting the notifications when the flag value changed to 5 or 25. But there is no notification when the flag value changed to 50 because by this time the observer has unregistered himself from the subject.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:

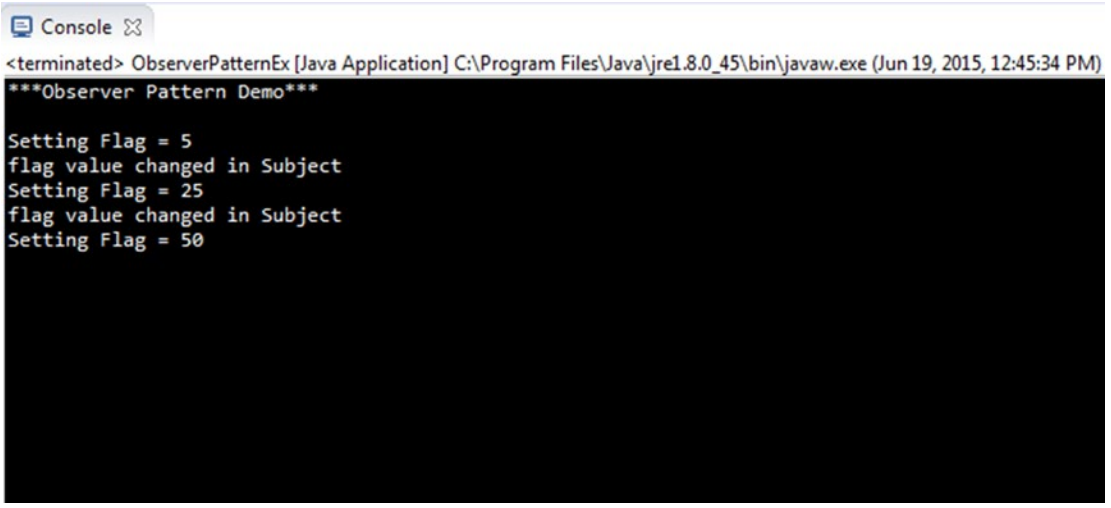


Implementation

```
package observer.pattern.demo;
import java.util.*;

class Observer
    {
        public void update()
        {
                System.out.println("flag value changed in Subject");
        }
    }

    interface ISubject
    {
        void register(Observer o);
        void unregister( Observer o);
        void notifyObservers();
    }
    class Subject implements ISubject
    {
        List<Observer> observerList = new ArrayList<Observer>();
        private int _flag;
        public int getFlag()
        {
           return _flag;
        }
```

```
        public void setFlag(int _flag)
        {
          this._flag=_flag;
          //flag value changed .So notify observer(s)
                notifyObservers();
        }
        @Override
        public void register(Observer o)
        {
            observerList.add(o);
        }
        @Override
        public void unregister(Observer o)
        {
            observerList.remove(o);
        }
        @Override
        public void notifyObservers()
        {
           for(int i=0;i<observerList.size();i++)
                {
                    observerList.get(i).update();
                }
        }
    }
class  ObserverPatternEx
{
        public static void main(String[] args)
    {
                System.out.println("***Observer Pattern Demo***\n");
        Observer o1 = new Observer();
        Subject sub1 = new Subject();
        sub1.register(o1);
        System.out.println("Setting Flag = 5 ");
        sub1.setFlag(5);
        System.out.println("Setting Flag = 25 ");
        sub1.setFlag(25);
        sub1.unregister(o1);
        //No notification this time to o1 .Since it is unregistered.
        System.out.println("Setting Flag = 50 ");
        sub1.setFlag(50);
    }
}
```

Output



Note

The above example is one of the simple illustrations for this pattern. Let us consider a relatively complex problem. Let us assume the following:

1. *Now we need to have a multiple observer class.*
2. *And we also want to know about the exact change in the subject. If you notice our earlier implementation, you can easily understand that there we are getting some kind of notification but our observer does not know about the changed value in the subject*

Obviously now we need to make some change in the above implementation. Note that now we cannot use concrete observers like the following:

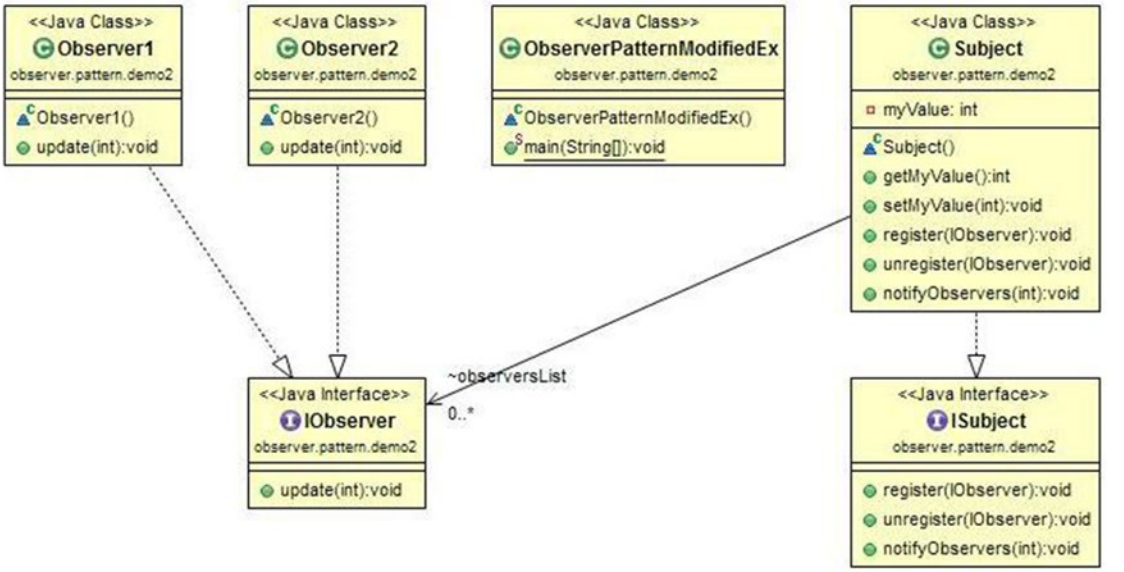
```
List<Observer> observerList = new ArrayList<Observer>();
```

Instead we need to use :

```
List<IObserver> observersList=new ArrayList<IObserver>();
```

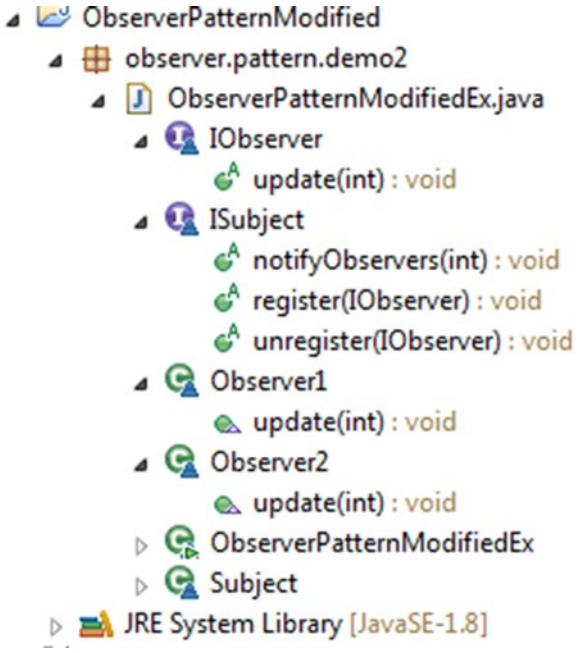
And so we need to replace Observer with IObserver in corresponding places also. We have also modified our update function to see the changed values in the Observers themselves.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the modified program is as follows:



Implementation

```
package observer.pattern.demo2;
import java.util.*;

    interface IObserver
    {
        void update(int i);
    }
    class Observer1 implements IObserver
    {
        @Override
                public void update(int i)
                {
                System.out.println("Observer1: myValue in Subject is now: "+i);
                }
    }
    class Observer2 implements IObserver
    {
        @Override
                public void update(int i)
                {
                        System.out.println("Observer2: observes ->myValue is changed in
                        Subject to :"+i);
                }
    }

    interface ISubject
    {
        void register(IObserver o);
        void unregister(IObserver o);
        void notifyObservers(int i);
    }

    class Subject implements ISubject
    {
        private int myValue;

        public int getMyValue() {
                        return myValue;
                }

                public void setMyValue(int myValue) {
                        this.myValue = myValue;
                         //Notify observers
                        notifyObservers(myValue);
                }
```

```
        List<IObserver> observersList=new ArrayList<IObserver>();

        @Override
    public void register(IObserver o)
    {
        observersList.add(o);
    }
        @Override
    public void unregister(IObserver o)
    {
        observersList.remove(o);
    }
    @Override
    public void notifyObservers(int updatedValue)
    {
       for(int i=0;i<observersList.size();i++)
        {
                observersList.get(i).update(updatedValue);
        }
    }
}

class ObserverPatternModifiedEx
{
    public static void main(String[] args)
    {
        System.out.println("*** Modified Observer Pattern Demo***\n");
        Subject sub = new Subject();
        Observer1 ob1 = new Observer1();
        Observer2 ob2 = new Observer2();

        sub.register(ob1);
        sub.register(ob2);

        sub.setMyValue(5);
        System.out.println();
        sub.setMyValue(25);
        System.out.println();

        //unregister ob1 only
        sub.unregister(ob1);
        //Now only ob2 will observe the change
        sub.setMyValue(100);
    }
}
```

Output

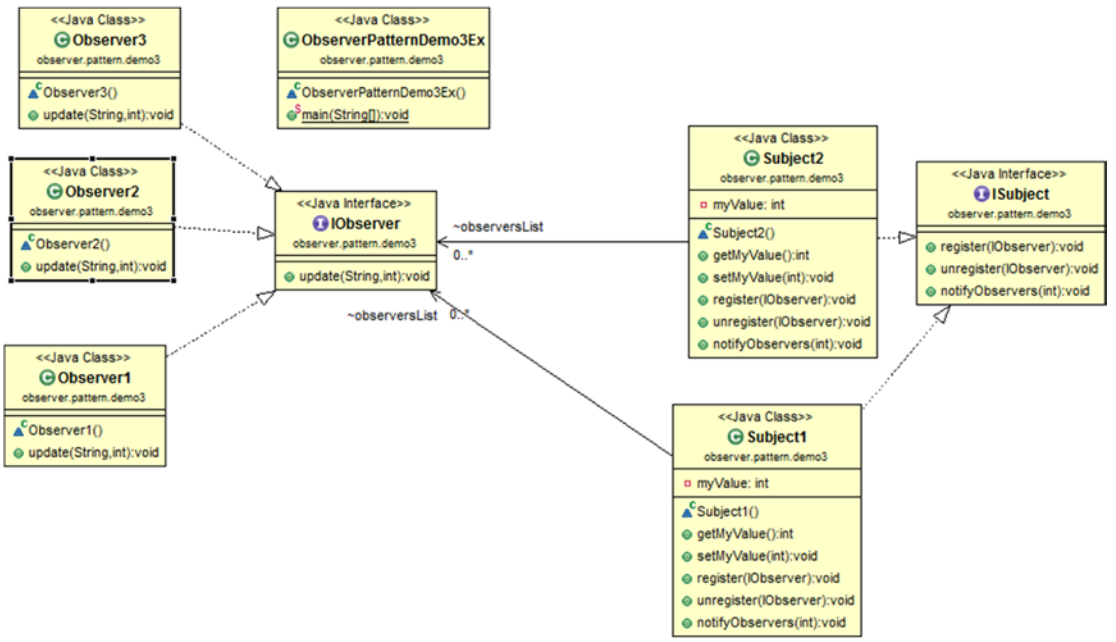
```
Console
<terminated> ObserverPatternModifiedEx (1) [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Jun 19, 2015, 12:59:39 PM)
*** Modified Observer Pattern Demo***

Observer1: myValue in Subject is now: 5
Observer2: observes ->myValue is changed in Subject to :5

Observer1: myValue in Subject is now: 25
Observer2: observes ->myValue is changed in Subject to :25

Observer2: observes ->myValue is changed in Subject to :100
```

Assignment

Implement an observer pattern where you have multiple observers and multiple subjects.

■ **Note** I always suggests that you do the assignments by yourself. Once you complete your task, come back here to match the output.

UML Class Diagram



Implementation

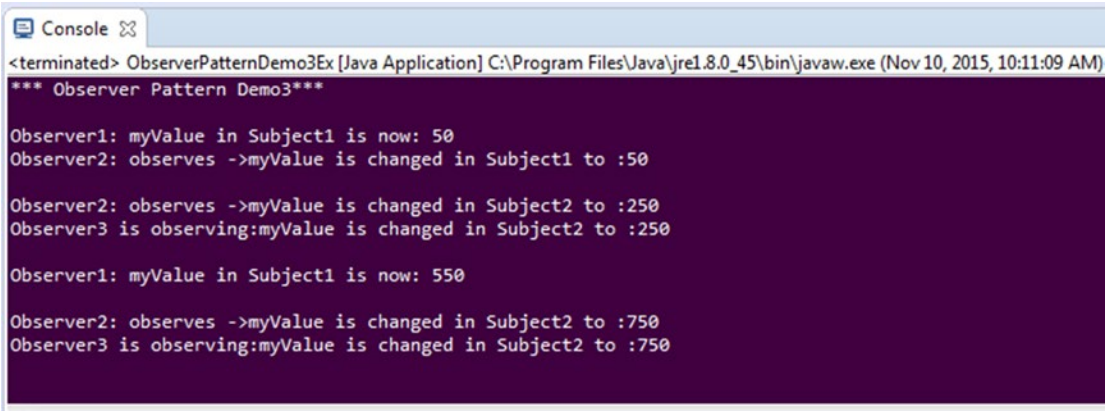
```
package observer.pattern.demo3;

import java.util.*;

    interface IObserver
    {
        void update(String s,int i);
    }
    class Observer1 implements IObserver
    {
        @Override
                public void update(String s,int i)
                {
                System.out.println("Observer1: myValue in "+ s+ " is now: "+i);
                }
    }
    class Observer2 implements IObserver
    {
        @Override
                public void update(String s,int i)
                {
                        System.out.println("Observer2: observes ->myValue is changed in
                        "+s+" to :"+i);
                }
    }
    class Observer3 implements IObserver
    {
        @Override
                public void update(String s,int i)
                {
                        System.out.println("Observer3 is observing:myValue is changed in
                        "+s+" to :"+i);
                }
    }

    interface ISubject
    {
        void register(IObserver o);
        void unregister(IObserver o);
        void notifyObservers(int i);
    }

    class Subject1 implements ISubject
    {
        private int myValue;
```

```
    public int getMyValue() {
                    return myValue;
            }

            public void setMyValue(int myValue) {
                    this.myValue = myValue;
                     //Notify observers
                    notifyObservers(myValue);
            }

            List<IObserver> observersList=new ArrayList<IObserver>();

            @Override
    public void register(IObserver o)
    {
            observersList.add(o);
    }
            @Override
    public void unregister(IObserver o)
    {
            observersList.remove(o);
    }
    @Override
    public void notifyObservers(int updatedValue)
    {
       for(int i=0;i<observersList.size();i++)
            {
                    observersList.get(i).update(this.getClass().getSimpleName(),
                    updatedValue);
            }
    }
}
class Subject2 implements ISubject
{
    private int myValue;

    public int getMyValue() {
                    return myValue;
            }

            public void setMyValue(int myValue) {
                    this.myValue = myValue;
                     //Notify observers
                    notifyObservers(myValue);
            }

            List<IObserver> observersList=new ArrayList<IObserver>();
```

```
        @Override
    public void register(IObserver o)
    {
        observersList.add(o);
    }
        @Override
    public void unregister(IObserver o)
    {
        observersList.remove(o);
    }
    @Override
    public void notifyObservers(int updatedValue)
    {
       for(int i=0;i<observersList.size();i++)
        {
            observersList.get(i).update(this.getClass().getSimpleName(),
            updatedValue);
        }
    }
}

class ObserverPatternDemo3Ex
{
    public static void main(String[] args)
    {
        System.out.println("*** Observer Pattern Demo3***\n");
      Subject1 sub1 = new Subject1();
      Subject2 sub2 = new Subject2();

      Observer1 ob1 = new Observer1();
      Observer2 ob2 = new Observer2();
      Observer3 ob3 = new Observer3();

    //Observer1 and Observer2 registers to //Subject 1
      sub1.register(ob1);
      sub1.register(ob2);
    //Observer2 and Observer3 registers to //Subject 2
      sub2.register(ob2);
      sub2.register(ob3);
    //Set new value to Subject 1
    //Observer1 and Observer2 get //notification
      sub1.setMyValue(50);
      System.out.println();
    //Set new value to Subject 2
    //Observer2 and Observer3 get //notification
      sub2.setMyValue(250);
      System.out.println();
      //unregister Observer2 from Subject 1
      sub1.unregister(ob2);
```

```
        //Set new value to Subject & only //Observer1 is notified
          sub1.setMyValue(550);
          System.out.println();
          //ob2 can still notice change in //Subject 2
          sub2.setMyValue(750);

      }
  }
```

Output



CHAPTER 3

Singleton Patterns

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

Concept

A particular class should have only one instance. We will use only that instance whenever we are in need.

Real-Life Example

Suppose you are a member of a cricket team. And in a tournament your team is going to play against another team. As per the rules of the game, the captain of each side must go for a toss to decide which side will bat (or bowl) first. So, if your team does not have a captain, you need to elect someone as a captain first. And at the same time, your team cannot have more than one captain.

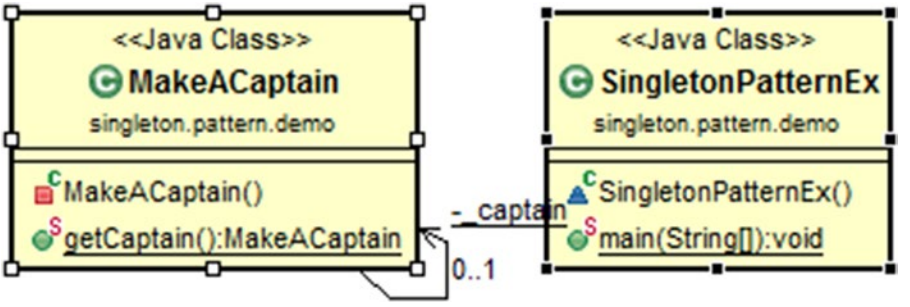
Computer World Example

In a software system sometimes we may decide to use only one file system. Usually we may use it for the centralized management of resources.

Illustration

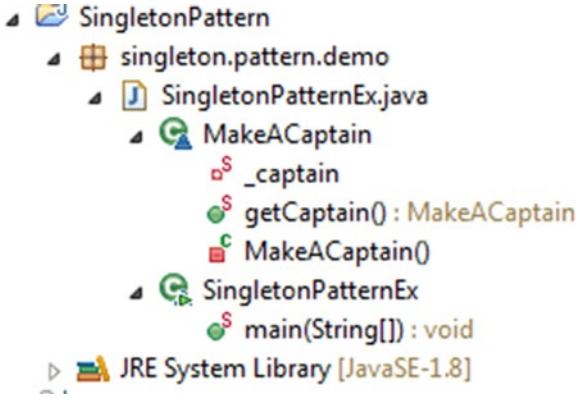
In this example, we have made the constructor private first, so that we cannot instantiate in normal fashion. When we attempt to create an instance of the class, we are checking whether we already have one available copy. If we do not have any such copy, we'll create it; otherwise, we'll simply reuse the existing copy.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:

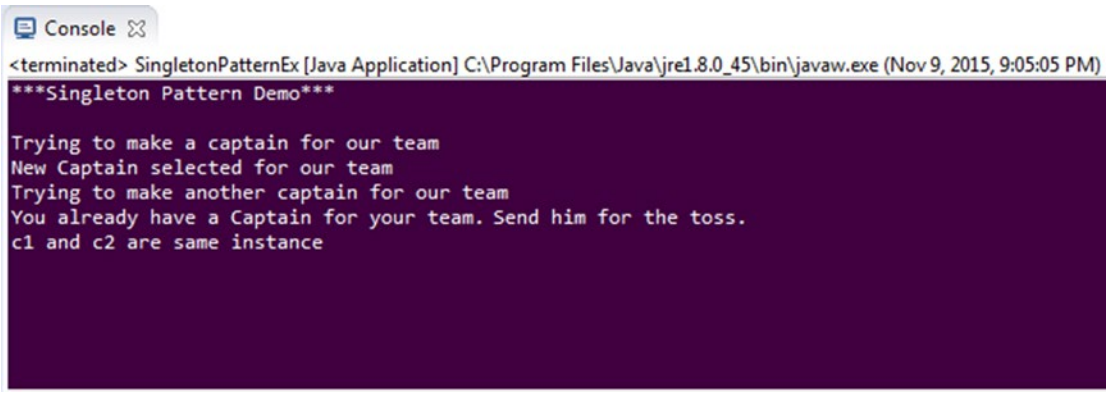


Implementation

```
package singleton.pattern.demo;
class MakeACaptain
{
        private static MakeACaptain _captain;
        //We make the constructor private to prevent the use of "new"
        private MakeACaptain() { }
    public static MakeACaptain getCaptain()
    {

                // Lazy initialization
                if (_captain == null)
                { _captain = new MakeACaptain();
                   System.out.println("New Captain selected for our team");
                }
```

```
                else
                {
                        System.out.print("You already have a Captain for your team.");
                        System.out.println("Send him for the toss.");
                }
                return _captain;

        }
  }
class SingletonPatternEx
{
        public static void main(String[] args)
    {
                System.out.println("***Singleton Pattern Demo***\n");
                System.out.println("Trying to make a captain for our team");
                MakeACaptain c1 = MakeACaptain.getCaptain();
                System.out.println("Trying to make another captain for our team");
                MakeACaptain c2 = MakeACaptain.getCaptain();
                        if (c1 == c2)
                        {
                                System.out.println("c1 and c2 are same instance");
                        }
                }
}
```

Output



Note

Why in the code have we used the term "Lazy initialization"?

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

Point out one improvement area in the above implementation?

The above implementation is not thread safe. So, there may be a situation when two or more threads come into picture and they create more than one object of the singleton class.

So what do we need to do to incorporate thread safety in the above implementation?

There are many discussions on this topic. People came up with their own solutions. But there are always pros and cons. I want to highlight some of them here:

Case (i): Use of "synchronized" keyword:

```
public static synchronized MakeACaptain getCaptain()
{
  //our code
}
```

With the above solution we need to pay for the performance cost associated with this synchronization method.

Case (ii): There is another method} called "Eager initialization" (opposite of "Lazy initialization" mentioned in our original code) to achieve thread safety.

```
class MakeACaptain
      {
          //Early initialization
          private static MakeACaptain _captain = new MakeACaptain();
          //We make the constructor private to prevent the use of "new"
          private MakeACaptain() { }

          // Global point of access //MakeACaptain.getCaptain() is a public static //method
          public static MakeACaptain getCaptain()
          {
              return _captain;
          }
      }
```

In the above solution an object of the singleton class is always instantiated.

Case (iii): To deal with this kind of situation, Bill Pugh came up with a different approach:

```
class MakeACaptain
{
        private static MakeACaptain _captain;
        private MakeACaptain() { }

      //Bill Pugh solution
      private static class SingletonHelper{
      //Nested class is referenced after getCaptain() is called
```

```
        private static final MakeACaptain _captain = new MakeACaptain();
    }

    public static MakeACaptain getCaptain()
    {
        return SingletonHelper._captain;
    }

}
```

This method does not need to use the synchronization technique and eager initialization. It is regarded as the standard method to implement singletons in Java.

CHAPTER 4

Proxy Patterns

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

Concept

We want to use a class which can perform as an interface to something else.

Real–Life Example

In a classroom, when one student is absent, during roll call, his best friend may try to mimic the student's voice to try to keep his friend from being marked as absent.

Computer World Example

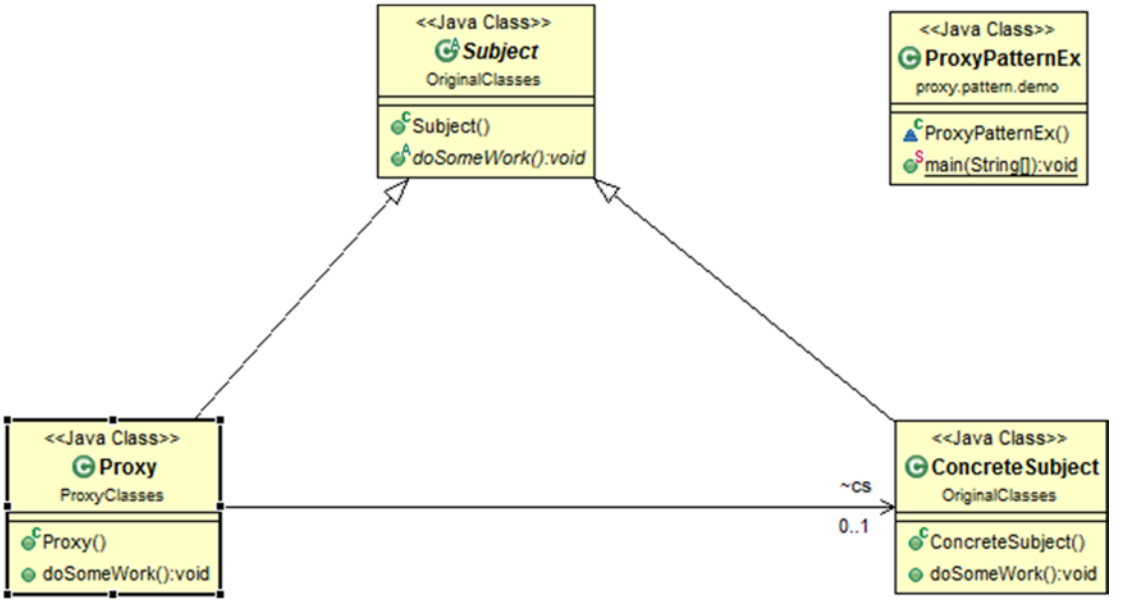
Consider an ATM implementation for a bank. Here we will find multiple proxy objects. Actual bank information will be stored in a remote server. We must remember that in the real programming world, the creation of multiple instances of a complex object (heavy object) is very costly. In such situations, we can create multiple proxy objects (which must point to an original object) and the total creation of actual objects can be carried out on a demand basis. Thus we can save both the memory and creational time.

Illustration

In the following program, we are calling the doSomework() function of the proxy object, which in turn calls the doSomework() of the concrete object. With the output, we are getting the result directly through the concrete object.

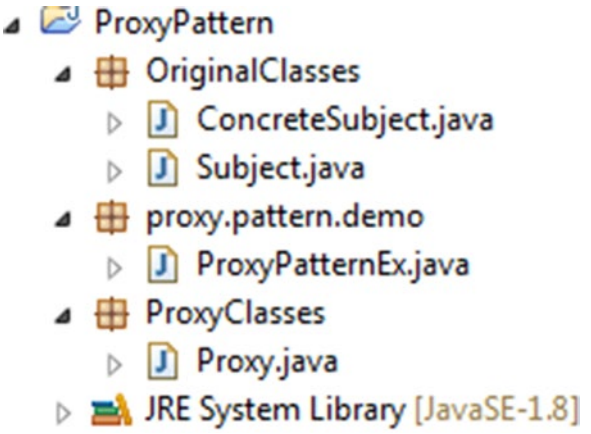
In this example we have followed this structure. Related parts are separated by the packages for better readability. Our concrete or original implementations reside in the package OriginalClasses, Proxy implementation (Proxy.java) is in the package ProxyClasses. A proxy is created and tested in ProxyPatternEx.java.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:

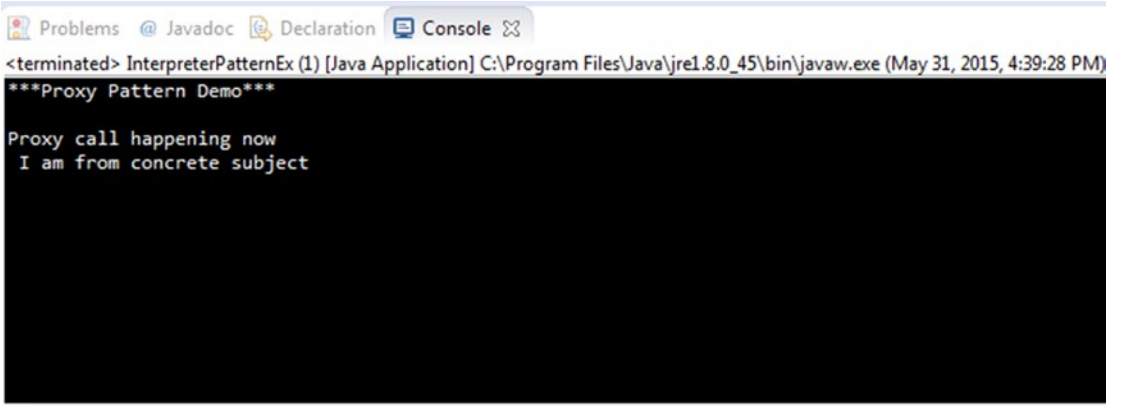


Implementation

```
// Subject.java
package OriginalClasses;

public abstract class Subject
{
    public abstract void doSomeWork();
}

// ConcreteSubject.java
package OriginalClasses;
import OriginalClasses.Subject;

public class ConcreteSubject extends Subject
{

        @Override
        public void doSomeWork()
        {
                System.out.println(" I am from concrete subject");
        }
}
// Proxy.java
package ProxyClasses;
import OriginalClasses.*;
public class Proxy extends Subject
{:
    ConcreteSubject cs;
    @Override
        public void doSomeWork()
        {
                System.out.println("Proxy call happening now");
            //Lazy initialization
            if (cs == null)
            {
                cs = new ConcreteSubject();
            }
            cs.doSomeWork();
        }
}
// ProxyPatternEx.java
package proxy.pattern.demo;
import ProxyClasses.Proxy;
class ProxyPatternEx
    {
        public static void main(String[] args)
        {
```

```
            System.out.println("***Proxy Pattern Demo***\n");
        Proxy px = new Proxy();
        px.doSomeWork();
    }
}
```

Output



Note

What are the different types of proxies?

Mainly we are familiar with the following types:

Remote proxies. They will hide that actual object which is in a different address space.
Virtual proxies. They are used to perform optimization techniques like the creation of a heavy object on a demand basis.
Protection proxies. They generally deal with different access rights.
Smart reference. It can also perform some additional housekeeping work when an object is accessed. A typical operation is counting the number of references to the actual object.

CHAPTER 5

Decorator Patterns

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

Concept

This main principle of this pattern says that we cannot modify existing functionalities but we can extend them. In other words, this pattern is open for extension but closed for modification. The core concept applies when we want to add some specific functionalities to some specific object instead of to the whole class.

Real-Life Example

Suppose you already own a house. Now you have decided to add an additional floor. Obviously, you do not want to change the architecture of ground floor (or existing floors). You may want to change the design of the architecture for the newly added floor without affecting the existing architecture for existing floor(s).

Computer World Example

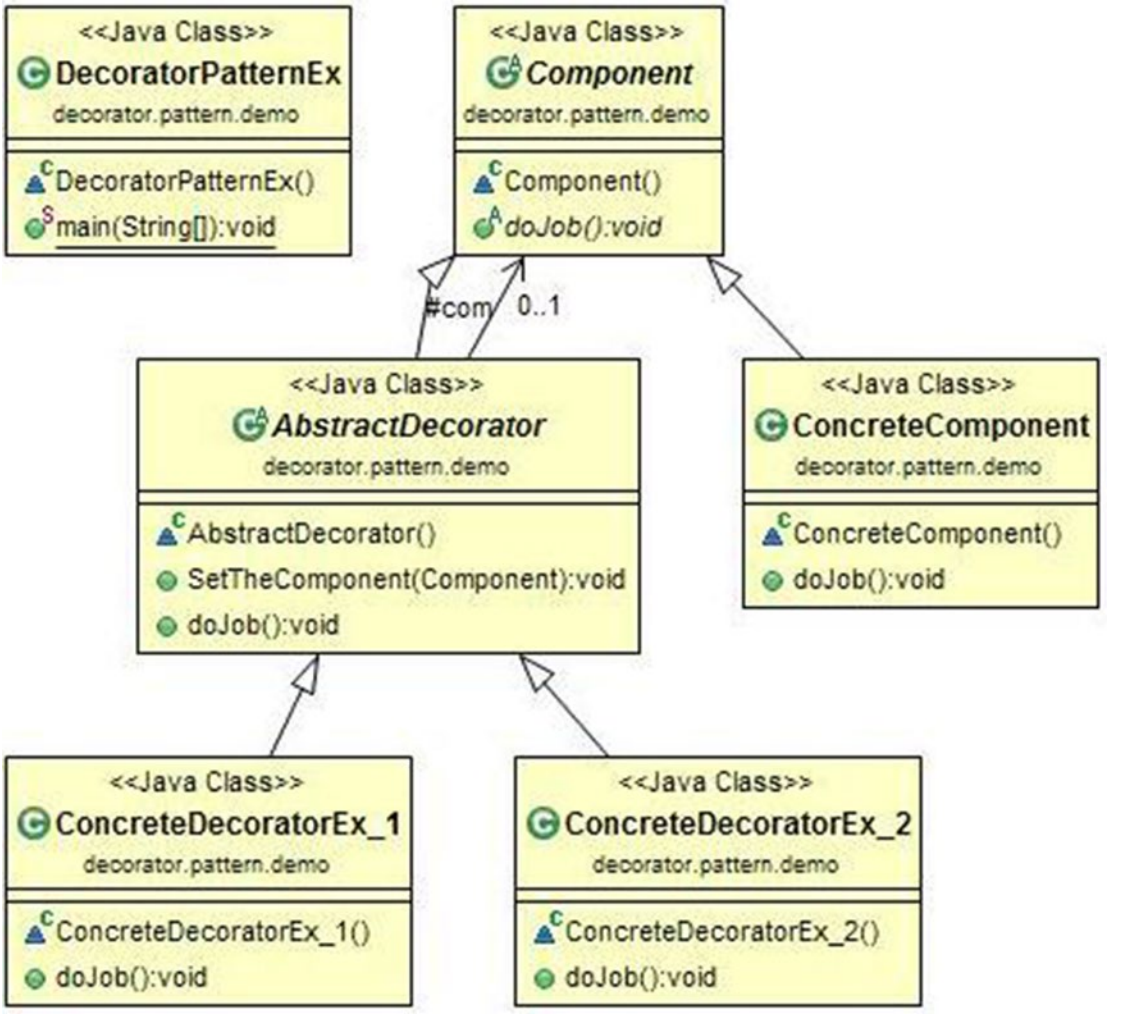
Suppose in a GUI-based toolkit, we want to add some border properties. We can do this by inheritance. But it cannot be treated as the best solution because our user or client cannot have absolute control from the creation. The core of that choice is static there.

Decorator can offer us a more flexible approach: here we may surround the component in another object. The enclosing object is termed "decorator." It conforms to the interface of the component it decorates. It forwards requests to the component. It can perform additional operations before or after those forwarding requests. An unlimited number of responsibilities can be added with this concept.

Illustration

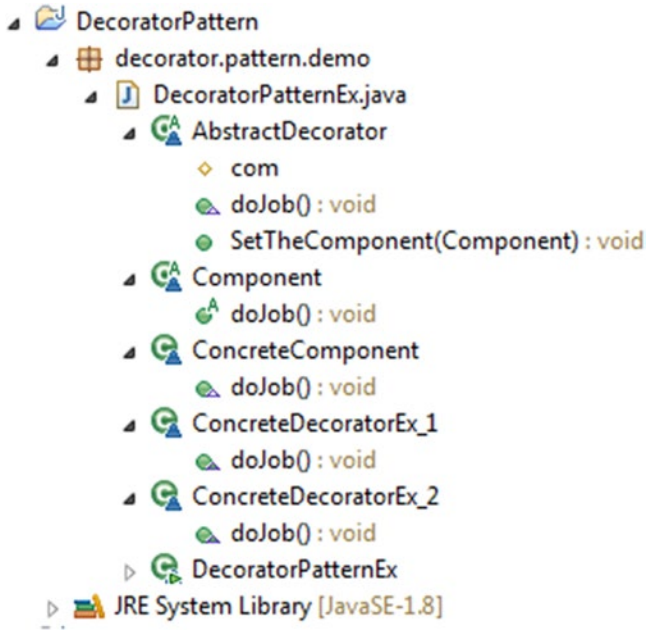
Please go through this example. Here we have not tried to modify the original doJob() method's functionality. Two decorators, ConcreteDecoratorEx_1 and ConcreteDecoratorEx_2, are added here to enhance functionality, but the original doJob()'s working is not disturbed due to this addition.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



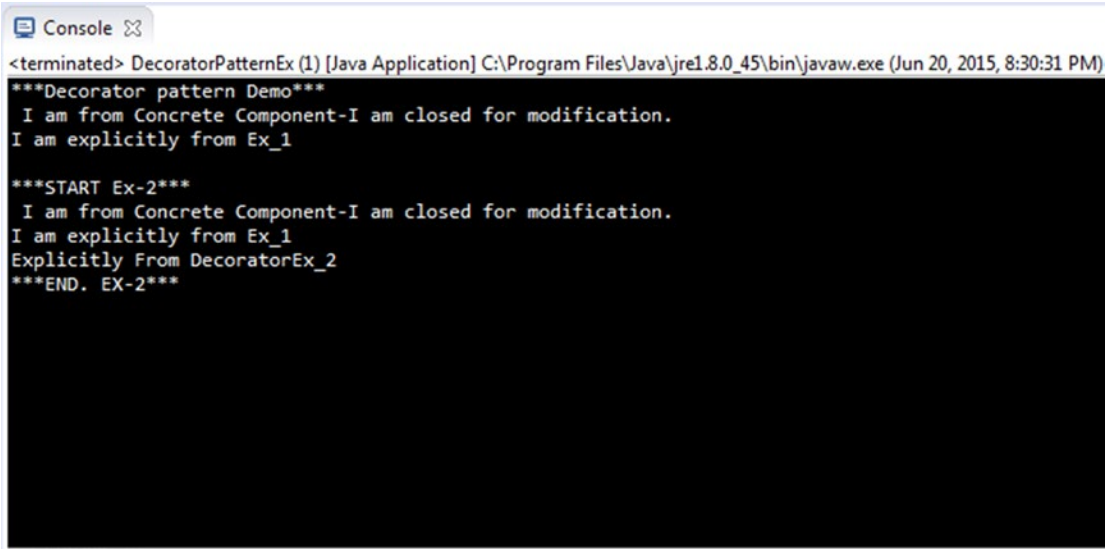
Implementation

```
package decorator.pattern.demo;

abstract class Component
    {
        public abstract void doJob();

    }
    class ConcreteComponent extends Component
    {
        @Override
          public void doJob()
        {
                System.out.println(" I am from Concrete Component-I am closed for
                modification.");

                }
    }
```

```
abstract class AbstractDecorator extends Component
{
    protected Component com ;
    public void SetTheComponent(Component c)
    {
        com = c;
    }
    public void doJob()
    {
        if (com != null)
        {
            com.doJob();
        }
    }

}
class ConcreteDecoratorEx_1 extends AbstractDecorator
{
   public void doJob()
    {
        super.doJob();
        //Add additional thing if necessary
        System.out.println("I am explicitly from Ex_1");
    }
}
class ConcreteDecoratorEx_2 extends AbstractDecorator
{
    public  void doJob()
    {
            System.out.println("");
            System.out.println("***START Ex-2***");
        super.doJob();
        //Add additional thing if necessary
        System.out.println("Explicitly From DecoratorEx_2");
        System.out.println("***END. EX-2***");
    }
}
class DecoratorPatternEx
{
    public static void main(String[] args)
    {
            System.out.println("***Decorator pattern Demo***");
        ConcreteComponent cc = new ConcreteComponent();

        ConcreteDecoratorEx_1 cd_1 = new ConcreteDecoratorEx_1();
        // Decorating ConcreteComponent Object //cc with ConcreteDecoratorEx_1
        cd_1.SetTheComponent(cc);
        cd_1.doJob();

        ConcreteDecoratorEx_2 cd_2= new ConcreteDecoratorEx_2();
        // Decorating ConcreteComponent Object //cc with ConcreteDecoratorEx_1 &
        //ConcreteDecoratorEX_2
```

```
            cd_2.SetTheComponent(cd_1);//Adding //results from cd_1 now
            cd_2.doJob();
        }
    }
```

Output



Note

What are the main advantages of using decorator patterns?

1. Without disturbing existing objects in the system, we can add new functionality to a particular object.
2. We can code incrementally. For example, we'll make a simple class first and then one by one we can add decorator objects to them as needed. As a result, we do not need to take care of each and every possible scenario in the beginning. On the other hand, we must acknowledge that making a complex class first and then extending its functionality is a much more complex procedure.

How is this pattern different from inheritance?

We can add or remove responsibilities by simply attaching or detaching decorators. But with the simple inheritance technique, we need to create a new class for new responsibilities. So, there will be many classes inside the system, which in turn can make the system complex.

What is the major disadvantage of using this pattern?

First of all, if we are careful enough, there is no significant disadvantage. But if we create too many decorators in the system, the system will be hard to maintain and debug. At the same time, the decorators can create unnecessary confusion.

CHAPTER 6

Template Method Patterns

GoF Definition: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. The template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Concept

In a template method, we define the minimum or essential structure of an algorithm. Then we defer some functionalities (responsibilities) to the subclasses. As a result, we can redefine certain steps of an algorithm by keeping the key structure fixed for that algorithm.

Real-Life Example

Suppose we want to make pizza. The basic mechanism is the same, but extra materials are added based upon the customer's choice—whether he/she wants a vegetarian pizza or a non-vegetarian pizza.

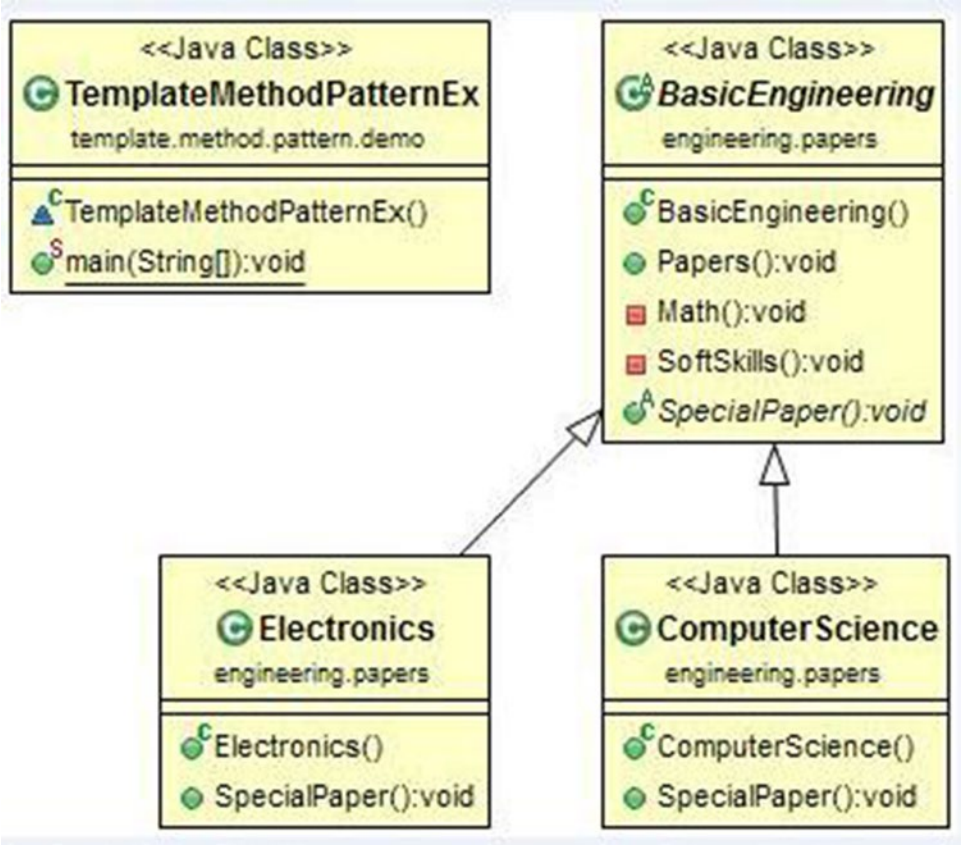
Computer World Example

For an engineering student, in general, most of the subjects in the first semester are common for all concentrations. Later, additional papers are added in his/her course based on his/her specialization (Computer Science, Electronics, etc.).

Illustration

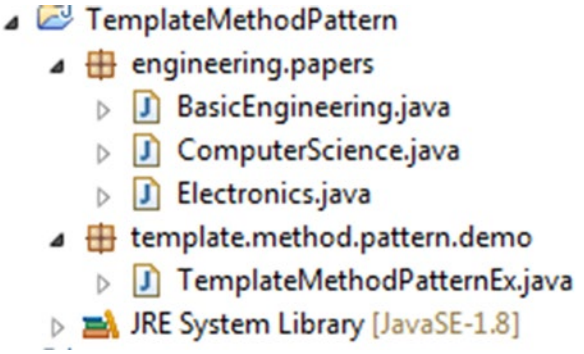
Here we have tried to implement our example with a similar concept. The parts of the program are described by the following structure in the solution. We have implemented a simple program to design engineering courses to illustrate the template method pattern. We are assuming that all engineering students need to pass mathematics and soft skills in their initial semesters to obtain their final degree. Later, some special papers will be added to their course based on their concentration (e.g., computer science or electronics).

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:

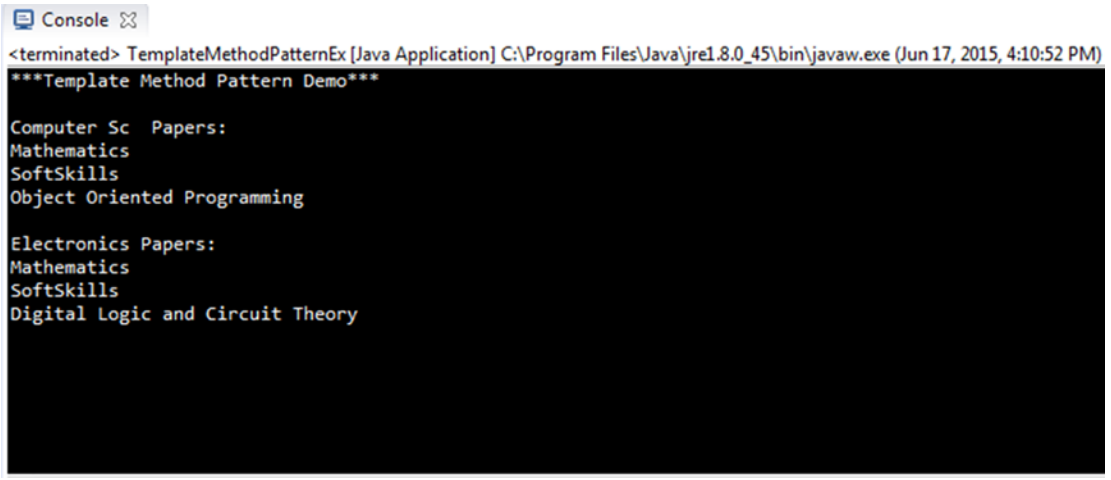


Implementation

```
// BasicEngineering.java

package engineering.papers;

public abstract class BasicEngineering
{
    // Papers() in the template method
    public void Papers()
    {
        //Common Papers:
        Math();
        SoftSkills();
        //Specialized Paper:
        SpecialPaper();
    }
    //Non-Abstract method Math(), SoftSkills() are //already implemented by Template class
    private void Math()
    {
        System.out.println("Mathematics");
    }
    private void SoftSkills()
    {
        System.out.println("SoftSkills");
    }
    //Abstract method SpecialPaper() must be implemented in derived classes
    public abstract void SpecialPaper();
}
```

```
// ComputerScience.java

package engineering.papers;

public class ComputerScience extends BasicEngineering
{
        @Override
        public void SpecialPaper()
    {
           System.out.println("Object Oriented Programming");
    }
}

// Electronics.java

package engineering.papers;

public class Electronics extends BasicEngineering
{
        @Override
    public void SpecialPaper()
    {
                System.out.println("Digital Logic and Circuit Theory");
    }
}

// TemplateMethodPatternEx.java
package template.method.pattern.Demo;
import engineeringPapers.*;

class TemplateMethodPatternEx
{
    public static void main(String[] args)
    {
        System.out.println("***Template Method Pattern Demo***\n");

        BasicEngineering bs = new ComputerScience();
        System.out.println("Computer Sc  Papers:");
        bs.Papers();
        System.out.println();
        bs = new Electronics();
        System.out.println("Electronics Papers:");
        bs.Papers();

    }
}
```

Output



Note

1. "Reuse of code" is the fundamental aim of this method. This is why, in general, we can see the use of this pattern in many class libraries.
2. GoF suggests that we explicitly decide which operations should be *hook operations* (we have freedom—we may or may not override the methods) and which operations should be *abstract operations* (we do not have a choice—overriding is a must in this case) during the development of a template method.

What is the major precaution we should take for implementing this method?

We need to minimize the number of incomplete/abstract operations. (In Java, remember: an abstract method does not have a body). Otherwise, each of the subclasses needs to override them and the overall process will lose the effectiveness of this design pattern.

CHAPTER 7

Strategy Patterns (Or, Policy Patterns)

GoF Definition: Define a family of algorithms, encapsulate each one, and make them interchangeable. The strategy pattern lets the algorithm vary independently from client to client.

Concept

We can select the behavior of an algorithm dynamically at runtime.

Real–Life Example

In a football match, at the last moment, in general, if Team A is leading Team B by a score of 1-0, instead of attacking, Team A becomes defensive. On the other hand, Team B goes for an all-out attack to score.

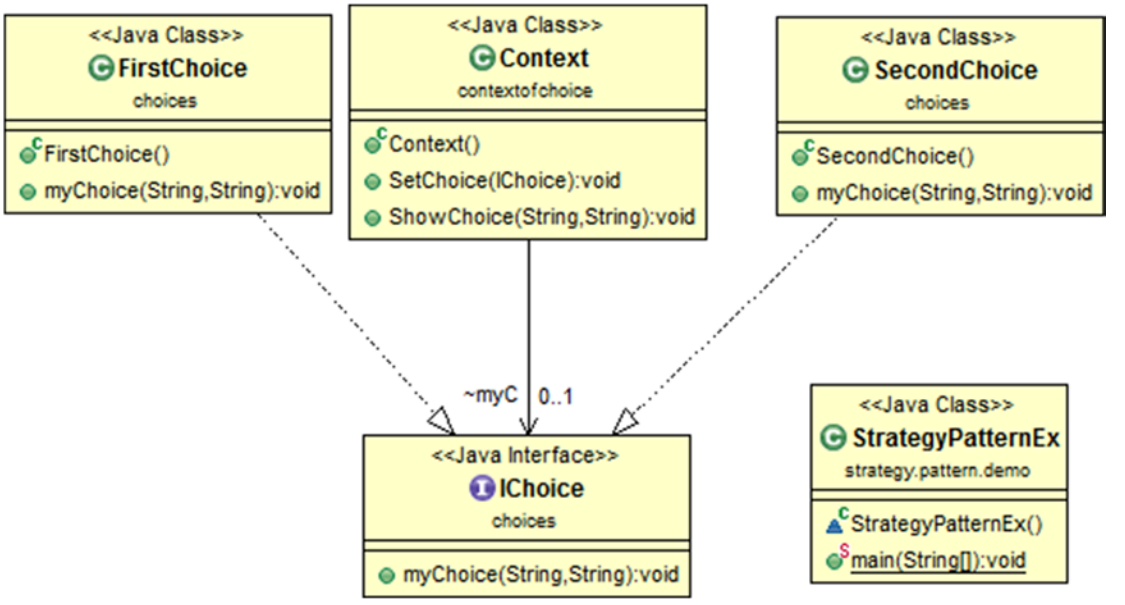
Computer World Example

The above concept is applicable to a computer football game also. Or, we can think of two dedicated memories where upon fulfillment of one memory, we start storing the data in the second available memory. So, a runtime check is necessary before the storing of data, and based on the situation, we'll proceed.

Illustration

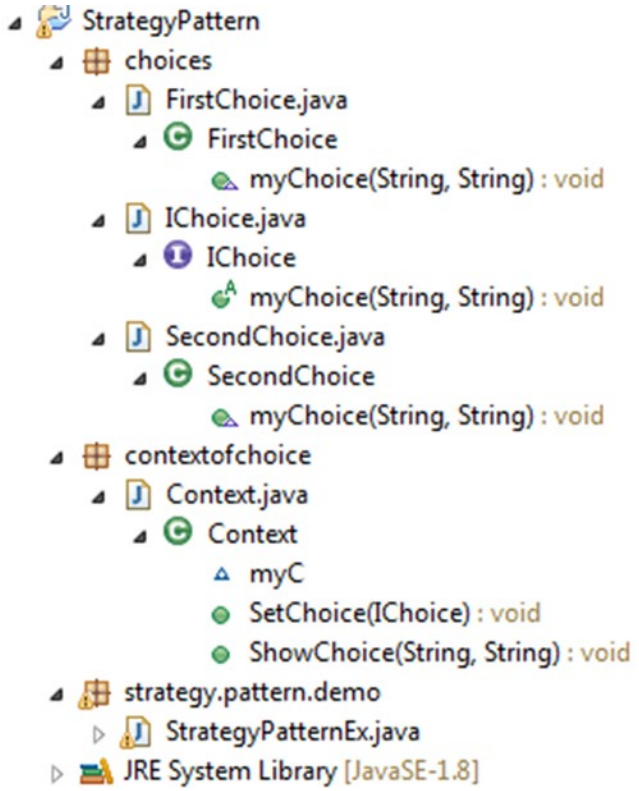
We can understand the strategy pattern through the following program and class diagram. In the main function, we have tested for two arbitrary choices of users (though you can do as many as wish). Depending on the user's input, our context objects will decide what choice should be set and displayed.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



Implementation

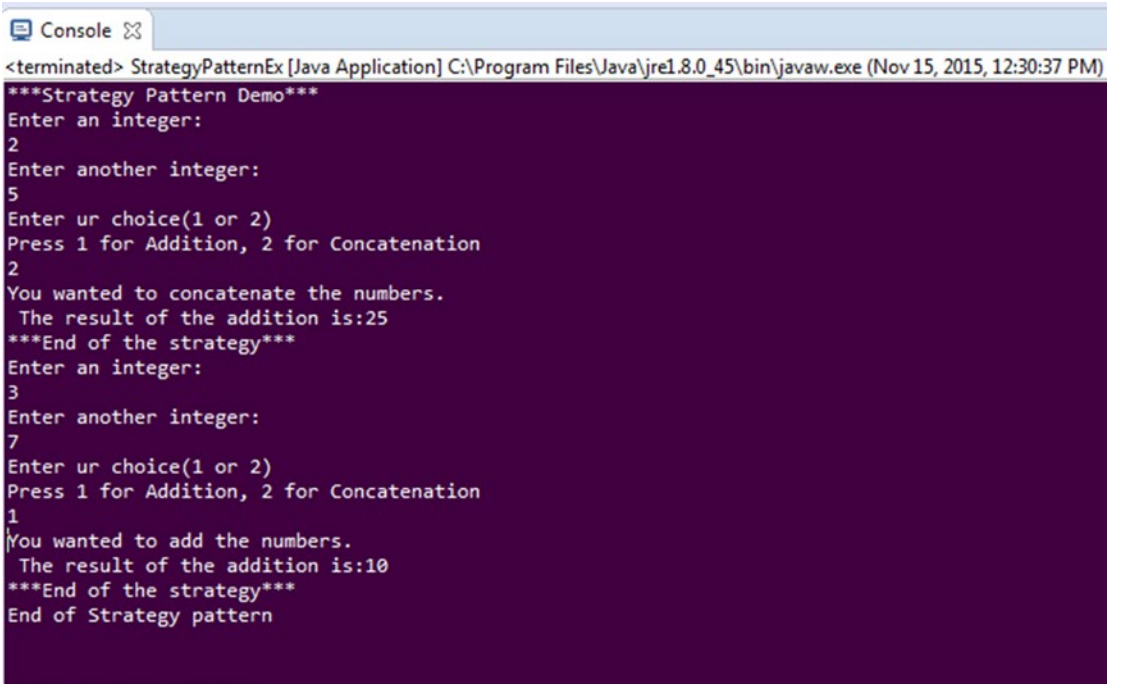
```
// IChoice.java

package choices;
public interface IChoice
{
        void myChoice(String s1, String s2);
}

//FirstChoice.java

package choices;
```

```
public class FirstChoice implements IChoice
{
        public void myChoice(String s1, String s2)
        {
                System.out.println("You wanted to add the numbers.");
                int int1, int2,sum;
                int1=Integer.parseInt(s1);
                int2=Integer.parseInt(s2);
                sum=int1+int2;
                System.out.println(" The result of the addition is:"+sum);
                System.out.println("***End of the strategy***");
        }
}
// SecondChoice.java

package choices;

public class SecondChoice implements IChoice
{
        public void myChoice(String s1, String s2)
        {
                System.out.println("You wanted to concatenate the numbers.");
                System.out.println(" The result of the addition is:"+s1+s2);
                System.out.println("***End of the strategy***");
        }
}

//Context.java

package contextofchoice;

import choices.IChoice;

public class Context
{
        IChoice myC;
        // Set a Strategy or algorithm in the Context
        public void SetChoice(IChoice ic)
        {
                myC = ic;
        }
        public void ShowChoice(String s1,String s2)
        {
                myC.myChoice(s1,s2);
        }
}
```

```
// StrategyPatternEx.java

package strategy.pattern.demo;
import java.io.IOException;
//For Java old versions-to take input from user
//import java.io.BufferedReader;
//import java.io.InputStreamReader;
/* Java 5  added a nice utility class called Scanner, to get input from user */
import java.util.Scanner;

import contextofchoice.Context;
//import choices.*;
import choices.FirstChoice;
import choices.IChoice;
import choices.SecondChoice;

class StrategyPatternEx
{
        public static void main(String[] args) throws IOException
        {
                System.out.println("***Strategy Pattern Demo***");
                Scanner in= new Scanner(System.in);//To take input from user
                IChoice ic = null;
                Context cxt = new Context();
                String input1,input2;;
                //Looping twice to test two different choices
                try
                {
                        for (int i = 1; i <= 2; i++)
                        {
                                System.out.println("Enter an integer:");
                                input1 = in.nextLine();
                                System.out.println("Enter another integer:");
                                input2 = in.nextLine();
                                System.out.println("Enter ur choice(1 or 2)");
                                System.out.println("Press 1 for Addition,
                                2 for Concatenation");
                                String c = in.nextLine();

                                /*For Java old versions-use these lines to collect input
                                from user
                                BufferedReader br = new BufferedReader
                                ( new InputStreamReader( System.in ));
                                String c = br.readLine();*/

                                if (c.equals("1"))
                                {
                                        /*If user input is 1, create object of FirstChoice
                                        (Strategy 1)*/
                                        ic = new FirstChoice();
                                }
```

```
                                else
                                {
                                        /*If user input is 2, create object of SecondChoice
                                        (Strategy 2)*/
                                        ic = new SecondChoice();
                                }
                                /*Associate the Strategy with Context*/
                                cxt.SetChoice(ic);
                                cxt.ShowChoice(input1,input2);
                        }
                }
                finally
                {
                        in.close();
                }
                System.out.println("End of Strategy pattern");
        }
}
```

Output



Note

What is the power behind the strategy pattern?

1. This pattern can provide dynamic behavior for us. It can help us to avoid dealing with complex algorithm-specific data structures.
2. With this pattern, the same behavior can be expressed differently. So, users can have a wide variety of choices.

What are the challenges associated with the strategy pattern?

1. The number of objects are increased in the system.
2. Additional overhead is needed due to communication between the strategies and their contexts.
3. Users need to be fully aware of all kinds of possible behaviors to avoid confusion.

CHAPTER 8

Adapter Patterns

GoF Definition: Convert the interface of a class into another interface that clients expect. The adapter pattern lets classes work together that couldn't otherwise because of incompatible interfaces.

Concept

The core concept is best described by the examples given below.

Real–Life Example

The most common example of this type can be found with mobile charging devices. If our charger is not supported by a particular kind of switchboard, we need to use an adapter. Even the translator who is translating language for one person is following this pattern in real life.

Computer World Example

In real-life development, in many cases, we cannot communicate between two interfaces directly. They contain some kind of constraint within themselves. To deal with this kind of incompatibility between those interfaces, we may need to introduce adapters.

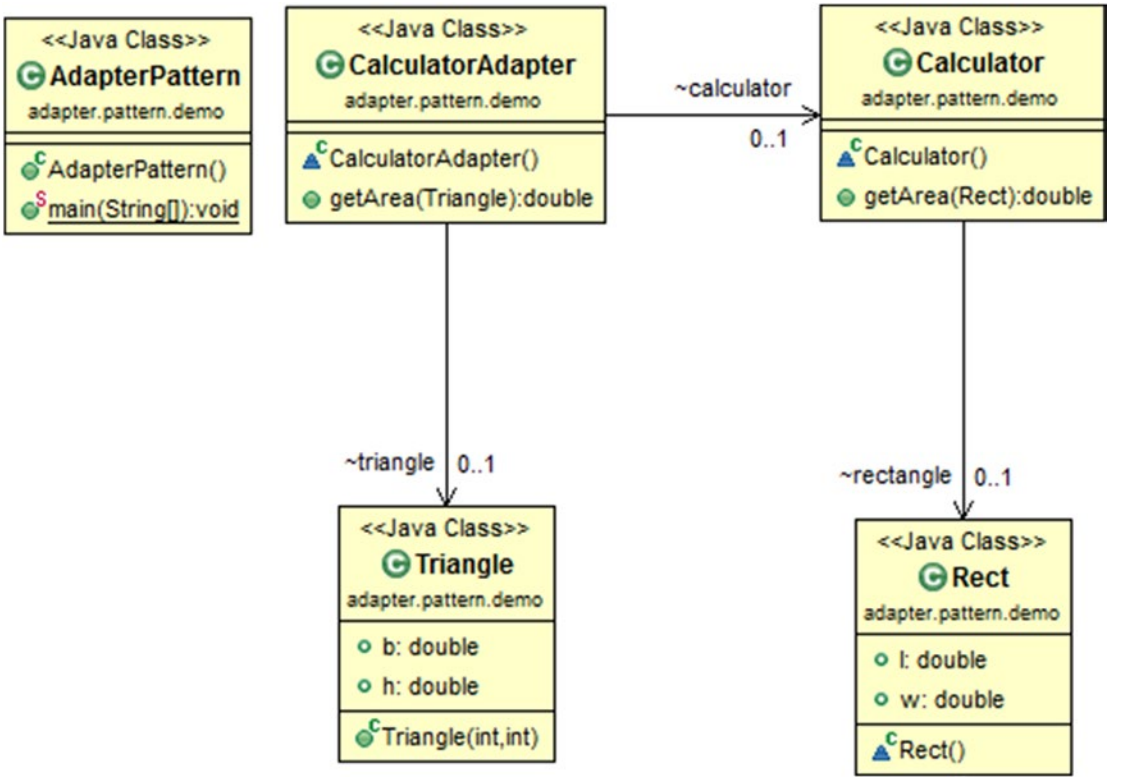
A very common use of this pattern is illustrated below.

Illustration

In this example, we can calculate the area of a rectangle easily. If we see the Calculator class and its getArea() method, we'll know that we need to supply a rectangle as an input in the getArea() method to get the area of the rectangle. Now suppose we want to calculate the area of a triangle, but we need to get the area of the triangle through the getArea() method of Calculator. How can we do that?

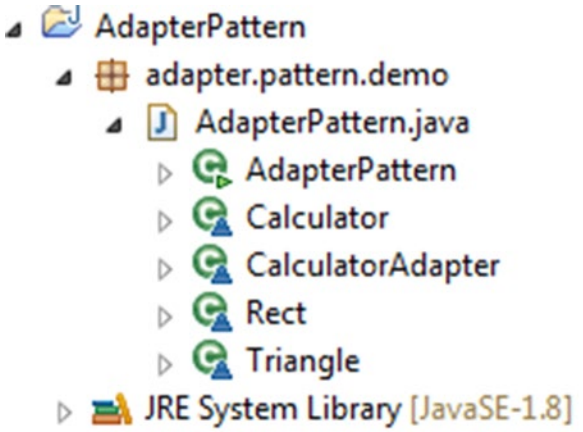
To do that we have made a CalculatorAdapter for the triangle and passed a triangle in its getArea() method. The method will translate the triangle input to rectangle input and in turn, it will call the getArea() of Calculator to get the area of it. *But from the user's point of view, it is very simple: it seems to the user that he/she is passing a triangle to get the area of that triangle.*

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:

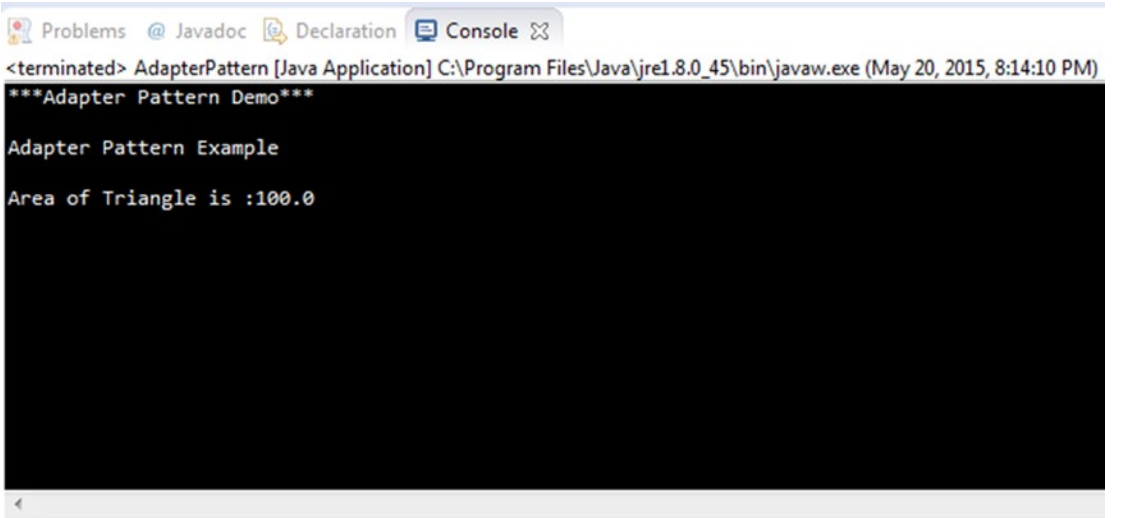


Implementation

```
package adapter.pattern.demo;
class Rect
    {
        public double l;
        public double w;
    }
class Triangle
{
    public double b;//base
    public double h;//height
    public Triangle(int b, int h)
    {
        this.b = b;
        this.h = h;
    }
}
/*Calculator can calculate the area of a rectangle. To calculate the area we need a
Rectangle input.*/
class Calculator
{   Rect rectangle;
        public double getArea(Rect r)
        {
                rectangle=r;
                return rectangle.l * rectangle.w;
        }
}
```

```
/*Calculate the area of a Triangle using CalculatorAdapter. Please note here: this time our
input is a Triangle.*/
class CalculatorAdapter
{
        Calculator calculator;
        Triangle triangle;
        public double getArea(Triangle t)
        {
                calculator = new Calculator();
                triangle=t;
                Rect r = new Rect();
                //Area of Triangle=0.5*base*height
                r.l = triangle.b;
                r.w = 0.5*triangle.h;
                return calculator.getArea(r);
        }

}

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

Output



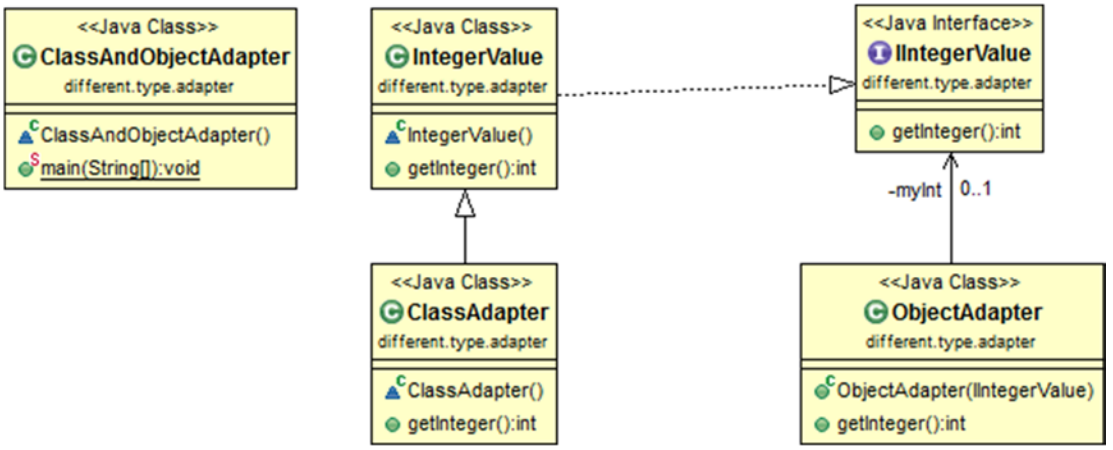
Note

GoF tells us about two major kinds of adapters:

A. Class adapters. They generally use multiple inheritance to adapt one interface to another. (But we must remember, in Java, multiple inheritance through classes is not supported. We need interfaces to implement the concept of multiple inheritance.)

B. Object adapters. They depend on the object compositions.

To illustrate the concepts, I'll present a simple example for your ready reference:

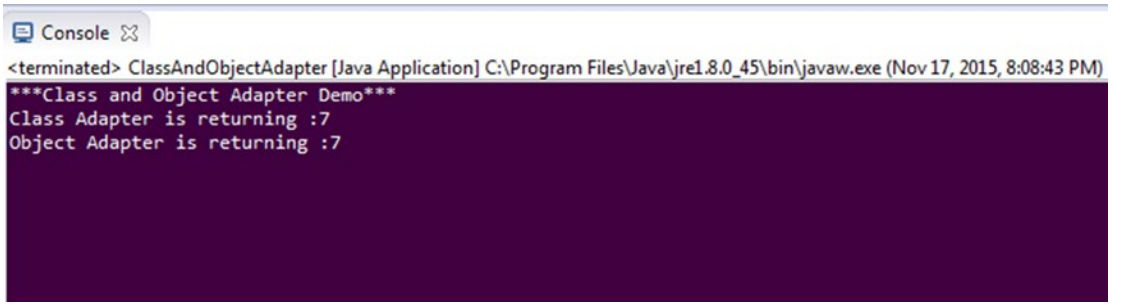


Illustration

```
package different.type.adapter;
interface IIntegerValue
{
        public int getInteger();

}
class IntegerValue implements IIntegerValue
{
        @Override
        public int getInteger()
        {
                return 5;
        }
}
class ClassAdapter extends IntegerValue
{
        //Incrementing by 2
        public int getInteger()
        {
                return 2+super.getInteger();
        }
}
```

```
class ObjectAdapter
{
        private IIntegerValue myInt;
        public ObjectAdapter(IIntegerValue myInt)
        {
                this.myInt=myInt;
        }
        //Incrementing by 2
        public int getInteger()
        {
                return 2+this.myInt.getInteger();
        }

}

class ClassAndObjectAdapter
{
        public static void main(String args[])
        {
                System.out.println("***Class and Object Adapter Demo***");
                ClassAdapter ca1=new ClassAdapter();
                System.out.println("Class Adapter is returning :"+ca1.getInteger());

                ClassAdapter ca2=new ClassAdapter();
                ObjectAdapter oa=new ObjectAdapter(new IntegerValue());
                System.out.println("Object Adapter is returning :"+oa.getInteger());
        }
}
```

Output



To what extent does an adapter need to take care when it adapts an adaptee?

It depends on that particular case. If our target interface is very similar, then adapters do not have much work. If there is not much similarity, then adapters must do some extra work to match those functionalities.

CHAPTER 9

Command Patterns

GoF Definition: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Concept

Here requests are encapsulated as objects. *In general, four terms are associated—invoker, client, command, and receiver*. A *command object* is capable of calling a particular method in the receiver. It stores the parameters of the methods in *receiver*. An *invoker* only knows about the command interface, but it is totally unware about the concrete commands. The *client* object holds the invoker object and the command object(s). The client decides which of these commands needs to execute at a particular point in time. To do that, he/she passes the command object to the invoker to execute that particular command.

Real-Life Example

We cannot change our past, but frequently we wish we could do so. Unfortunately, we do not have any such device yet to fulfill that wish. But we can undo and redo many other operations in our daily life. We can erase a pencil drawing with a rubber. We can re-architect our living places. And, most important, we can forget bad memories and start a fresh journey. So, you must acknowledge that undo/redo operations are part of our life and we are doing those through some commands—either externally or internally.

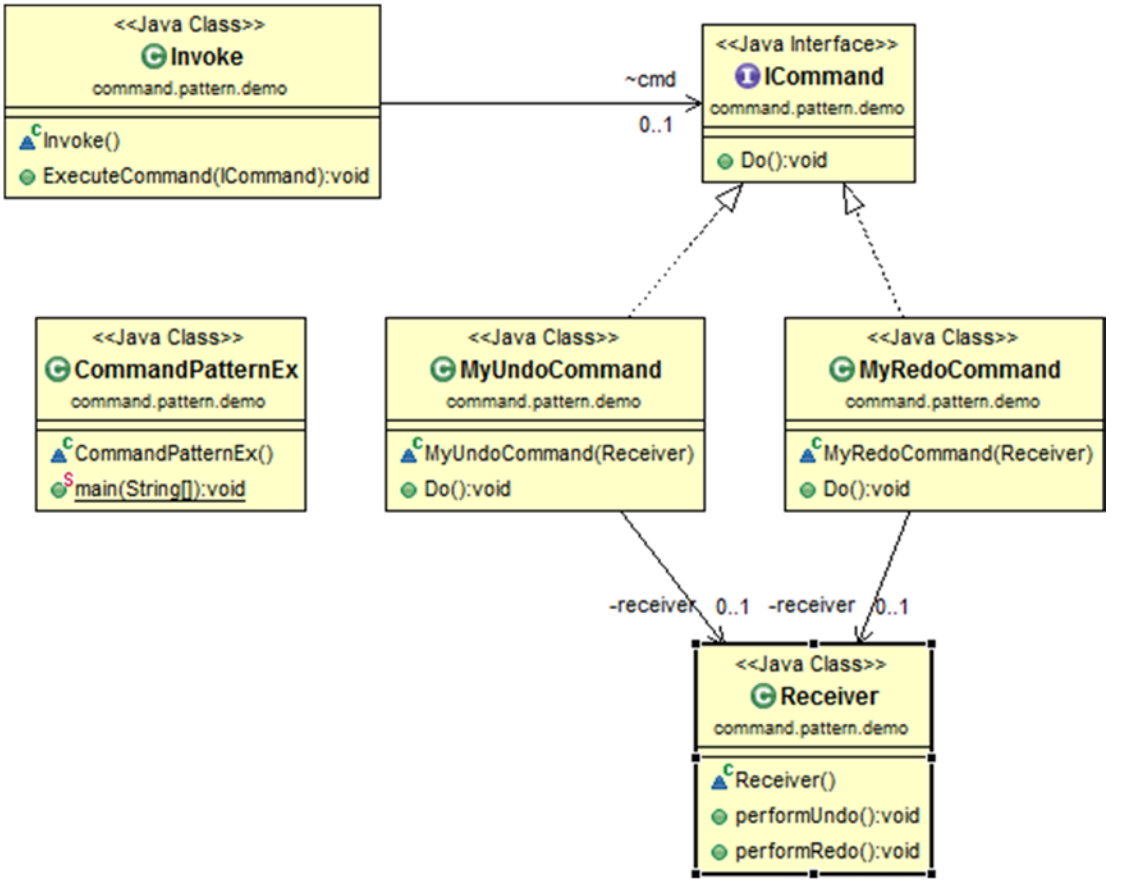
Computer World Example

The above scenario applies with Microsoft paint also. There we can do the undo/redo operations easily through some menu options or shortcut keys.

Illustration

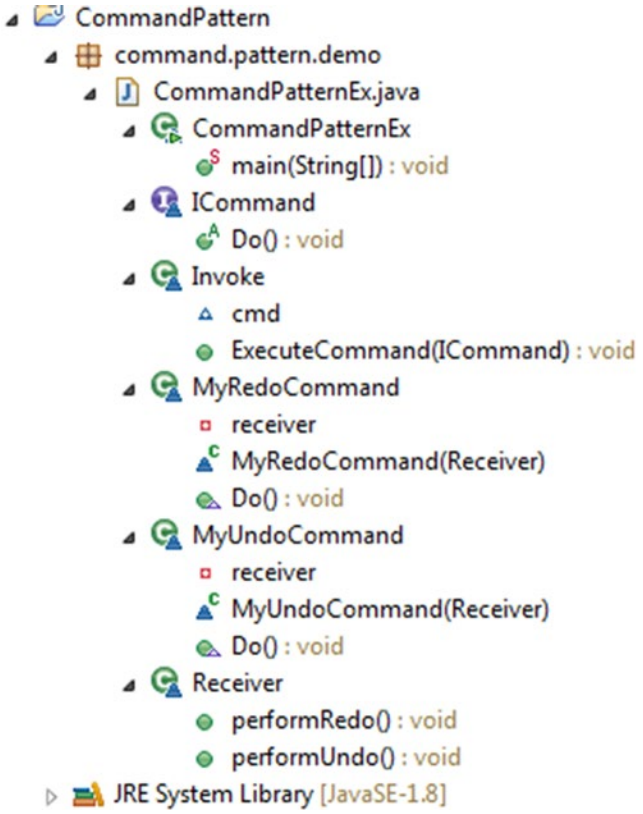
Consider this simple example in this context. For simplicity, we are dealing with only two commands, MyUndoCommand and MyRedoCommand. All naming conventions are used for your easy reference.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



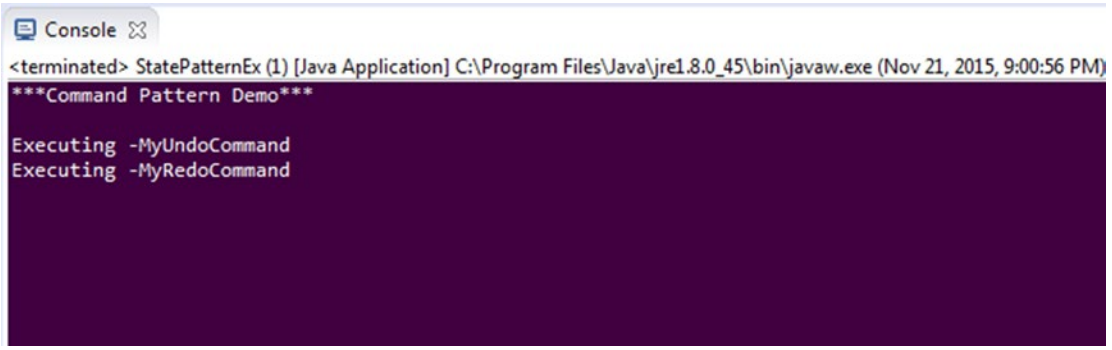
Implementation

```
package command.pattern.demo;

interface ICommand
{
        void Do();
}
class MyUndoCommand implements ICommand
{
        private Receiver receiver;
        MyUndoCommand(Receiver recv)
        {
                receiver=recv;
        }
```

```
        @Override
        public void Do()
        {
                //Call undo in receiver
                receiver.performUndo();
        }
}
class MyRedoCommand implements ICommand
{
        private Receiver receiver;
        MyRedoCommand(Receiver recv)
        {
         receiver=recv;
        }
        @Override
        public void Do()
        {
                //Call redo in receiver
                receiver.performRedo();
        }
}
//Receiver Class
class Receiver
{
        public void performUndo()
        {
                System.out.println("Executing -MyUndoCommand");
        }
        public void performRedo()
        {
                System.out.println("Executing -MyRedoCommand");
        }
}
//Invoker Class
class Invoke
{
        ICommand cmd;
        public void ExecuteCommand(ICommand cmd)
        {
                this.cmd=cmd;
                cmd.Do();
        }
}
```

```
//Client Class
class CommandPatternEx
{
        public static void main(String[] args)
        {
                System.out.println("***Command Pattern Demo***\n");
                Receiver intendedreceiver=new Receiver();
                /*Client holds Invoker and Command Objects*/
                Invoke inv = new Invoke();
                MyUndoCommand unCmd = new MyUndoCommand(intendedreceiver);
                MyRedoCommand reCmd = new MyRedoCommand(intendedreceiver);
                inv.ExecuteCommand(unCmd);
                inv.ExecuteCommand(reCmd);
        }
}
```

Output



Note

1. This pattern is widely used for undo/redo operations.
2. A callback function can be designed with this pattern.
3. This pattern is very useful when we model transactions (which can be responsible for changes in data).
4. Commands can be extended easily. They operate like any other objects. And the beauty of using them is that while we use them, we do not need to change the classes in the system.
5. *There is another pattern called chain of responsibility. There we forward a request along a chain of objects with the hope that any one of the objects along that chain will handle the request. But in command pattern, we'll forward the request to the specific object.*

CHAPTER 10

Iterator Patterns

GoF Definition: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Concept

Iterators are generally used to traverse a container to access its elements.

Real-Life Example

Suppose there are two companies: Company A and Company B. Company A stores its employee records (name, etc.) in a linked list and Company B stores its employee data in a big array. One day the two companies decide to work together. The iterator pattern is handy in such a situation. We need not write codes from scratch. We'll have a common interface through which we can access data for both companies. We'll simply call the same methods without rewriting the codes.

Computer World Example

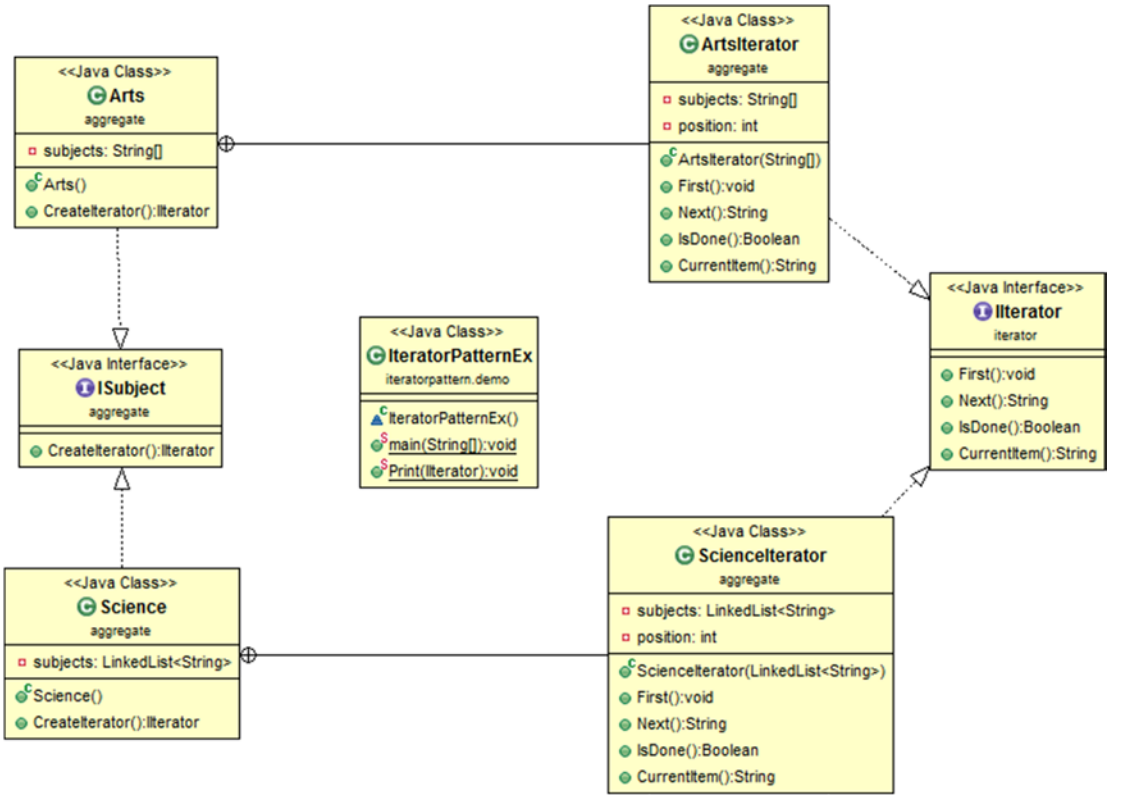
Similarly, say, in a college, the arts department may use array data structure and the science department may use linked list data structure to store their students' records. The main administrative department will access those data through the common methods—it doesn't care which data structure is used by individual departments.

Illustration

It has many parts. I have created the related folders for these. So, first see the structure. Here science subjects are stored in a linked list and arts subjects are stored in an array. We are printing the papers using the iterators. IIterator is the common interface here. Its methods are specifically implemented in ScienceIterator (contained in science class) and ArtsIterator (contained in arts class). We are printing the papers through the common methods IsDone() and Next() here. (You can use the other two methods also, namely, First() and currentItem()—those are also implemented.)

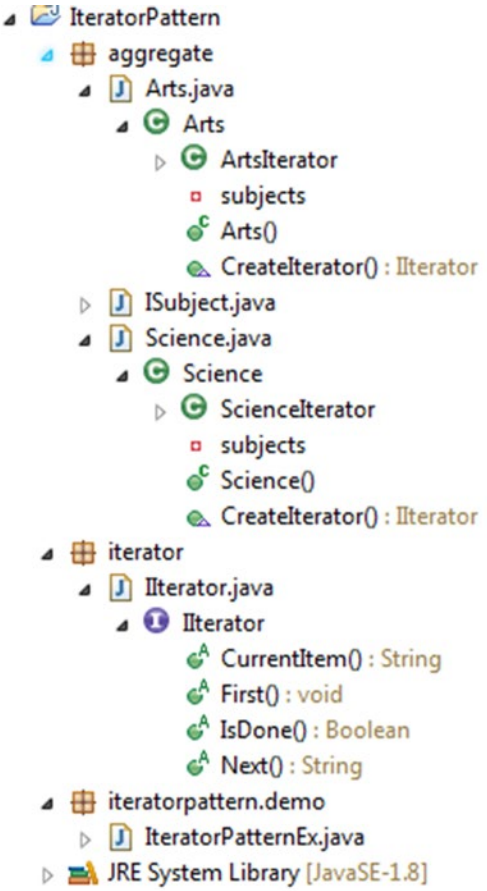
For simplicity, you can omit either Science(and ScienceIterator) or Arts(and ArtsIterator). But we kept both to show you the power of this pattern.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



Implementation

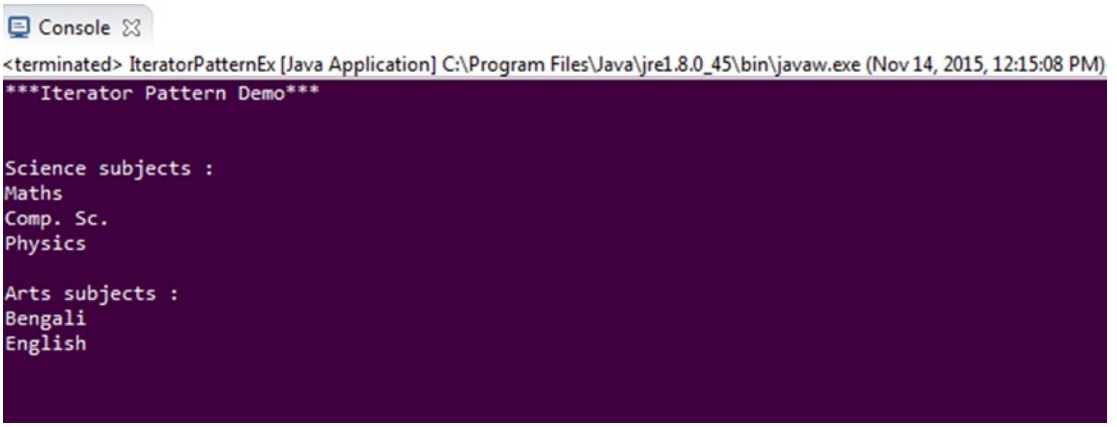
```
//ISubject.java

package aggregate;
import iterator.*;

public interface ISubject
{
        public IIterator CreateIterator();
}

//IIterator.java

package iterator;
```

```
public interface IIterator
{
         void First();//Reset to first element
     String Next();//get next element
     Boolean IsDone();//End of collection check
     String CurrentItem();//Retrieve Current Item
}
//Arts.java

package aggregate;
import iterator.*;

public class Arts implements ISubject
{
        private String[] subjects;

        public Arts()
        {
                subjects = new String[2];
                subjects[0] = "Bengali";
                subjects[1] = "English" ;
        }

        public IIterator CreateIterator()
        {
                return new ArtsIterator(subjects);
        }
        //Containing the ArtsIterator
        public class ArtsIterator implements IIterator
        {
                private String[] subjects;
                private int position;
                public ArtsIterator(String[] subjects)
                {
                        this.subjects = subjects;
                        position = 0;
                }
                public void First()
                {
                        position = 0;
                }

                public String Next()
                {
                        return subjects[position++];
                }

                public Boolean IsDone()
                {
                        return position >= subjects.length;
                }
```

```
                public String CurrentItem()
                {
                        return subjects[position];
                }
        }

}

// Science.java

package aggregate;
//for Linked List data structure used here

import java.util.LinkedList;
import iterator.*;

public class Science implements ISubject
{
        private LinkedList<String> subjects;

        public Science()
        {
                subjects = new LinkedList<String>();
                subjects.addLast("Maths");
                subjects.addLast("Comp. Sc.");
                subjects.addLast("Physics");
        }
        @Override
        public IIterator CreateIterator()
        {
                return new ScienceIterator(subjects);
        }
        //Containing the ScienceIterator
        public class ScienceIterator implements IIterator
        {
                private LinkedList<String> subjects;
                private int position;
                public ScienceIterator(LinkedList<String> subjects)
                {
                        this.subjects = subjects;
                        position = 0;
                }

                public void First()
                {
                        position = 0;
                }
```

```
                public String Next()
                {
                        return subjects.get(position++);
                }

                public Boolean IsDone()
                {
                        return position >= subjects.size();
                }

                public String CurrentItem()
                {
                        return subjects.get(position);
                }
        }
}

//IteratorPatternEx.java

package iteratorpattern.demo;
import iterator.*;
import aggregate.*;

class IteratorPatternEx
{
        public static void main(String[] args)
        {
                System.out.println("***Iterator Pattern Demo***\n");
                ISubject Sc_subject = new Science();
                ISubject Ar_subjects = new Arts();

                IIterator Sc_iterator = Sc_subject.CreateIterator();
                IIterator Ar_iterator = Ar_subjects.CreateIterator();

                System.out.println("\nScience subjects :");
                Print(Sc_iterator);

                System.out.println("\nArts subjects :");
                Print(Ar_iterator);
        }

        public static void Print(IIterator iterator)
        {
                while (!iterator.IsDone())
                {
                        System.out.println(iterator.Next());
                }

        }
}
```

Output



Note

If you have gone through the above code, probably you now have a fair idea of the power of an iterator and the significance of this pattern. We can support different variations for the traversal of an *aggregate* (the interface to create an Iterator object), and above all, it simplifies the interface.

But we must be careful while traversing and *any kind of modification during that traversal period can cause damage to us*. We can take a backup first to deal with this type of scenario, but it is obvious that taking the backup and reexamining it at some later stage is also a costly operation.

CHAPTER 11

Facade Patterns

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

Concept

It is one of those patterns that supports loose coupling. Here we emphasize the abstraction and hide the complex details by exposing a simple interface.

Real-Life Example

Suppose you are going to organize a birthday party and you have invited 100 people. Nowadays, you can go to any party organizer and let him/her know the minimum information— (party type, date and time of the party, number of attendees, etc.). The organizer will do the rest for you. You do not even think about how he/she will decorate the party room, whether people will take food from self-help counter or will be served by a caterer, and so on.

Computer World Example

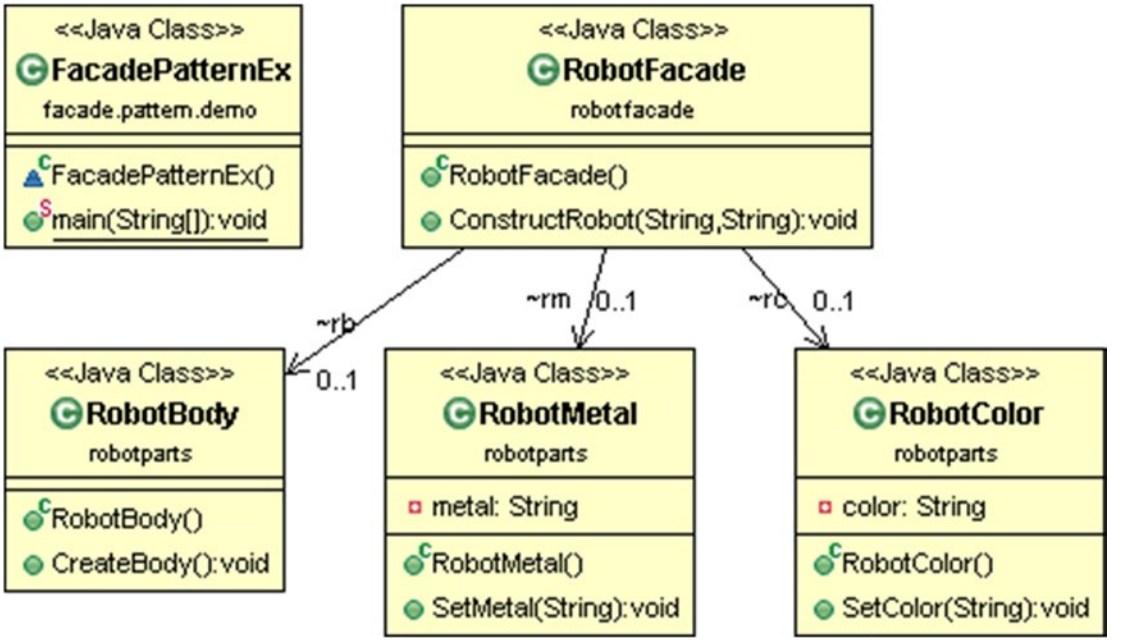
We can think about a case where we use a method from a library. The user doesn't care how the method is implemented in the library. He/she just calls the method to serve his/her easy purpose. The pattern can be best described by the example that follows.

Illustration

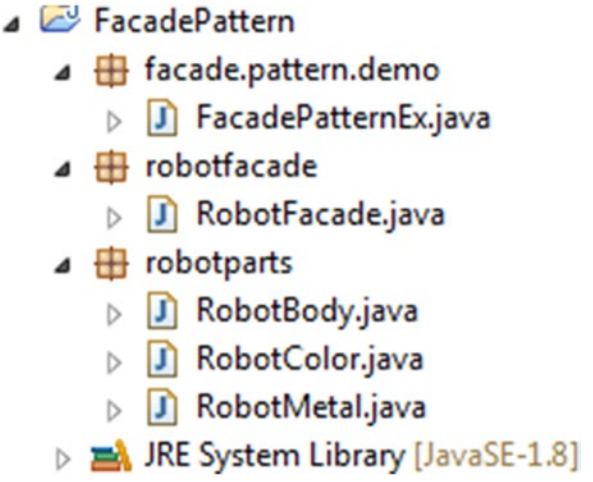
Here our aim is to build/construct robots. And from a user point of view he/she needs to supply only the color and material for his/her robot through the RobotFacade (See our FacadePatternEx.java file.) Our RobotFacade (RobotFacade.java) will in turn create objects for RobotBody, RobotColor, RobotMetal and will do the rest for the user. *We need not worry about the creation of these separate classes and their calling sequence*. All of the classes have their corresponding implementation here.

In this example I have followed this structure. Note that the related parts are separated by the packages for better readability.

UML Class Diagram



Package Explorer view

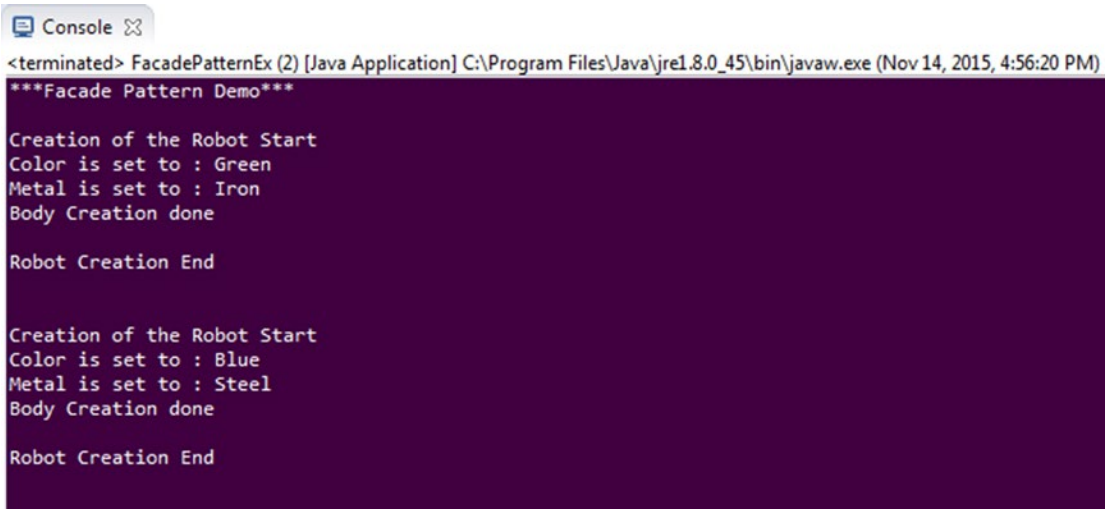


Implementation

```
//RobotBody.java
package robotparts;

public class RobotBody
{
        public void CreateBody()
        {
                System.out.println("Body Creation done");
        }
}

//RobotColor.java
package robotparts;

public class RobotColor
{
        private String color;
        public void SetColor(String color)
        {
                this.color = color;
                System.out.println("Color is set to : "+ this.color);
        }
}

//RobotMetal.java
package robotparts;

public class RobotMetal
{
        private String metal;
        public void SetMetal(String metal)
        {
                this.metal=metal;
                System.out.println("Metal is set to : "+this.metal);
        }
}
```

```
//RobotFacade.java
package robotfacade;
import robotparts.*;

public class RobotFacade
{
        RobotColor rc;
        RobotMetal rm ;
        RobotBody rb;
        public RobotFacade()
        {
                rc = new RobotColor();
                rm = new RobotMetal();
                rb = new RobotBody();

        }
        public void ConstructRobot(String color,String metal)
        {
                System.out.println("\nCreation of the Robot Start");
                rc.SetColor(color);
                rm.SetMetal(metal);
                rb.CreateBody();
                System.out.println(" \nRobot Creation End");
                System.out.println();
        }
}

//FacadePatternEx.java
package facade.pattern.demo;
import robotfacade.RobotFacade;

class FacadePatternEx
{
        public static void main(String[] args)
        {
                System.out.println("***Facade Pattern Demo***");
                RobotFacade rf1 = new RobotFacade();
                rf1.ConstructRobot("Green", "Iron");
                RobotFacade rf2 = new RobotFacade();
                rf2.ConstructRobot("Blue", "Steel");

        }
}
```

Output



Note

1. We use Facade pattern to represent a simple interface instead of a complex subsystem.
2. Here we promote weak coupling among subsystems—so, in this way, we are making them portable.
3. We already mentioned that we separate subsystems from clients by a simple interface. With this model, we not only make the system easier to use but also reduce the number of objects that the clients need to deal with.
4. There is truly no major disadvantage associated with this pattern. On the other hand, it has proven its usefulness in libraries like jQuery also.

CHAPTER 12

Factory Method Patterns

GoF Definition: Define an interface for creating an object, but let subclasses decide which class to instantiate. The factory method lets a class defer instantiation to subclasses.

Concept

The concept can be best described with the examples below.

Real-Life Example

Suppose you have two different types of televisions—one with an LED screen and another with an LCD screen. If any of these starts malfunctioning, you will call a TV repairman to request a visit to your residence. The repairman must ask first what kind of TV is nonoperational. As per your input, he'll carry the required instruments with him.

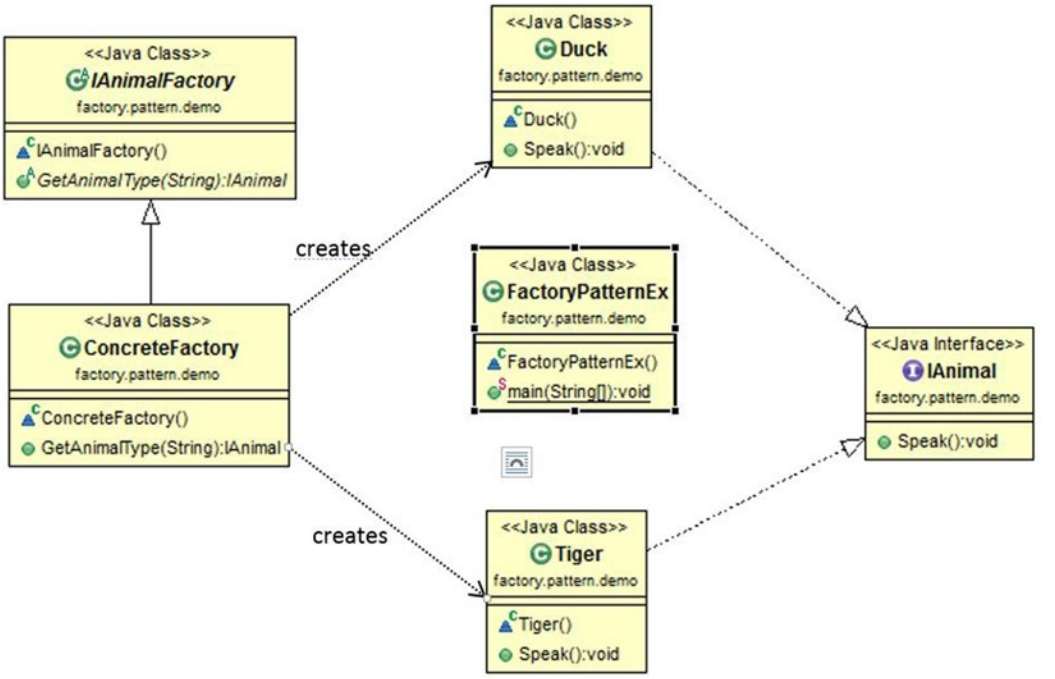
Computer World Example

In a Windows application, we may have different database users (e.g., one user uses Oracle and one may use Sql Server). Now whenever we need to insert data in our database we need to create either an SqlConnection or an OracleConnection first; only then we can proceed. If we put them into simple if-else, we need to repeat lots of codes and it doesn't look good. We can use the factory pattern to solve these types of problems. The basic structure is defined with an abstract class; our subclasses will be derived from this class. The subclasses will take the responsibility of the instantiation process.

Illustration

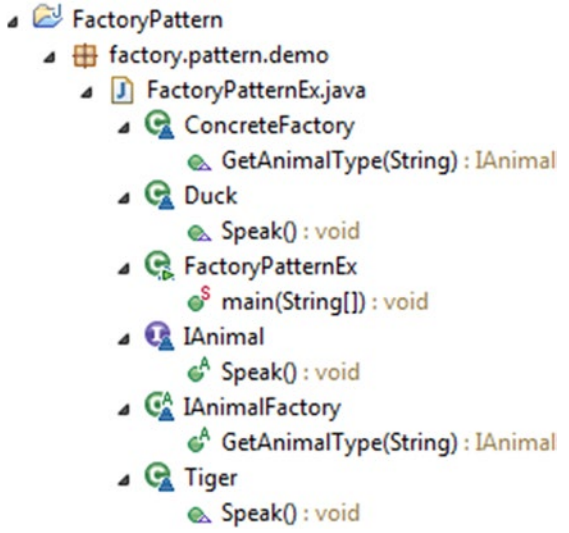
Here I have created all classes into a single file. Please go through the code. Note that here we have two animal types—Duck and Tiger. *And whenever we try to use a new type, Lion, which was not implemented earlier, an exception will be thrown. If you want to omit this extra part, you can do that. In that case, we also do not need to take precautions for any exceptions.*

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



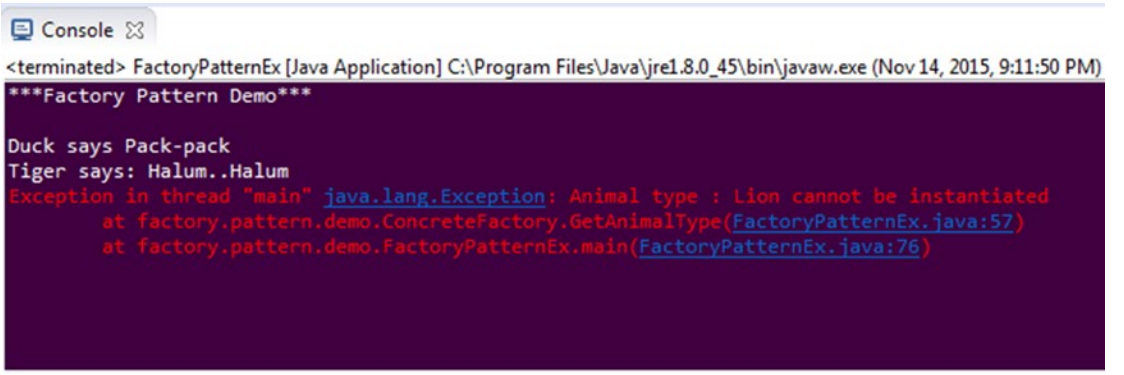
Implementation

```
package factory.pattern.demo;
interface IAnimal
{
        void Speak();
}
class Duck implements IAnimal
{
        @Override
        public void Speak()
        {
                System.out.println("Duck says Pack-pack");
        }
}
class Tiger implements IAnimal
{
        @Override
        public void Speak()
        {
                System.out.println("Tiger says: Halum..Halum");

        }
}
abstract class IAnimalFactory
{
        public abstract IAnimal
        /*if we cannot instantiate in later stage, we'll throw exception*/
        GetAnimalType(String type) throws Exception;
}
class ConcreteFactory extends IAnimalFactory
{

        @Override
        public  IAnimal GetAnimalType(String type) throws Exception
        {
                switch (type)
                {
                case "Duck":
                    return new Duck();
                case "Tiger":

                    return new Tiger();
                default:
                        throw new Exception( "Animal type : "+type+" cannot be
                        instantiated");
                }
        }
}
```

```
class FactoryPatternEx
{
        public static void main(String[] args) throws Exception
        {
                System.out.println("***Factory Pattern Demo***\n");
                IAnimalFactory animalFactory = new ConcreteFactory();
                IAnimal DuckType=animalFactory.GetAnimalType("Duck");
                DuckType.Speak();

                IAnimal TigerType = animalFactory.GetAnimalType("Tiger");
                TigerType.Speak();
                //There is no Lion type. So, an exception will be thrown
                IAnimal LionType = animalFactory.GetAnimalType("Lion");
                LionType.Speak();

        }
}
```

Output

If you do not want to see the exception, just do not try to create the Lion type here (i.e., comment out the code).



Note

1. This pattern is useful when our classes shift the responsibilities of objects creation to its subclasses.
2. This pattern is also useful when implementing parallel class hierarchies (when some of the responsibilities shift from one class to another) and when making a system with loose coupling is possible.
3. One issue that we need to address is that making too many objects often can decrease performance.

CHAPTER 13

Memento Patterns

GoF Definition: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Concept

Our aim is to save the state of an object, so that in the future, we can go back to the specified state. Three objects are playing the game here—a memento, a caretaker, and the originator. The memento will store the internal states of the originator. The originator can have the internal states and it has the ability to restore into its earlier state. An originator can also retrieve information from the memento. The caretaker takes care of the memento's safekeeping or protection and it should not examine the contents of the memento.

Real-Life Example

In notepad we use undo frequently by pressing ctrl+Z.

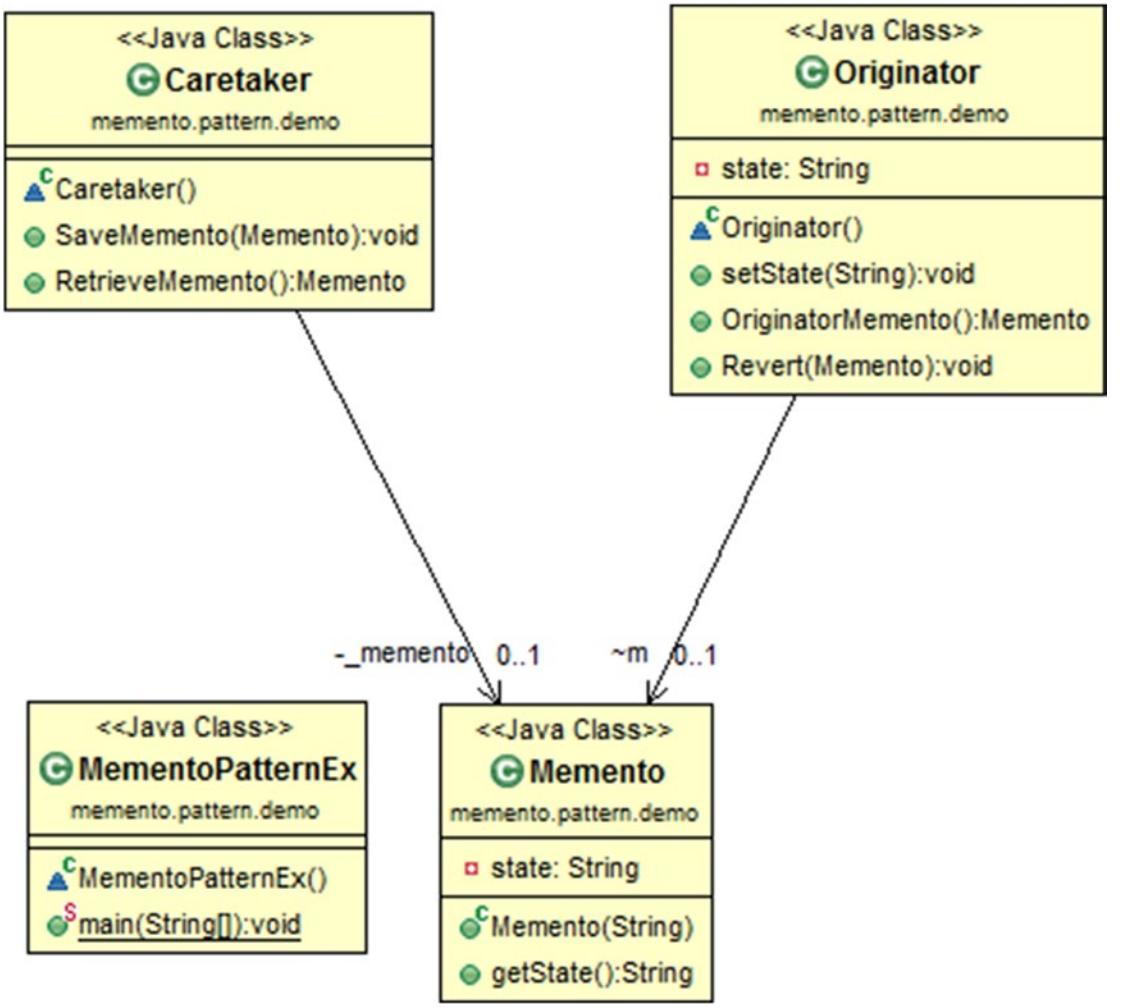
Computer World Example

A classic example in this category includes the state in a finite state machine. Apart from this, in real-world database programming, often we may need to roll back a transaction operation.

Illustration

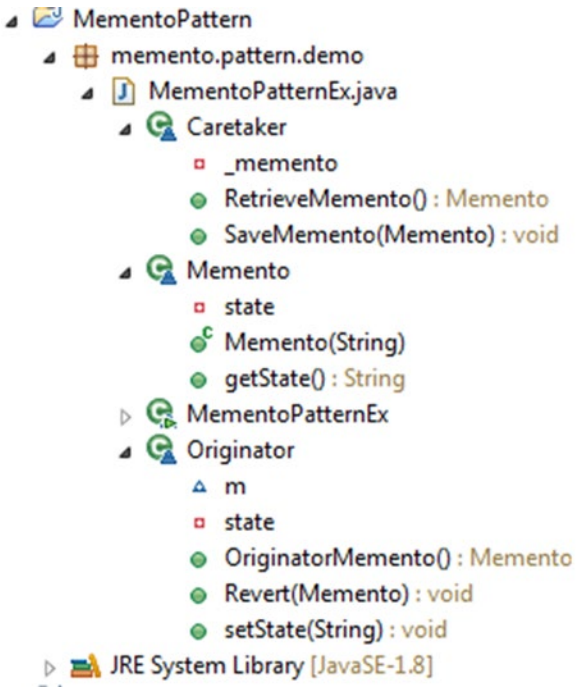
Please go through the code. Use the comments for your ready reference. Please also note that if you are familiar with C#, you can also use C# properties in place of the getter and setter operations used here.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



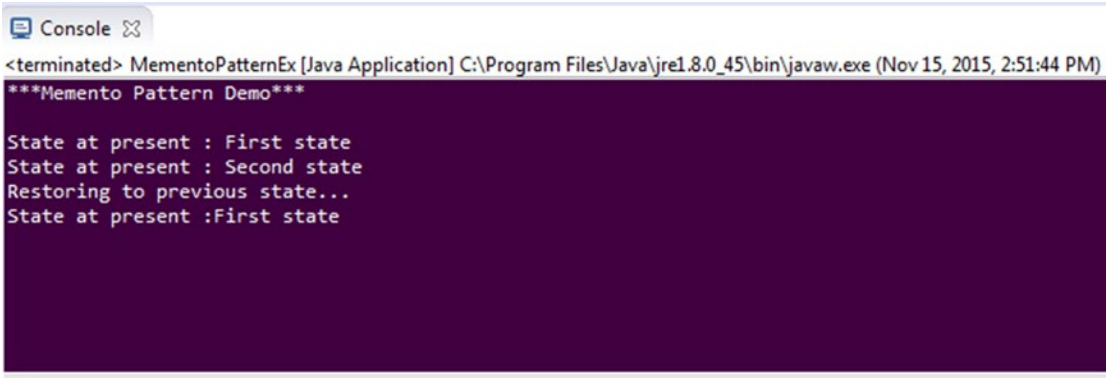
Implementation

```
package memento.pattern.demo;

// Memento class

class Memento
{
        private String state;

        public Memento(String state)
        {
                this.state = state;
        }
```

```
        public String getState()
        {
                return state;
        }

}

//  Originator class

class Originator
{
        private String state;
        Memento m;

        public void setState(String state)
        {
                this.state = state;
                System.out.println("State at present : " +state);
        }

        // Creates memento
        public Memento OriginatorMemento()
        {
                m = new Memento(state);
                return m;
        }

        // Back to old state
        public void Revert(Memento memento)
        {
                System.out.println("Restoring to previous state...");
                state = memento.getState();
                System.out.println("State at present :" +state);
        }
}

//Caretaker Class
class Caretaker
{
        private Memento _memento;

        public void SaveMemento(Memento m)
        {
                _memento = m;
        }
        public Memento RetrieveMemento()
        {
                return _memento;
        }

}
```

```
class MementoPatternEx
{
        public static void main(String[] args)
        {
                System.out.println("***Memento Pattern Demo***\n");
                Originator o = new Originator();
                o.setState("First state");

                // Holding old state
                Caretaker c = new Caretaker();
                c.SaveMemento(o.OriginatorMemento());

                //Changing state
                o.setState("Second state");

                // Restore saved state
                o.Revert(c.RetrieveMemento());

        }
}
```

Output



Note

1. We are advised to treat the memento object as an opaque object (i.e., ideally, caretakers should not be allowed to change them).
2. We should pay special attention so that other objects are not affected by the change made in the originator to the memento.
3. Sometimes, use of this pattern can cost more (e.g., if we want to store and restore large amount of data frequently). Also, from a caretaker point of view, the caretaker has no idea about how much state is kept in the memento that it wants to delete.

CHAPTER 14

State Patterns

GoF Definition: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Concept

The concept is best described by the examples that follow.

Real-Life Example

Consider a network connection for the moment. Here the object (that is responsible for communication) can be in various states (e.g., already a connection is established, a connection is closed, or the object is listening through the connection). We can also think of a traffic signal in this context.

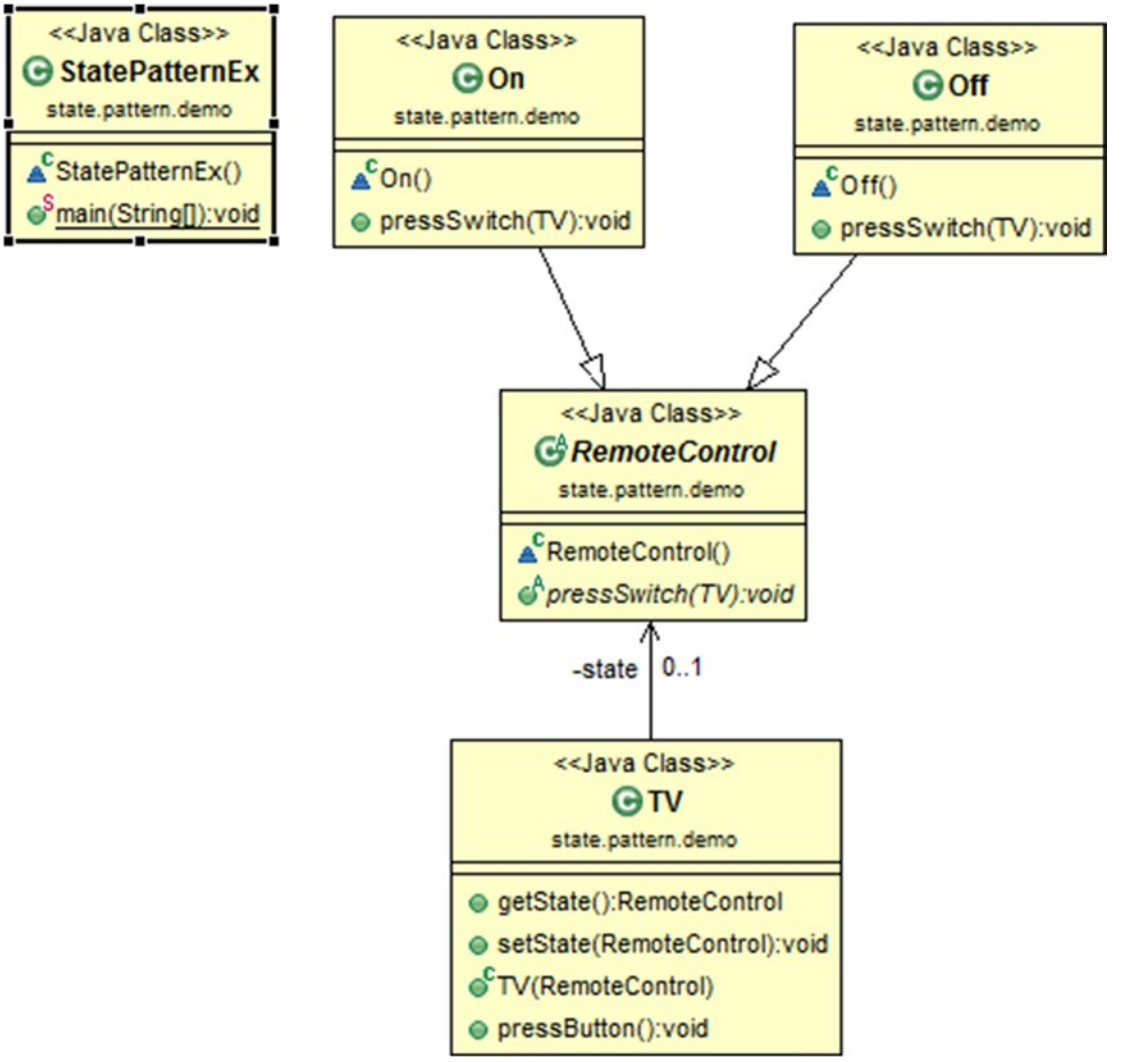
Computer World Example

The above example is applicable in the computer world also. Let us look at an additional example: We have a job processing application where we can process only one job (or any certain number of jobs) at a time. Now if a new job appears, either the application will process that job or it will signal that the new job cannot be processed at this moment because the system is already processing the maximum number of jobs in it (i.e., its number of job processing capabilities has reached the ceiling).

Illustration

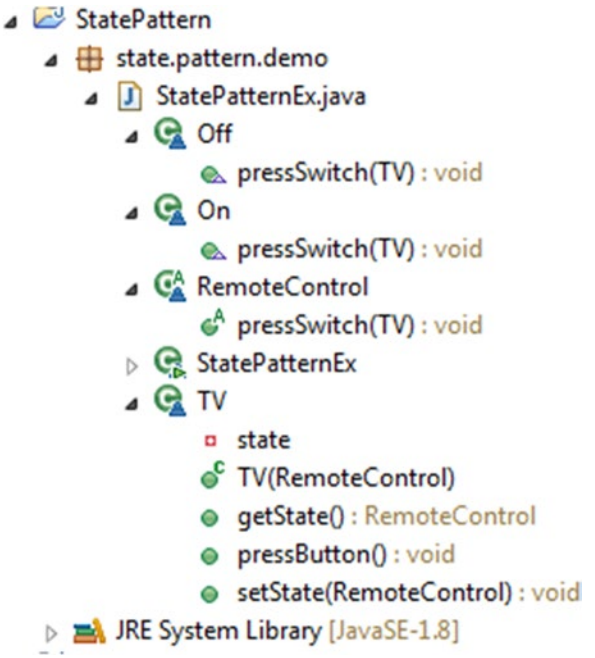
Here we have picked a very simple switching mechanism for turning a TV on/off. Suppose, we have a remote control to turn the TV on/off. Initially the TV is in the off state. When we press the power button, the TV will be on; upon the next press of the button, the TV will go off. We have implemented this concept with the state design pattern here.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:

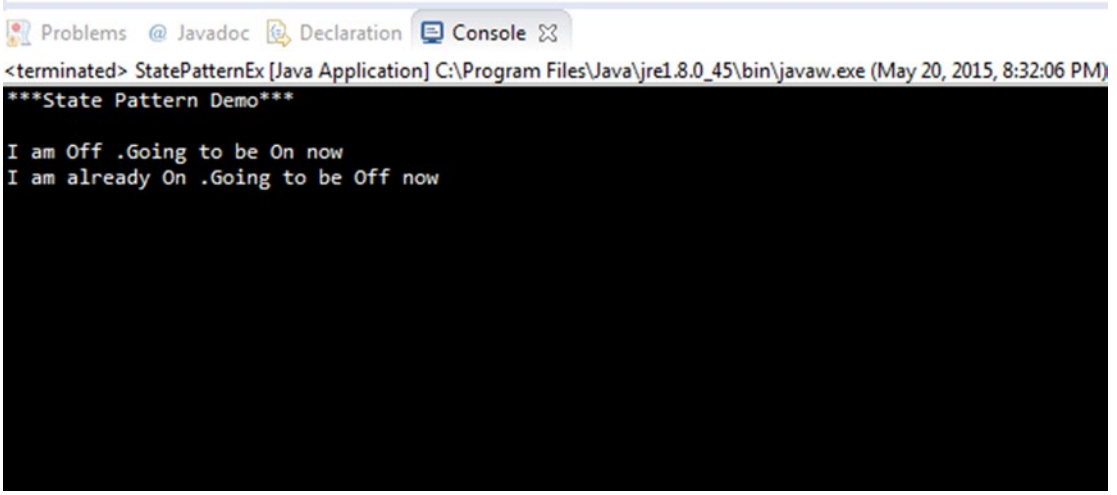


Implementation

```
package state.pattern.demo;
abstract class RemoteControl
{
    public abstract void pressSwitch(TV context);
}

class Off extends RemoteControl
{
        @Override
        public void pressSwitch(TV context){
        System.out.println("I am Off .Going to be On now");
        context.setState(new On());
    }
}
```

```
class On extends RemoteControl
{
        @Override
        public void pressSwitch(TV context)
        {
                System.out.println("I am already On .Going to be Off now");
        context.setState(new Off());
    }
}

class TV
{
    private RemoteControl state;

    public RemoteControl getState() {
                return state;
        }
        public void setState(RemoteControl state) {
                this.state = state;
        }
        public TV(RemoteControl state)
    {
        this.state=state;
    }
        public void pressButton()
    {
        state.pressSwitch(this);
    }
}
 class StatePatternEx
{
         public static void main(String[] args)
    {
                 System.out.println("***State Pattern Demo***\n");
                 //Initially TV is Off
        Off initialState = new Off();
        TV tv = new TV(initialState);
        //First time press
        tv.pressButton();
        //Second time press
        tv.pressButton();
    }
}
```

Output



Note

1. *As human beings, we perform best when we are happy and free of tension and our behavior can clearly reflect our mental state.* It is obvious that when we are in a happy and relaxed mode, we can perform better and we can talk to others in a friendlier tone. But consider the reverse scenario: when we are full of tension. In that scenario, in most cases, our efforts cannot produce a great result. That is why it is always suggested that we should work in relaxed mode. You can relate this simple philosophy with the foregoing illustration. If the TV is on, it can entertain you; if it is off, it cannot—right? *So, when we want to design similar kinds of behavior changes of an object when its internal state changes, this pattern becomes handy.*
2. If the number of states increases significantly in the system, then it becomes extremely hard to maintain that system.

CHAPTER 15

Builder Patterns

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

Concept

The pattern is useful when a creational algorithm of a complex object is independent of the assembly of the parts of the object. The construction process is also capable of building a different representation of that object under consideration.

Real-Life Example

To create a computer, different parts are assembled depending upon the order received by the customer (e.g., a customer can demand a 500 GB hard disk with an Intel processor; another customer can choose a 250 GB hard disk with an AMD processor).

Computer World Example

We sometimes need to convert one text format to another text format (e.g., RTF to ASCII text).

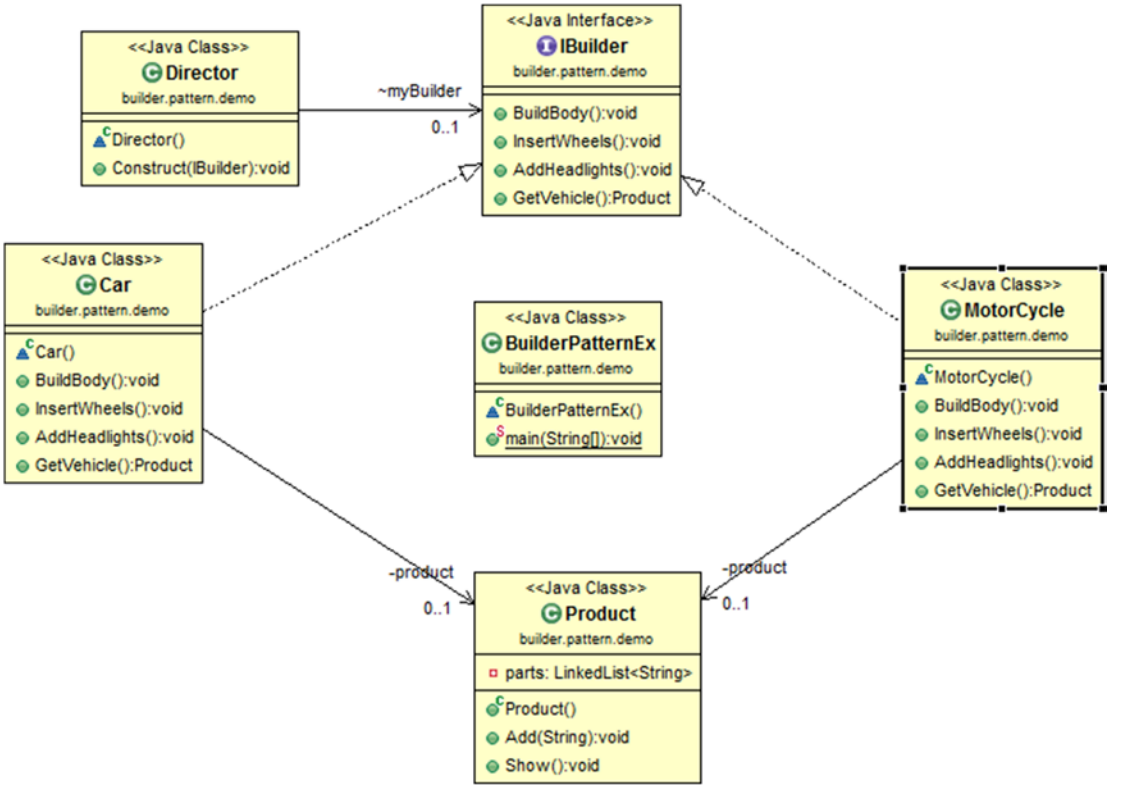
Illustration

Here the participants are *IBuilder, Car, MotorCycle, Product, and Director*. The first three are very straightforward—Car and MotorCycle are implementing the IBuilder interface. IBuilder is used to create parts of the Product object where Product represents the complex object under construction. The assembly process is described in Product. We can see that we have used the Linked List data structure in Product for this assembly operation.

Car and MotorCycle are the concrete implementations. They have implemented IBuilder interface. That's why they needed to BuildBody(), InsertWheels(), AddHeadlights(), and finally GetVehicle(). We use the first three methods to build the body of the vehicle, insert the number of wheels into it, and add headlights to the vehicle. GetVehicle() will return the ultimate product. Finally, Director will be responsible for constructing the ultimate vehicle. Director will build the product with IBuilder interface. He is calling the same Construct() method to create different types of vehicles.

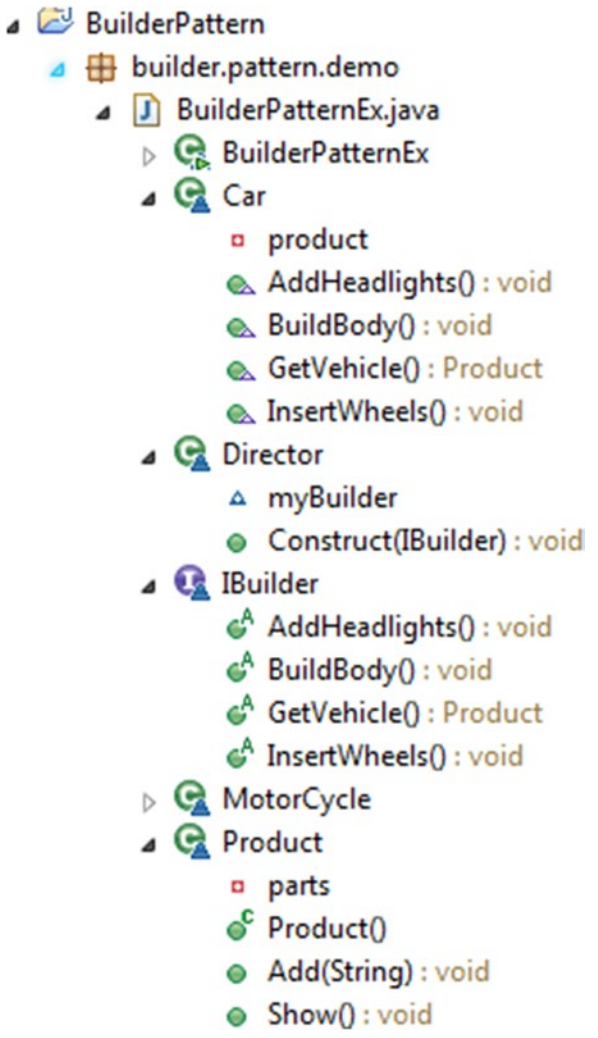
Please go through the code to see how different parts are assembled for this pattern.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



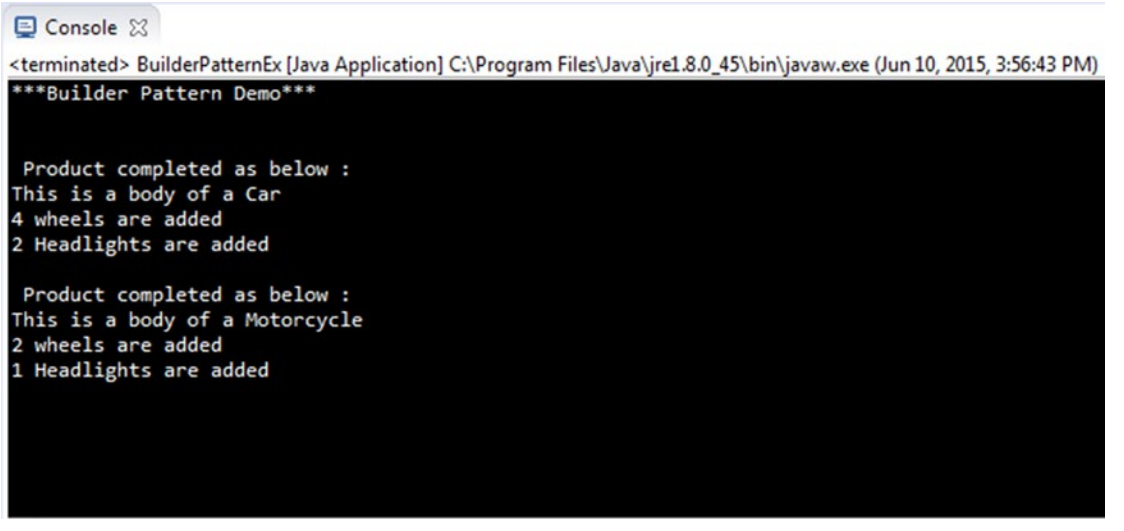
Implementation

```
package builder.pattern.demo;
import java.util.LinkedList;

// Builders common interface
interface IBuilder
{
        void BuildBody();
        void InsertWheels();
        void AddHeadlights();
        Product GetVehicle();

}

// Car is ConcreteBuilder
class Car implements IBuilder
{
        private Product product = new Product();

        @Override
        public void BuildBody()
        {
                product.Add("This is a body of a Car");
        }
        @Override
        public void InsertWheels()
        {
                product.Add("4 wheels are added");
        }
        @Override
        public void AddHeadlights()
        {
                product.Add("2 Headlights are added");
        }
        @Override
        public  Product GetVehicle()
        {
                return product;
        }

}

// Motorcycle is a ConcreteBuilder
class MotorCycle implements IBuilder
{
        private Product product = new Product();
        @Override
```

```
        public  void BuildBody()
        {
                product.Add("This is a body of a Motorcycle");
        }
        @Override
        public void InsertWheels()
        {
                product.Add("2 wheels are added");
        }
        @Override
        public void AddHeadlights()
        {
                product.Add("1 Headlights are added");
        }
        @Override
        public  Product GetVehicle()
        {
                return product;
        }
}

// "Product"
class Product
{
        // We can use any data structure that you prefer. We have used LinkedList here.
        private LinkedList<String> parts;
        public Product()
        {
                parts = new LinkedList<String>();
        }

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

        public void Show()
        {
                System.out.println("\n Product completed as below :");
                for(int i=0;i<parts.size();i++)
                {
                        System.out.println(parts.get(i));
                }
        }
}
// "Director"
class Director
{
        IBuilder myBuilder;
```

```
        // A series of steps—for the production
        public void Construct(IBuilder builder)
        {
                myBuilder=builder;
                myBuilder.BuildBody();
                myBuilder.InsertWheels();
                myBuilder.AddHeadlights();
        }
}
class BuilderPatternEx
{
        public static void main(String[] args)
        {
                System.out.println("***Builder Pattern Demo***\n");

                Director director = new Director();

                IBuilder carBuilder = new Car();
                IBuilder motorBuilder = new MotorCycle();

                // Making Car
                director.Construct(carBuilder);
                Product p1 = carBuilder.GetVehicle();
                p1.Show();

                //Making MotorCycle
                director.Construct(motorBuilder);
                Product p2 = motorBuilder.GetVehicle();
                p2.Show();
        }
}
```

Output



Note

1. Here we separate the code of assembling from its representation. So, it hides the complex construction process and represents it as a simple process.
2. Here we focus on "how the product will be made."
3. In general, we have only one method which will finally return the complete object. Other methods will be responsible for partial creation process only.
4. *It requires some amount of code duplication—which is considered a drawback with this pattern.*
5. *Also, if we want a mutable object (an object which can be modified after the creational process is over), we should not use this pattern.*

CHAPTER 16

Flyweight Patterns

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

Concept

A flyweight is an object through which we try to minimize memory usage by sharing data as much as possible. Two common terms are used here—*intrinsic* state and *extrinsic* state. The first category (intrinsic) can be stored in the flyweight and is shareable. The other one depends on the flyweight's context and is non-shareable. Client objects need to pass the extrinsic state to the flyweight.

Real-Life Example

In all real-world business applications, we want to avoid storing similar objects. The concept of this pattern is applicable in those places. I'll also share a story with you: a few years ago, two of my friends were each searching for an apartment to stay nearby their office. However, neither of them was satisfied with the available options. Then, one day, they found a place with all kind of facilities that they both desired. But there were two constraints—first of all, there was only one apartment, and second, the rent was high. So, to avail themselves of all those facilities, they decided to stay together and share the rent. This can be considered a real-life example of the flyweight pattern.

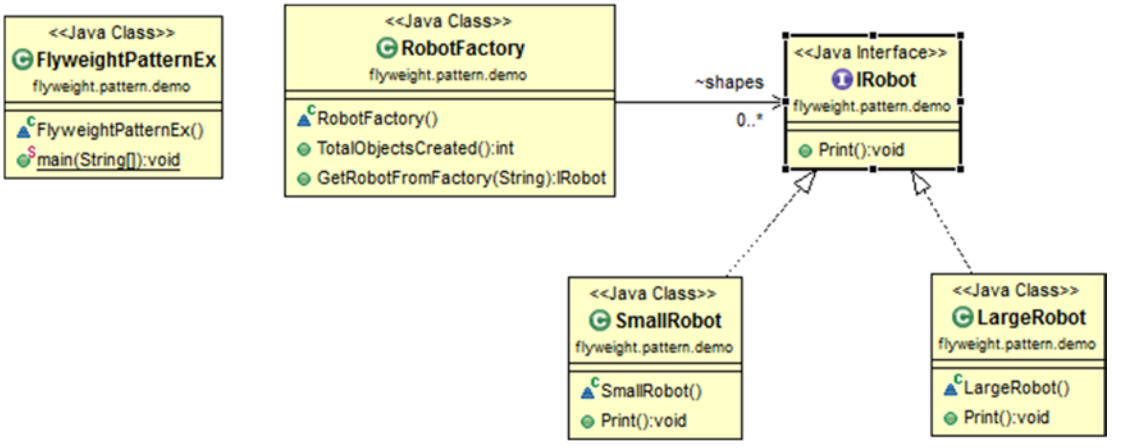
Computer World Example

The graphical representation of characters in word processors is a common example of this pattern. Also, we can think of a computer game where we have a large number of participants whose looks are same but who differ from each other in their performances (or color, dresses, weapons, etc.). We can use the flyweight pattern in all those scenarios.

Illustration

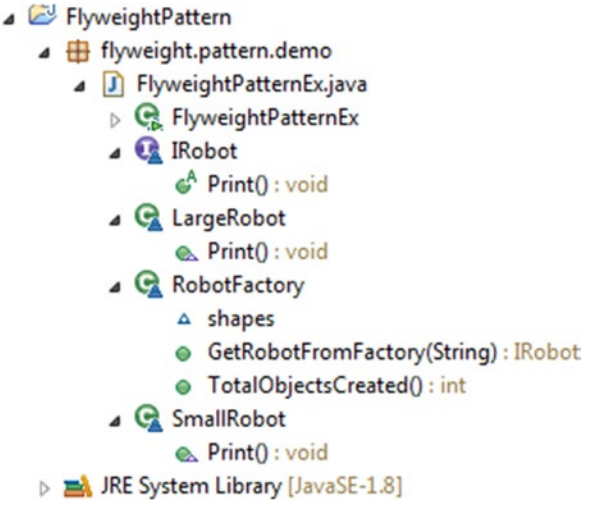
Please go through the code with the comments for your ready reference for an implementation of this pattern. Note that, we'll not create any new object (here, small or large robots) if we have created such a type already. If we have at least one instance of our desired object, we'll reuse that object from this point onward.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



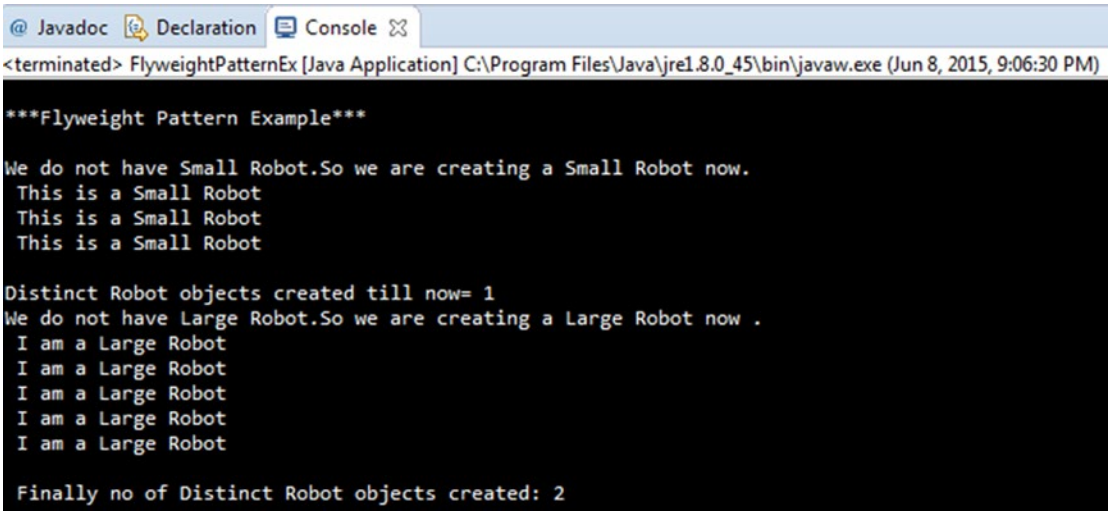
Implementation

```
package flyweight.pattern.demo;
import java.util.HashMap;
import java.util.Map;

/**
 * @author sarcarv
 * Our interface
 *
 */
interface IRobot
{
        void Print();
}

/**
 * @author sarcarv
 * A 'ConcreteFlyweight' class-SmallRobot
 *
 */
class SmallRobot implements IRobot
{
        @Override
        public void Print()
        {
                System.out.println(" This is a Small Robot");
        }
}

/**
 * @author sarcarv
 * A 'ConcreteFlyweight' class-LargeRobot
 *
 */
class LargeRobot implements IRobot
{
        @Override
        public void Print()
        {
                System.out.println(" I am a Large Robot");
        }
}
```

```
/**
 * @author sarcarv
 * The 'FlyweightFactory' class
 *
 */
class RobotFactory
{
        Map<String, IRobot> shapes = new HashMap<String, IRobot>();

        public int TotalObjectsCreated()
        {
                return shapes.size();
        }

        public IRobot GetRobotFromFactory(String RobotCategory) throws Exception
        {
                IRobot robotCategory = null;
                if (shapes.containsKey(RobotCategory))
                {
                        robotCategory = shapes.get(RobotCategory);
                }
                else
                {
                        switch (RobotCategory)
                        {
                        case "small":
                                System.out.println("We do not have Small Robot. So we are
                                creating a Small Robot now.");
                                robotCategory = new SmallRobot();
                                shapes.put("small", robotCategory);
                                break;
                        case "large":
                                System.out.println("We do not have Large Robot. So we are
                                creating a Large Robot now .");
                                robotCategory = new LargeRobot();
                                shapes.put("large", robotCategory);
                                break;
                        default:
                                throw new Exception(" Robot Factory can create only small
                                and large shapes");
                        }
                }
                return robotCategory;
        }

}
```

```
/**
 * @author sarcarv
 *FlyweightPattern is in action.
 */

class FlyweightPatternEx
{
        public static void main(String[] args) throws Exception
        {
                RobotFactory myfactory = new RobotFactory();
                System.out.println("\n***Flyweight Pattern Example***\n");

                IRobot shape = myfactory.GetRobotFromFactory("small");
                shape.Print();
                /*Here we are trying to get the objects additional 2 times. Note that from
                now onward we do not need to create additional small robots as we have
                already created this category*/
                for (int i = 0; i < 2; i++)
                {
                        shape = myfactory.GetRobotFromFactory("small");
                        shape.Print();
                }
                int NumOfDistinctRobots = myfactory.TotalObjectsCreated();
                System.out.println("\nDistinct Robot objects created till now=
                "+ NumOfDistinctRobots);

                /*Here we are trying to get the objects 5 times. Note that the second time
                onward we do not need to create additional large robots as we have already
                created this category in the first attempt(at i=0)*/
                for (int i = 0; i < 5; i++)
                {
                        shape = myfactory.GetRobotFromFactory("large");
                        shape.Print();
                }

                NumOfDistinctRobots = myfactory.TotalObjectsCreated();
                System.out.println("\n Finally no of Distinct Robot objects created:
                "+ NumOfDistinctRobots);
        }
}
```

Output

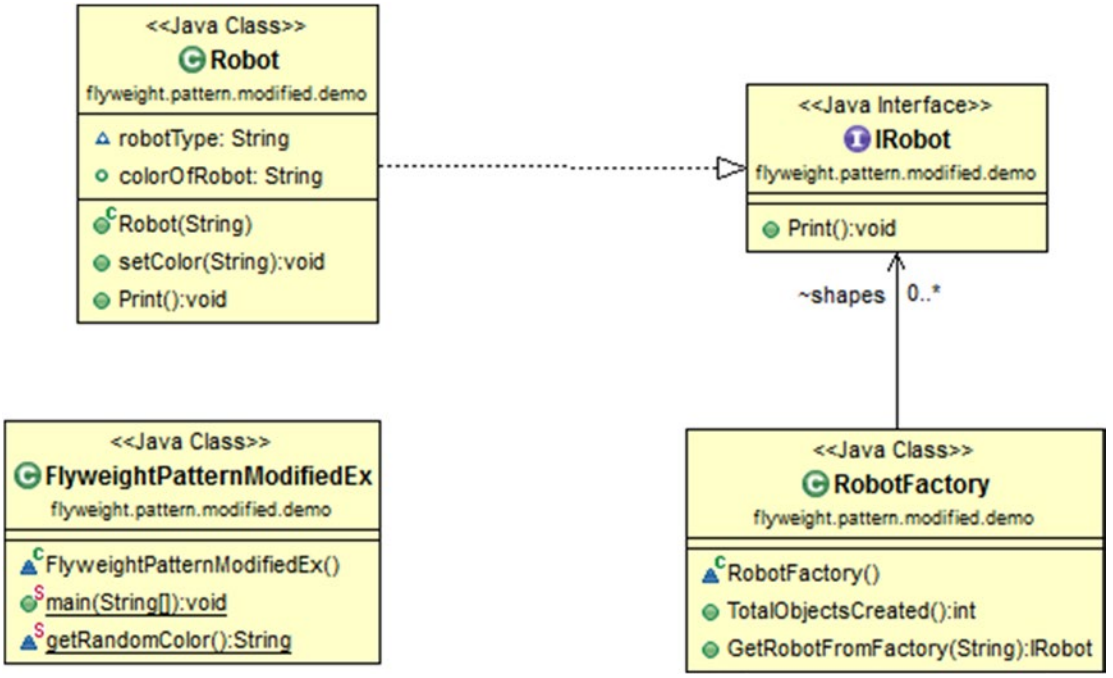


Improvement to the program

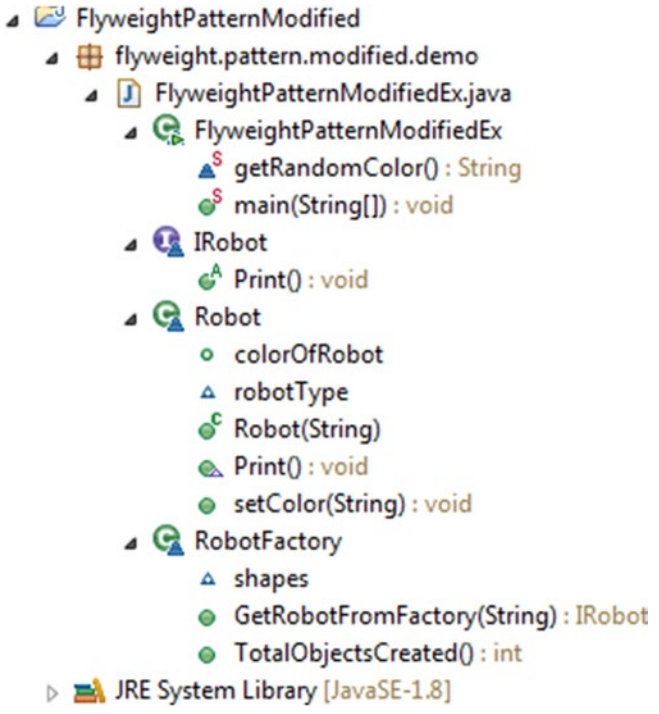
Now take a closer look to our program. *Here it seems that this pattern is behaving similar to a singleton pattern because we cannot create two distinct types of small (or large) robots*. But there may be situations where we want different types of small (or large) robots in which the basic structure should be same but it will differ only with some special characteristics. So, we are going to present another example here. This illustration will clear your doubt and you can make the distinction between the flyweight and the singleton patterns.

To make the program simple, we are dealing with robots which can be either king type or queen type. Each of these types can be either green or red. Before making any robot, we'll consult with our factory. If we already have king or queen types of robots, we'll not create them again. We will collect the basic structure from our factory and after that we'll color them. *Note that color is extrinsic data here, but the category of robot (king or queen) is intrinsic.*

UML Class Diagram



Package Explorer view



Implementation

```
package flyweight.pattern.modified.demo;
import java.util.HashMap;
import java.util.Map;
import java.util.Random;

/**
 * @author sarcarv
 * Our interface
 *
 */
interface IRobot
{
        void Print();
}

/**
 * @author sarcarv
 * A 'ConcreteFlyweight' class-SmallRobot
 *
 */
```

```
class Robot implements IRobot
{
        String robotType;
        public String colorOfRobot;
        public Robot(String robotType)
        {
                this.robotType=robotType;
        }
        public void setColor(String colorOfRobot)
        {
                this.colorOfRobot=colorOfRobot;
        }
        @Override
        public void Print()
        {
                System.out.println(" This is a " +robotType+ " type robot with
                "+colorOfRobot+ "color");
        }
}

/**
 * @author sarcarv
 * The 'FlyweightFactory' class
 *
 */
class RobotFactory
{
        Map<String, IRobot> shapes = new HashMap<String, IRobot>();

        public int TotalObjectsCreated()
        {
                return shapes.size();
        }

        public IRobot GetRobotFromFactory(String robotType) throws Exception
        {
                IRobot robotCategory= null;
                if (shapes.containsKey(robotType))
                {
                        robotCategory = shapes.get(robotType);
                }
                else
                {
                        switch (robotType)
                        {
                        case "King":
                                System.out.println("We do not have King Robot. So we are
                                creating a King Robot now.");
                                robotCategory = new Robot("King");
                                shapes.put("King",robotCategory);
                                break;
```

```
                        case "Queen":
                                System.out.println("We do not have Queen Robot. So we are
                                creating a Queen Robot now.");
                                robotCategory = new Robot("Queen");
                                shapes.put("Queen",robotCategory);
                                break;
                        default:
                                throw new Exception(" Robot Factory can create only king and
                                queen type robots");
                        }
                }
                return robotCategory;
        }

}
/**
 * @author sarcarv
 *FlyweightPattern is in action.
 */

class FlyweightPatternModifiedEx
{
        public static void main(String[] args) throws Exception
        {
                RobotFactory myfactory = new RobotFactory();
                System.out.println("\n***Flyweight Pattern Example Modified***\n");
        Robot shape;
                /*Here we are trying to get 3 king type robots*/
                for (int i = 0; i < 3; i++)
                {
                        shape =(Robot) myfactory.GetRobotFromFactory("King");
                        shape.setColor(getRandomColor());
                        shape.Print();
                }
                /*Here we are trying to get 3 queen type robots*/
                for (int i = 0; i < 3; i++)
                {
                        shape =(Robot) myfactory.GetRobotFromFactory("Queen");
                        shape.setColor(getRandomColor());
                        shape.Print();
                }
                int NumOfDistinctRobots = myfactory.TotalObjectsCreated();
                //System.out.println("\nDistinct Robot objects created till now =
                "+ NumOfDistinctRobots);
                System.out.println("\n Finally no of Distinct Robot objects created:
                "+ NumOfDistinctRobots);
        }
        static String getRandomColor()
        {
                Random r=new Random();
                /*You can supply any number of your choice in nextInt argument.
```

```
                 * we are simply checking the random number generated is an even number
                 * or an odd number. And based on that we are choosing the color. For
                 simplicity, we'll use only two colors—red and green
                 */
                int random=r.nextInt(20);
                if(random%2==0)
                {
                        return "red";
                }
                else
                {
                        return "green";
                }
        }
}
```

Output

Note

1. Minimization of storage is one of the key concerns here. If we can have more flyweights to share, we can save more memory.
2. If we can compute extrinsic states rather than storing them, we can save a significant amount of memory.
3. Sometimes in a tree structure, to share leaf nodes, we combine this pattern with composite pattern.
4. A flyweight interface may or may not enable sharing. In some cases, we may have unshared flyweight objects, which in turn may have concrete flyweight objects as children.
5. *In simple terms: intrinsic data make the instance unique, whereas extrinsic data are passed as arguments.*

CHAPTER 17

Abstract Factory Patterns

GoF Definition: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Concept

In this pattern, we provide an encapsulation mechanism to a group of individual factories. These factories have a theme in common. In this process, an interface is used to create related objects. Here we do not call their implementer or concrete classes directly. We sometimes refer to this pattern as a *factory of factories* or a Super *factory*.

With this pattern, we can interchange the specific implementations without changing the user's code. But to achieve this, we need to compensate for the complexity of the system. As a result, debugging may be difficult in many scenarios.

Real-Life Example

Suppose we are decorating our room. Now suppose we need two different types of almirah (or, say, table)—one must be made of wood and one of steel. For the wooden almirah, we need to visit a carpenter shop and for the other type, we can go to a readymade steel almirah shop. Both of these are almirah (or table) factories. Based on our demand, we decide what kind of factory we need. This scenario can be considered an example of this pattern.

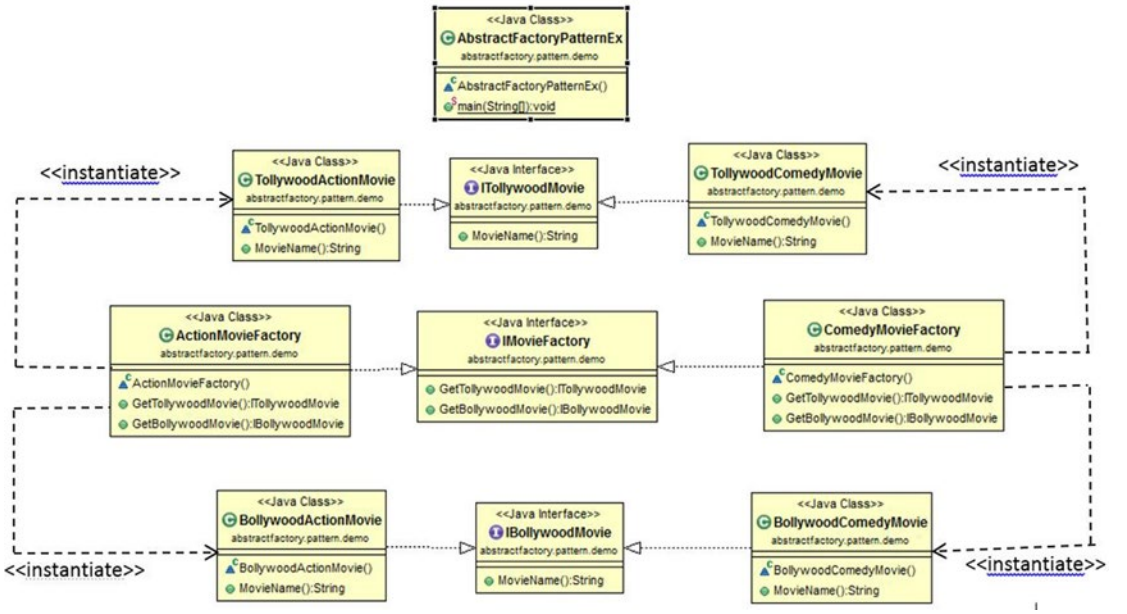
Computer World Example

ADO.NET has already implemented similar concepts to establish a connection to a database.

Illustration

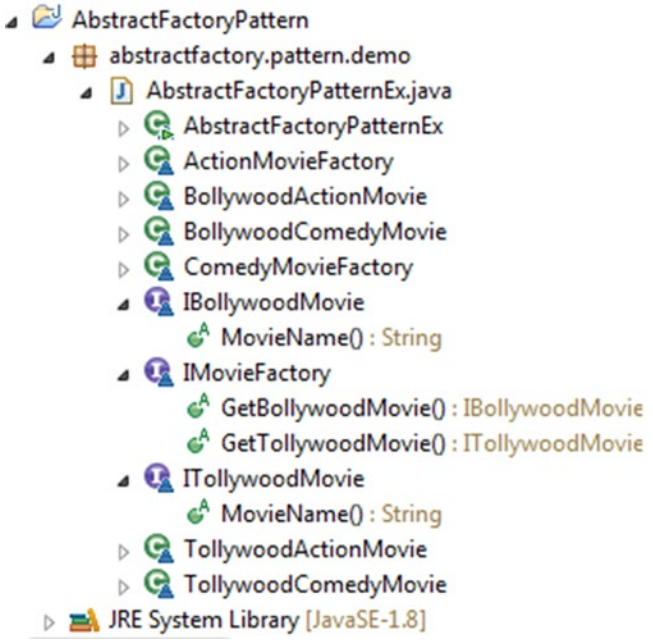
In this example, our client is looking for movies and he/she needs to access an Abstract Factory, IMovieFactory, and Abstract Products, ITollywoodMovie and IBollywoodMovie. The client does not care which of these factories is giving the concrete object for him/her. He/she uses only the generic interfaces of their products. The naming conventions are chosen for your easy reference.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



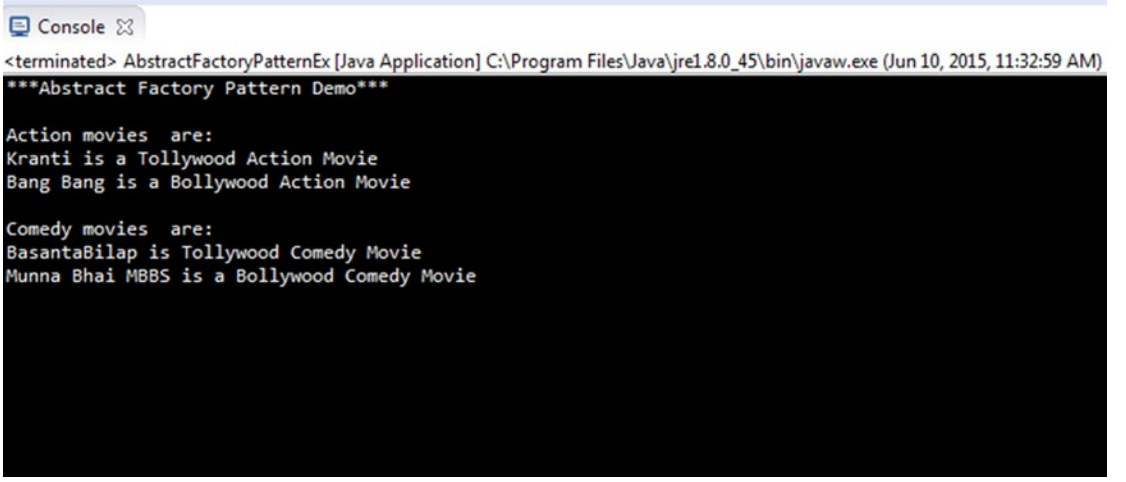
Implementation

```
package abstractfactory.pattern.demo;
interface ITollywoodMovie
{
    String MovieName();
}

interface IBollywoodMovie
{
    String MovieName();
}

interface IMovieFactory
{
    ITollywoodMovie GetTollywoodMovie();
    IBollywoodMovie GetBollywoodMovie();
}
```

```
//Tollywood Movie collections
class TollywoodActionMovie implements ITollywoodMovie
{
    @Override
     public String MovieName()
    {
        return "Kranti is a Tollywood Action Movie";
    }

}

class TollywoodComedyMovie implements ITollywoodMovie
{
    @Override
    public String MovieName()
    {
        return "BasantaBilap is a Tollywood Comedy Movie";
    }
}

// Bollywood Movie collections
class BollywoodActionMovie implements IBollywoodMovie
{
       @Override
       public String MovieName()
    {
        return "Bang Bang is a Bollywood Action Movie";
    }
}

class BollywoodComedyMovie implements IBollywoodMovie
{
       @Override
       public String MovieName()
    {
        return "Munna Bhai MBBS is a Bollywood Comedy Movie";
    }
}

//Action Movie Factory
class ActionMovieFactory implements IMovieFactory
{
    public ITollywoodMovie GetTollywoodMovie()
    {
        return new TollywoodActionMovie();
    }
```

```
    public IBollywoodMovie GetBollywoodMovie()
    {
        return new BollywoodActionMovie();
    }
}
//Comedy Movie Factory
class ComedyMovieFactory implements IMovieFactory
{
    public ITollywoodMovie GetTollywoodMovie()
    {
        return new TollywoodComedyMovie();
    }

    public IBollywoodMovie GetBollywoodMovie()
    {
        return new BollywoodComedyMovie();
    }
}
class AbstractFactoryPatternEx
{
        public static void main(String[] args)
    {
            System.out.println("***Abstract Factory Pattern Demo***");
            ActionMovieFactory actionMovies = new ActionMovieFactory();
        ITollywoodMovie tAction = actionMovies.GetTollywoodMovie();
        IBollywoodMovie bAction = actionMovies.GetBollywoodMovie();

        System.out.println("\nAction movies  are:");
        System.out.println(tAction.MovieName());
        System.out.println(bAction.MovieName());

        ComedyMovieFactory comedyMovies = new ComedyMovieFactory();
        ITollywoodMovie tComedy = comedyMovies.GetTollywoodMovie();
        IBollywoodMovie bComedy = comedyMovies.GetBollywoodMovie();

        System.out.println("\nComedy movies  are:");
        System.out.println(tComedy.MovieName());
        System.out.println(bComedy.MovieName());

    }
}
```

Output



Note

1. We use this pattern when our system does not care about how its products will be created or composed.
2. We use this pattern when we need to deal with multiple factories.
3. This pattern separates concrete classes and makes interchanging the products easier. It can also enhance the reliabilities among products. But, at the same time, we must acknowledge the fact that creating a new product is difficult with this pattern (because we need to extend the interface and, as a result, changes will be required in all of the subclasses that already implemented the interface).

CHAPTER 18

Mediator Patterns

GoF Definition: Define an object that encapsulates how a set of objects interacts. The mediator pattern promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Concept

A mediator is the one who takes the responsibility of communication among a group of objects. The mediator acts as an intermediary who can track the communication between two objects. The other objects in the system are also aware of this mediator and they know that if they need to communicate among themselves, they need to go through the mediator. *The advantage of using such a mediator is that we can reduce the direct interconnections among the objects and thus lower the coupling.*

Real-Life Example

In an airplane application, before taking off the flight undergoes a series of checks. These checks confirm that all components/parts (which are dependent on each other) are in perfect condition.

Also, the pilot needs to communicate with the towers at the airport. In general, one pilot from one airline will not communicate with another pilot from a different airline before taking off or landing operations. Towers are acting as the mediator here.

Computer World Example

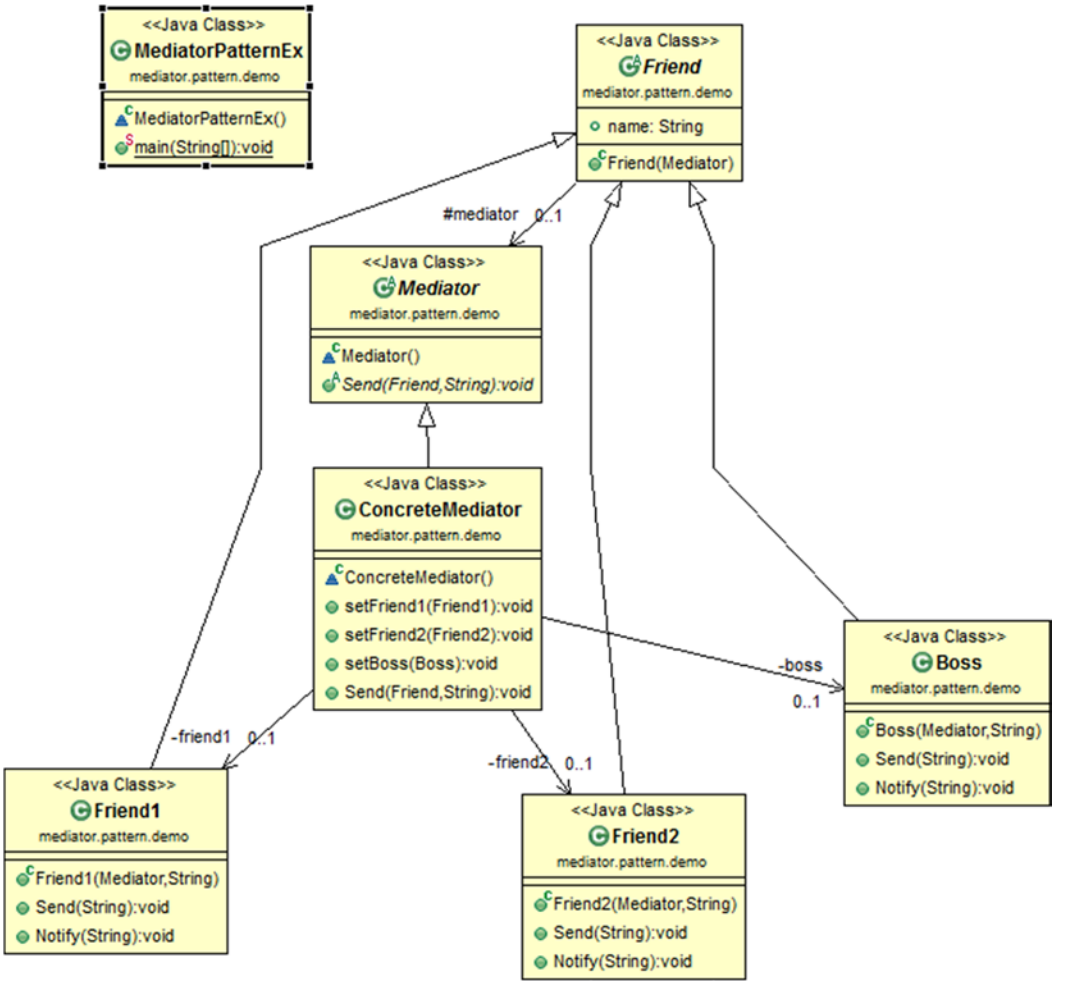
In a business application, in many cases we need to apply constraints (e.g., suppose we have a form for which we need to supply a user id and then a password for that account). In the same form, we may need to supply other mandatory information (e-mail id, communication address, age, etc.). Now suppose the functions are implemented as follows: once a user supplies his user id, the form will check whether that user id (supplied by user) is valid. If it is a valid id, then only the password field will be enabled. After supplying these two fields, we may need to check whether the user has provided any e-mail id. Let's assume here after providing a user id with a valid e-mail and all other mandatory details, our submit button will be enabled (i.e., the overall submit button will be enabled if we supply a valid user id, password, e-mail id, and other mandatory details only). We must also ensure that the user id is an integer, so if the user by mistake provides any character in that field, the submit button still will be in disabled mode. The mediator pattern becomes handy in such a scenario.

So, when a program consists of many classes and the logic is distributed among them, it becomes harder to read and maintain. If we need to make some kind of change, it becomes a challenging task for us. The mediator pattern is handy in such a scenario.

Illustration

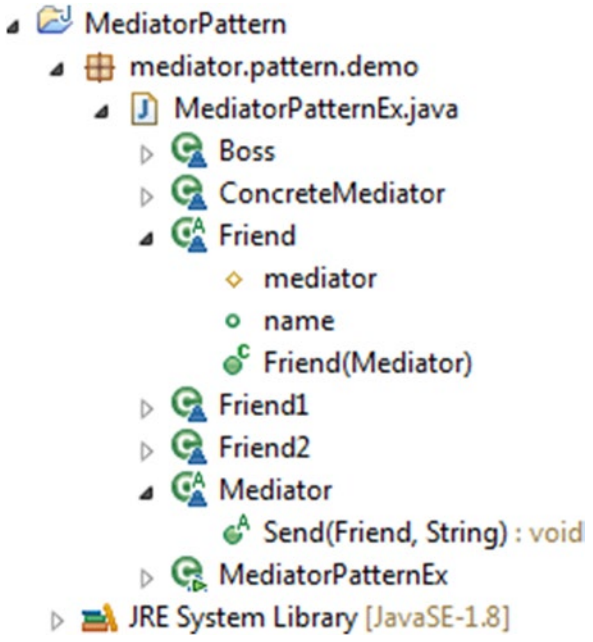
In the example here, we have a group of three friends—Amit, Sohel, and Raghu. Among these three friends, Raghu is the boss and he needs to coordinate things properly. Amit and Sohel work in Raghu's team. Whenever Amit and Sohel talk to each other (say, through a chat server), Raghu can see who is sending messages to him (though he does not care about the message). Raghu wants to coordinate things smoothly and whenever he wants to send messages, he wants his messages to reach others instantly. Analyze the code and output and it'll be clear to you.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



Implementation

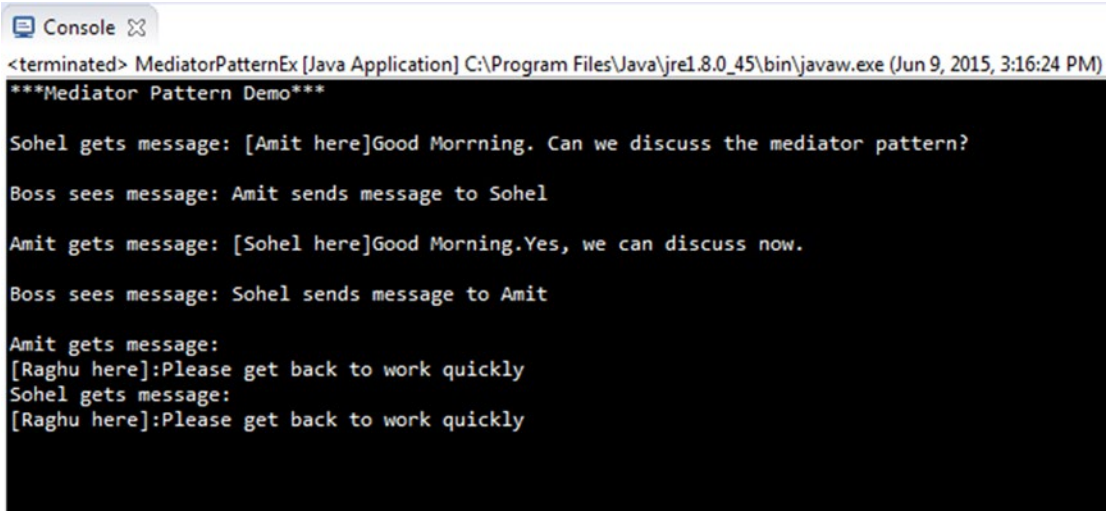
```
package mediator.pattern.demo;

abstract class Mediator
{
    public abstract void Send(Friend frd, String msg);
}

// ConcreteMediator
class ConcreteMediator extends  Mediator
{
    private Friend1 friend1;
    private Friend2 friend2;
    private Boss boss;
```

```
        //In this example, setters are sufficient.

                public void setFriend1(Friend1 friend1) {
                        this.friend1 = friend1;
                }

                public void setFriend2(Friend2 friend2) {
                        this.friend2 = friend2;
                }

                public void setBoss(Boss boss) {
                        this.boss = boss;
                }

        @Override
        public void Send(Friend frd,String msg)
        {
            //In all cases, boss is notified
            if (frd == friend1)
            {
                friend2.Notify(msg);
                boss.Notify(friend1.name + " sends message to " + friend2.name);
            }
            if(frd==friend2)
            {
                friend1.Notify(msg);
                boss.Notify(friend2.name + " sends message to " + friend1.name);

            }
            //Boss is sending message to others
            if(frd==boss)
            {
                friend1.Notify(msg);
                friend2.Notify(msg);
            }
        }

    }

    // Friend
    abstract class Friend
    {
        protected Mediator mediator;
        public String name;
```

```
    // Constructor
    public Friend(Mediator _mediator)
    {
        mediator = _mediator;
    }
}

// Friend1—first participant
class Friend1 extends Friend
{
    public Friend1(Mediator mediator,String name)
    {
            super(mediator);
        this.name = name;
    }

    public void Send(String msg)
    {
        mediator.Send(this,msg);
    }

    public void Notify(String msg)
    {
            System.out.println("Amit gets message: "+ msg);
    }
}

// Friend2—Second participant
class Friend2 extends Friend
{
    // Constructor
    public Friend2(Mediator mediator,String name)
    {
        super(mediator);
            this.name = name;
    }

    public void Send(String msg)
    {
        mediator.Send(this,msg);
    }

    public void Notify(String msg)
    {
            System.out.println("Sohel gets message: "+ msg);
    }
}
```

```
// Friend3–Third participant. He is the boss. He is notified whenever friend1 and
friend2 communicate.
class Boss extends Friend
{
    // Constructor
    public Boss(Mediator mediator,String name)
    {
            super(mediator);
        this.name = name;
    }

    public void Send(String msg)
    {
        mediator.Send(this, msg);
    }

    public void Notify(String msg)
    {
            System.out.println("\nBoss sees message: " + msg);
            System.out.println("");
    }
}

class MediatorPatternEx
{
    public static void main(String[] args)
    {
            System.out.println("***Mediator Pattern Demo***\n");
            ConcreteMediator m = new ConcreteMediator();

        Friend1 Amit= new Friend1(m,"Amit");
        Friend2 Sohel = new Friend2(m,"Sohel");
        Boss Raghu = new Boss(m,"Raghu");

        m.setFriend1(Amit);
        m.setFriend2(Sohel);
        m.setBoss(Raghu);

        Amit.Send("[Amit here]Good Morrning. Can we discuss the mediator pattern?");
        Sohel.Send("[Sohel here]Good Morning.Yes, we can discuss now.");
        Raghu.Send("\n[Raghu here]:Please get back to work quickly");
    }
}
```

Output



Note

1. Now you should have a clear idea that this pattern is very useful when we observe complex communications in the system. Communication (among objects) is much simpler with this pattern.
2. This pattern reduces the number of subclasses in the system and it also enhances the loose coupling in the system.
3. Here the “many-to-many” relationship is replaced with the “one-to-many” relationship—which is much easier to read and understand.
4. We can provide a centralized control with this pattern.
5. Sometimes the encapsulation process becomes tricky and we find it difficult to maintain or implement.

CHAPTER 19

Prototype Patterns

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

Concept

The prototype pattern provides an alternative method for instantiating new objects by copying or cloning an instance of an existing one. Creating a new instance, in a real-world scenario, is normally treated as an expensive operation. This pattern helps us to deal with this issue. Our focus here is to reduce the expense of this creational process of a new instance.

Real-Life Example

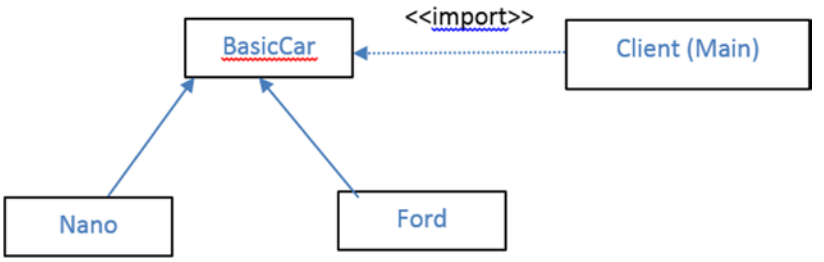
Suppose we have a master copy of a valuable document. We want to make some change to it to get a different feel. We can make a photocopy of this document and then try to edit our changes.

Computer World Example

Suppose we have made an application. The next time we want to create a similar application with some small changes, we must start with a copy from our master copy application and make the changes. We'll not start from the scratch.

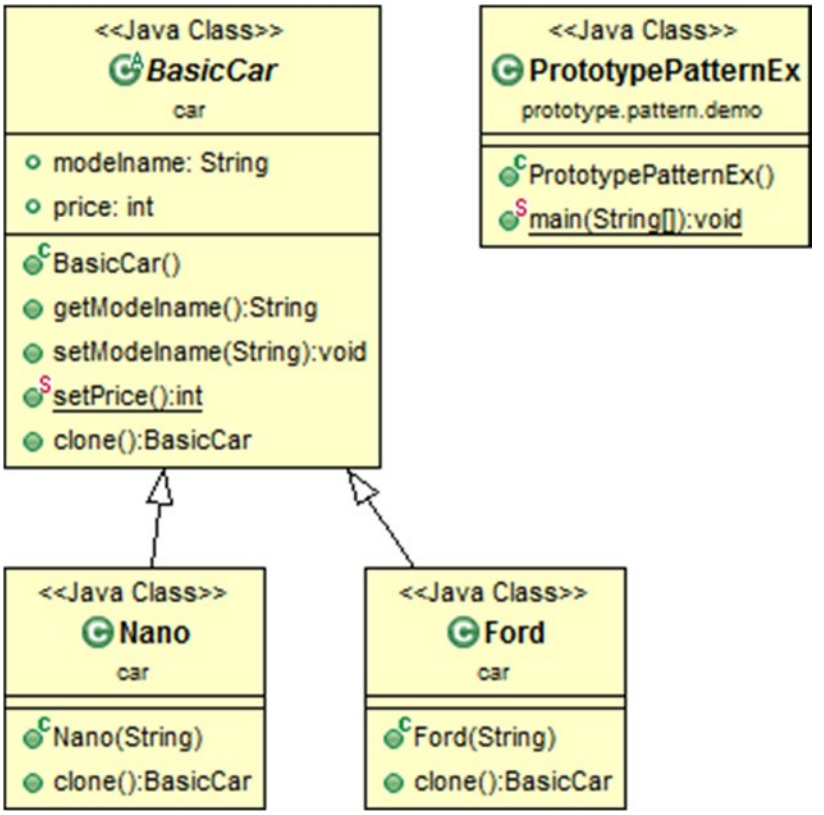
Illustration

In my example, I am going to follow the structure shown here:



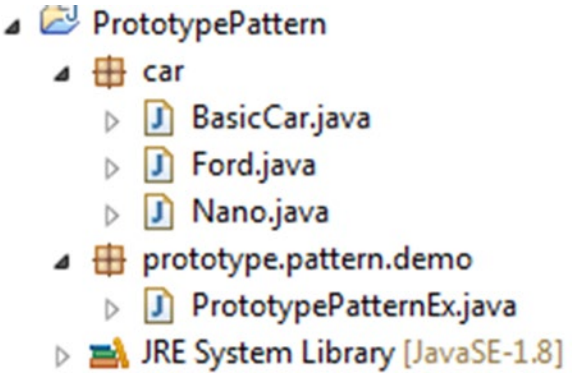
Here BasicCar is our prototype. Nano and Ford are our Concrete Prototypes and they need to implement the Clone() method defined in BasicCar. Here we notice that a BasicCar model is created with some default price. Later we have modified that price as per the model. Please also note that PrototypePatternEx is the client here. As usual, the related parts are separated by the packages for better readability.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



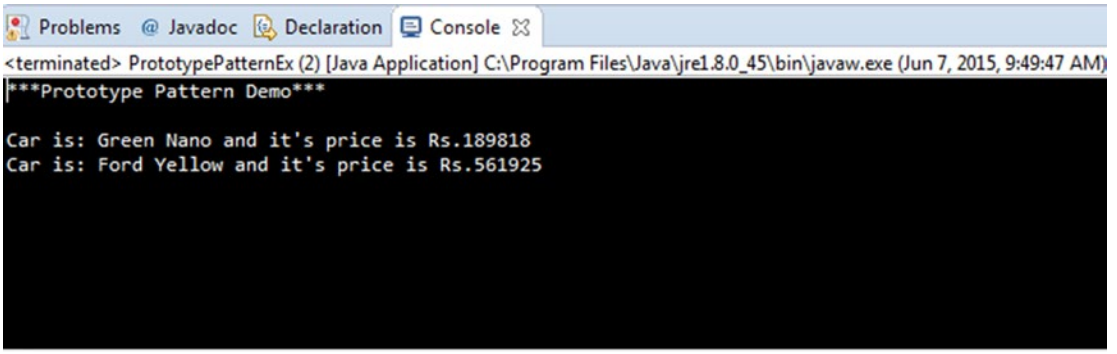
Implementation

```
//BasicCar.java
package car;
import java.util.Random;

public abstract class BasicCar implements Cloneable
{
    public String modelname;
    public int price;

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

    public static int setPrice()
    {
        int price = 0;
        Random r = new Random();
        int p = r.nextInt(100000);
        price = p;
       return price;
    }
```

```
   public BasicCar clone() throws CloneNotSupportedException
        {
                return  (BasicCar)super.clone();
        }

}
//Ford.java
package car;

public class Ford extends BasicCar
{
    public Ford(String m)
    {
        modelname = m;
    }

        @Override
        public BasicCar clone() throws CloneNotSupportedException
        {
                 return (Ford)super.clone();
        }
}
//Nano.java
package car;

public class Nano extends BasicCar
{
    public Nano(String m)
    {
        modelname = m;
    }
        @Override
        public BasicCar clone() throws CloneNotSupportedException
        {
                 return (Nano)super.clone();

        }
}
//PrototypePatternEx.java
package prototype.pattern.demo;
import car.*;

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

```
        BasicCar ford_basic = new Ford("Ford Yellow");
        ford_basic.price=500000;

        BasicCar bc1;
        //Clone Nano Object
        bc1 =nano_base.clone();
       //Price will be more than 100000 for sure
        bc1.price = nano_base.price+BasicCar.setPrice();
        System.out.println("Car is: "+ bc1.modelname+" and it's price is Rs."+bc1.price);

        //Clone Ford Object
        bc1 =ford_basic.clone();
        //Price will be more than 500000 for sure
        bc1.price = ford_basic.price+BasicCar.setPrice();
        System.out.println("Car is: "+ bc1.modelname+" and it's price is Rs."+bc1.price);

    }
}
```

Output



Note

1. When the system does not care about the creational mechanism of the products, this pattern is very helpful.
2. We can use this pattern when we need to instantiate classes at runtime.
3. In our example, we have used the default clone() method in Java, which is a shallow copy. Thus, it is inexpensive compared to a deep copy.

What are the advantages of the prototype pattern?

1. We can include or discard products at runtime.
2. We can create new instances with a cheaper cost.

What are the disadvantages of the prototype pattern?

1. Each subclass has to implement the cloning mechanism.
2. Implementing the cloning mechanism can be challenging if the objects under consideration do not support copying or if there is any kind of circular reference.

CHAPTER 20

Chain of Responsibility Patterns

GoF Definition: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Concept

Here we process a series of objects one by one (i.e., in a sequential manner). A source will initiate this processing. With this pattern, we constitute a chain where each of the processing objects can have some logic to handle a particular type of command object. After one's processing is done, if anything is still pending, it can be forwarded to the next object in the chain. We can add new objects anytime at the end of a chain.

Real-Life Example

In an organization, there are some customer care executives who handle feedback/issues from customers and forward those customer issues/escalations to the appropriate department in the organization. Not all departments will start fixing an issue. The department that seems to be responsible will take a look first, and if the department staff believe that the issue should be forwarded to another department, he/she will do that.

Computer World Example

Consider an application which is handling e-mail and faxes. As usual, we need to take care of the issues reported in each of these communications. We need to introduce two error handlers—EmailErrorHandler and FaxErrorHandler. EmailErrorHandler will handle e-mail errors only and is not responsible for fax errors. In the same manner, FaxErrorHandler will handle fax errors and does not care about e-mail errors.

Then we can make a chain as follows: whenever our main application finds an error, it will just raise this and forward the error with the hope that one of those handlers will handle it. The request will first come to FaxErrorhandler—if it finds that it is a fax issue, it'll handle the request; otherwise, it will forward the issue to EmailErrorHandler.

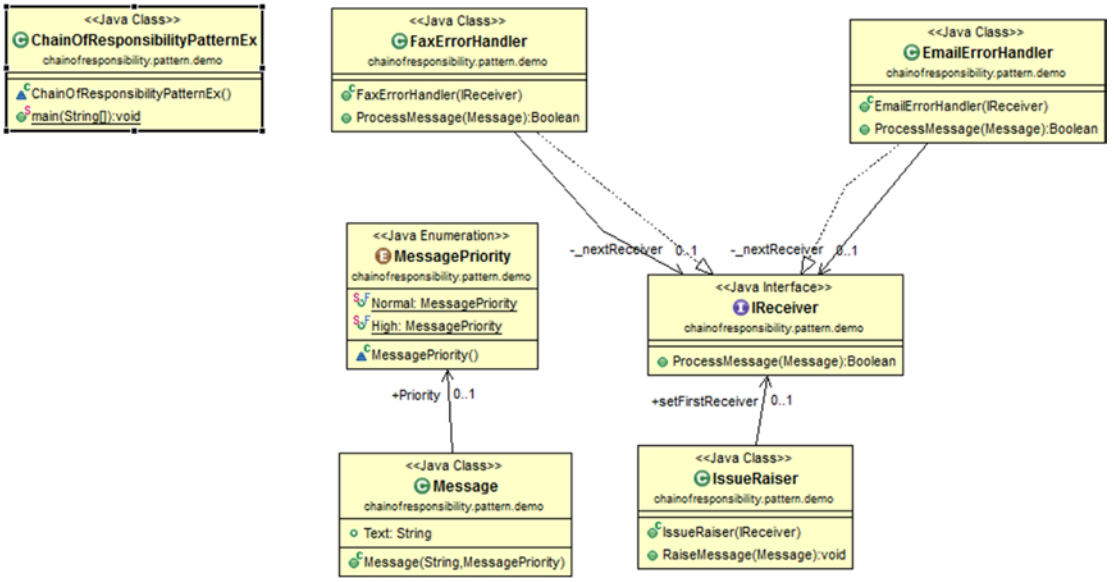
Note that here our chain ends with EmailErrorHandler. But if we need to handle another type of issue (e.g., Authentication), we can make an AuthenticationErrorHandler and put it after EmailErrorHandler. So, now, whenever the issue cannot be fixed by EmailErrorHandler, the issue can be forwarded to AuthenticationErrorHandler and the chain will end there.

Thus, the bottom line is as follows: the chain will end if the issue is being processed by some handler or there are no more handlers to process it (i.e., we have reached the end of the chain).

Illustration

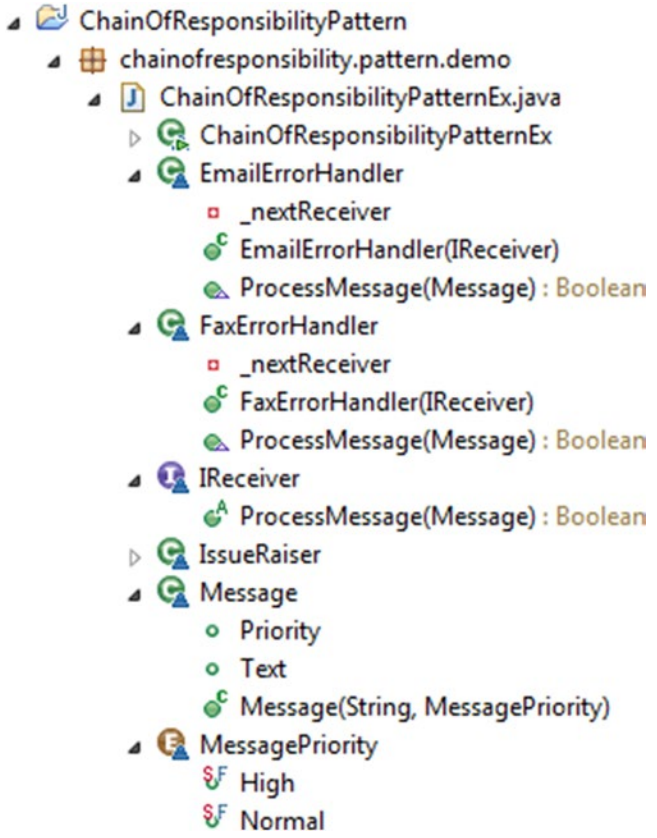
In this example, we are processing both normal and high-priority issues from e-mail and fax communications.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



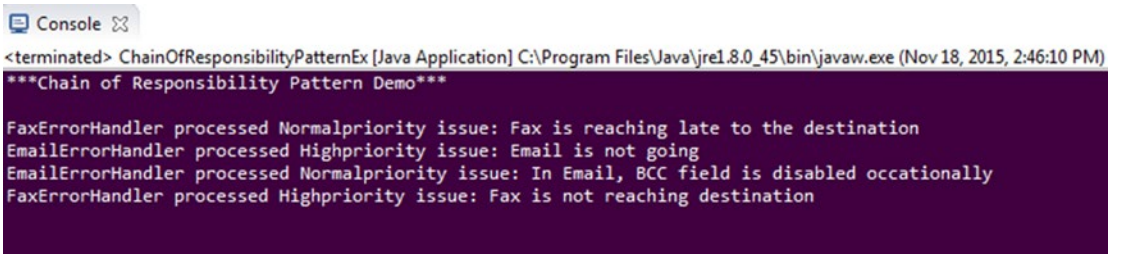
Implementation

```
package chainofresponsibility.pattern.demo;
enum MessagePriority
{
        Normal,
        High
}
class Message
{
        public String Text;
        public MessagePriority Priority;
        public Message(String msg, MessagePriority p)
```

```
        {
                Text = msg;
                this.Priority = p;
        }
}

interface IReceiver
{
        Boolean ProcessMessage(Message msg);
}
Class IssueRaiser
{
        public IReceiver setFirstReceiver;
        public IssueRaiser(IReceiver firstReceiver)
        {
                this.setFirstReceiver = firstReceiver;
        }
        public void RaiseMessage(Message msg)
        {
                if (setFirstReceiver != null)
                        setFirstReceiver.ProcessMessage(msg);
        }
}
class FaxErrorHandler implements IReceiver
{
        private IReceiver _nextReceiver;
        public FaxErrorHandler(IReceiver nextReceiver)
        {
                _nextReceiver = nextReceiver;
        }
        public Boolean ProcessMessage(Message msg)
        {
                if (msg.Text.contains("Fax"))
                {
                        System.out.println("FaxErrorHandler processed "+  msg.Priority+
                        "priority issue: "+ msg.Text);
                        return true;
                }
                else
                {
                        if (_nextReceiver != null)
                                _nextReceiver.ProcessMessage(msg);
                }
                return false;
        }
}
```

```
class EmailErrorHandler implements IReceiver
{
        private IReceiver _nextReceiver;
        public EmailErrorHandler(IReceiver nextReceiver)
        {
                _nextReceiver = nextReceiver;
        }
        public Boolean ProcessMessage(Message msg)
        {
                if (msg.Text.contains("Email"))
                {
                        System.out.println("EmailErrorHandler processed "+ msg.Priority+
                        "priority issue: "+ msg.Text);
                        return true;
                }
                else
                {
                        if (_nextReceiver != null)
                                _nextReceiver.ProcessMessage(msg);
                }
                return false;
        }
}
class ChainOfResponsibilityPatternEx
{
        public static void main(String[] args)
        {
                System.out.println("***Chain of Responsibility Pattern Demo***\n");
                //Making the chain first: IssueRaiser->FaxErrorhandler->EmailErrorHandler
                IReceiver faxHandler, emailHandler;
                //end of chain
                emailHandler = new EmailErrorHandler(null);
                //fax handler is before email
                faxHandler = new FaxErrorHandler(emailHandler);

                //starting point: raiser will raise issues and set the first handler
                IssueRaiser raiser = new IssueRaiser (faxHandler);

                Message m1 = new Message("Fax is reaching late to the destination",
                MessagePriority.Normal);
                Message m2 = new Message("Email is not going", MessagePriority.High);
                Message m3 = new Message("In Email, BCC field is disabled occasionally",
                MessagePriority.Normal);
                Message m4 = new Message("Fax is not reaching destination",
                MessagePriority.High);

                raiser.RaiseMessage(m1);
                raiser.RaiseMessage(m2);
                raiser.RaiseMessage(m3);
                raiser.RaiseMessage(m4);

        }
}
```

Output



Note

1. *This pattern is used when we issue a request without specifying the receiver. We expect any of our receivers to handle that request.*
2. There may be situation in which more than one receiver can handle the request but the receivers do not know the priority. However, we want to handle the request by the receiver based on the priority. This pattern can help us to design such a scenario.
3. We may need to have the capability to specify objects (that can handle a request) in runtime.
4. We can either define a new link or use an existing link when we need to implement a successor chain.
5. Sometimes we can try to implement an automatic mechanism for forwarding a request. *The advantage is that we can avoid implementing a specific forwarding mechanism* from one point to another point in our chain. Smalltalk's doesnotUnderstand is a typical example in this context.

CHAPTER 21

Composite Patterns

GoF Definition: Compose objects into tree structures to represent part-whole hierarchies. The composite pattern lets clients treat individual objects and compositions of objects uniformly.

Concept

This pattern can show part-whole hierarchy among objects. A client can treat a composite object just like a single object. In object-oriented programming, we make a composite object when we have many objects with common functionalities. This relationship is also termed a "has-a" relationship among objects.

Real-Life Example

We can think of any organization that has many departments, and in turn each department has many employees to serve. Please note that actually all are employees of the organization. Groupings of employees create a department, and those departments ultimately can be grouped together to build the whole organization.

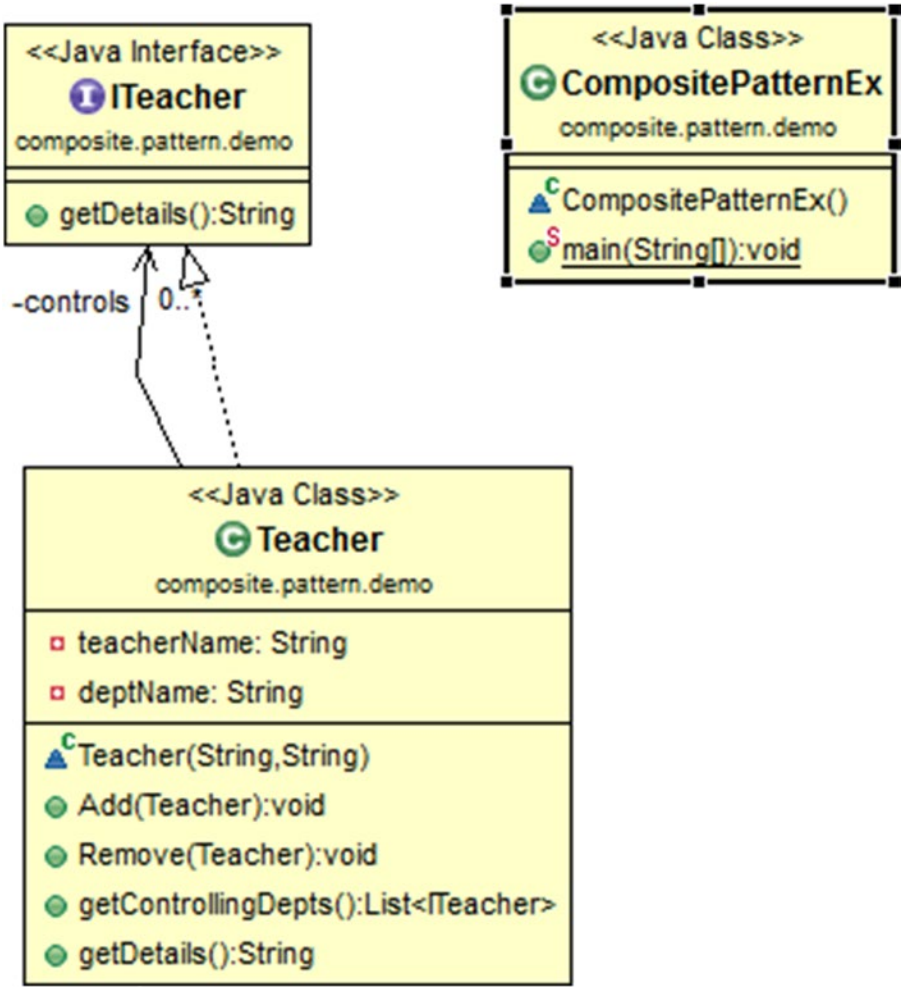
Computer World Example

Any tree structure in computer science can follow a similar concept.

Illustration

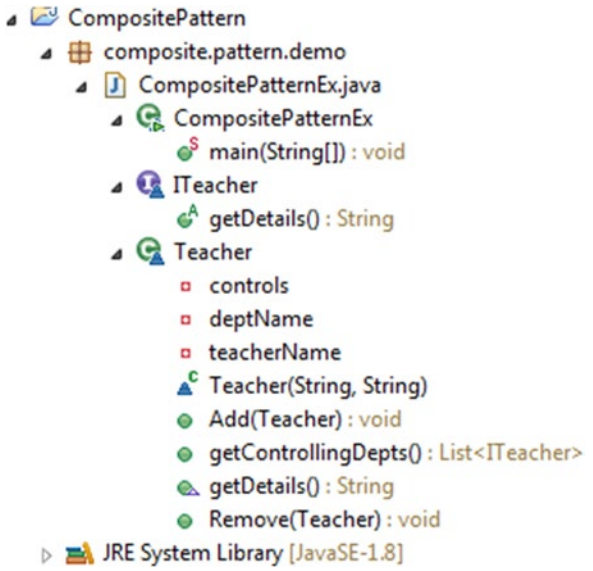
In this example we are showing a college organization. We have a Principal and two Heads of Departments: one for computer science and one for mathematics. At present, in the mathematics department, we have two lecturers; in the computer science department we have three lecturers. At the end, one lecturer from the computer science department retires/leaves. We have represented the scenario with the following simple example.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



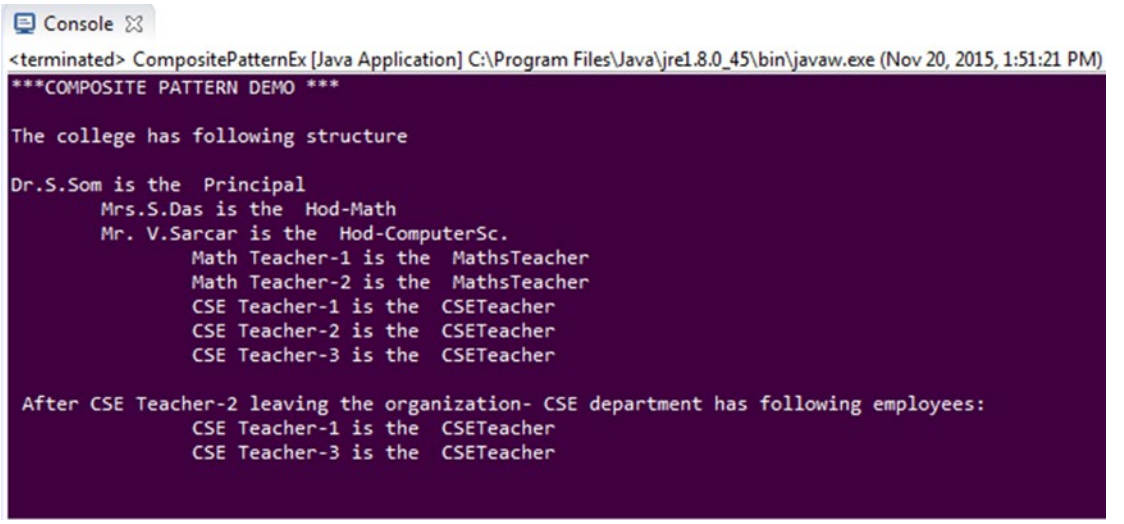
Implementation

```
package composite.pattern.demo;
import java.util.*;
interface ITeacher
{
        public String getDetails();
}
class Teacher implements ITeacher
{
        private String teacherName;
        private String deptName;
        private List<ITeacher> controls;

        Teacher(String teacherName, String deptName)
        {
                this.teacherName = teacherName;
                this.deptName = deptName;
                controls = new ArrayList<ITeacher>();
        }
        public void Add(Teacher teacher)
        {
                controls.add(teacher);
        }
```

```
        public void Remove(Teacher teacher)
        {
                controls.remove(teacher);
        }
        public List<ITeacher> getControllingDepts()
        {
                return controls;
        }
        @Override
        public String getDetails() {
                return (teacherName + " is the  " + deptName);
        }

}
class CompositePatternEx
{
        public static void main(String[] args)
        {
                Teacher Principal = new Teacher("Dr.S.Som","Principal");

                Teacher hodMaths = new Teacher("Mrs.S.Das","Hod-Math");

                Teacher hodCompSc = new Teacher("Mr. V.Sarcar","Hod-ComputerSc.");

                Teacher mathTeacher1 = new Teacher("Math Teacher-1","MathsTeacher");
                Teacher mathTeacher2 = new Teacher("Math Teacher-2","MathsTeacher");

                Teacher cseTeacher1 = new Teacher("CSE Teacher-1","CSETeacher");
                Teacher cseTeacher2 = new Teacher("CSE Teacher-2","CSETeacher");
                Teacher cseTeacher3 = new Teacher("CSE Teacher-3","CSETeacher");

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

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

                /*Teachers of Computer Sc. directly reports to HOD-Comp.Sc.*/
                hodCompSc.Add(cseTeacher1);
                hodCompSc.Add(cseTeacher2);
                hodCompSc.Add(cseTeacher3);

                /*Leaf nodes. There is no department under Mathematics*/
                mathTeacher1.Add(null);
                mathTeacher2.Add(null);
```

```
                /*Leaf nodes. There is no department under CSE.*/
                cseTeacher1.Add(null);
                cseTeacher2.Add(null);
                cseTeacher3.Add(null);

                //Printing the details
                System.out.println("***COMPOSITE PATTERN DEMO ***");
                System.out.println("\nThe college has following structure\n");
                System.out.println(Principal.getDetails());
                List<ITeacher> hods=Principal.getControllingDepts();
                for(int i=0;i<hods.size();i++)
                {
                        System.out.println("\t"+hods.get(i).getDetails());
                }

                List<ITeacher> mathTeachers=hodMaths.getControllingDepts();
                for(int i=0;i<mathTeachers.size();i++)
                {
                        System.out.println("\t\t"+mathTeachers.get(i).getDetails());
                }

                List<ITeacher> cseTeachers=hodCompSc.getControllingDepts();
                for(int i=0;i<cseTeachers.size();i++)
                {
                        System.out.println("\t\t"+cseTeachers.get(i).getDetails());
                }

                //One computer teacher is leaving
                hodCompSc.Remove(cseTeacher2);
                System.out.println("\n After CSE Teacher-2 leaving the organization- CSE
                department has following employees:");
                cseTeachers = hodCompSc.getControllingDepts();
                for(int i=0;i<cseTeachers.size();i++)
                {
                        System.out.println("\t\t"+cseTeachers.get(i).getDetails());
                }
        }
}
```

Output



Note

1. The pattern is ideal to represent the part-whole hierarchy among objects.
2. Here the client can treat composition of objects like a single object.
3. Clients can add new types of component easily.
4. If we are forced to maintain child ordering (e.g., parse trees as components), we need to take special care to maintain that order.

What is the best data structure to store components?

There is no universal rule. It depends on the requirement (e.g., efficiency). We can use linked list, trees, arrays, etc., based on our demand. GoF also suggests that it is not mandatory to use any general-purpose data structure.

CHAPTER 22

Bridge Patterns (Or Handle/Body Patterns)

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

Concept

In this pattern, the abstract class is separated from the implementation class and we provide a bridge interface between them. This interface helps us to make concrete class functionalities independent from the interface implementer class. We can alter these different kind of classes structurally without affecting each other.

Real-Life Example

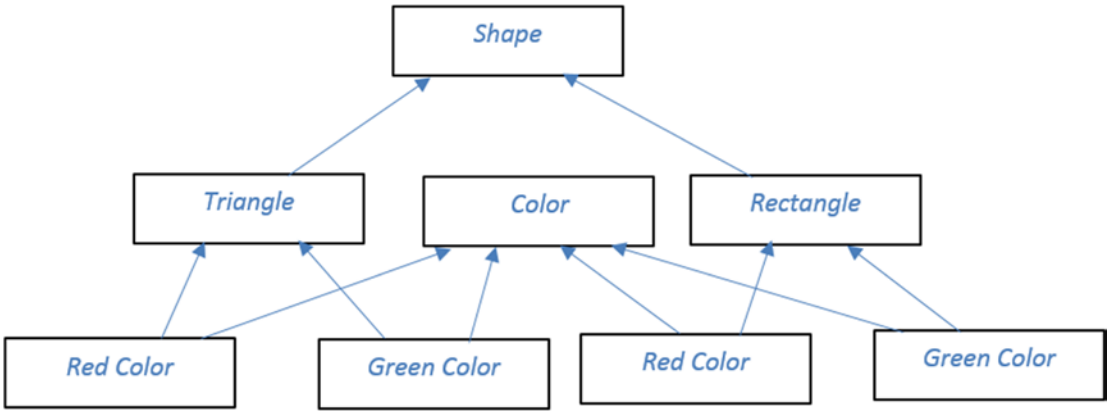
In a software product development company, the development team and technical support team both play a crucial role. A change in the operational strategy of one team should not have a direct impact on the other team. Here the technical support team plays the role of a bridge between the clients and the development team that implements the product.

Computer World Example

This pattern is used in a GUI framework. It separates Window abstraction from Window implementation in Linux/Mac OSs. The following illustration is a classical example in the software development field.

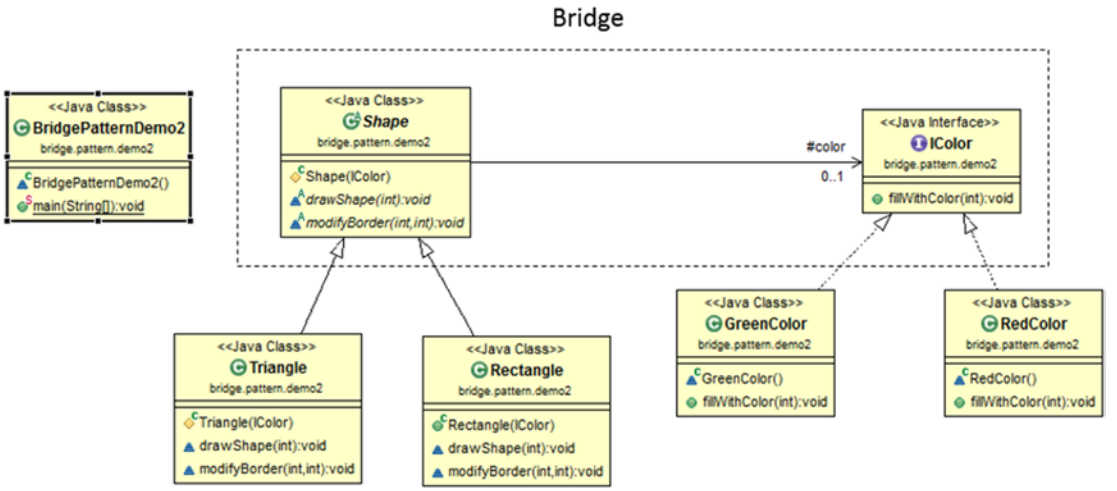
Illustration

Consider a situation like this:



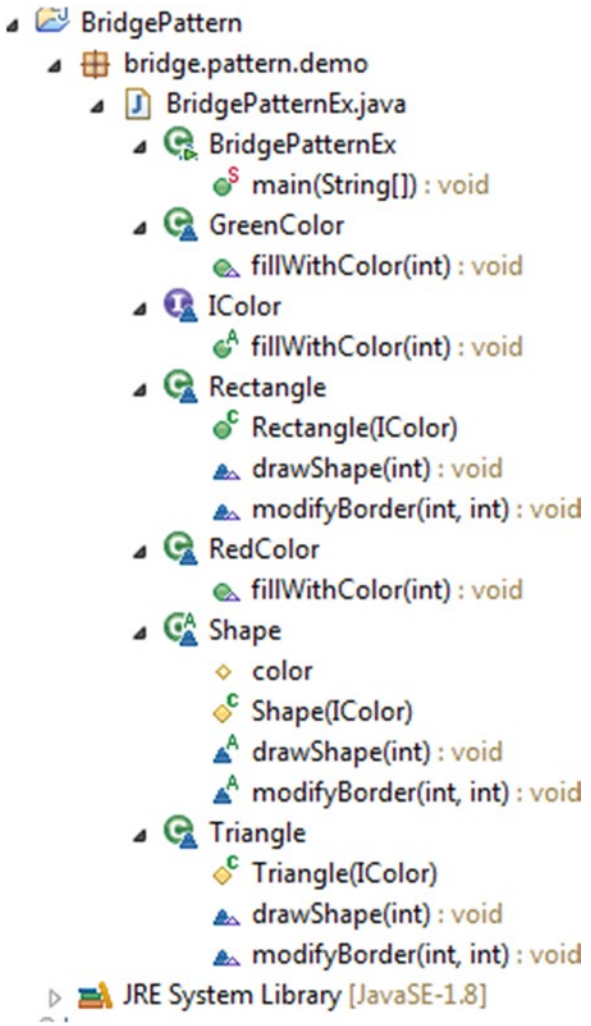
We'll use bridge pattern to decouple the interfaces in our example from the implementations. After our implementation it will have a cleaner look (follow our UML diagram).

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



Implementation

Here we have implemented both an abstraction-specific and an implementer-specific method to represent the power and usefulness of this pattern. We can draw a triangle and a rectangle with a particular color with the implementer-specific method drawShape(). We can change the thickness of the border by the abstraction-specific method modifyBorder(). Please go through the code.

```
package bridge.pattern.demo;

//Colors-The Implementer

interface IColor
{
        void fillWithColor(int border);
}
class RedColor implements IColor
{
        @Override
        public void fillWithColor(int border)
        {
                System.out.print("Red color with " +border+" inch border");

        }
}
class GreenColor implements IColor
{
        @Override
        public void fillWithColor(int border)
        {
                System.out.print("Green color with " +border+" inch border.");

        }
}

//Shapes-The Abstraction

abstract class Shape
{
        //Composition
        protected IColor color;
        protected Shape(IColor c)
        {
                this.color = c;
        }
        abstract void drawShape(int border);
        abstract void modifyBorder(int border,int increment);
}

class Triangle extends Shape
{
        protected Triangle(IColor c)
        {
                super(c);
        }
        //Implementer-specific method
        @Override
```

```
        void drawShape(int border) {
                System.out.print(" This Triangle is colored with: ");
                color.fillWithColor(border);
        }
        //Abstraction-specific method
        @Override
        void modifyBorder(int border,int increment) {
                System.out.println("\nNow we are changing the border length "+increment+ "
                times");
                border=border*increment;
                drawShape(border);
        }
}

class Rectangle extends Shape
{
        public Rectangle(IColor c)
        {
                super(c);
        }
        //Implementer-specific method
        @Override
        void drawShape(int border)
        {
                System.out.print(" This Rectangle is colored with: ");
                color.fillWithColor(border);
        }
        //Abstraction-specific method
        @Override
        void modifyBorder(int border,int increment) {
                System.out.println("\n Now we are changing the border length "+increment+ "
                times");
                border=border*increment;
                drawShape(border);
        }
}

class BridgePatternEx
{
    public static void main(String[] args)
    {
            System.out.println("*****BRIDGE PATTERN*****");
                //Coloring Green to Triangle
                System.out.println("\nColoring Triangle:");
                IColor green = new GreenColor();
                Shape triangleShape = new Triangle(green);
                triangleShape.drawShape(20);
                triangleShape.modifyBorder(20, 3);

                //Coloring Red to Rectangle
                System.out.println("\n\nColoring Rectangle :");
                IColor red = new RedColor();
```

```
                Shape rectangleShape = new Rectangle(red);
                rectangleShape.drawShape(50);
                //Modifying the border length twice
                rectangleShape.modifyBorder(50,2);
    }
}
```

Output

Note

1. The pattern is extremely helpful when our class and its associated functionalities may change in frequent intervals.
2. Here we remove the concrete binding between an abstraction and the corresponding implementation. As a result, both hierarchies (abstraction and its implementations) can extend through child classes.
3. Both hierarchies can grow independently. Here if we make any change in abstraction methods, they do not have an impact on the implementer method (i.e., fillWithColor()).

We have repeatedly referred here to the two hierarchies: abstraction and implementer. An abstraction should be an abstract class." Is the statement correct?

No. We can use either an abstract class or an interface. And the same rule applies for the implementer class also.

What are refined abstractions?

Children of an abstraction are termed "refined abstractions."

Who are concrete implementers?

Children of an implementer.

How can you differentiate an abstraction from its implementer?

In general, an abstraction contains the reference to its implementer.

How can you change the implementers dynamically or at runtime?

By changing the reference in the abstraction.

CHAPTER 23

Visitor Patterns

GoF Definition: Represent an operation to be performed on the elements of an object structure. The visitor pattern lets you define a new operation without changing the classes of the elements on which it operates.

Concept

This pattern helps us to add new functionalities to an existing object structure in such a way that the old structure remains unaffected by these changes. So, we can follow the open/close principle here (i.e., extension allowed but modification disallowed for entities like class, function, modules, etc.).

Real-Life Example

Consider a taxi booking scenario. The taxi arrives at our defined location for the pickup. Once we enter into it, the visiting taxi takes control of the transportation. It can choose a different way toward our destination and we may or may not have any prior knowledge of that way.

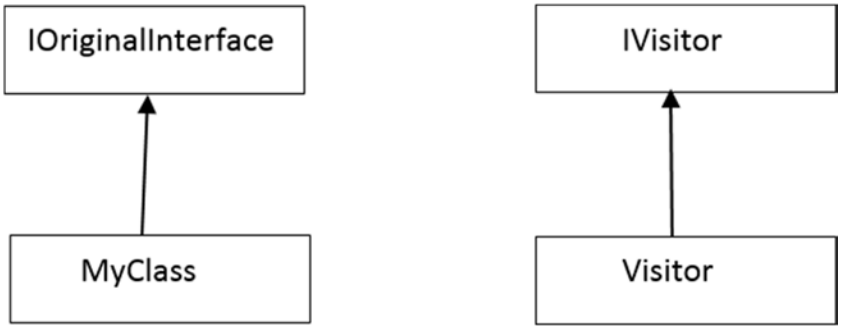
Computer World Example

This pattern is very useful when plugging happens into public APIs. Clients can then perform operations on a class with a visiting class without modifying the source.

Plugging into public APIs is a common example. Then a client can perform his desired operations without modifying the actual code (with a visiting class).

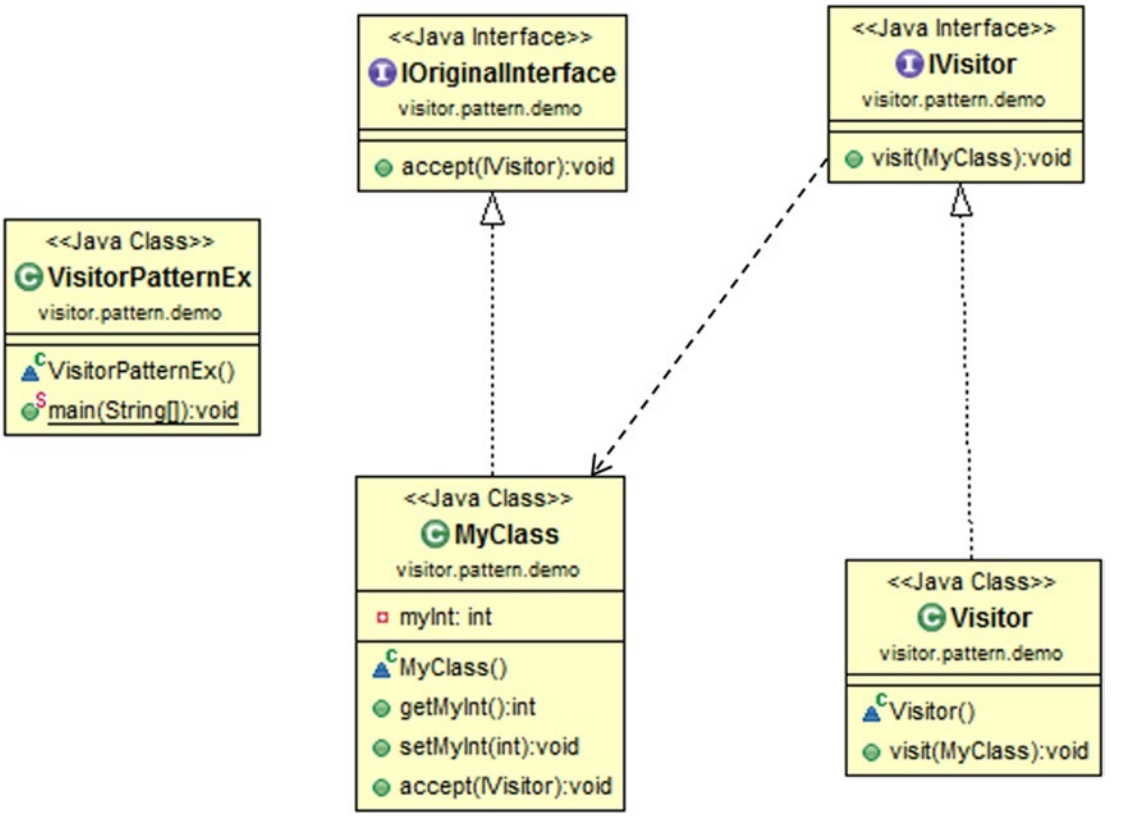
Illustration

Here we have illustrated a simple example to represent a visitor pattern. In order to do this, we have implemented a new class hierarchy (IVisitor hierarchy) and we have implemented the algorithms there. So, any modification/update operation in the IOriginalInterface hierarchy can be done through this new class hierarchy without affecting the code in the IOriginalInterface hierarchy.



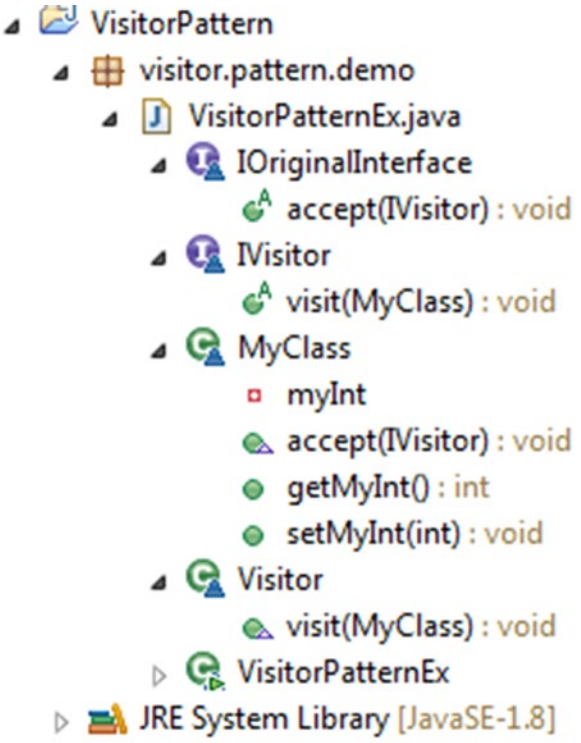
In the following example, we want to modify the initial integer value in MyClass (which implements the Interface IOriginalInterface) through the visitor pattern. Note that we are not touching the code in IOriginalInterface. We are separating functionality implementations (i.e., algorithms) from the class hierarchy where these algorithms operate.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:

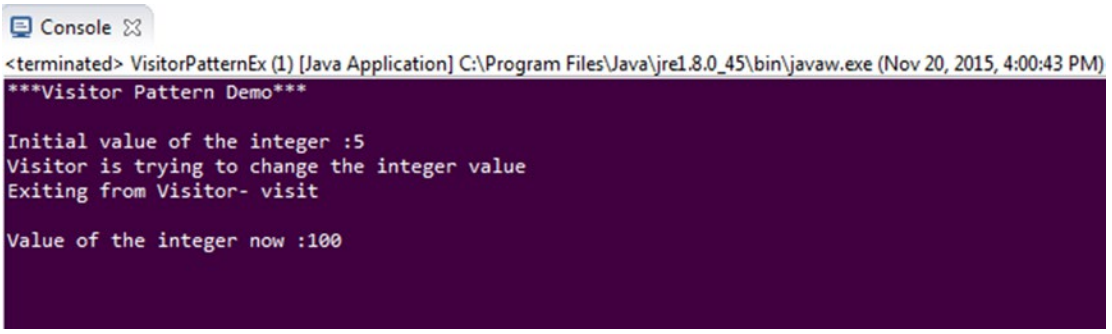


Implementation

```
package visitor.pattern.demo;

interface IOriginalInterface
{
        void accept(IVisitor visitor);
}
class MyClass implements IOriginalInterface
{
        //Initial or default value
        private int myInt = 5;
        public int getMyInt()
        {
                return myInt;
        }
```

```
        public void setMyInt(int myInt)
        {
                this.myInt = myInt;
        }

        @Override
        public void accept(IVisitor visitor)
        {
                System.out.println("Initial value of the integer :"+ myInt);
                visitor.visit(this);
                System.out.println("\nValue of the integer now :"+ myInt);
        }
}

interface IVisitor
{
        void visit(MyClass myClassElement);
}
class Visitor implements IVisitor
{
        @Override
        public void visit(MyClass myClassElement)
        {
                System.out.println("Visitor is trying to change the integer value");
                myClassElement.setMyInt(100);
                System.out.println("Exiting from Visitor- visit");
        }
}
class VisitorPatternEx
{
        public static void main(String[] args)
        {
                System.out.println("***Visitor Pattern Demo***\n");
                IVisitor v = new Visitor();
                MyClass myClass = new MyClass();
                myClass.accept(v);
        }
}
```

Output



Note

1. As mentioned earlier, the visitor pattern is very useful for adding new operations without affecting the existing structure, which was the key aim behind this pattern.
2. Visitor operations are controlled in a unified manner.
3. On the other hand, the class encapsulation may need to be compromised when visitors are used. If the existing structure is really complex, the traversal mechanism becomes complex.
4. The visitor hierarchy becomes difficult to maintain when we need to add new concrete classes to our existing architecture frequently (e.g., in our program, if we now add Myclass2, we need to add additional operations in the visitor class hierarchy to support this pattern).
5. Sometimes we need to perform some unrelated operations on the objects in the existing architecture. But these operations can directly/indirectly affect the classes in the system. In those situations, this pattern can help us by putting all of these operations in the visitor hierarchy.

CHAPTER 24

Interpreter Patterns

GoF Definition: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Concept

Here, in general, we define a grammatical representation for a language and provide an interpreter to deal with that grammar (e.g., in our example we have interpreted a string input as binary data). In simple words, this pattern says how to evaluate sentences in a language.

Real–Life Example

A language translator who translates a language for us provides a classic example for this pattern. Or, we can also consider music notes as our grammar and musicians as our interpreters.

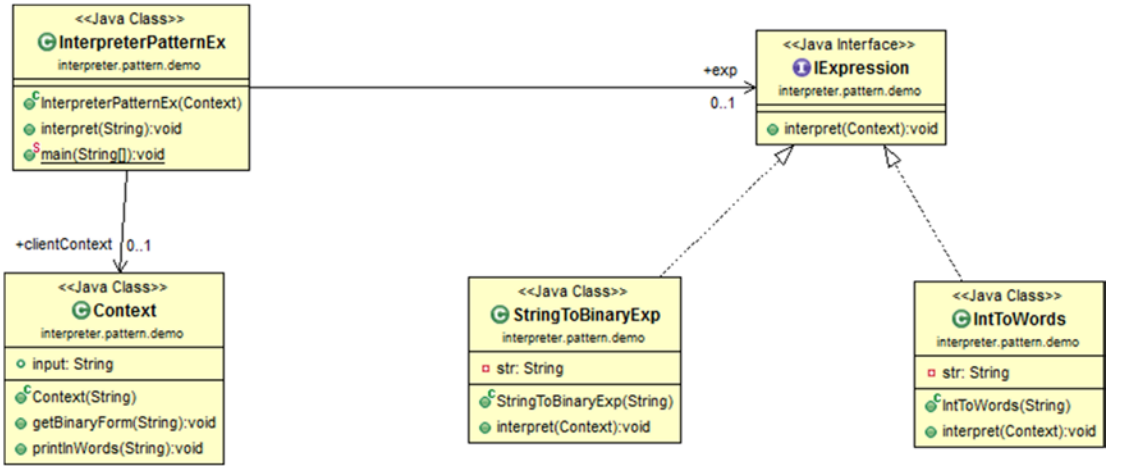
Computer World Example

A Java compiler interprets the source code into byte code. This byte code is understandable by JVM (Java virtual machine). In C# also, our source code is converted to MSIL (Microsoft intermediate language) code, which is interpreted by CLR (common language runtime). Upon execution, this MSIL (intermediate code) is converted to native code (binary executable code) by a JIT (Just In time) compiler.

Illustration

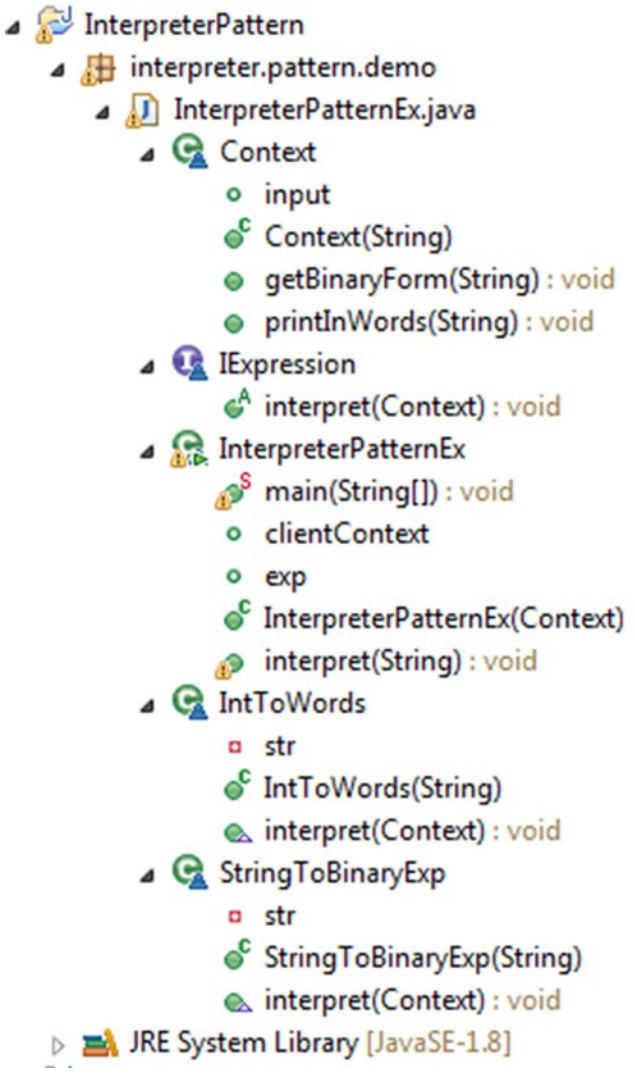
Here we need to create an interpreter context engine for our interpretation work. Then we need to create an expression implementation (ideally we should have more implementations—e.g., here we may also be interested in converting the data to hexadecimal or any other format) that will consume the functionality provided by the interpreter context. At last we have created the client who will accept the user input to generate the required output. The client here will also decide which expression to use if we have more than one expression. Go through the example now. Here we have two choices: we can interpret an input string (basically a decimal number) as binary data or we can simply interpret it and print the digits in the input into their equivalent English words.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:



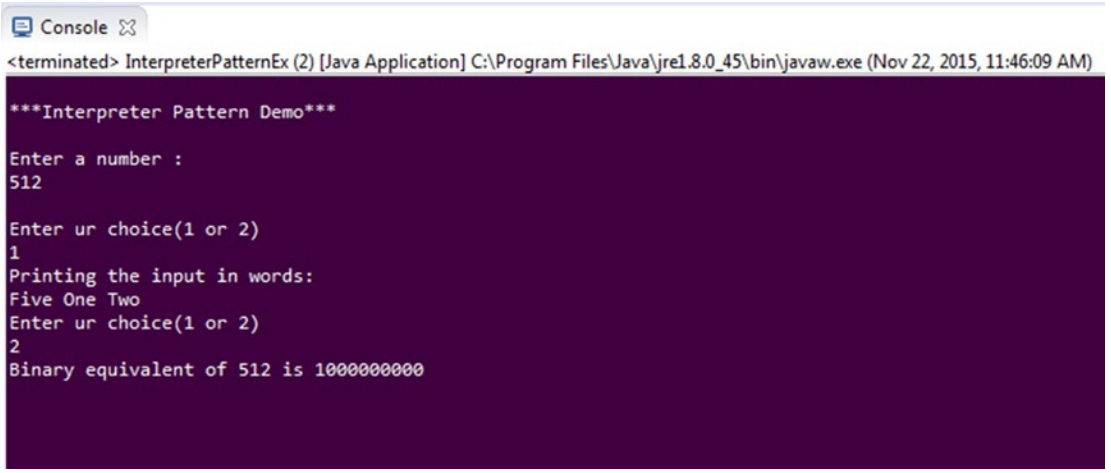
Implementation

```
package interpreter.pattern.demo;

import java.util.Scanner;

/*Context class: interpretation is carried out based on our implementation.*/
class Context
{
        public String input;
        public Context(String input)
        {
                this.input=input;
        }
        public void getBinaryForm(String input)
        {
                int i = Integer.parseInt(input);
                //integer to its equivalent binary string representation
                String binaryString = Integer.toBinaryString(i);
                System.out.println("Binary equivalent of "+input+ " is "+ binaryString);
        }
        public void printInWords(String input)
        {
                this.input = input;
                System.out.println("Printing the input in words:");
                char c[]=input.toCharArray();
                for(int i=0;i<c.length;i++)
                {
                        switch (c[i])
                        {
                        case '1':
                                System.out.print("One ");
                                break;
                        case '2':
                                System.out.print("Two ");
                                break;
                        case '3':
                                System.out.print("Three ");
                                break;
                        case '4':
                                System.out.print("Four ");
                                break;
                        case '5':
                                System.out.print("Five ");
                                break;
                        case '6':
                                System.out.print("Six ");
                                break;
                        case '7':
                                System.out.print("Seven ");
                                break;
```

```
                        case '8':
                                System.out.print("Eight ");
                                break;
                        case '9':
                                System.out.print("Nine ");
                                break;
                        case '0':
                                System.out.print("Zero ");
                                break;
                        default:
                                System.out.print("* ");
                                break;
                        }
                }
        }
}

interface IExpression
{
        void interpret(Context ic);
}
class StringToBinayExp implements IExpression
{
        private String str;
        public StringToBinaryExp(String s)
        {
                str = s;
        }

        @Override
        public void interpret(Context ic)
        {
                ic.getBinaryForm(str);
        }
}
class IntToWords implements IExpression
{
        private String str;
        public IntToWords(String str)
        {
                this.str = str;
        }
        @Override
        public void interpret(Context ic)
        {
                ic.printInWords(str);
        }
}
```

```
class InterpreterPatternEx
{
        public Context clientContext=null;
        public IExpression exp=null;
        public InterpreterPatternEx(Context c)
        {
                clientContext = c;
        }
        public void interpret(String str)
        {
                //We'll test 2 consecutive inputs at a time
                for(int i=0;i<2;i++){
                        System.out.println("\nEnter ur choice(1 or 2)");
                        Scanner in = new Scanner(System.in);
                        String c = in.nextLine();
                        if (c.equals("1"))
                        {
                                exp = new IntToWords(str);
                                exp.interpret(clientContext);
                        }
                        else
                        {
                                exp = new StringToBinaryExp(str);
                                exp.interpret(clientContext);
                        }
                }
        }

        public static void main(String[] args)
        {
                System.out.println("\n***Interpreter Pattern Demo***\n");
                System.out.println("Enter a number :");
                Scanner in = new Scanner(System.in);
                String input = in.nextLine();
                                Context context=new Context(input);
                InterpreterPatternEx client = new InterpreterPatternEx(context);
                client.interpret(input);
        }
}
```

Output



Note

1. This pattern is widely used to interpret the statements in a language as abstract syntax trees. It performs best when the grammar is easy to understand and simple.
2. We can represent, modify, or implement a grammar easily.
3. We can evaluate an expression in our preferred ways. It is up to us how we'll interpret those expressions.
4. *If the grammar is complex (e.g., it may have many constraints/rules), implementing this pattern becomes hard. For each rule, we may need to implement a new class, and obviously it is a cumbersome process.*

APPENDIX A

FAQ

Which of these patterns is/are best?

There is no straightforward answer for this type of question. It depends on many factors (the situation, the demand, the constraints, etc.). But if you know all of these patterns, you will have the flexibility to decide. In real life, it is quite possible that we need to use a combination of these patterns to design a requirement.

Why should we use design patterns?

These are general reusable solutions for software design problems which we see repeatedly in real-world software development. They help us to avoid unnecessary and scattered implementations. Please refer to the section "Introduction" in Chapter 1 for the detailed answer.

What is the difference between a command and a memento pattern?

All actions are stored for the command pattern, but the memento pattern saves the state only on request.

In general, in the command pattern, we frequently see operations like undo's and redo's for every action, but the memento pattern does not need that.

What is the difference between the facade pattern and the builder pattern?

The aim of the facade pattern is to make a specific portion of code easier for use. It abstracts details away from the developer.

The builder pattern splits the construction process of an object from the representation. (See our code.) Our Director is calling the same Construct() method to create different types of vehicles (i.e., we can use the same construction process to create multiple types).

What is the difference between the builder pattern and the strategy pattern? It appears that they have similar UML representation.

First of all we must take care of the intent first. The builder pattern falls into the creational pattern and the strategy pattern falls into the behavioral pattern. Their areas of focus are different. With the builder pattern we can use the same construction process to create multiple types and with strategy pattern, we have the freedom to select an algorithm in runtime.

What is the difference between the command pattern and the interpreter pattern?

For the command pattern, commands are basically objects. On the other hand, for the interpreter pattern the commands are sentences. With interpreter, we try to evaluate an expression in an easy manner. Sometimes the interpreter pattern looks convenient, but we must note that the cost of building an interpreter may be significant if the grammar is complex—because we need a new class to evaluate a new rule in the grammar.

What is the difference between the chain of responsibility pattern and the observer pattern?

For observer patterns, all registered observers will be notified/get request (for the change in subject), but in the chain of responsibility, it is possible that we do not need to reach to the end of chain, so all processing objects need not to handle the same scenario. The request can be processed much earlier by some processing object that is placed at the beginning of the chain.

What is the difference between the mediator pattern and the observer pattern?

GOF also mentioned: *These are competing patterns. The difference between them is that observer distributes communication by introducing observer and subject objects, whereas a mediator object encapsulates the communication between other objects*. Here I'll request you to consider our example of the mediator pattern. Two workers are always getting messages from their boss. It doesn't matter whether they like those messages or not. But if they are simple observers, they should have an option to unregister their boss's control on them, saying "I do not want to see messages from the boss."

GoF also found that we may face fewer challenges to make reusable observers and subjects than to make reusable mediators, but if we try to understand the flow of communication, the mediator scores higher than the observer.

Consider a situation. You have already implemented an interpreter pattern. Later you found that you need to incorporate an additional way to interpret a special type of expression. What will your preferred option be?

We can combine the power of visitor patterns here. It will help us not to disturb the existing grammar classes.

Can we combine the iterator pattern with the composite pattern?

Yes. To design a recursive structure, this combination is quite common.

In which design pattern must opaque objects be present?

In the memento pattern. Memento itself is an opaque object. Remember that the caretaker is not allowed to make any change there.

What is the difference between the command pattern and the chain of responsibility pattern?

With the chain of responsibility, a request is forwarded to a chain with an expectation that one node of that chain will handle that request. However, we have no idea about who is going to handle that request. But with the command pattern, a request will go to an intended receiver (object).

What is the difference between a singleton class and a static class?

With a singleton class we can create objects. This is the main difference. And it means that we can use the concepts of inheritance and polymorphism (by extending the base class) with a singleton class. Experts also believe that static classes are not easy to mock and test.

Note: We cannot override the static methods in Java, but these methods are bounded during compile time, so in certain situations, a static class that consists of static methods can perform better than a singleton class with several non-static methods.

References

- Design Patterns–Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995
- Design Patterns in C# by Vaskaran Sarcar, 2015
- Design Patterns in C# by Jean Paul V.A, 2012

- Java Design Patterns: A Tutorial by James W. Cooper, Addison Wesley, 2000
- www.youtube.com/watch?v=ffQZIGTTM48&list=PL8C53D99ABAD3F4C8
- www.dofactory.com/net/decorator-design-pattern
- www.c-sharpcorner.com/UploadFile/40e97e/adapter-pattern-in-C-Sharp/
- www.dotnet-tricks.com/Tutorial/designpatterns/1N4c140713-Facade-Design-Pattern---C
- www.dotnet-tricks.com/Tutorial/designpatterns/FUcV280513-Factory-Method-Design-Pattern---C
- www.codeproject.com/Articles/29036/Patterns-in-Real-Life
- www.youtube.com/watch?v=CuyzHO-Nx14
- www.youtube.com/watch?v=Y3xP2YSJ2JQ
- www.youtube.com/watch?v=9jIgSsIfh_8
- www.youtube.com/watch?v=S8XL1L_1Lyw
- www.dofactory.com/net/memento-design-pattern
- http://en.wikipedia.org/wiki/Memento_pattern
- http://sourcemaking.com/design_patterns/builder/c-sharp-dot-net
- www.dofactory.com/net/builder-design-pattern
- www.dreamincode.net/forums/topic/185616-design-patterns-state/
- https://weblogs.java.net/blog/ryano/archive/2005/01/implementing_th.html
- http://sourcemaking.com/design_patterns/State/c-sharp-dot-net
- http://en.wikipedia.org/wiki/Flyweight_pattern
- www.c-sharpcorner.com/UploadFile/f6c0e9/flyweight-pattern/
- www.dofactory.com/net/flyweight-design-pattern
- www.tutorialspoint.com/design_pattern/flyweight_pattern.htm
- www.dotnetexamples.com/2013/07/flyweight-design-pattern.html
- www.dotnet-tricks.com/Tutorial/designpatterns/cWHV140713-Flyweight-Design-Pattern---C
- www.dotnet-tricks.com/Tutorial/designpatterns/4EJN020613-Abstract-Factory-Design-Pattern---C
- www.expertbloggingon.net/post/2013/06/21/CSharp-Abstract-Factory-Pattern-Design-Patterns-in-Action.aspx
- http://en.wikipedia.org/wiki/Abstract_factory_pattern#Structure
- www.c-sharpcorner.com/UploadFile/851045/command-design-pattern-in-C-Sharp/
- http://sourcemaking.com/design_patterns/mediator
- www.dofactory.com/javascript/mediator-design-pattern

- http://en.wikipedia.org/wiki/Mediator_pattern
- http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern
- http://en.wikipedia.org/wiki/Composite_pattern
- www.tutorialspoint.com/design_pattern/composite_pattern.htm
- www.dofactory.com/net/composite-design-pattern
- www.tutorialspoint.com/design_pattern/bridge_pattern.htm
- www.oodesign.com/bridge-pattern.html
- www.codeproject.com/Articles/890/Bridge-Pattern-Bridging-the-gap-between-Interface
- www.journaldev.com/1491/bridge-pattern-in-java-example-tutorial
- www.youtube.com/watch?v=UYUopyMcWjw
- http://en.wikipedia.org/wiki/Visitor_pattern
- http://java.dzone.com/articles/design-patterns-visitor
- www.journaldev.com/1635/interpreter-design-pattern-in-java-example-tutorial
- http://cplus.about.com/od/introductiontoprogramming/p/profileofcsh.htm
- http://java.dzone.com/articles/design-patterns-uncovered-14
- http://stackoverflow.com/questions/2760843/differences-between-facade-pattern-and-other-patterns
- www.codeproject.com/Articles/430590/Design-Patterns-of-Creational-Design-Patterns
- http://www.coderanch.com/t/151598/java-Architect-SCEA/certification/Builder-Strategy-Patterns
- http://stackoverflow.com/questions/11344814/why-java-lang-object-can-not-be-cloned
- http://javarevisited.blogspot.in/2012/10/java-program-to-get-input-from-user.html
- www.journaldev.com/1377/java-singleton-design-pattern-best-practices-with-examples#enum-singleton
- www.codeproject.com/Tips/468254/Proxy-Design-Pattern-in-Java
- https://en.wiktionary.org/wiki/smart-reference_proxy
- http://java67.blogspot.in/2013/05/difference-between-deep-copy-vs-shallow-cloning-java.html
- http://www.programcreek.com/2013/02/java-design-pattern-iterator/
- https://carldanley.com/js-facade-pattern/
- www.quora.com/What-are-the-pros-and-cons-of-the-factory-design-pattern
- https://javarevealed.wordpress.com/2013/08/12/builder-design-pattern/

- http://stackoverflow.com/questions/2829106/disadvantages-of-builder-design-pattern
- http://stackoverflow.com/questions/5467005/adapter-pattern-class-adapter-vs-object-adapter
- www.avajava.com/tutorials/lessons/bridge-pattern.html
- http://stackoverflow.com/questions/11722352/why-do-we-need-the-visit-method-in-the-visitor-design-pattern
- www.vainolo.com/2012/07/30/the-visitor-design-pattern-with-sequence-diagrams/

Index

G, H

I, J, K, L

M, N

O

P, Q, R

S

T, U

V, W, X, Y, Z