Java Institute for Advanced Technology

UNIT NAME: OBJECT ORIENTED DESIGN PATTERN I

UNIT ID: HF2P 04

ASSESSMENT NAME: RESEARCH ASSIGNMENT

NAME: M.R.P.N.THARUKSHA RAJAPAKSHA

STUDENT ID: 2019/2020/CO/SE/I2/029

SCN NO: 207977608

NIC: 200019401866

BRANCH: JAVA INSTITUTE, COLOMBO

Contents

1. Introduction to Object Oriented Programming

As the name implies, Object Oriented Programming or OOPs are languages that use objects in programming. Object oriented programming aims to implement real-world entities such as inheritance, hiding, polymorphism, etc. in programming. The main purpose of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Object Oriented Programming provides several concepts. They are called OOPs Concepts (Object Oriented Programming Concepts).

The OOPs Concepts are:

- **Class** - Class is a user-defined data type that has data and methods. A class can be accessed by creating an instance (object) from that class. Class is the blueprint of an object. Using that one can create as many objects as they need.

- **Objects** - Objects are the primary component of object oriented programming. The object is an instance of a class that is in computer memory. Objects can interact with other objects.

- **Abstraction** - Abstractions can be used to hide unnecessary details about code and implementation from the outside world.

- **Encapsulation** - Variables in a class can be hidden and stop direct access from other classes. Variables can only be accessed with getter and setter methods in that class. This is known as encapsulation.

- **Inheritance** - A class can inherit the properties and behaviors of its parent class. Because of this concept programmers don't have to write the same properties again in all the needed classes.

- **Polymorphism** - It is the ability to display a message in many forms.

- **Dynamic Binding** - The code that needs to be executed in a method call will be decided at runtime.

- **Message Passing** - Message parsing is a communication method in object oriented programming that is used by objects to communicate with one another specifying the object name, function, and the information to be sent. Is done in message parsing.

2. Evaluation of Object Oriented Design Patterns

In Object Oriented Programming, a design pattern is a solution that is being provided for the set of commonly occurring errors while programming by the programmers. It is a general repeatable solution to a problem that is occurring commonly in software design. A design pattern is not a design that is fully completed and is not capable of being directly being implemented in the relevant section of the program. It is a template that is designed on illustrating the way to solve the issues that occur while programming. So, after understanding this we can implement the code according to the way we want and solve the issue of the program. There are several design patterns that are being designed in order to provide solutions for various errors that occur while coding. All the Object Oriented Design Patterns can be mainly categorized into 3 major sections according to the way how to solutions are being provided. Those 3 major design pattern types are,

- Behavioral Design Pattern
- Creational Design Pattern
- Structural Design Pattern

2.1 Behavioral Design Pattern

The behavioral design pattern is the third main type of all the other design pattern types in Object Oriented Programming. This design pattern is frequently used in the instances when two objects are being executed mutually with each other in a specific Object Oriented program. There is also a list of design patterns that are included in this type that can be used according to the relevant situation. Those types are as follows,

- Chain of responsibility Design Pattern
- Command Design Pattern
- Interpreter Design Pattern
- Iterator Design Pattern
- Mediator Design Pattern
- Memento Design Pattern
- Observer Design Pattern
- State Design Pattern
- Strategy Design Pattern
- Template Method Design Pattern
- Visitor Design Pattern

When we consider the Object Oriented Design Patterns overall, there is a list of benefits that we gain by using these patterns in coding. Some of those key benefits are mentioned as follows,

- Helps in speeding up the process of development by providing the development paradigms that are being well tested and proven.
- The reusing of these Object Oriented Design Patterns helps on preventing minor flaws/errors before becoming the errors that are large concerns, also these patterns provide another benefit on improving the readability status of the relevant code.
- This design pattern is a set of general answers that are documented in a way that isn't specific to a given situation.
- The Object Oriented Design Patterns also enable programmers to express well-known, well-understood names for software interactions, and they can be refined over time, making them more robust than ad-hoc design.

2.2 Creational Design Pattern

This type of design pattern is an Object Oriented Design Pattern type which mainly provides solutions for the errors that are occurred while creating an Object in Object Oriented Programming. There is a list of design patterns that are included within this section. They are,

- Builder Design Pattern
- Factory Design Pattern
- Prototype Design Pattern
- Singleton Design Pattern

2.3 Structural Design Pattern

This type of design pattern is an Object Oriented Design Pattern type which mainly provides solutions for the errors that are occurred while creating relationships in Object Oriented Programming. There is a list of design patterns that are included in this structural design pattern type. They are,

- Adapter Design Pattern
- Bridge Design Pattern
- Composite Design Pattern
- Decorator Design Pattern
- Facade Design Pattern
- Flyweight Design Pattern
- Proxy Design Pattern

3. Explanation of Object Oriented Design Patterns

In Object Oriented Design Patterns 1 Subject we have discussed about the following Design Patterns,

- Singleton Design Pattern
- Adapter Design Pattern
- Strategy Design Pattern
- Facade Design Pattern
- Iterator Design Pattern
- Template Design Pattern
- Observer Design Pattern
- Command Design Pattern
- State Design Pattern
- Factory Design Pattern
- Proxy Design Pattern

3.1 Singleton Design Pattern

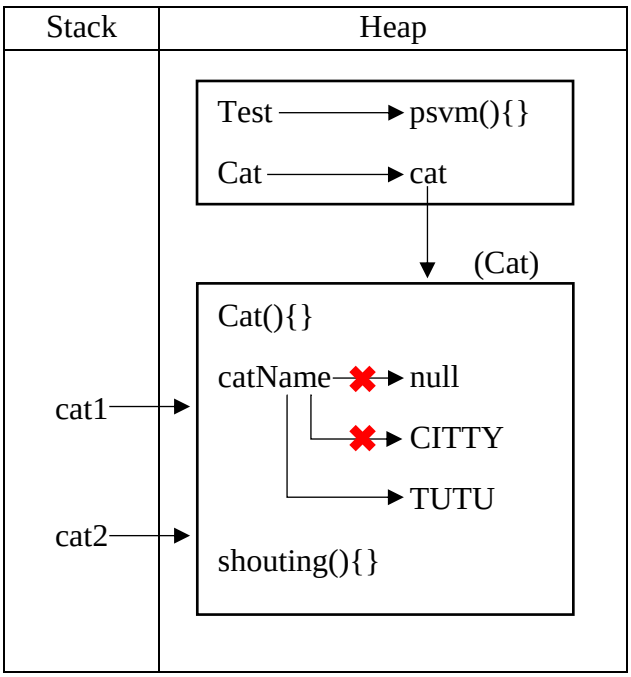
The Singleton Design Pattern is one of the simplest design patterns in Java. This type of design pattern falls under the creational pattern as it provides one of the best ways to create an object. This pattern includes the single class responsible for creating an object and ensures that only a single object is created. This class provides a way to access the only object that can be accessed directly, without the need to instantiate the class object.

Example:

Cat.java

```
public class Cat {


   private Cat() {}


   private static Cat cat;


   public static synchronized Cat getCat() {
      if (cat == null) {
         cat = new Cat();
      }
      return cat;
```

```
    }

    public String catName;

    public void shouting() {
        System.out.println(catName + " Miyauu Miyaau");
    }
}
```

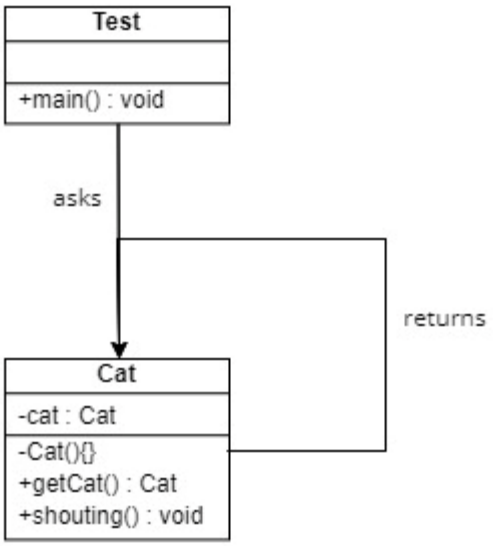


Test.java

```
public class Test {
    public static void main(String[] args) {

        Cat cat1 = Cat.getCat();
        cat1.catName = "CITTY";
        cat1.shouting();

        Cat cat2 = Cat.getCat();
        cat2.catName = "TUTU";
        cat2.shouting();
    }
}
```

Output

```
CITTY Miyauu Miyaau
TUTU Miyauu Miyaau
```



Singleton Pattern Class Diagram

When the main() method of the Test class is run in the above code, the above output is obtained but no two Cat class objects are created. What actually happened was that the same Cat class object was captured by two variables and the catName was changed, as shown in the Stack Heap Diagram above. This is because the Cat class is built using the Singleton Design Pattern so that the client can use only one Cat class object.

In building the Cat class according to the Singleton Design Pattern, the class is modified as public so that it can be used anywhere. The constructor of that class has been modified as private so that it cannot be used elsewhere. A static variable called cat has been created so that it can capture a Cat class object and is modified as private so that it cannot be used elsewhere. Allows a client to create a Cat class object using the getCat() method. The getCat() method is modified as public so that it can be used anywhere, is modified as static for general use, and is modified as synchronized so that only one thread can be run at a time. The getCat() method uses an if block to check if a Cat class object has already been created.

In this way, the Cat class is built according to the Singleton Design Pattern. Accordingly, the client will be able to use the Cat class object only as specified in the main() method of the Test class. However, a client can only use one Cat class object.

3.2 Adapter Design Pattern

The Adapter Design Pattern acts as a bridge between two incompatible interfaces. This type of design pattern comes under a structural pattern as it combines the capabilities of two independent interfaces. This pattern includes a single class responsible for connecting to the functionality of independent or incompatible interfaces.

Example:

Test.java

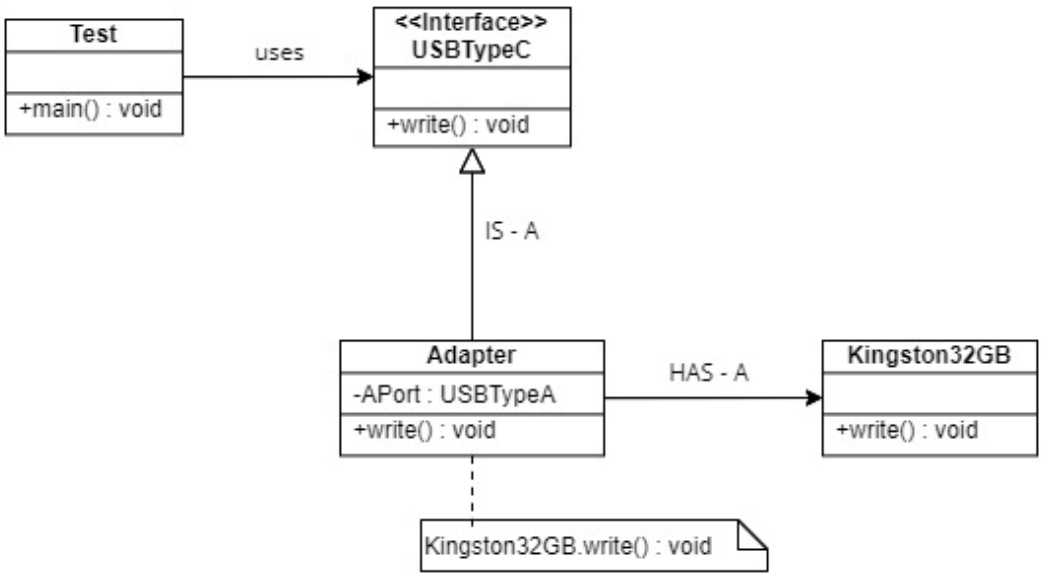
```
class MackBook {


   USBTypeC CPort;


   public void copyData() {
      System.out.println("Copying File");
      CPort.write();
   }
}
```

```
interface USBTypeC {
    public abstract void write();
}

class Kingston16GB implements USBTypeC {
    @Override
    public void write() {
        System.out.println("Writing Speed 20MB/s");
    }
}

interface USBTypeA {
    public abstract void write();
}

class Kingston32GB implements USBTypeA {
    @Override
    public void write() {
        System.out.println("Writing Speed 10MB/s");
    }
}

class Adapter implements USBTypeC {

    USBTypeA APort;

    @Override
    public void write() {
        APort.write();
```

```
    }
}

public class Test {
    public static void main(String[] args) {

        MackBook mackBook = new MackBook();

        Kingston16GB CPen = new Kingston16GB();
        mackBook.CPort = CPen;
        mackBook.copyData();

        Kingston32GB APen = new Kingston32GB();

        Adapter adapter = new Adapter();
        adapter.APort = APen;
        mackBook.CPort = adapter;
        mackBook.copyData();

    }
}
```

Output

```
Copying File
Writing Speed 20MB/s
Copying File
Writing Speed 10MB/s
```



Adapter Design Pattern Class Diagram

When the main() method of the Test class is run in the above code, the above output is obtained. In the above code, two interfaces are created as USBTypeC and USBTypeA. The USBTypeC interface is implemented into the Kingston16GB class and the USBTypeA interface into the Kingston32GB class. In the code above, a class called MackBook is created, which can only catch USBTypeC class objects. Accordingly, Kingston16GB class objects can be caught by the MackBook class and Kingston32GB class objects cannot be caught by the MackBook class. But according to the Adapter Design Pattern, building an Adapter class according to the code above can catch the Kingston32GB class objects indirectly into the MackBook class, though not directly.

To that adapter class, the USBTypeC interface is implemented and an instance variable is created to catch a USBTypeA class object. The write() methods in the caught USBTypeA objects are called by override write() method in the Adapter class.

In this way, the Adapter class is built according to the Adapter Design Pattern. Accordingly, a client can build relationships between mismatched interfaces or mismatched classes using Adapter class objects and other class objects as specified in the main() method of the Test class.

3.3 Strategy Design Pattern (Policy Design Pattern)

In the Strategy Design Pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under a behavior pattern. Within the Strategy Design Pattern, we create objects that represent different strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

Example:

Test.java

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;


interface Strategy{
   public abstract boolean find(String text, String symbol);
}


class Strategy1 implements Strategy{


   @Override
   public boolean find(String text, String symbol){
      if(text.contains(symbol)){
         return true;
      }else{
         return false;
      }
   }


}


class Strategy2 implements Strategy{
```

```
    @Override
    public boolean find(String text, String symbol){
        if(text.indexOf(symbol)>0){
            return true;
        }else{
            return false;
        }
    }

}

class Strategy3 implements Strategy{

    @Override
    public boolean find(String text, String symbol){
        Matcher matcher = Pattern.compile(symbol + "\\w+").matcher(text);
        if(matcher.find()){
            return true;
        }else{
            return false;
        }
    }

}

public class Test {

    public static void main(String[] args) {
```

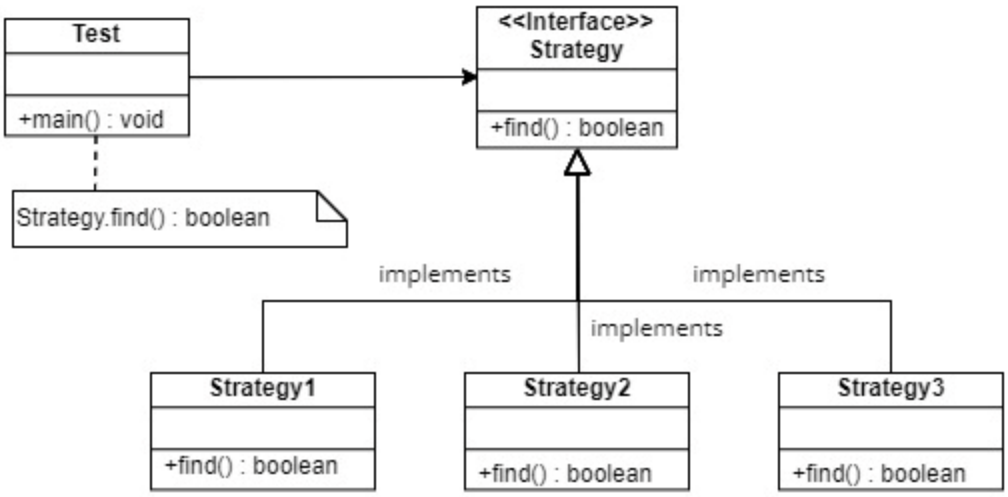
```
        String text = "strategy@pattern.edu.lk";

        Strategy1 s1 = new Strategy1();
        boolean result1 = s1.find(text, "@");
        if(result1){
            System.out.println("Strategy1 Success");
        }else{
            System.out.println("Strategy1 Error");
        }

        Strategy2 s2 = new Strategy2();
        boolean result2 = s2.find(text, "@");
        if(result2){
            System.out.println("Strategy2 Success");
        }else{
            System.out.println("Strategy2 Error");
        }

        Strategy3 s3 = new Strategy3();
        boolean result3 = s3.find(text, "@");
        if(result3){
            System.out.println("Strategy3 Success");
        }else{
            System.out.println("Strategy3 Error");
        }

    }
}
```

Output

Strategy1 Success

Strategy2 Success

Strategy3 Success



Strategy Design Pattern Class Diagram

When the main() method of the Test class is run in the above code, the above output is obtained. The above code builds three classes as Strategy1, Strategy2, and Strategy3. These three classes are built to identify any symbol in a text. The strategies used by these three classes to accomplish that task are different from each other. But when a client is allowed to use this code, he must have a common format to use any of these classes (strategies). Otherwise, the client may face various problems (different method names, different return types, different parameter lists). Therefore, according to the code above, the Strategy interface is built according to the Strategy Design Pattern and it is implemented for all the three classes Strategy1, Strategy2, and Strategy3.

The find() method is built into the Strategy interface. The method can pass two parameters as text and symbol and the return type is boolean. The find() method has been modified as public so that it can be used anywhere, and the Strategy interface has also been modified abstract to make it mandatory to override all the implementing classes.

Accordingly, the find() method in the Strategy interface has been modified abstractly, so all classes implementing the Strategy interface must override the find() method. Therefore, all the classes that implement the Strategy interface have to build their Strategies according to the same format.

In this way, the above code is built according to the Strategy Design Pattern and the client can identify the symbols of the text they want by using the class objects and passing the parameters through the methods as shown in the main() method of the Test class.

3.4 Facade Design Pattern

The Facade Design Pattern hides the intricacies of the system and provides an interface through which the client can access the system. This type of design pattern comes under a structural pattern as it adds an interface to an existing system to hide its intricacies. This pattern involves a single class that provides simplified methods required by the client and delegates calls to methods of existing system classes.

Example:

Shape.java

```
package shape;


public interface Shape {
   void draw();
}
```

Circle.java

```
package circle;


import shape.Shape;


public class Circle implements Shape {


  @Override
  public void draw() {
    System.out.println("Circle::draw()");
  }
}
```

Rectangle.java

```
package rectangle;
```

```
import shape.Shape;

public class Rectangle implements Shape {

  @Override
  public void draw() {
    System.out.println("Rectangle::draw()");
  }
}
```

Square.java

```
package square;

import shape.Shape;

public class Square implements Shape {

  @Override
  public void draw() {
    System.out.println("Square::draw()");
  }
}
```

ShapeMaker.java

```
import circle.Circle;
import rectangle.Rectangle;
import shape.Shape;
import square.Square;
```

```
public class ShapeMaker {

   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }

   public void drawCircle() {
      circle.draw();
   }

   public void drawRectangle() {
      rectangle.draw();
   }

   public void drawSquare() {
      square.draw();
   }
}
```

Test.java

```
public class Test {
   public static void main(String[] args) {
     ShapeMaker shapeMaker = new ShapeMaker();
```

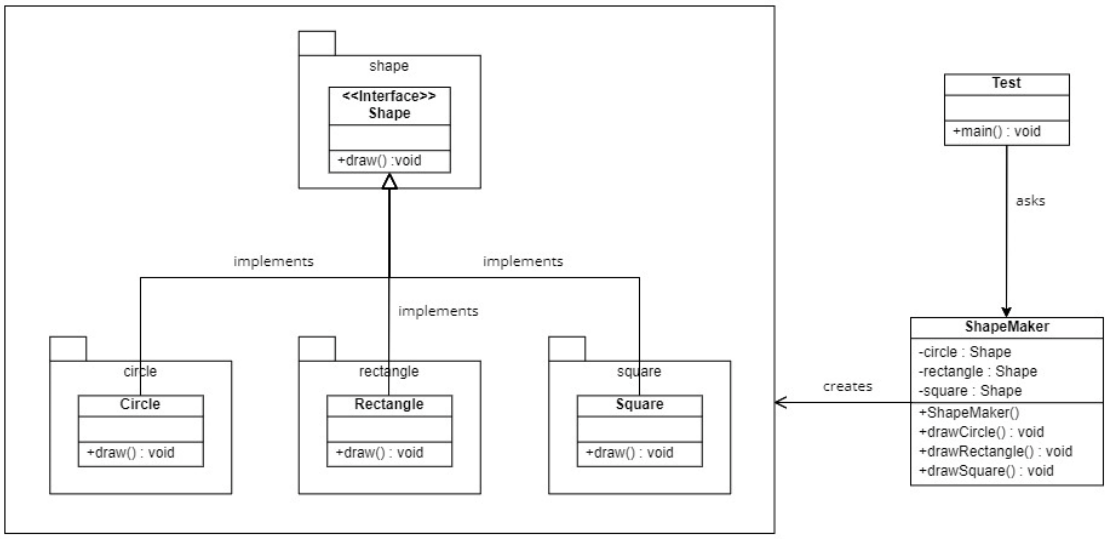
```
    shapeMaker.drawCircle();
    shapeMaker.drawRectangle();
    shapeMaker.drawSquare();
  }
}
```

Output

Circle::draw()

Rectangle::draw()

Square::draw()



Facade Design Pattern Class Diagram

When the main() method of the Test class is run in the above code, the above output is obtained. In the above code, four packages are created namely circle, rectangle, square, and shape. The Circle class is built in the circle package, the Rectangle class is built in the rectangle package, the Square class is built in the square package, and the Shape interface is built in the shape package. The shape interface has been implemented for all three classes Circle, Rectangle and Square. Also, the ShapeMaker class and the Test class are built into the default package.

This Facade Design Pattern creates a sequence for the client to use when there are several different classes. Accordingly, the ShapeMaker class has created an order to use the Circle, Rectangle, and Square classes. In the ShapeMaker class, three instance variables called a circle, rectangle, and square are created to capture the class objects Circle, Rectangle and Square. The three variables have been modified as private so that they cannot be used elsewhere. In the ShapeMaker class constructor, we create class objects for the class Circle, Rectangle and Square and assign them to the three instance variables circle, rectangle, and square. The ShapeMaker class also builds the three methods drawCircle(), drawRectangle(), and

drawSquare() as shown in the code above. The ShapeMaker class, its constructor, and drawCircle(), drawRectangle(), drawSquare() methods have been modified as public so that they can be used anywhere.

In this way, the above code is built according to the Facade Design Pattern and a client can create the class objects and call the methods according to the main() method of the Test class.

3.5 Iterator Design Pattern

The Iterator Design Pattern is a design pattern commonly used in Java and .Net programming environments. This pattern is used to get away to access the elements of a collection object in a sequential manner without any need to know its underlying representation. The Iterator Design Pattern belongs to the category of behavior patterns.

Example:

Test.java

```
interface Iterator {
   public abstract boolean hasNext();
   public abstract Object next();
}

interface Container {
   public Iterator getIterator();
}

class NameRepository implements Container {

   public String names[] = {"Nimal" , "Kamal" ,"Saman" , "Nadun"};

   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }
```

```
  private class NameIterator implements Iterator {

    int index;

    @Override
    public boolean hasNext() {

      if(index < names.length){
        return true;
      }
      return false;
    }

    @Override
    public Object next() {

      if(this.hasNext()){
        return names[index++];
      }
      return null;
    }
  }
}

class Test {
   public static void main(String[] args) {
    NameRepository namesRepository = new NameRepository();
```

```
    for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
      String name = (String)iter.next();
      System.out.println("Name : " + name);
    }
  }
}
```

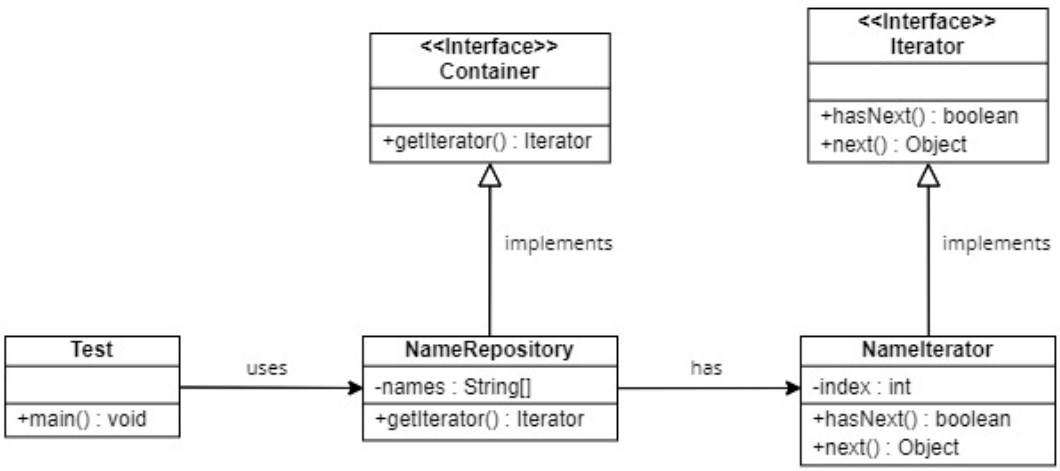
Output

Name : Nimal

Name : Kamal

Name : Saman

Name : Nadun



Iterator Design Pattern Class Diagram

When the main() method of the Test class is run in the above code, the above output is obtained. The above code mainly has two interfaces called Iterator and Container, a class called NameRepository, and an inner class called NameIterator. Here the Iterator interface is built according to the Iterator Design Pattern. This interface is called "The Heart of the Iterator Design Pattern". An Iterator is a common model for collections. But this is not an Array, Set, Map, or any collection. An Iterator is used to see if there is a value and to get the value.

In the Iterator interface a method called hasNext() which is the return type boolean and a method called next() which is the return type Object. Both methods have been modified as abstract. Then for each class that implements the Iterator interface both the hasNext() and next() methods must be overridden. Both methods have been modified as public so that they can be used anywhere. The Container interface is implemented to the NameRepository class as shown in the code above and its getIterator() method is overridden and built. In the NameRepository class, create a string array of names[] and it has been modified as public. Also, within the

NameRepository class, an inner class called NameIterator is built. The Iterator interface is implemented to the NameIterator class and its hasNext() and next() methods are overridden and built as shown in the code above. In the NameIterator class, an instance variable named index has been created to catch an int value. The nameIterator inner class has been modified as private so that it cannot be used elsewhere.

In this way, the above code is built according to the Iterator Design Pattern and the client can print the data in the String Array by creating class the objects and calling the methods as described in the main() method of the Test class.

3.6 Template Design Pattern

In the Template Design Pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class. Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Design Pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. This pattern falls under the category of behavior patterns.

Example:

Test.java

```
abstract class Prepare { // Template Class


   public void boilWater() {
      System.out.println("Boil Water");
   }


   public void pourInCup() {
      System.out.println("Pour In Cup");
   }


   public abstract void step2();
   public abstract void step4();


   public final void templateMethod() { // Template Method
```

```
        boilWater();
        step2();
        pourInCup();
        if (doStep4()) {
            step4();
        }
    }

    public boolean doStep4() {
        return true;
    }

}

class PrepareTea extends Prepare {

    @Override
    public void step2() {
        System.out.println("Steep Tea Bag");
    }

    @Override
    public void step4() {
        System.out.println("Add Ginger");
    }

    @Override
    public boolean doStep4() {
        return false;
```

```
    }
}

class PrepareCoffee extends Prepare {

    @Override
    public void step2() {
        System.out.println("Brew Coffee Grinds");
    }

    @Override
    public void step4() {
        System.out.println("Add Sugar And Milk");
    }

}

 class Test {
    public static void main(String[] args) {

        PrepareTea tea1 = new PrepareTea();
        PrepareTea tea2 = new PrepareTea();
        PrepareCoffee coffee1 = new PrepareCoffee();

        tea1.templateMethod();
        tea2.templateMethod();
        coffee1.templateMethod();

    }
```

}

Output

Boil Water

Steep Tea Bag

Pour In Cup

Boil Water

Steep Tea Bag

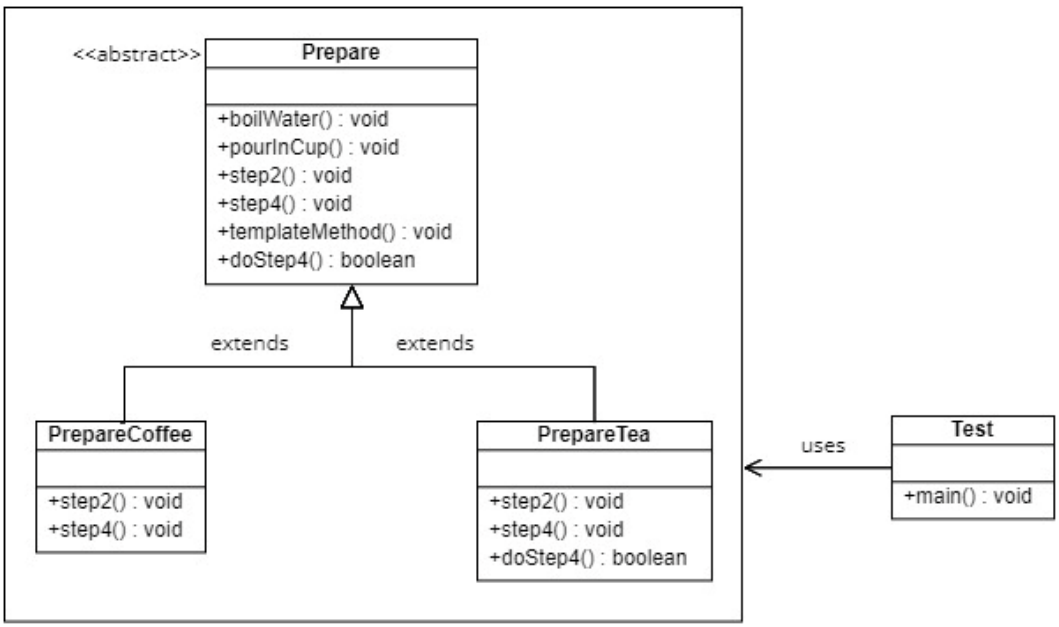
Pour In Cup

Boil Water

Brew Coffee Grinds

Pour In Cup

Add Sugar And Milk



Template Design Pattern Class Diagram

When the main() method of the Test class is run in the above code, the above output is obtained. In the above code, we have created an abstract class called Prepare and two classes called PrepareTea and PrepareCoffee. Here the Prepare abstract class is built according to the Template Design Pattern.

The Prepare abstract class includes six methods called boilWater(), pourInCup(), step2(), step4(), templateMethod() and doStep4(). The return method of boilWater(), pourInCup(), step2(), step4() and templateMethod() is void and the return type of doStep4() method is boolean. The step2() and step4() methods have been modified as abstract. Then for every class

that extends the Prepare abstract class, both methods must be overridden. Within the templateMethod() method calls the other five methods in sequence, and the templateMethod() method has been modified as final so that it cannot be changed again. All six methods have been modified as public so that they can be used anywhere. The Prepare abstract class extends to both the PrepareTea and PrepareCoffee classes. Within the PrepareTea class, the three methods step2(), step4(), and doStep4() of the Prepare abstract class are overridden and built as shown in the code above. Within the PrepareCoffee class, the two methods step2() and step4() of the Prepare abstract class are overridden and built as shown in the code above.

In this way, the above code is built according to the Template Design Pattern and a client can create the class objects and call the methods as per the main() method of the Test class.

3.7 Observer Design Pattern

The Observer Design Pattern is used when there is a one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. The Observer Design Pattern defines a one-to-many dependency between objects so that when an object changes state, all of its dependents are notified and updated automatically. Observer Design Pattern falls under the behavioral pattern category.

Example:

Test.java

```
import java.util.ArrayList;


class YoutubeChannel {


   ArrayList<Subscriber> subscribers = new ArrayList<>();


   public void addSubscriber(Subscriber s) {
      subscribers.add(s);
   }


   public void removeSubscriber(Subscriber s) {
      subscribers.remove(s);
   }
```

```
    public void notifySubscriber(String msg) {
        for (Subscriber s : subscribers) {
            s.update(msg);
        }
    }

}

interface Subscriber {
    public abstract void subscribe(YoutubeChannel yc);
    public abstract void unSubscribe();
    public abstract void update(String msg);
}

class Subscriber1 implements Subscriber {

    YoutubeChannel youtubeChannel;
    String notification;

    @Override
    public void subscribe(YoutubeChannel yc) {
        youtubeChannel = yc;
        youtubeChannel.addSubscriber(this);
    }

    @Override
    public void unSubscribe() {
        youtubeChannel.removeSubscriber(this);
```

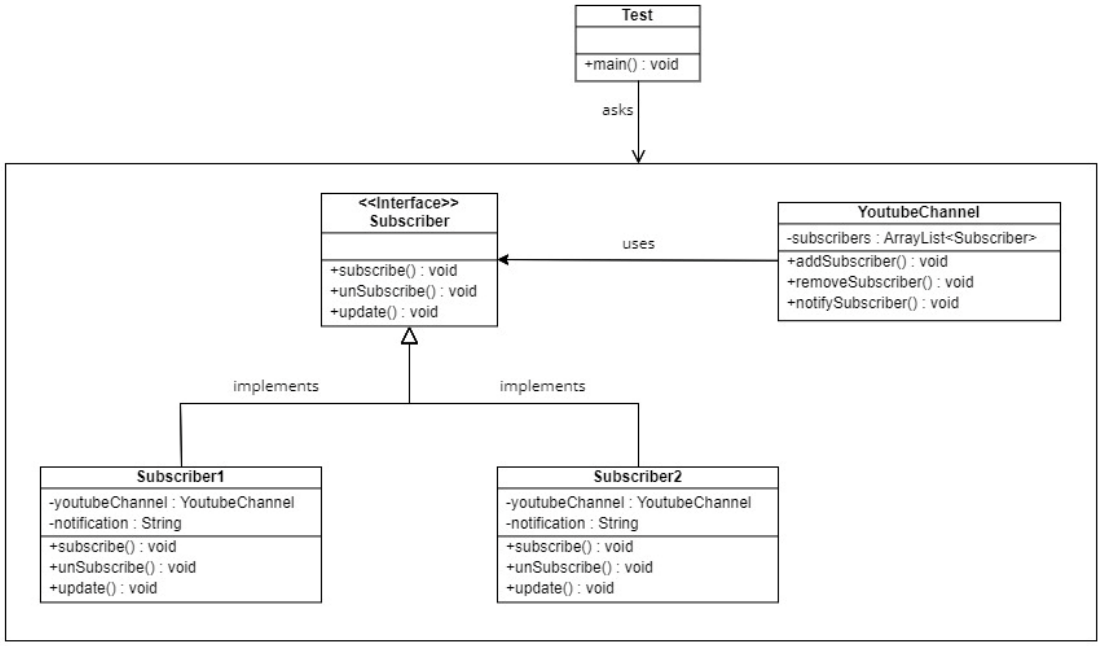
```
        youtubeChannel = null;
    }

    @Override
    public void update(String msg) {
        notification = msg;
        System.out.println("Subscriber1: "+notification);
    }

}

class Subscriber2 implements Subscriber {

    YoutubeChannel youtubeChannel;
    String notification;

    @Override
    public void subscribe(YoutubeChannel yc) {
        youtubeChannel = yc;
        youtubeChannel.addSubscriber(this);
    }

    @Override
    public void unSubscribe() {
        youtubeChannel.removeSubscriber(this);
        youtubeChannel = null;
    }

    @Override
```

```
    public void update(String msg) {
        notification = msg;
        System.out.println("Subscriber2: "+notification);
    }

}

public class Test {
    public static void main(String[] args) {

        YoutubeChannel c = new YoutubeChannel();

        Subscriber1 s1 = new Subscriber1();
        Subscriber2 s2 = new Subscriber2();

        s1.subscribe(c);
        s2.subscribe(c);

        c.notifySubscriber("Avengers 1 Released!");

        s2.unSubscribe();

        c.notifySubscriber("Avengers 2 Released!");

    }
}
```

Output

```
Subscriber1: Avengers 1 Released!
Subscriber2: Avengers 1 Released!
```

Subscriber1: Avengers 2 Released!



Observer Design Pattern Class Diagram

When the main method of the Test class is run in the above code, the above output is obtained. Built according to the Observer Design Pattern, this code shows how to send the same notification to all the subscribers of a YouTube channel at once.

The above code mainly has been built two classes called YoutubeChannel, Subscriber1, and Subscriber2, and an interface called Subscriber. An ArrayList called subscribers has been created to keep a list of subscribers in the YoutubeChannel class. Also in that class, have built three methods addSubscriber(), removeSubscriber(), and notifySubscriber() as shown in the code above. In all three methods, the return type is void and modified as public so that it can be used anywhere. The subscriber interface has three built-in methods, subscribe(), unSubscribe(), and update(). In all three methods, the return type is void and modified as public so that it can be used anywhere. Also, the three methods unSubscribe() and update() have been modified as abstract, so all three methods must be overridden for every class that implements the Subscriber interface. The Subscriber interface is implemented for both Subscriber1 and Subscriber2 classes. Therefore, both Subscriber1 and Subscriber2 classes override the three subscribe(), unSubscribe(), and update() methods that belong to the Subscriber interface and build as shown in the code above. In both Subscriber1 and Subscriber2 classes there are two instance variables called youtubeChannel to capture Youtube Channels and notification to capture Notification Messages.

In this way, the above code is built according to the Observer Design Pattern and the client sends the same notification to all the subscribers of a YouTube channel according to the above code by creating the class objects and passing the parameters through the methods as per the main method of the Test class.

3.8 Command Design Pattern

The Command Design Pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. And it is a data-driven design pattern that falls under the behavior pattern category. A request is wrapped as a command under an object and passed to the Invoker object. The Invoker object looks for the appropriate object to handle this command and passes the command to the corresponding object that executes the command.

Example:

Test.java

```
interface Command {
   public abstract void execute(int x, int y);
}

class PlusCommand implements Command {

   Maths m;

   public PlusCommand(Maths m) {
      this.m = m;
   }

   @Override
   public void execute(int x, int y) {
      m.plus(x, y);
   }

}

class MinusCommand implements Command {

   Maths m;
```

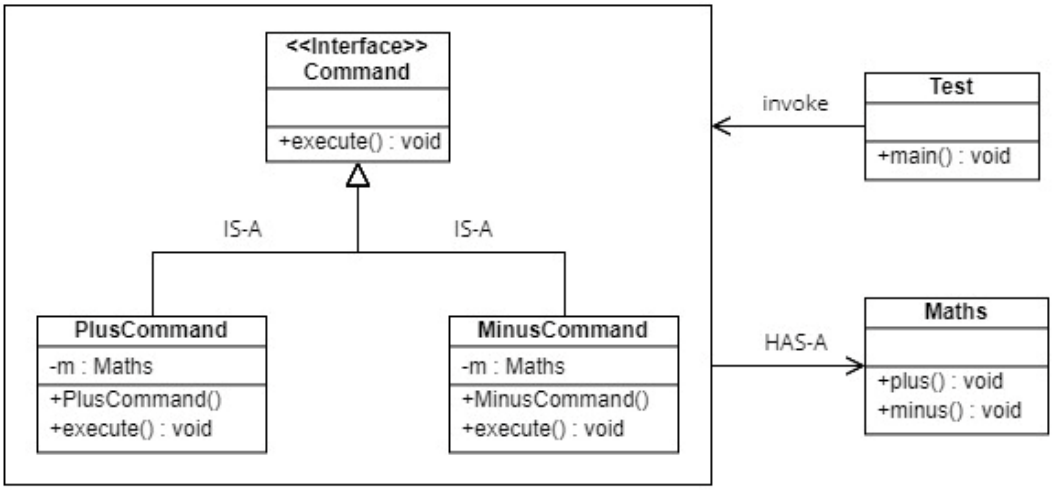
```
    public MinusCommand(Maths m) {
        this.m = m;
    }

    @Override
    public void execute(int x, int y) {
        m.minus(x, y);
    }

}

class Maths {

    public void plus(int x, int y) { //action method
        System.out.println(x + y);
    }

    public void minus(int x, int y) { //action method
        System.out.println(x - y);
    }

}

class Test { //invoker
    public static void main(String[] args) { //invoke

        Maths m = new Maths();
```

```
        PlusCommand pc = new PlusCommand(m);
        MinusCommand mc = new MinusCommand(m);


        pc.execute(10, 20);
        mc.execute(10, 20);


    }
}
```

Output

30

-10



Command Design Pattern Class Diagram

When the main() method of the Test class is run in the above code, the above output is obtained. This code, which is built according to the Command Design Pattern, shows how to perform certain arithmetic operations.

The above code has been built four classes namely PlusCommand, MinusCommand, Maths and Test, and an interface called Command. A method called execute() is built into the Command interface. The return type of that method is void and has been modified as public so that it can be used anywhere. Also, the execute() method has been modified as abstract, so this method must be overridden for every class that implements the Command interface. The Command interface is implemented to both the PlusCommand and MinusCommand classes. Therefore, the execute() method, which belongs to the Command interface for both the PlusCommand and MinusCommand classes, is overridden and built as shown in the code above. Also, an instance variable named m has been created to catch Maths class objects in

both classes. The constructors of both the PlusCommand and MinusCommand classes are built as shown in the code above and are modified as public so that they can be used anywhere. In the Maths class, the two methods plus() and minus() are constructed as shown in the code above, and the return type of both methods is void and modified as public so that it can be used anywhere. According to the Command Design Pattern, the plus() and minus() methods are the action method, the Test class is the invoker and the Test class's main method is the invoke.

In this way, the above code is constructed according to the Command Design Pattern and the client can perform the arithmetic operations according to the above code by creating the class objects and passing the parameters through the methods as per the main() method of the Test class.

3.9 State Design Pattern

The State Design Pattern is a behavioral design pattern that allows an object to change the behavior when it's internal state changes. Within the State Design Pattern, class behavior varies based on its status. This type of design pattern comes under a pattern of behavior. In the State Design Pattern, we create objects that represent different states and a contextual object whose behavior changes as its state object changes.

Example:

Test.java

```
interface PhoneState {
   public abstract void incomingCall();
}

class On implements PhoneState {
   @Override
   public void incomingCall() {
      System.out.println("Play Ringtone");
   }
}

class Off implements PhoneState {
   @Override
```

```
    public void incomingCall() {
        System.out.println("Not Responding");
    }
}

class Silent implements PhoneState {
    @Override
    public void incomingCall() {
        System.out.println("Vibrating");
    }
}

class Phone implements PhoneState {

    PhoneState state;

    public void setState(PhoneState state) {
        this.state = state;
    }

    @Override
    public void incomingCall() {
        state.incomingCall();
    }

}

class Test {
    public static void main(String[] args) {
```

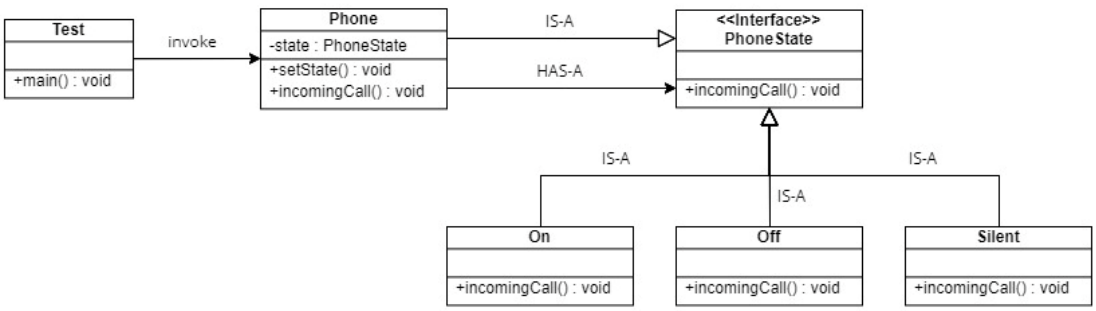
```
        Phone p = new Phone();

        On on = new On();
        Off off = new Off();
        Silent silent = new Silent();

        p.setState(on);
        p.incomingCall();

        p.setState(off);
        p.incomingCall();

        p.setState(silent);
        p.incomingCall();

    }
}
```

Output

Play Ringtone

Not Responding

Vibrating



State Design Pattern Class Diagram

When the main() method of the Test class is run in the above code, the above output is obtained. This code, which is built according to the State Design Pattern, is an example of how incoming calls respond to changes in the state of a phone.

The above code mainly has been built four classes namely On, Off, Silent, and Phone, and an interface called PhoneState. The incomingCall() method is built into the PhoneState interface. The return type of that method is void and has been modified as public so that it can be used anywhere. The incomingCall() method has also been modified as abstract, so this method must be overridden for every class that implements the PhoneState interface. The PhoneState interface implements all the classes On, Off, Silent, and Phone. Accordingly, the incomingCall() method that belongs to the PhoneState interface for all the four classes On, Off, Silent and Phone has been overridden and built as shown in the code above. Created an instance variable called the state to catch PhoneState class objects in the phone class. Also in the Phone class, a method called setState() is built as shown in the code above and its return type is void and modified as public so that it can be used anywhere.

In this way, the above code is built according to the State Design Pattern and the client can change the state of a phone according to the above code by creating the class objects and passing the parameters through the methods as per the main() method of the Test class and get different responses to incoming calls.

3.10 Factory Design Pattern

The Factory Design Pattern is one of the most used design patterns in Java. This type of design pattern falls under the creational pattern as it provides the best way to create an object. A Factory Design Pattern or Factory Method Pattern defines an interface or abstract class for creating an object but lets the subclasses decide which class to instantiate. In other words, subclasses are responsible to create the instance of the class. In the Factory Design Pattern, we create objects without exposing the creation logic to the client and refer to the newly created objects using a common interface.

Example:

Test.java

```
interface AppleDevice {

   public abstract void getModel();

}


class IPhone implements AppleDevice {

   @Override
```

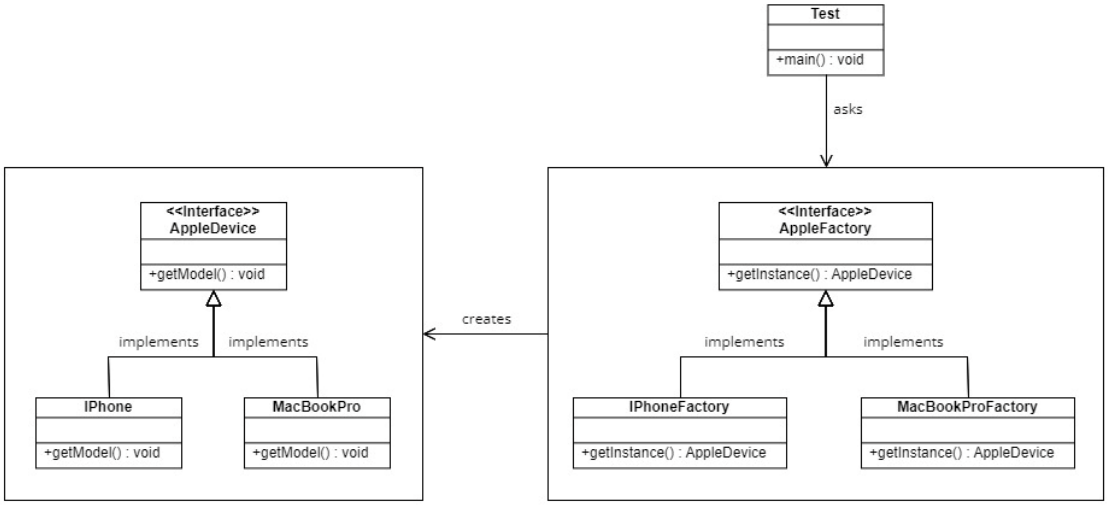
```
    public void getModel() {
        System.out.println("Apple iPhone");
    }
}

class MacBookPro implements AppleDevice {
    @Override
    public void getModel() {
        System.out.println("Apple MacBook Pro");
    }
}

interface AppleFactory { // Factory
    public abstract AppleDevice getInstance();
}

class IPhoneFactory implements AppleFactory {
    @Override
    public AppleDevice getInstance() {
        return new IPhone();
    }
}

class MacBookProFactory implements AppleFactory {
    @Override
    public AppleDevice getInstance() {
        return new MacBookPro();
    }
}
```

```
class Test {
    public static void main(String[] args) {

        AppleFactory factory1 = new IPhoneFactory();
        AppleFactory factory2 = new MacBookProFactory();

        AppleDevice device1 = factory1.getInstance();
        device1.getModel();

        AppleDevice device2 = factory2.getInstance();
        device2.getModel();

    }
}
```

Output

Apple iPhone

Apple MacBook Pro



Factory Design Pattern Class Diagram

When the main() method of the test class is run in the above code, the above output is obtained. For this code, which is built according to the Factory Design Pattern, the functionality of an Apple factory is taken as an example.

The above code mainly has been built four classes namely IPhone, MacBookPro, IPhoneFactory, and MacBookProFactory, and two interfaces called AppleDevice and AppleFactory. A method called getModel() is built into the AppleDevice interface. The return type of that method is void and has been modified as public so that it can be used anywhere. The getModel() method is also modified as abstract, so this method must be overridden for every class that implements the AppleDevice interface. The AppleDevice interface has been implemented for both the IPhone and MacBookPro classes, and the GetModel() method, which belongs to the AppleDevice interface for both classes, has been overridden and built as shown in the code above. A method called getInstance() is built into the AppleFactory interface. The return type of that method is AppleDevice and has been modified as public so that it can be used anywhere. The getInstance() method has also been modified as abstract, so this method must be overridden for every class that implements the AppleFactory interface. The AppleFactory interface has been implemented for both the IPhoneFactory and MacBookProFactory classes, and the getInstance() method, which belongs to the AppleFactory interface for both classes, has been overridden and built as shown in the code above.

In this way, the above code is built according to the Factory Design Pattern and a client can create the class objects and call the methods as specified in the main() method of the Test class.

3.11 Proxy Design Pattern

The Proxy Design Pattern is a structural design pattern that lets you provide a substitute or placeholder for another object. In the Proxy Design Pattern, one class represents the functionality of another class. This type of design pattern comes under a structural pattern. In the Proxy Design Pattern, we create an object that has an original object to interface its functionality to the outside world. A proxy controls access to the original object allowing you to perform something either before or after the request gets through to the original object.

Example:

Test.java

```
interface Windows11OS {

    public abstract void access(String serialKey);

}


class Windows11OSGenuine implements Windows11OS {

    @Override
```

```
    public void access(String serialKey) {
        System.out.println("You're using Full Version of Windows 11");
    }
}

class Windows11OSTrial implements Windows11OS {

    private Windows11OSGenuine os = new Windows11OSGenuine(); //Proxy HAS - Real Object

    @Override
    public void access(String serialKey) {

        if (serialKey.equals("Q43FJ05")) {
            os.access(serialKey);
        } else {
            System.out.println("You're using Trial Version of Windows 11 with Limited Features");
        }
    }

}

class Test {
    public static void main(String[] args) {

        Windows11OSTrial os1 = new Windows11OSTrial();
        os1.access(""); //Without serial key
        os1.access("Q43FJree"); //With incorrect serial key
        os1.access("Q43FJ05"); //With correct serial key
```

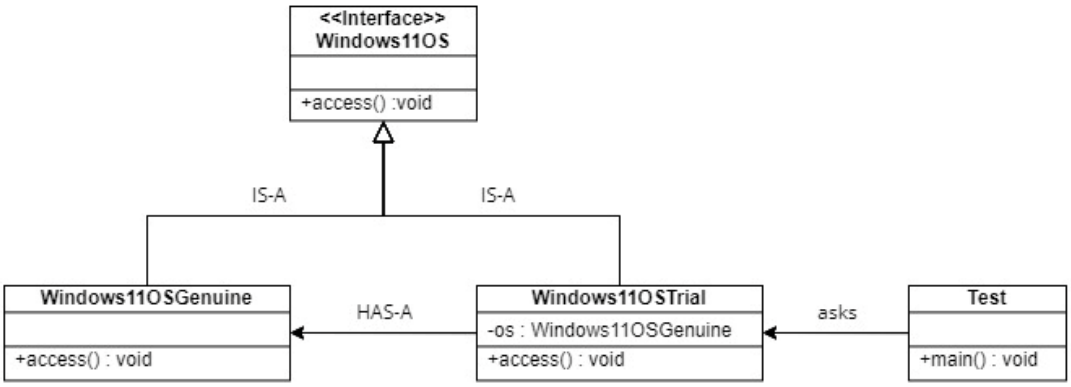
```
    }
}
```

Output

You're using Trial Version of Windows 11 with Limited Features

You're using Trial Version of Windows 11 with Limited Features

You're using Full Version of Windows 11



Proxy Design Pattern Class Diagram

When the main() method of the Test class is run in the above code, the above output is obtained. This code, which is built according to the Proxy Design Pattern, how to activate Windows 11 OS using a serial key is taken as an example.

The above code mainly has been built three classes called Windows11OSGenuine, Windows11OSTrial and Test, and an interface called Windows11OS. A method called access() is built into the Windows11OS interface. The return type of that method is void and has been modified as public so that it can be used anywhere. The access() method has also been modified as abstract, so this method must be overridden for every class that implements the Windows11OS interface. The Windows11OS interface implements both Windows11OSGenuine and Windows11OSTrial classes. Accordingly, the access() method, which belongs to the Windows11OS interface for both Windows11OSGenuine and Windows11OSTrial classes, has been overridden and built as shown in the code above. The access() method, which overrides the Windows11OSTrial class has an if block is created as shown in the code above to check the accuracy of the serial key. In the Windows11OSTrial class, an instance variable called os was created to catch Windows11OSGenuine class objects, and that instance variable was modified as private so that they could not be used elsewhere. Thus the Windows11OSGenuine class object created within the Windows11OSTrial class is the real/original object.

In this way, the above code is built according to the Proxy Design Pattern and the client can activate Windows 11 OS by creating class objects as per the main() method of the Test class and passing the serial keys as parameters through the methods.

Reference

https://www.tutorialspoint.com/design_pattern/index.htm

https://www.geeksforgeeks.org/