JournalDev

Java Design Patterns

A Programmer's Approach

Pankaj Kumar

Table of Content

Design Patterns Overview

Design Patterns are very popular among software developers. A design pattern is a well-described solution to a common software problem.

Some of the benefits of using design patterns are:

1. Design Patterns are already defined and provides industry standard approach to solve a recurring problem, so it saves time if we sensibly use the design pattern.
2. Using design patterns promotes reusability that leads to more robust and highly maintainable code. It helps in reducing total cost of ownership (TCO) of the software product.
3. Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of team understand it easily.

Java Design Patterns are divided into three categories – **creational**, **structural**, and **behavioral** design patterns.

Creational Design Patterns

Creational design patterns provide solution to instantiate an object in the best possible way for specific situations.

The basic form of object creation could result in design problems or add unwanted complexity to the design. Creational design patterns solve this problem by controlling the object creation by different ways.

There are five creational design patterns that we will discuss in this eBook.

1. Singleton Pattern
2. Factory Pattern
3. Abstract Factory Pattern
4. Builder Pattern
5. Prototype Pattern

All these patterns solve specific problems with object creation, so you should understand and use them when needed.

1. Singleton Pattern

Singleton is one of the **Gangs of Four Design patterns** and comes in the **Creational Design Pattern** category. From the definition, it seems to be a very simple design pattern but when it comes to implementation, it comes with a lot of implementation concerns. The implementation of Singleton pattern has always been a controversial topic among developers. Here we will learn about Singleton design pattern principles, different ways to implement Singleton and some of the best practices for its usage.

Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine. The singleton class must provide a global access point to get the instance of the class. Singleton pattern is used for logging, driver objects, caching and thread pool.
Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade etc. Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

To implement Singleton pattern, we have different approaches but all of them have following common concepts.

- Private constructor to restrict instantiation of the class from other classes.
- Private static variable of the same class that is the only instance of the class.
- Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

In further sections, we will learn different approaches of Singleton pattern implementation and design concerns with the implementation.

A. Eager Initialization

In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not be using it.

Here is the implementation of static initialization singleton class.

```
package com.journaldev.singleton;

public class EagerInitializedSingleton {

    private static final EagerInitializedSingleton instance = new
EagerInitializedSingleton();

    //private constructor to avoid client applications to use
constructor
    private EagerInitializedSingleton(){}

    public static EagerInitializedSingleton getInstance(){
        return instance;
    }
}
```

If your singleton class is not using a lot of resources, this is the approach to use. But in most of the scenarios, Singleton classes are created for resources such as File System, Database connections etc and we should avoid the instantiation until unless client calls the *getInstance* method. Also this method doesn't provide any options for exception handling.

B. Static block initialization

Static block initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for exception handling.

```
package com.journaldev.singleton;

public class StaticBlockSingleton {

    private static StaticBlockSingleton instance;

    private StaticBlockSingleton(){}

    //static block initialization for exception handling
    static{
        try{
            instance = new StaticBlockSingleton();
        }catch(Exception e){
            throw new RuntimeException("Exception occured in creating
singleton instance");
        }
    }

    public static StaticBlockSingleton getInstance(){
        return instance;
    }
}
```

Both eager initialization and static block initialization creates the instance even before it's being used and that is not the best practice to use. So in further sections, we will learn how to create Singleton class that supports lazy initialization.

C. Lazy Initialization

Lazy initialization method to implement Singleton pattern creates the instance in the global access method. Here is the sample code for creating Singleton class with this

```
package com.journaldev.singleton;

public class LazyInitializedSingleton {

    private static LazyInitializedSingleton instance;

    private LazyInitializedSingleton(){}
```

```
    public static LazyInitializedSingleton getInstance(){
        if(instance == null){
            instance = new LazyInitializedSingleton();
        }
        return instance;
    }
}
```

The above implementation works fine in case of single threaded environment but when it comes to multithreaded systems, it can cause issues if multiple threads are inside the if loop at the same time. It will destroy the singleton pattern and both threads will get the different instances of singleton class. In next section, we will see different ways to create a thread-safe singleton class.

D. Thread Safe Singleton

The easier way to create a thread-safe singleton class is to make the global access method synchronized, so that only one thread can execute this method at a time. General implementation of this approach is like the below class.

```
package com.journaldev.singleton;

public class ThreadSafeSingleton {

    private static ThreadSafeSingleton instance;

    private ThreadSafeSingleton(){}

    public static synchronized ThreadSafeSingleton getInstance(){
        if(instance == null){
            instance = new ThreadSafeSingleton();
        }
        return instance;
    }

}
```

Above implementation works fine and provides thread-safety but it reduces the performance because of cost associated with the synchronized method, although we need it only for the first few threads who might create the separate instances (Read: Java Synchronization). To avoid this extra overhead every time, **double checked locking** principle is used. In this approach, the synchronized block is used inside if condition with an additional check to ensure that only one instance of singleton class is created.

Below code snippet provides the double checked locking implementation.

```
public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){
    if(instance == null){
        synchronized (ThreadSafeSingleton.class) {
            if(instance == null){
                instance = new ThreadSafeSingleton();
            }
        }
    }
    return instance;
}
```

E. Bill Pugh Singleton Implementation

Prior to Java 5, java memory model had a lot of issues and above approaches used to fail in certain scenarios where too many threads try to get the instance of the Singleton class simultaneously. So Bill Pugh came up with a different approach to create the Singleton class using an inner static helper class. The Bill Pugh Singleton implementation goes like this;

```
package com.journaldev.singleton;

public class BillPughSingleton {

    private BillPughSingleton(){}

    private static class SingletonHelper{
        private static final BillPughSingleton INSTANCE = new
BillPughSingleton();
```

```
    }

    public static BillPughSingleton getInstance(){
        return SingletonHelper.INSTANCE;
    }
}
```

Notice the **private inner static class** that contains the instance of the singleton class. When the singleton class is loaded, SingletonHelper class is not loaded into memory and only when someone calls the *getInstance* method, this class gets loaded and creates the Singleton class instance.

This is the most widely used approach for Singleton class as it doesn't require synchronization. I am using this approach in many of my projects and it's easy to understand and implement also.

F. Using Reflection to destroy Singleton Pattern

Reflection can be used to destroy all the above singleton implementation approaches. Let's see this with an example class.

```
package com.journaldev.singleton;


import java.lang.reflect.Constructor;


public class ReflectionSingletonTest {

    public static void main(String[] args) {
        EagerInitializedSingleton instanceOne =
EagerInitializedSingleton.getInstance();
        EagerInitializedSingleton instanceTwo = null;
        try {
            Constructor[] constructors =
EagerInitializedSingleton.class.getDeclaredConstructors();
            for (Constructor constructor : constructors) {
                //Below code will destroy the singleton pattern
                constructor.setAccessible(true);
                instanceTwo = (EagerInitializedSingleton)
constructor.newInstance();
```

```
                break;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(instanceOne.hashCode());
        System.out.println(instanceTwo.hashCode());
    }

}
```

When you run the above test class, you will notice that hashCode of both the instances are not same that destroys the singleton pattern. Reflection is very powerful and used in a lot of frameworks like Spring and Hibernate, do check out **Java Reflection Tutorial**.

G. Enum Singleton

To overcome this situation with Reflection, Joshua Bloch suggests the use of Enum to implement Singleton design pattern as Java ensures that any enum value is instantiated only once in a Java program. Since Java Enum values are globally accessible, so is the singleton. The drawback is that the enum type is somewhat inflexible; for example, it does not allow lazy initialization.

```
package com.journaldev.singleton;

public enum EnumSingleton {

    INSTANCE;

    public static void doSomething(){
        //do something
    }
}
```

H. Serialization and Singleton

Sometimes in distributed systems, we need to implement Serializable interface in Singleton class so that we can store its state in file system and

retrieve it at later point of time. Here is a small singleton class that implements Serializable interface also.

```
package com.journaldev.singleton;

import java.io.Serializable;

public class SerializedSingleton implements Serializable{

    private static final long serialVersionUID = -7604766932017737115L;

    private SerializedSingleton(){}

    private static class SingletonHelper{
        private static final SerializedSingleton instance = new
SerializedSingleton();
    }

    public static SerializedSingleton getInstance(){
        return SingletonHelper.instance;
    }

}
```

The problem with above serialized singleton class is that whenever we deserialize it, it will create a new instance of the class. Let's see it with a simple program.

```
package com.journaldev.singleton;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;

public class SingletonSerializedTest {

    public static void main(String[] args) throws
FileNotFoundException, IOException, ClassNotFoundException {
        SerializedSingleton instanceOne =
SerializedSingleton.getInstance();
```

```
        ObjectOutput out = new ObjectOutputStream(new FileOutputStream(
                "filename.ser"));
        out.writeObject(instanceOne);
        out.close();

        //deserailize from file to object
        ObjectInput in = new ObjectInputStream(new FileInputStream(
                "filename.ser"));
        SerializedSingleton instanceTwo = (SerializedSingleton)
in.readObject();
        in.close();

        System.out.println("instanceOne
hashCode="+instanceOne.hashCode());
        System.out.println("instanceTwo
hashCode="+instanceTwo.hashCode());

    }

}
```

Output of the above program is;

```
instanceOne hashCode=2011117821
instanceTwo hashCode=109647522
```

So it destroys the singleton pattern, to overcome this scenario all we need to do it provide the implementation of readResolve() method.

```
protected Object readResolve() {
    return getInstance();
}
```

After this you will notice that hashCode of both the instances are same in test program.

2. Factory Pattern

Factory Pattern is one of the **Creational Design pattern** and it's widely used in JDK as well as frameworks like Spring and Struts.

Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern take out the responsibility of instantiation of a class from client program to the factory class. Let's first learn how to implement factory pattern in java and then we will learn its benefits and we will see its usage in JDK.

A. Super Class

Super class in factory pattern can be an interface, abstract class or a normal java class. For our example, we have super class as abstract class with overridden toString() method for testing purpose.

```
package com.journaldev.design.model;

public abstract class Computer {

    public abstract String getRAM();
    public abstract String getHDD();
    public abstract String getCPU();

    @Override
    public String toString(){
        return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+",
CPU="+this.getCPU();
    }
}
```

B. Sub Classes

Let's say we have two sub-classes PC and Server with below implementation.

```
package com.journaldev.design.model;


public class PC extends Computer {

    private String ram;
    private String hdd;
    private String cpu;

    public PC(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
    @Override
    public String getRAM() {
        return this.ram;
    }

    @Override
    public String getHDD() {
        return this.hdd;
    }

    @Override
    public String getCPU() {
        return this.cpu;
    }

}
```

Notice that both the classes are extending Computer class.

```
package com.journaldev.design.model;


public class Server extends Computer {

    private String ram;
    private String hdd;
    private String cpu;

    public Server(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
```

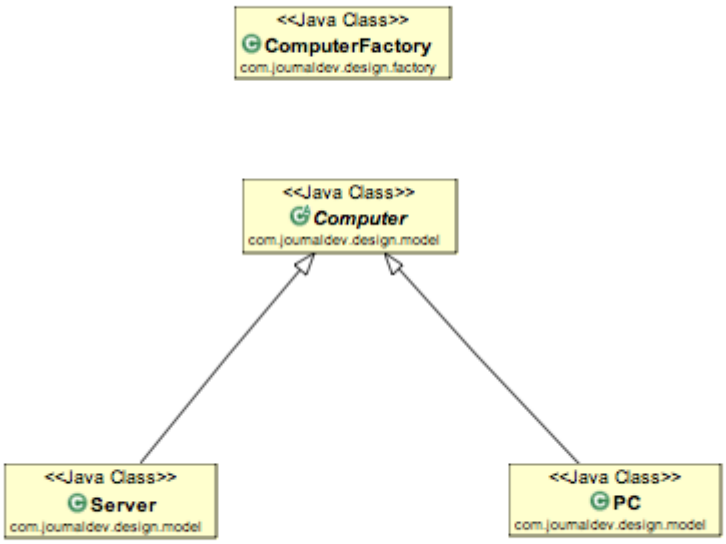
```
        this.cpu=cpu;
    }
    @Override
    public String getRAM() {
        return this.ram;
    }

    @Override
    public String getHDD() {
        return this.hdd;
    }

    @Override
    public String getCPU() {
        return this.cpu;
    }

}
```

C. Factory Class

Now that we have super classes and sub-classes ready, we can write our factory class. Here is the basic implementation.

```
package com.journaldev.design.factory;

import com.journaldev.design.model.Computer;
import com.journaldev.design.model.PC;
import com.journaldev.design.model.Server;

public class ComputerFactory {

    public static Computer getComputer(String type, String ram, String
hdd, String cpu){
        if("PC".equalsIgnoreCase(type)) return new PC(ram, hdd, cpu);
        else if("Server".equalsIgnoreCase(type)) return new Server(ram,
hdd, cpu);

        return null;
    }
}
```

1. We can keep Factory class Singleton or we can keep the method that returns the subclass as static.
2. Notice that based on the input parameter, different subclass is created and returned.



Here is a simple test client program that uses above factory pattern implementation.

```
package com.journaldev.design.test;

import com.journaldev.design.abstractfactory.PCFactory;
import com.journaldev.design.abstractfactory.ServerFactory;
import com.journaldev.design.factory.ComputerFactory;
import com.journaldev.design.model.Computer;

public class TestFactory {

    public static void main(String[] args) {
        Computer pc = ComputerFactory.getComputer("pc","2 GB","500
GB","2.4 GHz");
        Computer server = ComputerFactory.getComputer("server","16
GB","1 TB","2.9 GHz");
        System.out.println("Factory PC Config::"+pc);
        System.out.println("Factory Server Config::"+server);
    }
```

```
}
```

Output of above program is:

```
Factory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz
Factory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz
```

D. Benefits of Factory Pattern

1. Factory pattern provides approach to code for interface rather than implementation.
2. Factory pattern removes the instantiation of actual implementation classes from client code, making it more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because client program is unaware of this.
3. Factory pattern provides abstraction between implementation and client classes through inheritance.

E. Factory Pattern Examples in JDK

1. java.util.Calendar, ResourceBundle and NumberFormat getInstance() methods uses Factory pattern.
2. valueOf() method in wrapper classes like Boolean, Integer etc.

3. Abstract Factory Pattern

Abstract Factory is one of the **Creational pattern** and almost similar to **Factory Pattern** except the fact that it's more like factory of factories.

If you are familiar with **factory design pattern in java**, you will notice that we have a single Factory class that returns the different sub-classes based on the input provided and factory class uses if-else or switch statement to achieve this.

In Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class and then an Abstract Factory class that will return the sub-class based on the input factory class. At first it seems confusing but once you see the implementation, it's really easy to grasp and understand the minor difference between Factory and Abstract Factory pattern.

Like our factory pattern post, we will use the same super class and sub-classes.

A. Super Class and Sub-Classes

```
package com.journaldev.design.model;

public abstract class Computer {

    public abstract String getRAM();
    public abstract String getHDD();
    public abstract String getCPU();

    @Override
    public String toString(){
        return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+",
CPU="+this.getCPU();
    }
}


package com.journaldev.design.model;


public class PC extends Computer {
```

```
    private String ram;
    private String hdd;
    private String cpu;

    public PC(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
    @Override
    public String getRAM() {
        return this.ram;
    }

    @Override
    public String getHDD() {
        return this.hdd;
    }

    @Override
    public String getCPU() {
        return this.cpu;
    }

}
```

```
package com.journaldev.design.model;


public class Server extends Computer {

    private String ram;
    private String hdd;
    private String cpu;

    public Server(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
    @Override
    public String getRAM() {
        return this.ram;
    }
```

```
    @Override
    public String getHDD() {
        return this.hdd;
    }

    @Override
    public String getCPU() {
        return this.cpu;
    }

}
```

B. Factory Classes for Each sub-class

First of all we need to create an Abstract Factory interface or **abstract class**.

```
package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

public interface ComputerAbstractFactory {

    public Computer createComputer();

}
```

Notice that *createComputer()* method is returning an instance of super class *Computer*. Now our factory classes will implement this interface and return their respective sub-class.

```
package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;
import com.journaldev.design.model.PC;

public class PCFactory implements ComputerAbstractFactory {

    private String ram;
    private String hdd;
    private String cpu;
```

```
    public PCFactory(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
    @Override
    public Computer createComputer() {
        return new PC(ram,hdd,cpu);
    }

}
```

Similarly we will have a factory class for Server sub-class.

```
package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;
import com.journaldev.design.model.Server;

public class ServerFactory implements ComputerAbstractFactory {

    private String ram;
    private String hdd;
    private String cpu;

    public ServerFactory(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }

    @Override
    public Computer createComputer() {
        return new Server(ram,hdd,cpu);
    }

}
```

Now we will create a consumer class that will provide the entry point for the client classes to create sub-classes.

```
package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

public class ComputerFactory {

    public static Computer getComputer(ComputerAbstractFactory factory){
        return factory.createComputer();
    }
}
```

Notice that it's a simple class and *getComputer* method is accepting *ComputerAbstractFactory* argument and returning *Computer* object. At this point the implementation must be getting clear.

Let's write a simple test method and see how to use the abstract factory to get the instance of sub-classes.

```
package com.journaldev.design.test;

import com.journaldev.design.abstractfactory.PCFactory;
import com.journaldev.design.abstractfactory.ServerFactory;
import com.journaldev.design.factory.ComputerFactory;
import com.journaldev.design.model.Computer;

public class TestDesignPatterns {

    public static void main(String[] args) {
        testAbstractFactory();
    }

    private static void testAbstractFactory() {
        Computer pc =
com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new
PCFactory("2 GB","500 GB","2.4 GHz"));
        Computer server =
com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new
ServerFactory("16 GB","1 TB","2.9 GHz"));
        System.out.println("AbstractFactory PC Config::"+pc);
        System.out.println("AbstractFactory Server Config::"+server);
    }
}
```

```
Output of the above program will be:

AbstractFactory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz
AbstractFactory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz
```

Here is the class diagram of abstract factory implementation.

B. Benefits of Abstract Factory Pattern

- Abstract Factory pattern provides approach to code for interface rather than implementation.
- Abstract Factory pattern is "factory of factories" and can be easily extended to accommodate more products, for example we can add another sub-class Laptop and a factory LaptopFactory.
- Abstract Factory pattern is robust and avoid conditional logic of Factory pattern.

C. Abstract Factory Pattern Examples in JDK

- javax.xml.parsers.DocumentBuilderFactory#newInstance()
- javax.xml.transform.TransformerFactory#newInstance()
- javax.xml.xpath.XPathFactory#newInstance()

4. Builder Pattern

Builder design pattern is a **creational design pattern** like **Factory Pattern** and **Abstract Factory Pattern**. This pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

There are three major issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

1. Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side it's hard to maintain the order of the argument.
2. Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
3. If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.

We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters but the problem with this is that the Object state will be **inconsistent** until unless all the attributes are set explicitly.

Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

A. Builder Pattern Implementation

1. First of all you need to create a static nested class and then copy all the arguments from the outer class to the Builder class. We should follow the naming convention and if the class name is *Computer* then builder class should be named as *ComputerBuilder*.
2. The Builder class should have a public constructor with all the required attributes as parameters.

3. Builder class should have methods to set the optional parameters and it should return the same Builder object after setting the optional attribute.
4. The final step is to provide a *build()* method in the builder class that will return the Object needed by client program. For this we need to have a private constructor in the Class with Builder class as argument.

Here is the sample code where we have a Computer class and ComputerBuilder class to build it.

```
package com.journaldev.design.builder;

public class Computer {

    //required parameters
    private String HDD;
    private String RAM;

    //optional parameters
    private boolean isGraphicsCardEnabled;
    private boolean isBluetoothEnabled;


    public String getHDD() {
        return HDD;
    }

    public String getRAM() {
        return RAM;
    }

    public boolean isGraphicsCardEnabled() {
        return isGraphicsCardEnabled;
    }

    public boolean isBluetoothEnabled() {
        return isBluetoothEnabled;
    }

    private Computer(ComputerBuilder builder) {
        this.HDD=builder.HDD;
        this.RAM=builder.RAM;
        this.isGraphicsCardEnabled=builder.isGraphicsCardEnabled;
        this.isBluetoothEnabled=builder.isBluetoothEnabled;
```

```
    }

    //Builder Class
    public static class ComputerBuilder{

        // required parameters
        private String HDD;
        private String RAM;

        // optional parameters
        private boolean isGraphicsCardEnabled;
        private boolean isBluetoothEnabled;

        public ComputerBuilder(String hdd, String ram){
            this.HDD=hdd;
            this.RAM=ram;
        }

        public ComputerBuilder setGraphicsCardEnabled(boolean
isGraphicsCardEnabled) {
            this.isGraphicsCardEnabled = isGraphicsCardEnabled;
            return this;
        }

        public ComputerBuilder setBluetoothEnabled(boolean
isBluetoothEnabled) {
            this.isBluetoothEnabled = isBluetoothEnabled;
            return this;
        }

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

    }

}
```

Notice that Computer class has only getter methods and no public constructor, so the only way to get a Computer object is through the ComputerBuilder class.

Here is a test program showing how to use Builder class to get the object.

```
package com.journaldev.design.test;
```

```
import com.journaldev.design.builder.Computer;

public class TestBuilderPattern {

    public static void main(String[] args) {
        //Using builder to get the object in a single line of code and
                //without any inconsistent state or arguments
management issues
        Computer comp = new Computer.ComputerBuilder(
                "500 GB", "2 GB").setBluetoothEnabled(true)
                .setGraphicsCardEnabled(true).build();
    }

}
```

B. Builder Design Pattern Example in JDK

- java.lang.StringBuilder#append() (unsynchronized)
- java.lang.StringBuffer#append() (synchronized)

5. Prototype Pattern

Prototype pattern is one of the Creational Design pattern, so it provides a mechanism of object creation. Prototype pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing. So this pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. This pattern uses java cloning to copy the object.

It would be easy to understand this pattern with an example, suppose we have an Object that loads data from database. Now we need to modify this data in our program multiple times, so it's not a good idea to create the Object using *new* keyword and load all the data again from database. So the better approach is to clone the existing object into a new object and then do the data manipulation.

Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should not be done by any other class. However whether to use shallow or deep copy of the Object properties depends on the requirements and it's a design decision.

Here is a sample program showing implementation of Prototype pattern.

```
package com.journaldev.design.prototype;

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

public class Employees implements Cloneable{

    private List<String> empList;

    public Employees(){
        empList = new ArrayList<String>();
    }

    public Employees(List<String> list){
        this.empList=list;
    }
    public void loadData(){
        //read all employees from database and put into the list
```

```
        empList.add("Pankaj");
        empList.add("Raj");
        empList.add("David");
        empList.add("Lisa");
    }

    public List<String> getEmpList() {
        return empList;
    }

    @Override
    public Object clone() throws CloneNotSupportedException{
            List<String> temp = new ArrayList<String>();
            for(String s : this.getEmpList()){
                temp.add(s);
            }
            return new Employees(temp);
    }

}
```

Notice that the clone method is overridden to provide a deep copy of the employees list.

Here is the test program that will show the benefit of prototype pattern usage.

```
package com.journaldev.design.test;

import java.util.List;

import com.journaldev.design.prototype.Employees;

public class PrototypePatternTest {

    public static void main(String[] args) throws
CloneNotSupportedException {
        Employees emps = new Employees();
        emps.loadData();

        //Use the clone method to get the Employee object
        Employees empsNew = (Employees) emps.clone();
        Employees empsNew1 = (Employees) emps.clone();
        List<String> list = empsNew.getEmpList();
        list.add("John");
```

```
        List<String> list1 = empsNew1.getEmpList();
        list1.remove("Pankaj");

        System.out.println("emps List: "+emps.getEmpList());
        System.out.println("empsNew List: "+list);
        System.out.println("empsNew1 List: "+list1);
    }

}
```

Output of the above program is:

```
emps HashMap: [Pankaj, Raj, David, Lisa]
empsNew HashMap: [Pankaj, Raj, David, Lisa, John]
empsNew1 HashMap: [Raj, David, Lisa]
```

If the object cloning was not provided, every time we need to make database call to fetch the employee list and then do the manipulations that would have been resource and time consuming.

Structural Design Patterns

Structural patterns provide different ways to create a class structure, for example using inheritance and composition to create a large object from small objects.

1. Adapter Design Pattern

Adapter design pattern is one of the **structural design pattern** and it's used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an **Adapter**. As a real life example, we can think of a mobile charger as an adapter because mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket.

We will try to implement multi-adapter using adapter design pattern in this tutorial.

So first of all we will have two classes – Volt (to measure volts) and Socket (producing constant volts of 120V).

```
package com.journaldev.design.adapter;

public class Volt {

    private int volts;

    public Volt(int v){
        this.volts=v;
    }

    public int getVolts() {
        return volts;
    }

    public void setVolts(int volts) {
        this.volts = volts;
    }
```

```
}
```

```
package com.journaldev.design.adapter;

public class Socket {

    public Volt getVolt(){
        return new Volt(120);
    }
}
```

Now we want to build an adapter that can produce 3 volts, 12 volts and default 120 volts. So first of all we will create an adapter interface with these methods.

```
package com.journaldev.design.adapter;

public interface SocketAdapter {

    public Volt get120Volt();

    public Volt get12Volt();

    public Volt get3Volt();
}
```

A. Two Way Adapter Pattern

While implementing Adapter pattern, there are two approaches – class adapter and object adapter, however both these approaches produce same result.

1. **Class Adapter** – This form uses **java inheritance** and extends the source interface, in our case Socket class.
2. **Object Adapter** – This form uses **Java Composition** and adapter contains the source object.

B. Class Adapter Implementation

Here is the **class adapter** approach implementation of our adapter.

```
package com.journaldev.design.adapter;
```

```
//Using inheritance for adapter pattern
public class SocketClassAdapterImpl extends Socket implements
SocketAdapter{

    @Override
    public Volt get120Volt() {
        return getVolt();
    }

    @Override
    public Volt get12Volt() {
        Volt v= getVolt();
        return convertVolt(v,10);
    }

    @Override
    public Volt get3Volt() {
        Volt v= getVolt();
        return convertVolt(v,40);
    }

    private Volt convertVolt(Volt v, int i) {
        return new Volt(v.getVolts()/i);
    }

}
```

C. Object Adapter Implementation

Here is the **Object adapter** implementation of our adapter.

```
package com.journaldev.design.adapter;

public class SocketObjectAdapterImpl implements SocketAdapter{

    //Using Composition for adapter pattern
    private Socket sock = new Socket();

    @Override
    public Volt get120Volt() {
        return sock.getVolt();
    }
```

```
    @Override
    public Volt get12Volt() {
        Volt v= sock.getVolt();
        return convertVolt(v,10);
    }

    @Override
    public Volt get3Volt() {
        Volt v= sock.getVolt();
        return convertVolt(v,40);
    }

    private Volt convertVolt(Volt v, int i) {
        return new Volt(v.getVolts()/i);
    }
}
```

Notice that both the adapter implementations are almost same and they implement the *SocketAdapter* interface. The adapter interface can also be an **abstract class**.

Here is a test program to consume our adapter implementation.

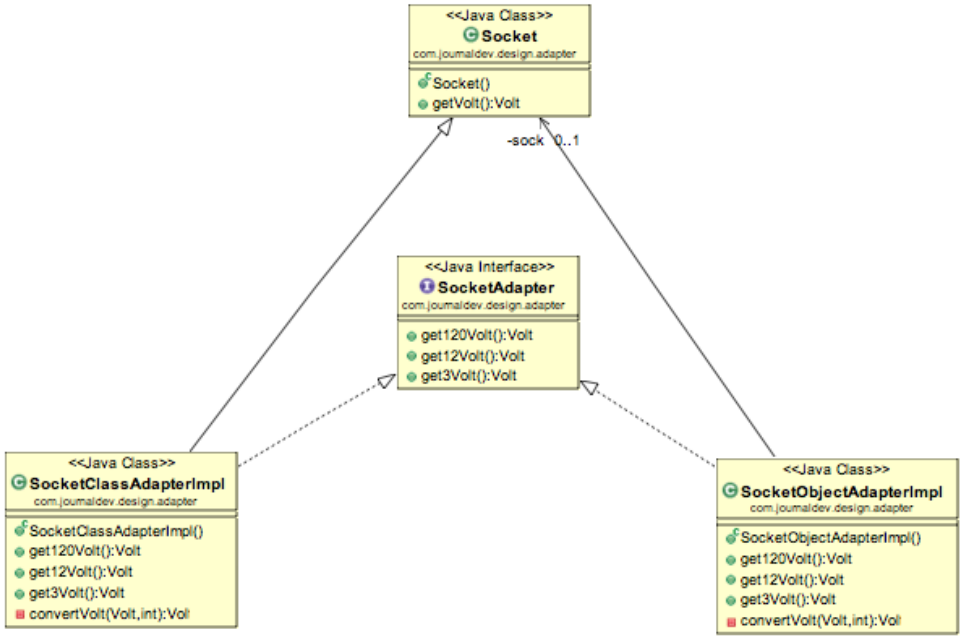
```
package com.journaldev.design.test;

import com.journaldev.design.adapter.SocketAdapter;
import com.journaldev.design.adapter.SocketClassAdapterImpl;
import com.journaldev.design.adapter.SocketObjectAdapterImpl;
import com.journaldev.design.adapter.Volt;

public class AdapterPatternTest {

    public static void main(String[] args) {

        testClassAdapter();
        testObjectAdapter();
    }

    private static void testObjectAdapter() {
        SocketAdapter sockAdapter = new SocketObjectAdapterImpl();
        Volt v3 = getVolt(sockAdapter,3);
        Volt v12 = getVolt(sockAdapter,12);
        Volt v120 = getVolt(sockAdapter,120);
```

```
        System.out.println("v3 volts using Object
Adapter="+v3.getVolts());
        System.out.println("v12 volts using Object
Adapter="+v12.getVolts());
        System.out.println("v120 volts using Object
Adapter="+v120.getVolts());
    }

    private static void testClassAdapter() {
        SocketAdapter sockAdapter = new SocketClassAdapterImpl();
        Volt v3 = getVolt(sockAdapter,3);
        Volt v12 = getVolt(sockAdapter,12);
        Volt v120 = getVolt(sockAdapter,120);
        System.out.println("v3 volts using Class
Adapter="+v3.getVolts());
        System.out.println("v12 volts using Class
Adapter="+v12.getVolts());
        System.out.println("v120 volts using Class
Adapter="+v120.getVolts());
    }

    private static Volt getVolt(SocketAdapter sockAdapter, int i) {
        switch (i){
        case 3: return sockAdapter.get3Volt();
        case 12: return sockAdapter.get12Volt();
        case 120: return sockAdapter.get120Volt();
        default: return sockAdapter.get120Volt();
        }
    }
}
```

When we run above test program, we get following output.

```
v3 volts using Class Adapter=3
v12 volts using Class Adapter=12
v120 volts using Class Adapter=120
v3 volts using Object Adapter=3
v12 volts using Object Adapter=12
v120 volts using Object Adapter=120
```

D. Adapter Pattern Class Diagram



E. Adapter Pattern Example in JDK

- java.util.Arrays#asList()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)

2. Composite Pattern

Composite pattern is one of the **Structural design pattern** and is used when we have to represent a part-whole hierarchy. When we need to create a structure in a way that the objects in the structure has to be treated the same way, we can apply composite design pattern.

Let's understand it with a real life example – A diagram is a structure that consists of Objects such as Circle, Lines, Triangle etc and when we fill the drawing with color (say Red), the same color also gets applied to the Objects in the drawing. Here drawing is made up of different parts and they all have same operations.

Composite Pattern consists of following objects.

1. **Base Component** – Base component is the interface for all objects in the composition, client program uses base component to work with the objects in the composition. It can be an interface or an **abstract class** with some methods common to all the objects.
2. **Leaf** – Defines the behaviour for the elements in the composition. It is the building block for the composition and implements base component. It doesn't have references to other Components.
3. **Composite** – It consists of leaf elements and implements the operations in base component.

Here I am applying composite design pattern for the drawing scenario.

A. Base Component

Base component defines the common methods for leaf and composites, we can create a *class Shape* with a method *draw(String fillColor)* to draw the shape with given color.

```
package com.journaldev.design.composite;

public interface Shape {

    public void draw(String fillColor);
}
```

B. Leaf Objects

Leaf implements base component and these are the building block for the composite. We can create multiple leaf objects such as Triangle, Circle etc.

```
package com.journaldev.design.composite;

public class Triangle implements Shape {

    @Override
    public void draw(String fillColor) {
        System.out.println("Drawing Triangle with color "+fillColor);
    }

}
```

```
package com.journaldev.design.composite;

public class Circle implements Shape {

    @Override
    public void draw(String fillColor) {
        System.out.println("Drawing Circle with color "+fillColor);
    }

}
```

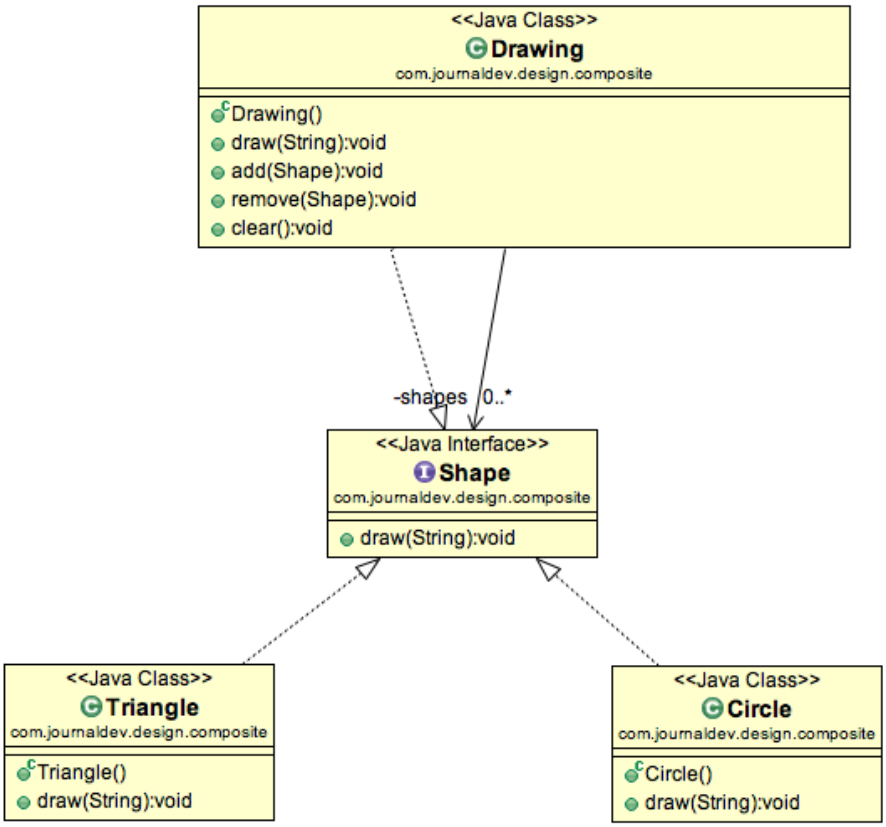
C. Composite

A composite object contains group of leaf objects and we should provide some helper methods to add or delete leafs from the group. We can also provide a method to remove all the elements from the group.

```
package com.journaldev.design.composite;

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

public class Drawing implements Shape{
```

```
    //collection of Shapes
    private List<Shape> shapes = new ArrayList<Shape>();

    @Override
    public void draw(String fillColor) {
        for(Shape sh : shapes)
        {
            sh.draw(fillColor);
        }
    }

    //adding shape to drawing
    public void add(Shape s){
        this.shapes.add(s);
    }

    //removing shape from drawing
    public void remove(Shape s){
        shapes.remove(s);
    }

    //removing all the shapes
    public void clear(){
        System.out.println("Clearing all the shapes from drawing");
        this.shapes.clear();
    }
}
```

Notice that composite also implements component and behaves similar to leaf except that it can contain group of leaf elements.



Our composite pattern implementation is ready and we can test it with a client program.

```
package com.journaldev.design.test;

import com.journaldev.design.composite.Circle;
import com.journaldev.design.composite.Drawing;
import com.journaldev.design.composite.Shape;
import com.journaldev.design.composite.Triangle;

public class TestCompositePattern {

    public static void main(String[] args) {
        Shape tri = new Triangle();
        Shape tri1 = new Triangle();
        Shape cir = new Circle();

        Drawing drawing = new Drawing();
        drawing.add(tri1);
        drawing.add(tri1);
```

```
        drawing.add(cir);

        drawing.draw("Red");

        drawing.clear();

        drawing.add(tri);
        drawing.add(cir);
        drawing.draw("Green");
    }

}
```

Output of the above program is:

```
Drawing Triangle with color Red
Drawing Triangle with color Red
Drawing Circle with color Red
Clearing all the shapes from drawing
Drawing Triangle with color Green
Drawing Circle with color Green
```

D. Important Points about Composite Pattern

- Composite pattern should be applied only when the group of objects should behave as the single object.
- Composite pattern can be used to create a tree like structure.

java.awt.Container#add(Component) is a great example of Composite pattern in java and used a lot in Swing.

3. Proxy Pattern

Proxy Design pattern is one of the **Structural design pattern** and in my opinion one of the simplest pattern to understand. Proxy pattern intent according to GoF is:

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

The definition itself is very clear and proxy pattern is used when we want to provide controlled access of a functionality. Let's say we have a class that can run some command on the system. Now if we are using it, its fine but if we want to give this program to a client application, it can have severe issues because client program can issue command to delete some system files or change some settings that you don't want. Here a proxy class can be created to provide controlled access of the program.

A. Main Class

Since we code Java in terms of interfaces, here is our interface and its implementation class.

```
package com.journaldev.design.proxy;

public interface CommandExecutor {

    public void runCommand(String cmd) throws Exception;
}

package com.journaldev.design.proxy;

import java.io.IOException;

public class CommandExecutorImpl implements CommandExecutor {

    @Override
    public void runCommand(String cmd) throws IOException {
```

```
            //some heavy implementation
        Runtime.getRuntime().exec(cmd);
        System.out.println("'" + cmd + "' command executed.");
    }

}
```

B. Proxy Class

Now we want to provide only admin users to have full access of above class, if the user is not admin then only limited commands will be allowed. Here is our very simple proxy class implementation.

```
package com.journaldev.design.proxy;

public class CommandExecutorProxy implements CommandExecutor {

    private boolean isAdmin;
    private CommandExecutor executor;

    public CommandExecutorProxy(String user, String pwd){
        if("Pankaj".equals(user) && "J@urnalD$v".equals(pwd))
isAdmin=true;
        executor = new CommandExecutorImpl();
    }

    @Override
    public void runCommand(String cmd) throws Exception {
        if(isAdmin){
            executor.runCommand(cmd);
        }else{
            if(cmd.trim().startsWith("rm")){
                throw new Exception("rm command is not allowed for non-
admin users.");
            }else{
                executor.runCommand(cmd);
            }
        }
    }

}
```

C. Proxy Pattern Client Test Program

```
package com.journaldev.design.test;

import com.journaldev.design.proxy.CommandExecutor;
import com.journaldev.design.proxy.CommandExecutorProxy;

public class ProxyPatternTest {

    public static void main(String[] args){
        CommandExecutor executor = new CommandExecutorProxy("Pankaj",
"wrong_pwd");
        try {
            executor.runCommand("ls -ltr");
            executor.runCommand(" rm -rf abc.pdf");
        } catch (Exception e) {
            System.out.println("Exception Message::"+e.getMessage());
        }

    }

}
```

Output of above test program is:

```
'ls -ltr' command executed.
Exception Message::rm command is not allowed for non-admin users.
```

Proxy pattern common uses are to control access or to provide a wrapper implementation for better performance.

Java RMI whole package uses proxy pattern.

4. Flyweight Pattern

According to GoF, **flyweight design pattern** intent is:

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

Flyweight design pattern is a **Structural design pattern** like Facade pattern, Adapter Pattern and Decorator pattern. Flyweight design pattern is used when we need to create a lot of Objects of a class. Since every object consumes memory space that can be crucial for low memory devices, such as mobile devices or embedded systems, flyweight design pattern can be applied to reduce the load on memory by sharing objects.

Before we apply flyweight design pattern, we need to consider following factors:

- The number of Objects to be created by application should be huge.
- The object creation is heavy on memory and it can be time consuming too.
- The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of an Object should be defined by the client program.

To apply flyweight pattern, we need to divide Object property into **intrinsic** and **extrinsic** properties. Intrinsic properties make the Object unique whereas extrinsic properties are set by client code and used to perform different operations. For example, an Object Circle can have extrinsic properties such as color and width.

For applying flyweight pattern, we need to create a **Flyweight factory** that returns the shared objects. For our example, let's say we need to create a drawing with lines and Ovals. So we will have an interface Shape and its concrete implementations as *Line* and *Oval*. Oval class will have intrinsic property to determine whether to fill the Oval with given color or not whereas Line will not have any intrinsic property.

A. Flyweight Interface and Concrete Classes

```
package com.journaldev.design.flyweight;

import java.awt.Color;
import java.awt.Graphics;

public interface Shape {

    public void draw(Graphics g, int x, int y, int width, int height,
            Color color);
}
```

package com.journaldev.design.flyweight;

```
import java.awt.Color;
import java.awt.Graphics;

public class Line implements Shape {

    public Line(){
        System.out.println("Creating Line object");
        //adding time delay
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void draw(Graphics line, int x1, int y1, int x2, int y2,
            Color color) {
        line.setColor(color);
        line.drawLine(x1, y1, x2, y2);
    }

}

package com.journaldev.design.flyweight;

import java.awt.Color;
import java.awt.Graphics;

public class Oval implements Shape {
```

```
    //intrinsic property
    private boolean fill;

    public Oval(boolean f){
        this.fill=f;
        System.out.println("Creating Oval object with fill="+f);
        //adding time delay
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void draw(Graphics circle, int x, int y, int width, int
height,
            Color color) {
        circle.setColor(color);
        circle.drawOval(x, y, width, height);
        if(fill){
            circle.fillOval(x, y, width, height);
        }
    }

}
```

Notice that I have intentionally introduced delay in creating the Object of concrete classes to make the point that flyweight pattern can be used for Objects that takes a lot of time while instantiated.

B. Flyweight Factory

The flyweight factory will be used by client programs to instantiate the Object, so we need to keep a map of Objects in the factory that should not be accessible by client application. Whenever client program makes a call to get an instance of Object, it should be returned from the HashMap, if not found then create a new Object and put in the Map and then return it. We need to make sure that all the intrinsic properties are considered while creating the Object.

Our flyweight factory class looks like below code.

```
package com.journaldev.design.flyweight;

import java.util.HashMap;

public class ShapeFactory {

    private static final HashMap<ShapeType,Shape> shapes = new
HashMap<ShapeType,Shape>();

    public static Shape getShape(ShapeType type) {
        Shape shapeImpl = shapes.get(type);

        if (shapeImpl == null) {
            if (type.equals(ShapeType.OVAL_FILL)) {
                shapeImpl = new Oval(true);
            } else if (type.equals(ShapeType.OVAL_NOFILL)) {
                shapeImpl = new Oval(false);
            } else if (type.equals(ShapeType.LINE)) {
                shapeImpl = new Line();
            }
            shapes.put(type, shapeImpl);
        }
        return shapeImpl;
    }

    public static enum ShapeType{
        OVAL_FILL,OVAL_NOFILL,LINE;
    }
}
```

Notice the use of Java Enum for type safety, Java Composition (shapes map) and Factory pattern in *getShape* method.

C. Flyweight Pattern Client Example

Below is a sample program that consumes flyweight pattern implementation.

```
package com.journaldev.design.flyweight;

import java.awt.BorderLayout;
import java.awt.Color;
```

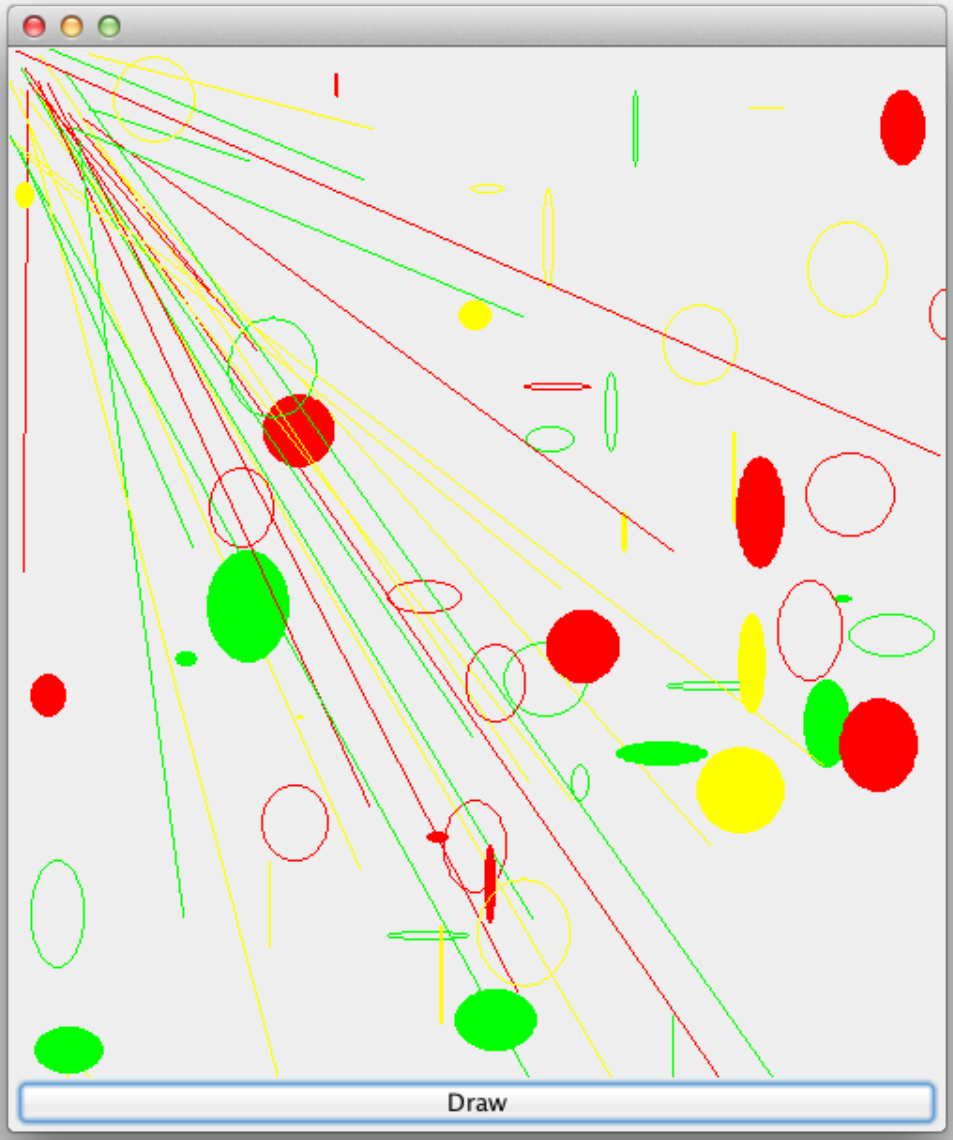
```
import java.awt.Container;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

import com.journaldev.design.flyweight.ShapeFactory.ShapeType;

public class DrawingClient extends JFrame{

    private static final long serialVersionUID = -1350200437285282550L;
    private final int WIDTH;
    private final int HEIGHT;

    private static final ShapeType shapes[] = { ShapeType.LINE,
ShapeType.OVAL_FILL,ShapeType.OVAL_NOFILL };
    private static final Color colors[] = { Color.RED, Color.GREEN,
Color.YELLOW };

    public DrawingClient(int width, int height){
        this.WIDTH=width;
        this.HEIGHT=height;
        Container contentPane = getContentPane();

        JButton startButton = new JButton("Draw");
        final JPanel panel = new JPanel();

        contentPane.add(panel, BorderLayout.CENTER);
        contentPane.add(startButton, BorderLayout.SOUTH);
        setSize(WIDTH, HEIGHT);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

        startButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                Graphics g = panel.getGraphics();
                for (int i = 0; i < 20; ++i) {
                    Shape shape =
ShapeFactory.getShape(getRandomShape());
                    shape.draw(g, getRandomX(), getRandomY(),
getRandomWidth(),
                            getRandomHeight(), getRandomColor());
```

```
                }
            }
        });
    }

    private ShapeType getRandomShape() {
        return shapes[(int) (Math.random() * shapes.length)];
    }

    private int getRandomX() {
        return (int) (Math.random() * WIDTH);
    }

    private int getRandomY() {
        return (int) (Math.random() * HEIGHT);
    }

    private int getRandomWidth() {
        return (int) (Math.random() * (WIDTH / 10));
    }

    private int getRandomHeight() {
        return (int) (Math.random() * (HEIGHT / 10));
    }

    private Color getRandomColor() {
        return colors[(int) (Math.random() * colors.length)];
    }

    public static void main(String[] args) {
        DrawingClient drawing = new DrawingClient(500,600);
    }
}
```

I have used random number generation to generate different type of Shapes in our frame.

If you run above client program, you will notice the delay in creating first Line Object and Oval objects with fill as true and false. After that the program executes quickly since its using the shared objects.

After clicking “Draw” button multiple times, the frame looks like below image.



And you will see following output in command line confirming that Objects are shared.

```
Creating Line object
Creating Oval object with fill=true
Creating Oval object with fill=false
```

That's all for flyweight pattern, we will look into more design patterns in future posts. If you liked it, please share your thoughts in comments section and share it with others too.

D. Flyweight Pattern Example in JDK

All the wrapper classes valueOf() method uses cached objects showing use of Flyweight design pattern. The best example is Java String class String Pool implementation.

E. Important Points

- In our example, the client code is not forced to create object using Flyweight factory but we can force that to make sure client code uses flyweight pattern implementation but its a complete design decision for particular application.
- Flyweight pattern introduces complexity and if number of shared objects are huge then there is a trade of between memory and time, so we need to use it judiciously based on our requirements.
- Flyweight pattern implementation is not useful when the number of intrinsic properties of Object is huge, making implementation of Factory class complex.

5. Facade Pattern

Facade Pattern is one of the **Structural design patterns** (such as Adapter pattern and Decorator pattern) and used to help client applications to easily interact with the system.

According to GoF Facade design pattern is:

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

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

Suppose we have an application with set of interfaces to use MySql/Oracle database and to generate different types of reports, such as HTML report, PDF report etc. So we will have different set of interfaces to work with different types of database. Now a client application can use these interfaces to get the required database connection and generate reports. But when the complexity increases or the interface behavior names are confusing, client application will find it difficult to manage it. So we can apply Facade pattern here and provide a wrapper interface on top of the existing interface to help client application.

A. Set of Interfaces

We can have two helper interfaces, namely MySqlHelper and OracleHelper.

```
package com.journaldev.design.facade;

import java.sql.Connection;

public class MySqlHelper {
```

```
    public static Connection getMySqlDBConnection(){
        //get MySql DB connection using connection parameters
        return null;
    }

    public void generateMySqlPDFReport(String tableName, Connection
con){
        //get data from table and generate pdf report
    }

    public void generateMySqlHTMLReport(String tableName, Connection
con){
        //get data from table and generate pdf report
    }
}
```

```
package com.journaldev.design.facade;

import java.sql.Connection;

public class OracleHelper {

    public static Connection getOracleDBConnection(){
        //get MySql DB connection using connection parameters
        return null;
    }

    public void generateOraclePDFReport(String tableName, Connection
con){
        //get data from table and generate pdf report
    }

    public void generateOracleHTMLReport(String tableName, Connection
con){
        //get data from table and generate pdf report
    }

}
```

B. Facade Interface

We can create a Facade interface like below. Notice the use of Java Enum for type safety.

```
package com.journaldev.design.facade;

import java.sql.Connection;

public class HelperFacade {

    public static void generateReport(DBTypes dbType, ReportTypes
reportType, String tableName){
        Connection con = null;
        switch (dbType){
        case MYSQL:
            con = MySqlHelper.getMySqlDBConnection();
            MySqlHelper mySqlHelper = new MySqlHelper();
            switch(reportType){
            case HTML:
                mySqlHelper.generateMySqlHTMLReport(tableName, con);
                break;
            case PDF:
                mySqlHelper.generateMySqlPDFReport(tableName, con);
                break;
            }
            break;
        case ORACLE:
            con = OracleHelper.getOracleDBConnection();
            OracleHelper oracleHelper = new OracleHelper();
            switch(reportType){
            case HTML:
                oracleHelper.generateOracleHTMLReport(tableName, con);
                break;
            case PDF:
                oracleHelper.generateOraclePDFReport(tableName, con);
                break;
            }
            break;
        }

    }
    public static enum DBTypes{
        MYSQL,ORACLE;
    }
    public static enum ReportTypes{
        HTML,PDF;
    }
}
```

C. Client Program

Now let's see client code without using Facade and using Facade interface.

```
package com.journaldev.design.test;

import java.sql.Connection;

import com.journaldev.design.facade.HelperFacade;
import com.journaldev.design.facade.MySqlHelper;
import com.journaldev.design.facade.OracleHelper;

public class FacadePatternTest {

    public static void main(String[] args) {
        String tableName="Employee";

        //generating MySql HTML report and Oracle PDF report without
using Facade
        Connection con = MySqlHelper.getMySqlDBConnection();
        MySqlHelper mySqlHelper = new MySqlHelper();
        mySqlHelper.generateMySqlHTMLReport(tableName, con);

        Connection con1 = OracleHelper.getOracleDBConnection();
        OracleHelper oracleHelper = new OracleHelper();
        oracleHelper.generateOraclePDFReport(tableName, con1);

        //generating MySql HTML report and Oracle PDF report using
Facade
        HelperFacade.generateReport(HelperFacade.DBTypes.MYSQL,
HelperFacade.ReportTypes.HTML, tableName);
        HelperFacade.generateReport(HelperFacade.DBTypes.ORACLE,
HelperFacade.ReportTypes.PDF, tableName);
    }

}
```

As you can see that using Facade interface is a lot easier and cleaner way and avoid having a lot of logic at client side. JDBC Driver Manager Class to get the database connection is a wonderful example of facade pattern.

D. Important Points

- Facade pattern is more like a helper for client applications, it doesn't hide subsystem interfaces from the client. Whether to use Facade or not is completely dependent on client code.
- Facade pattern can be applied at any point of development, usually when the number of interfaces grow and system gets complex.
- Subsystem interfaces are not aware of Facade and they shouldn't have any reference of the Facade interface.
- Facade pattern should be applied for similar kind of interfaces, its purpose is to provide a single interface rather than multiple interfaces that does the similar kind of jobs.
- We can use Factory pattern with Facade to provide better interface to client systems.

6. Bridge Pattern

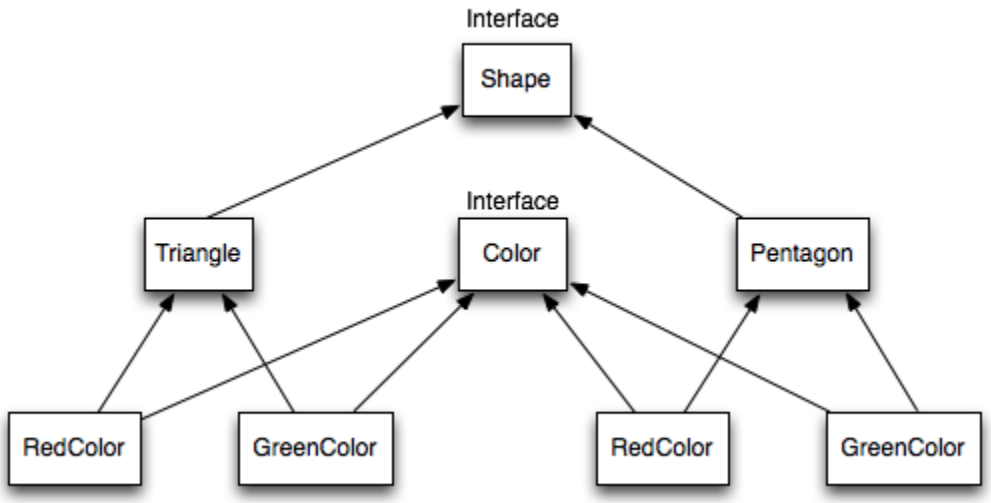
When we have interface hierarchies in both interfaces as well as implementations, then **builder design pattern** is used to decouple the interfaces from implementation and hiding the implementation details from the client programs. Like Adapter pattern, its one of the **Structural design pattern**.

According to GoF bridge design pattern is:

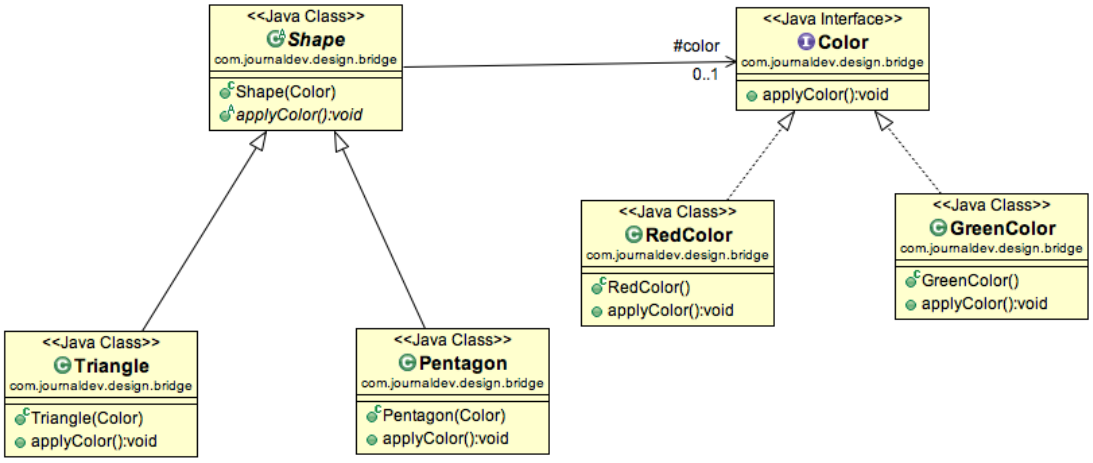
> "Decouple an abstraction from its implementation so that the two can vary independently"

The implementation of bridge design pattern follows the notion to prefer Composition over inheritance.

If we look into this design pattern with example, it will be easy to understand. Let's say we have an interface hierarchy in both interfaces and implementations like below image.



Now we will use bridge design pattern to decouple the interfaces from implementation and the UML diagram for the classes and interfaces after applying bridge pattern will look like below image.



Notice the bridge between *Shape* and *Color* interfaces and use of composition in implementing the bridge pattern.

Here is the java code for Shape and Color interfaces.

```
package com.journaldev.design.bridge;

public interface Color {

    public void applyColor();
}
package com.journaldev.design.bridge;

public abstract class Shape {
    //Composition - implementor
    protected Color color;

    //constructor with implementor as input argument
    public Shape(Color c){
        this.color=c;
    }

    abstract public void applyColor();
}
```

We have Triangle and Pentagon implementation classes as below.

```
package com.journaldev.design.bridge;

public class Triangle extends Shape{

    public Triangle(Color c) {
        super(c);
    }
    @Override
    public void applyColor() {
        System.out.print("Triangle filled with color ");
        color.applyColor();
    }
}
package com.journaldev.design.bridge;

public class Pentagon extends Shape{

    public Pentagon(Color c) {
        super(c);
    }

    @Override
    public void applyColor() {
        System.out.print("Pentagon filled with color ");
        color.applyColor();
    }

}
```

Here are the implementation classes for RedColor and GreenColor.

```
package com.journaldev.design.bridge;

public class RedColor implements Color{

    public void applyColor(){
        System.out.println("red.");
    }
}
package com.journaldev.design.bridge;

public class GreenColor implements Color{

    public void applyColor(){
        System.out.println("green.");
    }
}
```

Let's test our bridge pattern implementation with a test program.

```
package com.journaldev.design.test;

import com.journaldev.design.bridge.GreenColor;
import com.journaldev.design.bridge.Pentagon;
import com.journaldev.design.bridge.RedColor;
import com.journaldev.design.bridge.Shape;
import com.journaldev.design.bridge.Triangle;

public class BridgePatternTest {

    public static void main(String[] args) {
        Shape tri = new Triangle(new RedColor());
        tri.applyColor();

        Shape pent = new Pentagon(new GreenColor());
        pent.applyColor();
    }

}
```

Output of above class is:

```
Triangle filled with color red.
Pentagon filled with color green.
```

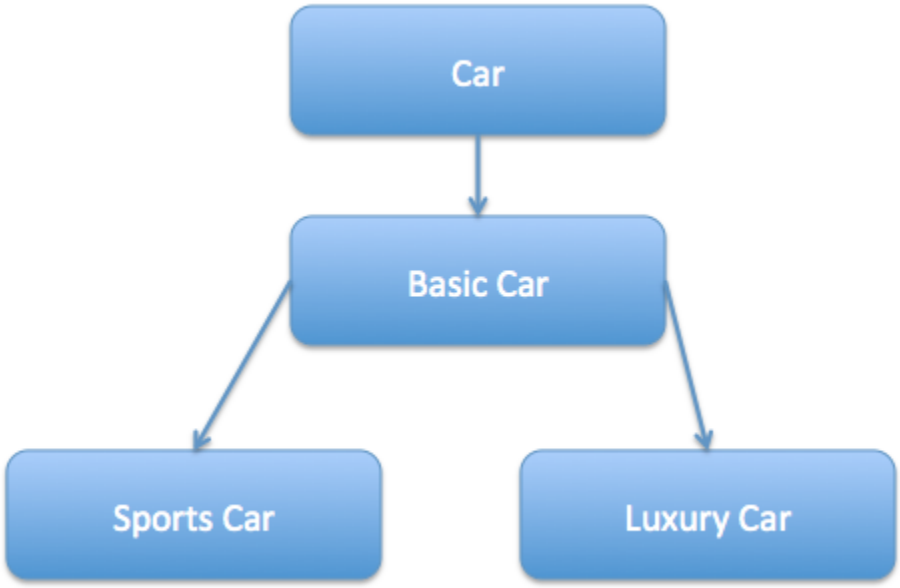
Bridge design pattern can be used when both abstraction and implementation can have different hierarchies independently and we want to hide the implementation from the client application.

7. Decorator Pattern

Decorator design pattern is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior. Decorator design pattern is one of the structural design pattern (such as Adapter Pattern, Bridge Pattern, Composite Pattern) and uses abstract classes or interface with composition to implement.

We use inheritance or composition to extend the behavior of an object but this is done at compile time and its applicable to all the instances of the class. We can't add any new functionality of remove any existing behavior at runtime – this is when Decorator pattern comes into picture.

Suppose we want to implement different kinds of cars – we can create interface Car to define the assemble method and then we can have a Basic car, further more we can extend it to Sports car and Luxury Car. The implementation hierarchy will look like below image.



But if we want to get a car at runtime that has both the features of sports car and luxury car, then the implementation gets complex and if further more we want to specify which features should be added first, it gets even more complex. Now image if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage. To solve this kind of programming situation, we apply decorator pattern.

We need to have following types to implement decorator design pattern.

A. Component Interface

The interface or **abstract class** defining the methods that will be implemented. In our case *Car* will be the component interface.

```
package com.journaldev.design.decorator;

public interface Car {

    public void assemble();
}
```

B. Component Implementation

The basic implementation of the component interface. We can have BasicCar class as our component implementation.

```
package com.journaldev.design.decorator;

public class BasicCar implements Car {

    @Override
    public void assemble() {
        System.out.print("Basic Car.");
    }

}
```

C. Decorator

Decorator class implements the component interface and it has a HAS-A relationship with the component interface. The component variable should be accessible to the child decorator classes, so we will make this variable protected.

```
package com.journaldev.design.decorator;

public class CarDecorator implements Car {

    protected Car car;

    public CarDecorator(Car c){
        this.car=c;
    }

    @Override
    public void assemble() {
        this.car.assemble();
    }

}
```

D: Concrete Decorators

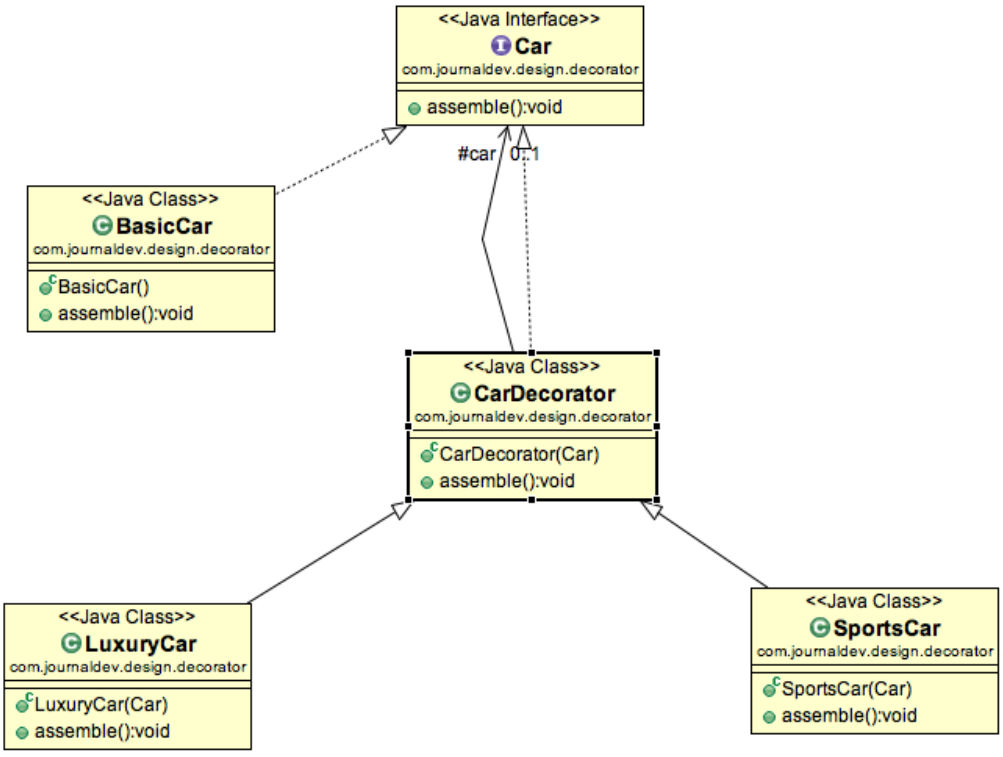
Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as LuxuryCar and SportsCar.

```
package com.journaldev.design.decorator;

public class SportsCar extends CarDecorator {

    public SportsCar(Car c) {
        super(c);
    }

    @Override
    public void assemble(){
        car.assemble();
```

```
        System.out.print(" Adding features of Sports Car.");
    }
}
```

```
package com.journaldev.design.decorator;

public class LuxuryCar extends CarDecorator {

    public LuxuryCar(Car c) {
        super(c);
    }

    @Override
    public void assemble(){
        car.assemble();
        System.out.print(" Adding features of Luxury Car.");
    }
}
```

D. Decorator Pattern Class Diagram



E. Decorator Pattern Client Program

```
package com.journaldev.design.test;

import com.journaldev.design.decorator.BasicCar;
import com.journaldev.design.decorator.Car;
import com.journaldev.design.decorator.LuxuryCar;
import com.journaldev.design.decorator.SportsCar;

public class DecoratorPatternTest {

    public static void main(String[] args) {
        Car sportsCar = new SportsCar(new BasicCar());
        sportsCar.assemble();
        System.out.println("\n*****");

        Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new
BasicCar()));
        sportsLuxuryCar.assemble();
    }
}
```

Notice that client program can create different kinds of Object at runtime and they can specify the order of execution too.

Output of above test program is:

```
Basic Car. Adding features of Sports Car.
*****
Basic Car. Adding features of Luxury Car. Adding features of Sports
Car.
```

F. Important Points

- Decorator pattern is helpful in providing runtime modification abilities and hence more flexible. It's easy to maintain and extend when the number of choices are more.
- The disadvantage of decorator pattern is that it uses a lot of similar kind of objects (decorators).
- Decorator pattern is used a lot in Java IO classes, such as FileReader, BufferedReader etc.

Behavioral Design Patterns

Behavioral patterns provide solution for the better interaction between objects and how to provide lose coupling and flexibility to extend easily.

1. Template Method Pattern

Template Method is a **behavioral design pattern** and it's used to create a method stub and deferring some of the steps of implementation to the subclasses. **Template method** defines the steps to execute an algorithm and it can provide default implementation that might be common for all or some of the subclasses.

Let's understand this pattern with an example, suppose we want to provide an algorithm to build a house. The steps need to be performed to build a house are – building foundation, building pillars, building walls and windows. The important point is that we can't change the order of execution because we can't build windows before building the foundation. So in this case we can create a template method that will use different methods to build the house.

Now building the foundation for a house is same for all type of houses, whether it's a wooden house or a glass house. So we can provide base implementation for this, if subclasses want to override this method, they can but mostly it's common for all the types of houses.

To make sure that subclasses don't override the template method, we should make it final.

A. Template Method Abstract Class

Since we want some of the methods to be implemented by subclasses, we have to make our base class as abstract class.

```
package com.journaldev.design.template;

public abstract class HouseTemplate {

    //template method, final so subclasses can't override
    public final void buildHouse(){
        buildFoundation();
        buildPillars();
        buildWalls();
        buildWindows();
        System.out.println("House is built.");
    }

    //default implementation
    private void buildWindows() {
        System.out.println("Building Glass Windows");
    }

    //methods to be implemented by subclasses
    public abstract void buildWalls();
    public abstract void buildPillars();

    private void buildFoundation() {
        System.out.println("Building foundation with cement,iron rods
and sand");
    }
}
```

buildHouse() is the template method and defines the order of execution for performing several steps.

B. Template Method Concrete Classes

We can have different type of houses, such as Wooden House and Glass House.

```
package com.journaldev.design.template;

public class WoodenHouse extends HouseTemplate {

    @Override
```

```
    public void buildWalls() {
        System.out.println("Building Wooden Walls");
    }

    @Override
    public void buildPillars() {
        System.out.println("Building Pillars with Wood coating");
    }

}
```

We could have overridden other methods also, but for simplicity I am not doing that.

```
package com.journaldev.design.template;

public class GlassHouse extends HouseTemplate {

    @Override
    public void buildWalls() {
        System.out.println("Building Glass Walls");
    }

    @Override
    public void buildPillars() {
        System.out.println("Building Pillars with glass coating");
    }

}
```

C. Template Method Pattern Client

Let's test our template method pattern example with a test program.

```
package com.journaldev.design.template;

public class HousingClient {

    public static void main(String[] args) {

        HouseTemplate houseType = new WoodenHouse();
```

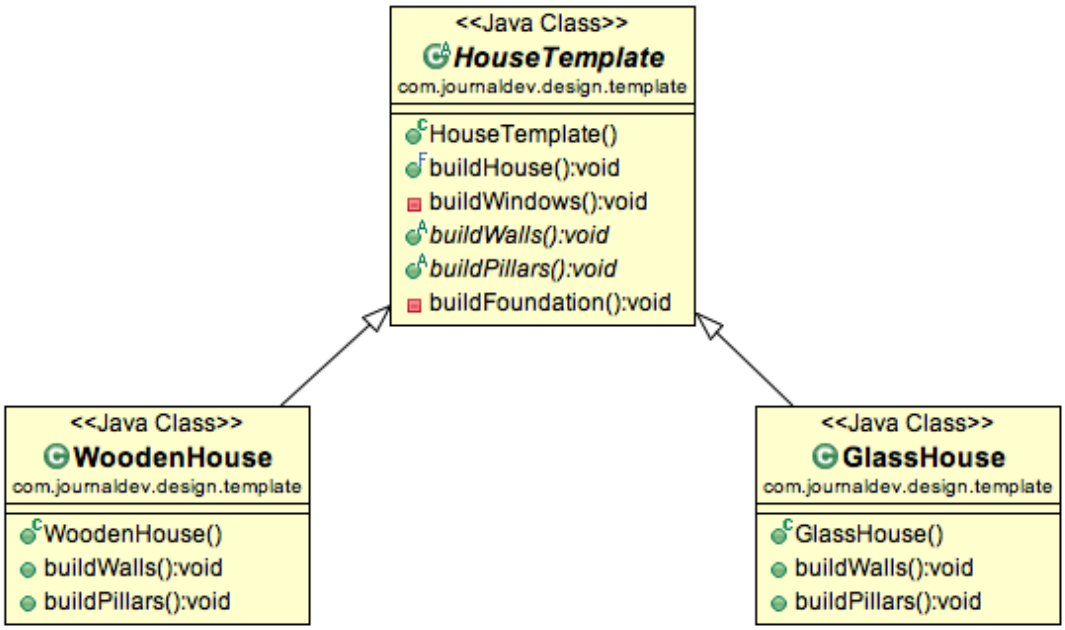
```
        //using template method
        houseType.buildHouse();
        System.out.println("************");

        houseType = new GlassHouse();

        houseType.buildHouse();
    }

}
```

Notice that client is invoking the template method of base class and depending of implementation of different steps, it's using some of the methods from base class and some of them from subclass.

Output of the above program is:

```
Building foundation with cement,iron rods and sand
Building Pillars with Wood coating
Building Wooden Walls
Building Glass Windows
House is built.
************
Building foundation with cement,iron rods and sand
Building Pillars with glass coating
Building Glass Walls
Building Glass Windows
House is built.
```

D. Template Method Class Diagram



E. Template Method Pattern in JDK

- All non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Reader and java.io.Writer.
- All non-abstract methods of java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap.

F. Important Points

- Template method should consists of certain steps whose order is fixed and for some of the methods, implementation differs from base class to subclass. Template method should be final.
- Most of the times, subclasses calls methods from super class but in template pattern, superclass template method calls methods from subclasses, this is known as Hollywood Principle – "don't call us, we'll call you".
- Methods in base class with default implementation are referred as **Hooks** and they are intended to be overridden by subclasses, if you want some of the methods to be not overridden, you can make them final, for example in our case we can make buildFoundation() method final because if we don't want subclasses to override it.

2. Mediator Pattern

Mediator Pattern is one of the **behavioral design pattern**, so it deals with the behaviors of objects. Mediator design pattern is used to provide a centralized communication medium between different objects in a system. According to GoF, mediator pattern intent is:

> "Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another"

Mediator design pattern is very helpful in an enterprise application where multiple objects are interacting with each other. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes maintainability cost higher and not flexible to extend easily. Mediator pattern focuses on provide a mediator between objects for communication and help in implementing lose-coupling between objects.

Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have it's own logic to provide way of communication.

The system objects that communicate each other are called Colleagues. Usually we have an interface or abstract class that provides the contract for communication and then we have concrete implementation of mediators.

For our example, we will try to implement a chat application where users can do group chat. Every user will be identified by its name and they can send and receive messages. The message sent by any user should be received by all the other users in the group.

A. Mediator Interface

First of all we will create Mediator interface that will define the contract for concrete mediators.

```
package com.journaldev.design.mediator;

public interface ChatMediator {

    public void sendMessage(String msg, User user);

    void addUser(User user);
}
```

B. Colleague Interface

Users can send and receive messages, so we can have User interface or abstract class. I am creating User as abstract class like below.

```
package com.journaldev.design.mediator;

public abstract class User {
    protected ChatMediator mediator;
    protected String name;

    public User(ChatMediator med, String name){
        this.mediator=med;
        this.name=name;
    }

    public abstract void send(String msg);

    public abstract void receive(String msg);
}
```

Notice that User has a reference to the mediator object, it's required for the communication between different users.

C. Concrete Mediator

Now we will create concrete mediator class, it will have a list of users in the group and provide logic for the communication between the users.

```
package com.journaldev.design.mediator;

import java.util.ArrayList;
import java.util.List;
 public class ChatMediatorImpl implements ChatMediator {
 private List<User> users;
     public ChatMediatorImpl(){
         this.users=new ArrayList<>();
    }
     @Override
    public void addUser(User user){
         this.users.add(user);
    }
     @Override
    public void sendMessage(String msg, User user) {
         for(User u : this.users){
             //message should not be received by the user sending it
             if(u != user){
                 u.receive(msg);
             }
         }
    }
}
```

C. Concrete Colleague

Now we can create concrete User classes to be used by client system.

```
package com.journaldev.design.mediator;
 public class UserImpl extends User {
 public UserImpl(ChatMediator med, String name) {
         super(med, name);
    }
 @Override
    public void send(String msg){
         System.out.println(this.name+": Sending Message="+msg);
         mediator.sendMessage(msg, this);
    }
```

```
    @Override
    public void receive(String msg) {
        System.out.println(this.name+": Received Message:"+msg);
    }
}
```

Notice that send() method is using mediator to send the message to the users and it has no idea how it will be handled by the mediator.

D. Mediator Pattern Client

Let's test this our chat application with a simple program where we will create mediator and add users to the group and one of the user will send a message.

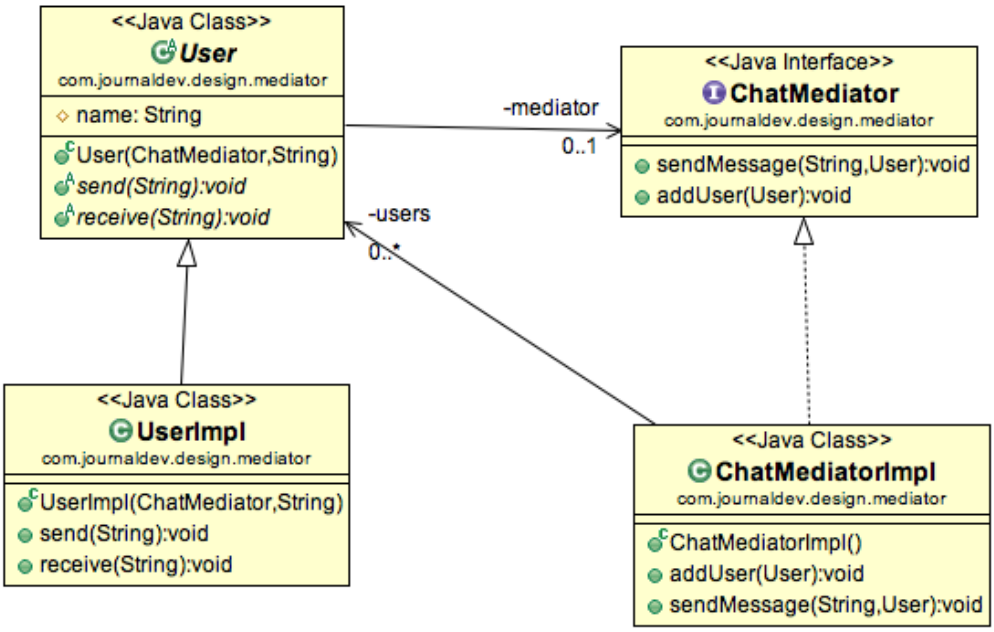
```
package com.journaldev.design.mediator;

public class ChatClient {

    public static void main(String[] args) {
        ChatMediator mediator = new ChatMediatorImpl();
        User user1 = new UserImpl(mediator, "Pankaj");
        User user2 = new UserImpl(mediator, "Lisa");
        User user3 = new UserImpl(mediator, "Saurabh");
        User user4 = new UserImpl(mediator, "David");
        mediator.addUser(user1);
        mediator.addUser(user2);
        mediator.addUser(user3);
        mediator.addUser(user4);

        user1.send("Hi All");
    }
}
```

Notice that client program is very simple and it has no idea how the message is getting handled and if mediator is getting user or not.

Output of the above program is:

```
Pankaj: Sending Message=Hi All
Lisa: Received Message:Hi All
Saurabh: Received Message:Hi All
David: Received Message:Hi All
```

E. Mediator Pattern Class Diagram



F. Mediator Pattern in JDK

java.util.Timer class scheduleXXX() methods
Java Concurrency Executor execute() method.
java.lang.reflect.Method invoke() method.

G. Important Points

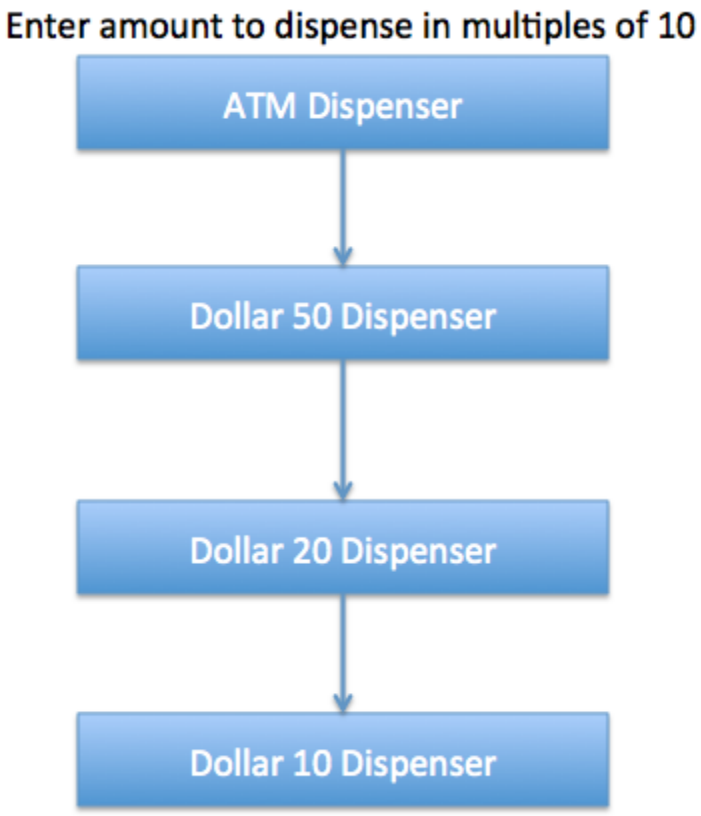
- Mediator pattern is useful when the communication logic between objects is complex, we can have a central point of communication that takes care of communication logic.
- Java Message Service (JMS) uses Mediator pattern along with **Observer pattern** to allow applications to subscribe and publish data to other applications.
- We should not use mediator pattern just to achieve lose-coupling because if the number of mediators will grow, then it will become hard to maintain them.

3. Chain of Responsibility Pattern

Chain of responsibility design pattern is one of the **behavioral design pattern**. Chain of responsibility pattern is used to achieve **lose coupling** in software design where a request from client is passed to a chain of objects to process them. Then the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

Let's see the example of chain of responsibility pattern in JDK and then we will proceed to implement a real life example of this pattern. We know that we can have multiple catch blocks in a try-catch block code. Here every catch block is kind of a processor to process that particular exception. So when any exception occurs in the try block, its send to the first catch block to process. If the catch block is not able to process it, it forwards the request to next object in chain i.e next catch block. If even the last catch block is not able to process it, the exception is thrown outside of the chain to the calling program.

One of the great example of Chain of Responsibility pattern is **ATM Dispense machine**. The user enters the amount to be dispensed and the machine dispense amount in terms of defined currency bills such as 50$, 20$, 10$ etc. If the user enters an amount that is not multiples of 10, it throws error. We will use Chain of Responsibility pattern to implement this solution. The chain will process the request in the same order as below image.



Note that we can implement this solution easily in a single program itself but then the complexity will increase and the solution will be tightly coupled. So we will create a chain of dispense systems to dispense bills of 50$, 20$ and 10$.

A. Base Classes and Interface

We can create a class *Currency* that will store the amount to dispense and used by the chain implementations.

```
package com.journaldev.design.chainofresponsibility;

public class Currency {

    private int amount;

    public Currency(int amt){
        this.amount=amt;
```

```
    }

    public int getAmount(){
        return this.amount;
    }
}
```

The base interface should have a method to define the next processor in the chain and the method that will process the request. Our ATM Dispense interface will look like below.

```
package com.journaldev.design.chainofresponsibility;

public interface DispenseChain {

    void setNextChain(DispenseChain nextChain);

    void dispense(Currency cur);
}
```

B. Concrete Chain Implementations

We need to create different processor classes that will implement the *DispenseChain* interface and provide implementation of dispense methods. Since we are developing our system to work with three types of currency bills – 50$, 20$ and 10$, we will create three concrete implementations.

```
package com.journaldev.design.chainofresponsibility;

public class Dollar50Dispenser implements DispenseChain {

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain=nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 50){
            int num = cur.getAmount()/50;
```

```
            int remainder = cur.getAmount() % 50;
            System.out.println("Dispensing "+num+" 50$ note");
            if(remainder !=0) this.chain.dispense(new
Currency(remainder));
        }else{
            this.chain.dispense(cur);
        }
    }

}
```

```
package com.journaldev.design.chainofresponsibility;

public class Dollar20Dispenser implements DispenseChain{

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain=nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 20){
            int num = cur.getAmount()/20;
            int remainder = cur.getAmount() % 20;
            System.out.println("Dispensing "+num+" 20$ note");
            if(remainder !=0) this.chain.dispense(new
Currency(remainder));
        }else{
            this.chain.dispense(cur);
        }
    }

}
```

```
package com.journaldev.design.chainofresponsibility;

public class Dollar10Dispenser implements DispenseChain {

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
```

```
        this.chain=nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 10){
            int num = cur.getAmount()/10;
            int remainder = cur.getAmount() % 10;
            System.out.println("Dispensing "+num+" 10$ note");
            if(remainder !=0) this.chain.dispense(new
Currency(remainder));
        }else{
            this.chain.dispense(cur);
        }
    }
}
```

The important point to note here is the implementation of dispense method, you will notice that every implementation is trying to process the request and based on the amount, it might process some or full part of it. If it's not able to process it fully, it sends the request to the next processor in chain to process the remaining request. If the processor is not able to process anything, it just forwards the same request to the next chain.

C. Creating the Chain

This is a very important step and we should create the chain carefully, otherwise a processor might not be getting any request at all. For example, in our implementation if we keep the first processor chain as *Dollar10Dispenser* and then *Dollar20Dispenser*, then the request will never be forwarded to the second processor and the chain will become useless.

Here is our ATM Dispenser implementation to process the user requested amount.

```
package com.journaldev.design.chainofresponsibility;

import java.util.Scanner;

public class ATMDispenseChain {

    private DispenseChain c1;
    public ATMDispenseChain() {
```

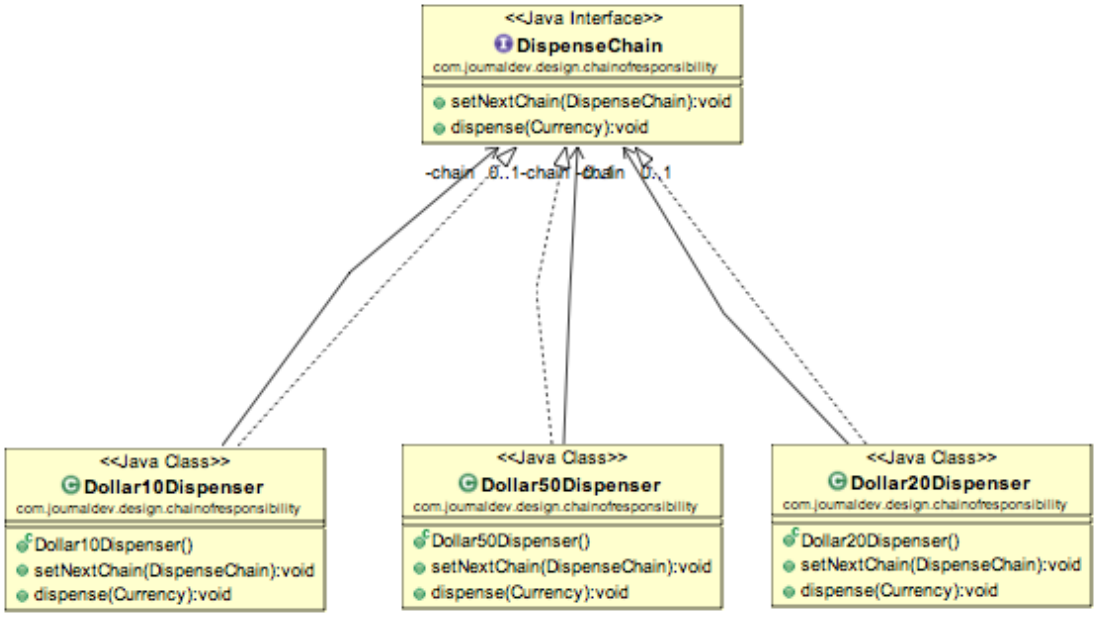
```
        // initialize the chain
        this.c1 = new Dollar50Dispenser();
        DispenseChain c2 = new Dollar20Dispenser();
        DispenseChain c3 = new Dollar10Dispenser();

        // set the chain of responsibility
        c1.setNextChain(c2);
        c2.setNextChain(c3);
    }
 public static void main(String[] args) {
        ATMDispenseChain atmDispenser = new ATMDispenseChain();
        while (true) {
            int amount = 0;
            System.out.println("Enter amount to dispense");
            Scanner input = new Scanner(System.in);
            amount = input.nextInt();
            if (amount % 10 != 0) {
                System.out.println("Amount should be in multiple of
10s.");
                return;
            }
            // process the request
            atmDispenser.c1.dispense(new Currency(amount));
        }
    }
}
```

When we run above application, we get output like below.

```
Enter amount to dispense
530
Dispensing 10 50$ note
Dispensing 1 20$ note
Dispensing 1 10$ note
Enter amount to dispense
100
Dispensing 2 50$ note
Enter amount to dispense
120
Dispensing 2 50$ note
Dispensing 1 20$ note
Enter amount to dispense
15
Amount should be in multiple of 10s.
```

D. Class Diagram

Our ATM dispense example of chain of responsibility implementation looks like below image.



E. Chain of Responsibility Pattern Examples in JDK

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

F. Important Points

- Client doesn't know which part of the chain will be processing the request and it will send the request to the first object in the chain. For example, in our program ATMDispenseChain is unaware of who is processing the request to dispense the entered amount.
- Each object in the chain will have its own implementation to process the request, either full or partial or to send it to the next object in the chain.

- Every object in the chain should have reference to the next object in chain to forward the request to, it's achieved by java composition.
- Creating the chain carefully is very important otherwise there might be a case that the request will never be forwarded to a particular processor or there are no objects in the chain who are able to handle the request. In my implementation, I have added the check for the user entered amount to make sure it gets processed fully by all the processors but we might not check it and throw exception if the request reaches the last object and there are no further objects in the chain to forward the request to. This is a design decision.
- Chain of Responsibility pattern is good to achieve lose coupling but it comes with the trade-off of having a lot of implementation classes and maintenance problems if most of the code is common in all the implementations.

4. Observer Pattern

Observer pattern is one of the behavioral design pattern. Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change. In observer pattern, the object that watch on the state of another object are called Observer and the object that is being watched is called Subject. According to GoF, observer pattern intent is;

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

Subject contains a list of observers to notify of any change in it's state, so it should provide methods using which observers can register and unregister themselves. Subject also contain a method to notify all the observers of any change and either it can send the update while notifying the observer or it can provide another method to get the update.

Observer should have a method to set the object to watch and another method that will be used by Subject to notify them of any updates.

Java provides inbuilt platform for implementing Observer pattern through *java.util.Observable* class and *java.util.Observer* interface. However it's not widely used because the implementation is really simple and most of the times we don't want to end up extending a class just for implementing Observer pattern as java doesn't provide multiple inheritance in classes.

Java Message Service (JMS) uses **Observer pattern** along with **Mediator pattern** to allow applications to subscribe and publish data to other applications.

Model-View-Controller (MVC) frameworks also use Observer pattern where Model is the Subject and Views are observers that can register to get notified of any change to the model.

A. Observer Pattern Example

For our example, we would implement a simple topic and observers can register to this topic. Whenever any new message will be posted to the topic, all the registers observers will be notified and they can consume the message.

Based on the requirements of Subject, here is the base Subject interface that defines the contract methods to be implemented by any concrete subject.

```
package com.journaldev.design.observer;

public interface Subject {

    //methods to register and unregister observers
    public void register(Observer obj);
    public void unregister(Observer obj);

    //method to notify observers of change
    public void notifyObservers();

    //method to get updates from subject
    public Object getUpdate(Observer obj);

}
```

Next we will create contract for Observer, there will be a method to attach the Subject to the observer and another method to be used by Subject to notify of any change.

```
package com.journaldev.design.observer;

public interface Observer {

    //method to update the observer, used by subject
    public void update();

    //attach with subject to observe
    public void setSubject(Subject sub);
}
```

Now our contract is ready, let's proceed with the concrete implementation of our topic.

```
package com.journaldev.design.observer;

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

public class MyTopic implements Subject {

    private List<Observer> observers;
    private String message;
    private boolean changed;
    private final Object MUTEX= new Object();

    public MyTopic(){
        this.observers=new ArrayList<>();
    }
    @Override
    public void register(Observer obj) {
        if(obj == null) throw new NullPointerException("Null
Observer");
        synchronized (MUTEX) {
        if(!observers.contains(obj)) observers.add(obj);
        }
    }

    @Override
    public void unregister(Observer obj) {
        synchronized (MUTEX) {
        observers.remove(obj);
        }
    }

    @Override
    public void notifyObservers() {
        List<Observer> observersLocal = null;
        //synchronization is used to make sure any observer registered
after message is received is not notified
        synchronized (MUTEX) {
            if (!changed)
                return;
            observersLocal = new ArrayList<>(this.observers);
            this.changed=false;
        }
```

```
            for (Observer obj : observersLocal) {
                obj.update();
            }

        }

        @Override
        public Object getUpdate(Observer obj) {
            return this.message;
        }

        //method to post message to the topic
        public void postMessage(String msg){
            System.out.println("Message Posted to Topic:"+msg);
            this.message=msg;
            this.changed=true;
            notifyObservers();
        }

}
```

The method implementation to register and unregister an observer is very simple, the extra method is *postMessage()* that will be used by client application to post String message to the topic. Notice the boolean variable to keep track of the change in the state of topic and used in notifying observers. This variable is required so that if there is no update and somebody calls *notifyObservers()* method, it doesn't send false notifications to the observers.

Also notice the use of synchronization in *notifyObservers()* method to make sure the notification is sent only to the observers registered before the message is published to the topic.

Here is the implementation of Observers that will watch over the subject.

```
package com.journaldev.design.observer;

public class MyTopicSubscriber implements Observer {

    private String name;
    private Subject topic;

    public MyTopicSubscriber(String nm){
        this.name=nm;
```

```
    }
    @Override
    public void update() {
        String msg = (String) topic.getUpdate(this);
        if(msg == null){
            System.out.println(name+":: No new message");
        }else
        System.out.println(name+":: Consuming message::"+msg);
    }

    @Override
    public void setSubject(Subject sub) {
        this.topic=sub;
    }

}
```

Notice the implementation of *update()* method where it's calling Subject *getUpdate()* method to get the message to consume. We could have avoided this call by passing message as argument to *update()* method.

Here is a simple test program to consume our topic implementation.

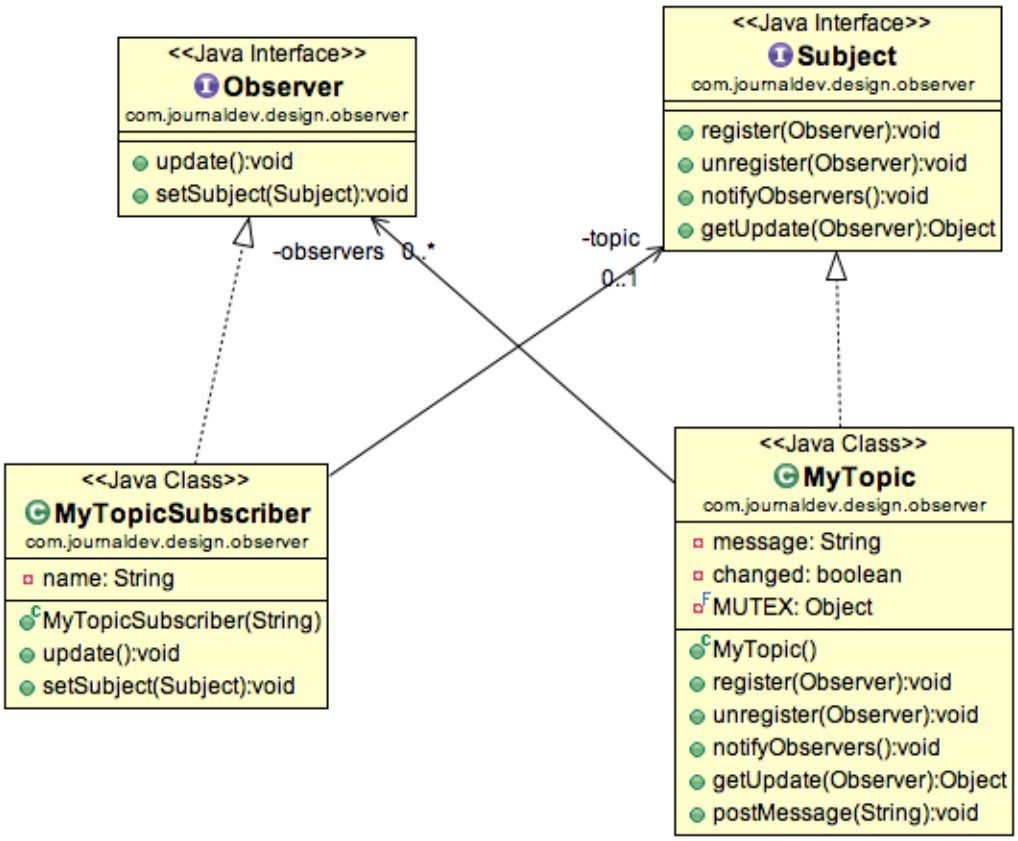
```
package com.journaldev.design.observer;

public class ObserverPatternTest {

    public static void main(String[] args) {
        //create subject
        MyTopic topic = new MyTopic();

        //create observers
        Observer obj1 = new MyTopicSubscriber("Obj1");
        Observer obj2 = new MyTopicSubscriber("Obj2");
        Observer obj3 = new MyTopicSubscriber("Obj3");

        //register observers to the subject
        topic.register(obj1);
        topic.register(obj2);
        topic.register(obj3);

        //attach observer to subject
        obj1.setSubject(topic);
        obj2.setSubject(topic);
        obj3.setSubject(topic);
```

```
        //check if any update is available
        obj1.update();

        //now send message to subject
        topic.postMessage("New Message");
    }

}
```

When we run above program, we get following output.

```
Obj1:: No new message
Message Posted to Topic:New Message
Obj1:: Consuming message::New Message
Obj2:: Consuming message::New Message
Obj3:: Consuming message::New Message
```

B. Observer Pattern Class Diagram



Observer pattern is also called as publish-subscribe pattern. Some of it's implementations are;

- java.util.EventListener in Swing
- javax.servlet.http.HttpSessionBindingListener
- javax.servlet.http.HttpSessionAttributeListener

That's all for Observer pattern in java, I hope you liked it. Share your love with comments and by sharing it with others.

5. Strategy Pattern

Strategy pattern is one of the **behavioral design pattern**. Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.

Strategy pattern is also known as **Policy Pattern**. We defines multiple algorithms and let client application pass the algorithm to be used as a parameter. One of the best example of this pattern is *Collections.sort()* method that takes Comparator parameter. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways, check this post for sorting objects in java using Java Comparable and Comparator.

For our example, we will try to implement a simple Shopping Cart where we have two payment strategies – using Credit Card or using PayPal.

First of all we will create the interface for our strategy, in our case to pay the amount passed as argument.

```
package com.journaldev.design.strategy;

public interface PaymentStrategy {

    public void pay(int amount);
}
```

Now we will have to create concrete implementations of algorithms for payment using credit/debit card or through paypal.

```
package com.journaldev.design.strategy;

public class CreditCardStrategy implements PaymentStrategy {

    private String name;
    private String cardNumber;
    private String cvv;
    private String dateOfExpiry;

    public CreditCardStrategy(String nm, String ccNum, String cvv,
String expiryDate){
        this.name=nm;
```

```
        this.cardNumber=ccNum;
        this.cvv=cvv;
        this.dateOfExpiry=expiryDate;
    }
    @Override
    public void pay(int amount) {
        System.out.println(amount +" paid with credit/debit card");
    }

}
package com.journaldev.design.strategy;

public class PaypalStrategy implements PaymentStrategy {

    private String emailId;
    private String password;

    public PaypalStrategy(String email, String pwd){
        this.emailId=email;
        this.password=pwd;
    }

    @Override
    public void pay(int amount) {
        System.out.println(amount + " paid using Paypal.");
    }

}
```

Now our algorithms are ready and we can implement Shopping Cart and payment method will require input as Payment strategy.

```
package com.journaldev.design.strategy;

public class Item {

    private String upcCode;
    private int price;

    public Item(String upc, int cost){
        this.upcCode=upc;
        this.price=cost;
    }

    public String getUpcCode() {
        return upcCode;
```

```
    }

    public int getPrice() {
        return price;
    }

}
package com.journaldev.design.strategy;

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.List;

public class ShoppingCart {

    //List of items
    List<Item> items;

    public ShoppingCart(){
        this.items=new ArrayList<Item>();
    }

    public void addItem(Item item){
        this.items.add(item);
    }

    public void removeItem(Item item){
        this.items.remove(item);
    }

    public int calculateTotal(){
        int sum = 0;
        for(Item item : items){
            sum += item.getPrice();
        }
        return sum;
    }

    public void pay(PaymentStrategy paymentMethod){
        int amount = calculateTotal();
        paymentMethod.pay(amount);
    }
}
```

Notice that payment method of shopping cart requires payment algorithm as argument and doesn't store it anywhere as instance variable.

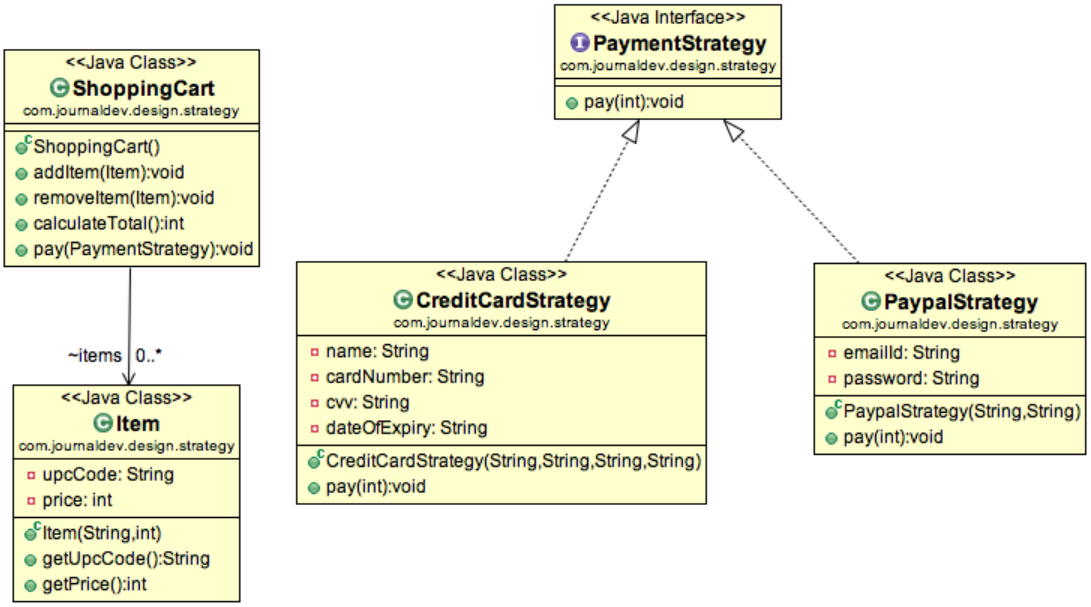
Let's test our setup with a simple program.

```
package com.journaldev.design.strategy;

public class ShoppingCartTest {

    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        Item item1 = new Item("1234",10);
        Item item2 = new Item("5678",40);

        cart.addItem(item1);
        cart.addItem(item2);

        //pay by paypal
        cart.pay(new PaypalStrategy("myemail@example.com", "mypwd"));

        //pay by credit card
        cart.pay(new CreditCardStrategy("Pankaj Kumar",
"1234567890123456", "786", "12/15"));
    }

}
```

Output of above program is:

```
50 paid using Paypal.
50 paid with credit/debit card
```

A. Strategy Pattern Class Diagram



B. Important Points

- We could have used composition to create instance variable for strategies but we should avoid that as we want the specific strategy to be applied for a particular task, same is followed in Collections.sort() and Arrays.sort() method that take comparator as argument.
- Strategy Pattern is very similar to **State Pattern**. One of the difference is that Context contains state as instance variable and there can be multiple tasks whose implementation can be dependent on the state whereas in strategy pattern strategy is passed as argument to the method and context object doesn't have any variable to store it.
- Strategy pattern is useful when we have multiple algorithms for specific task and we want our application to be flexible to choose any of the algorithm at runtime for specific task.

6. Command Pattern

Command Pattern is one of the **Behavioral Design Pattern** and it's used to implement **lose coupling** in a request-response model. In command pattern, the request is send to the *invoker* and invoker pass it to the encapsulated *command* object. Command object passes the request to the appropriate method of *Receiver* to perform the specific action. The client program create the receiver object and then attach it to the Command. Then it creates the invoker object and attach the command object to perform an action. Now when client program executes the action, it's processed based on the command and receiver object.

We will look at a real life scenario where we can implement Command pattern. Let's say we want to provide a File System utility with methods to open, write and close file and it should support multiple operating systems such as Windows and Unix.

To implement our File System utility, first of all we need to create the receiver classes that will actually do all the work. Since we code in terms of java interfaces, we can have FileSystemReceiver interface and it's implementation classes for different operating system flavors such as Windows, Unix, Solaris etc.

A. Receiver Classes

```
package com.journaldev.design.command;

public interface FileSystemReceiver {

    void openFile();
    void writeFile();
    void closeFile();
}
```

FileSystemReceiver interface defines the contract for the implementation classes. For simplicity, I am creating two flavors of receiver classes to work with Unix and Windows systems.

```
package com.journaldev.design.command;

public class UnixFileSystemReceiver implements FileSystemReceiver {

    @Override
    public void openFile() {
        System.out.println("Opening file in unix OS");
    }

    @Override
    public void writeFile() {
        System.out.println("Writing file in unix OS");
    }

    @Override
    public void closeFile() {
        System.out.println("Closing file in unix OS");
    }
}

package com.journaldev.design.command;

public class WindowsFileSystemReceiver implements FileSystemReceiver {

    @Override
    public void openFile() {
        System.out.println("Opening file in Windows OS");

    }

    @Override
    public void writeFile() {
        System.out.println("Writing file in Windows OS");
    }
    @Override
    public void closeFile() {
        System.out.println("Closing file in Windows OS");
    }
}
```

Did you noticed the Override annotation and if you wonder why it's used, please read java annotations and override annotation benefits.

Now that our receiver classes are ready, we can move to implement our Command classes.

B. Command Interface and Implementations

We can use interface or abstract class to create our base Command, it's a design decision and depends on your requirement. We are going with interface because we don't have any default implementations.

```
package com.journaldev.design.command;

public interface Command {

    void execute();
}
```

Now we need to create implementations for all the different types of action performed by the receiver, since we have three actions we will create three Command implementations and each Command implementation will forward the request to the appropriate method of receiver.

```
package com.journaldev.design.command;

public class OpenFileCommand implements Command {

    private FileSystemReceiver fileSystem;

    public OpenFileCommand(FileSystemReceiver fs){
        this.fileSystem=fs;
    }
    @Override
    public void execute() {
        //open command is forwarding request to openFile method
        this.fileSystem.openFile();
    }

}
```

```
package com.journaldev.design.command;

public class CloseFileCommand implements Command {

    private FileSystemReceiver fileSystem;

    public CloseFileCommand(FileSystemReceiver fs){
        this.fileSystem=fs;
```

```
    }
    @Override
    public void execute() {
        this.fileSystem.closeFile();
    }

package com.journaldev.design.command;

public class WriteFileCommand implements Command {

    private FileSystemReceiver fileSystem;

    public WriteFileCommand(FileSystemReceiver fs){
        this.fileSystem=fs;
    }
    @Override
    public void execute() {
        this.fileSystem.writeFile();
    }

}
```

Now we have receiver and command implementations ready, so we can move to implement the invoker class.

C. Invoker Class

Invoker is a simple class that encapsulates the Command and passes the request to the command object to process it.

```
package com.journaldev.design.command;

public class FileInvoker {

    public Command command;

    public FileInvoker(Command c){
        this.command=c;
    }

    public void execute(){
        this.command.execute();
    }
```

```
}
```

Our file system utility implementation is ready and we can move to write a simple client program but before that I will provide a utility method to create the appropriate *FileSystemReceiver* object. Since we can use System class to get the operating system information, we will use this or else we can use Factory pattern to return appropriate type based on the input from client program.

```
package com.journaldev.design.command;

public class FileSystemReceiverUtil {

    public static FileSystemReceiver getUnderlyingFileSystem(){
         String osName = System.getProperty("os.name");
         System.out.println("Underlying OS is:"+osName);
         if(osName.contains("Windows")){
             return new WindowsFileSystemReceiver();
         }else{
             return new UnixFileSystemReceiver();
         }
    }

}
```

Let's move now to create our client program that will consume our file system utility.

```
package com.journaldev.design.command;

public class FileSystemClient {

    public static void main(String[] args) {
        //Creating the receiver object
        FileSystemReceiver fs =
FileSystemReceiverUtil.getUnderlyingFileSystem();

        //creating command and associating with receiver
        OpenFileCommand openFileCommand = new OpenFileCommand(fs);

        //Creating invoker and associating with Command
        FileInvoker file = new FileInvoker(openFileCommand);

        //perform action on invoker object
```

```
        file.execute();

        WriteFileCommand writeFileCommand = new WriteFileCommand(fs);
        file = new FileInvoker(writeFileCommand);
        file.execute();

        CloseFileCommand closeFileCommand = new CloseFileCommand(fs);
        file = new FileInvoker(closeFileCommand);
        file.execute();
    }

}
```

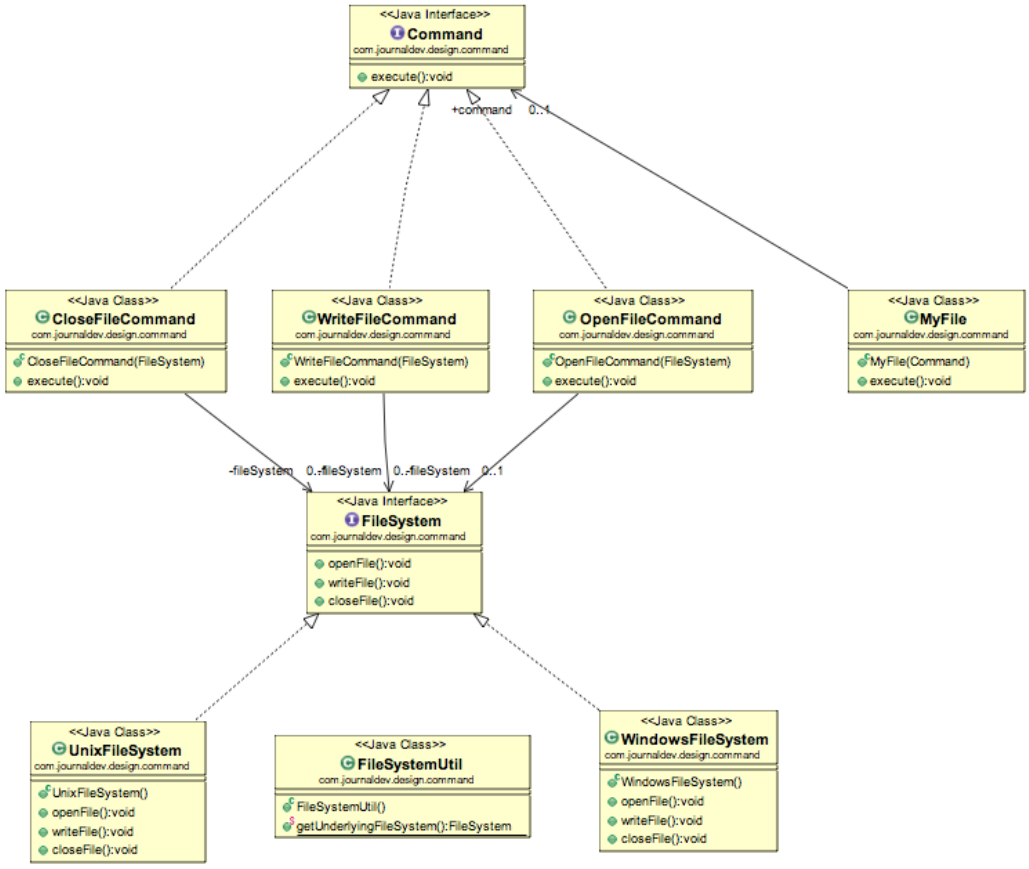
Notice that client is responsible to create the appropriate type of command object, for example if you want to write a file you are not supposed to create CloseFileCommand object. Client program is also responsible to attach receiver to the command and then command to the invoker class.

Output of the above program is:

```
Underlying OS is:Mac OS X
Opening file in unix OS
Writing file in unix OS
Closing file in unix OS
```

C. Class Diagram

Here is the class diagram for our file system utility implementation.



D. Command Pattern JDK Example

Runnable interface (java.lang.Runnable) and Swing Action (javax.swing.Action) uses command pattern.

E. Important Points

- Command is the core of this pattern that defines the contract for implementation.
- Receiver implementation is separate from command implementation.
- Command implementation classes chose the method to invoke on receiver object, for every method in receiver there will be a command implementation. It works as a bridge between receiver and action methods.

- Invoker class just forward the request from client to the command object.
- Client is responsible to instantiate appropriate command and receiver implementation and then associate them together.
- Client is also responsible for instantiating invoker object and associating command object with it and execute the action method.
- Command pattern is easily extendible, we can add new action methods in receivers and create new Command implementations without changing the client code.
- The drawback with Command pattern is that the code gets huge and confusing with high number of action methods and because of so many associations.

7. State Pattern

State pattern is one of the **behavioral design pattern**. State design pattern is used when an Object change its behavior based on its internal state.

If we have to change the behavior of an object based on its state, we can have a state variable in the Object and use **if-else** condition block to perform different actions based on the state. State pattern is used to provide a systematic and lose-coupled way to achieve this through **Context** and **State** implementations.

Context is the class that has a State reference to one of the concrete implementations of the State and forwards the request to the state object for processing. Let's understand this with a simple example.

Suppose we want to implement a TV Remote with a simple button to perform action, if the State is ON, it will turn on the TV and if state is OFF, it will turn off the TV.

We can implement it using if-else condition like below;

```
package com.journaldev.design.state;

public class TVRemoteBasic {

    private String state="";

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

    public void doAction(){
        if(state.equalsIgnoreCase("ON")){
            System.out.println("TV is turned ON");
        }else if(state.equalsIgnoreCase("OFF")){
            System.out.println("TV is turned OFF");
        }
    }

    public static void main(String args[]){
        TVRemoteBasic remote = new TVRemoteBasic();
```

```
        remote.setState("ON");
        remote.doAction();

        remote.setState("OFF");
        remote.doAction();
    }

}
```

Notice that client code should know the specific values to use for setting the state of remote, further more if number of states increase then the tight coupling between implementation and the client code will be very hard to maintain and extend.

Now we will use State pattern to implement above TV Remote example.

A. State Interface

First of all we will create State interface that will define the method that should be implemented by different concrete states and context class.

```
package com.journaldev.design.state;

public interface State {

    public void doAction();
}
```

B. Concrete State Implementations

In our example, we can have two states – one for turning TV on and another to turn it off. So we will create two concrete state implementations for these behaviors.

```
package com.journaldev.design.state;

public class TVStartState implements State {

    @Override
    public void doAction() {
        System.out.println("TV is turned ON");
```

```
    }

}
```

```
package com.journaldev.design.state;

public class TVStopState implements State {

    @Override
    public void doAction() {
        System.out.println("TV is turned OFF");
    }

}
```

Now we are ready to implement our Context object that will change it's behavior based on its internal state.

C. Context Implementation

```
package com.journaldev.design.state;

public class TVContext implements State {

    private State tvState;

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

    public State getState() {
        return this.tvState;
    }

    @Override
    public void doAction() {
        this.tvState.doAction();
    }

}
```

Notice that Context also implements State and keep a reference of its current state and forwards the request to the state implementation.

D. Test Program

Now let's write a simple program to test our implementation of TV Remote using State pattern.

```
package com.journaldev.design.state;

public class TVRemote {

    public static void main(String[] args) {
        TVContext context = new TVContext();
        State tvStartState = new TVStartState();
        State tvStopState = new TVStopState();

        context.setState(tvStartState);
        context.doAction();


        context.setState(tvStopState);
        context.doAction();

    }

}
```

Output of above program is same as the basic implementation of TV Remote without using any pattern.

The benefits of using State pattern to implement polymorphic behavior is clearly visible, the chances of error are less and it's very easy to add more states for additional behavior making it more robust, easily maintainable and flexible. Also State pattern helped in avoiding if-else or switch-case conditional logic in this scenario.

8. Visitor Pattern

Visitor Pattern is one of the **behavioral design pattern**. Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class.

For example, think of a Shopping cart where we can add different type of items (Elements), when we click on checkout button, it calculates the total amount to be paid. Now we can have the calculation logic in item classes or we can move out this logic to another class using visitor pattern. Let's implement this in our example of visitor pattern.

To implement visitor pattern, first of all we will create different type of items (Elements) to be used in shopping cart.

```
package com.journaldev.design.visitor;

public interface ItemElement {

    public int accept(ShoppingCartVisitor visitor);
}
```

Notice that accept method takes Visitor argument, we can have some other methods also specific for items but for simplicity I am not going into that much detail and focusing on visitor pattern only.

Let's create some concrete classes for different types of items.

```
package com.journaldev.design.visitor;

public class Book implements ItemElement {

    private int price;
    private String isbnNumber;

    public Book(int cost, String isbn){
        this.price=cost;
        this.isbnNumber=isbn;
    }
```

```
    public int getPrice() {
        return price;
    }

    public String getIsbnNumber() {
        return isbnNumber;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }

}

package com.journaldev.design.visitor;

public class Fruit implements ItemElement {

    private int pricePerKg;
    private int weight;
    private String name;

    public Fruit(int priceKg, int wt, String nm){
        this.pricePerKg=priceKg;
        this.weight=wt;
        this.name = nm;
    }

    public int getPricePerKg() {
        return pricePerKg;
    }

    public int getWeight() {
        return weight;
    }

    public String getName(){
        return this.name;
    }
    @Override
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }
}
```

Notice the implementation of accept () method in concrete classes, its calling visit () method of Visitor and passing itself as argument.

We have visit () method for different type of items in Visitor interface that will be implemented by concrete visitor class.

```
package com.journaldev.design.visitor;

public interface ShoppingCartVisitor {

    int visit(Book book);
    int visit(Fruit fruit);
}
```

Now we will implement visitor interface and every item will have its own logic to calculate the cost.

```
package com.journaldev.design.visitor;

public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {

    @Override
    public int visit(Book book) {
        int cost=0;
        //apply 5$ discount if book price is greater than 50
        if(book.getPrice() > 50){
            cost = book.getPrice()-5;
        }else cost = book.getPrice();
        System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost
="+cost);
        return cost;
    }

    @Override
    public int visit(Fruit fruit) {
        int cost = fruit.getPricePerKg()*fruit.getWeight();
        System.out.println(fruit.getName() + " cost = "+cost);
        return cost;
    }

}
```

Let's see how we can use it in client applications.

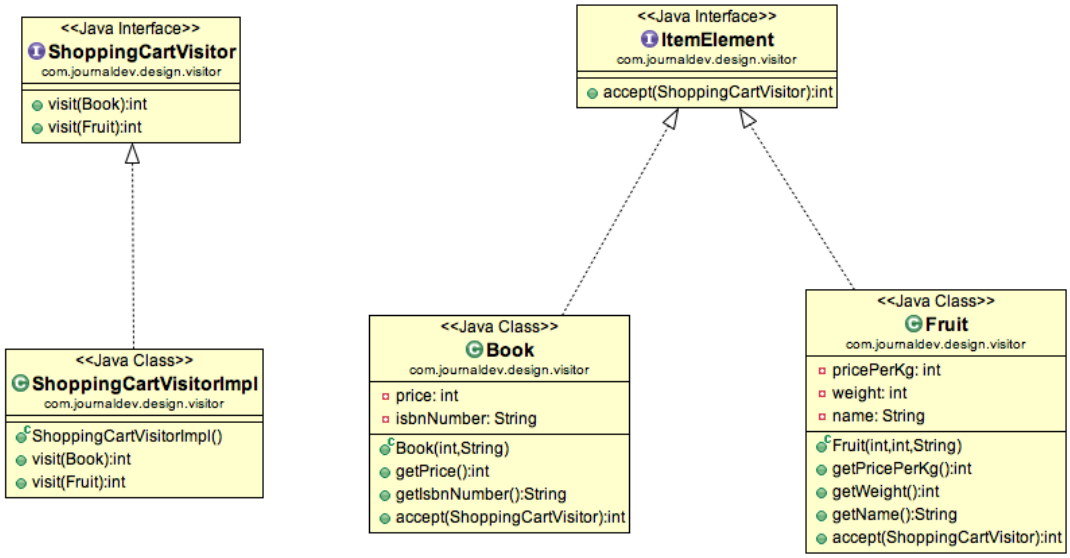
```
package com.journaldev.design.visitor;

public class ShoppingCartClient {

    public static void main(String[] args) {
        ItemElement[] items = new ItemElement[]{new Book(20,
"1234"),new Book(100, "5678"),
                new Fruit(10, 2, "Banana"), new Fruit(5, 5, "Apple")};

        int total = calculatePrice(items);
        System.out.println("Total Cost = "+total);
    }

    private static int calculatePrice(ItemElement[] items) {
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
        int sum=0;
        for(ItemElement item : items){
            sum = sum + item.accept(visitor);
        }
        return sum;
    }

}
```

When we run above program, we get following output.

```
Book ISBN::1234 cost =20
Book ISBN::5678 cost =95
Banana cost = 20
Apple cost = 25
Total Cost = 160
```

A. Visitor Pattern Class Diagram

Class diagram for our visitor pattern implementation is:



The benefit of this pattern is that if the logic of operation changes, then we need to make change only in the visitor implementation rather than doing it in all the item classes.

Another benefit is that adding a new item to the system is easy, it will require change only in visitor interface and implementation and existing item classes will not be affected.

The drawback of visitor pattern is that we should know the return type of visit () methods at the time of designing otherwise we will have to change the interface and all of its implementations. Another drawback is that if there are too many implementations of visitor interface, it makes it hard to extend.

9. Interpreter Pattern

Interpreter pattern is one of the **behavioral design pattern** and is used to define a grammatical representation for a language and provides an interpreter to deal with this grammar. The best example of this pattern is java compiler that interprets the java source code into byte code that is understandable by JVM. Google Translator is also an example of interpreter pattern where the input can be in any language and we can get the output interpreted in another language.

To implement interpreter pattern, we need to create Interpreter context engine that will do the interpretation work and then we need to create different Expression implementations that will consume the functionalities provided by the interpreter context. Finally we need to create the client that will take the input from user and decide which Expression to use and then generate output for the user.

Let's understand this with an example where the user input will be of two forms – "<Number> in Binary" or "<Number> in Hexadecimal" and our interpreter client should return it in format "<Number> in Binary= <Number_Binary_String>" and "<Number> in Hexadecimal= <Number_Binary_String>" respectively.

Our first step will be to write the Interpreter context class that will do the actual interpretation.

```
package com.journaldev.design.interpreter;

public class InterpreterContext {

    public String getBinaryFormat(int i){
        return Integer.toBinaryString(i);
    }

    public String getHexadecimalFormat(int i){
        return Integer.toHexString(i);
    }
}
```

Now we need to create different types of Expressions that will consume the interpreter context class.

```
package com.journaldev.design.interpreter;

public interface Expression {

    String interpret(InterpreterContext ic);
}
```

We will have two expression implementations, one to convert int to binary and other to convert int to hexadecimal format.

```
package com.journaldev.design.interpreter;

public class IntToBinaryExpression implements Expression {

    private int i;

    public IntToBinaryExpression(int c){
        this.i=c;
    }
    @Override
    public String interpret(InterpreterContext ic) {
        return ic.getBinaryFormat(this.i);
    }

}
```

```
package com.journaldev.design.interpreter;

public class IntToHexExpression implements Expression {

    private int i;

    public IntToHexExpression(int c){
        this.i=c;
    }

    @Override
    public String interpret(InterpreterContext ic) {
        return ic.getHexadecimalFormat(i);
    }

}
```

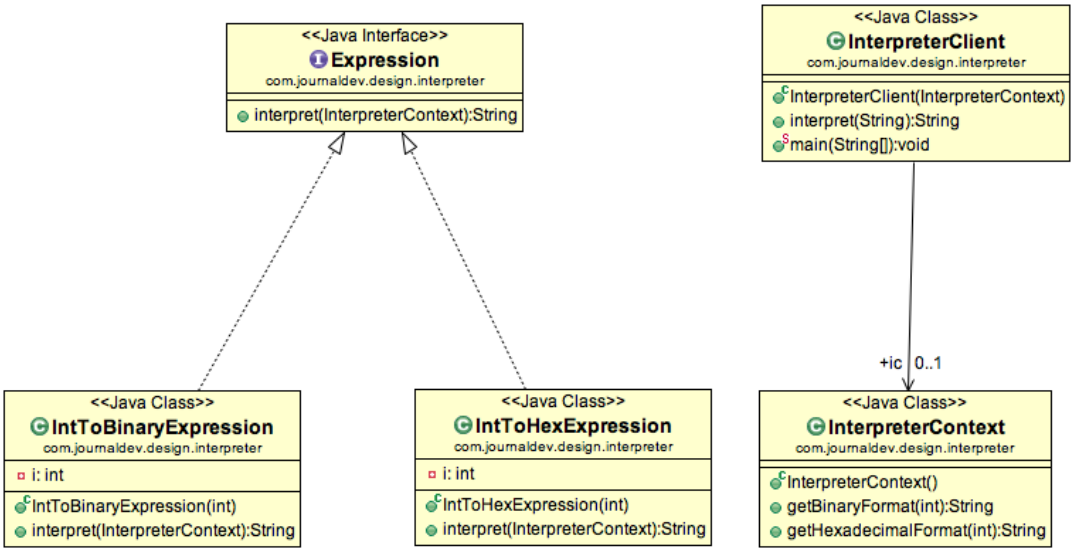
Now we can create our client application that will have the logic to parse the user input and pass it to correct expression and then use the output to generate the user response.

```
package com.journaldev.design.interpreter;

public class InterpreterClient {

    public InterpreterContext ic;

    public InterpreterClient(InterpreterContext i){
        this.ic=i;
    }

    public String interpret(String str){
        Expression exp=null;
        //create rules for expressions
        if(str.contains("Hexadecimal")){
            exp=new
IntToHexExpression(Integer.parseInt(str.substring(0,str.indexOf("
")))));
        }else if(str.contains("Binary")){
            exp=new
IntToBinaryExpression(Integer.parseInt(str.substring(0,str.indexOf("
")))));
        }else return str;

        return exp.interpret(ic);
    }

    public static void main(String args[]){
        String str1 = "28 in Binary";
        String str2 = "28 in Hexadecimal";

        InterpreterClient ec = new InterpreterClient(new
InterpreterContext());
        System.out.println(str1+"= "+ec.interpret(str1));
        System.out.println(str2+"= "+ec.interpret(str2));

    }
}
```

The client also has a main method for testing purpose, when we run above we get following output:

```
28 in Binary= 11100
28 in Hexadecimal= 1c
```

A. Class Diagram



B. Important Points

- Interpreter pattern can be used when we can create a syntax tree for the grammar we have.
- Interpreter pattern requires a lot of error checking and a lot of expressions and code to evaluate them, it gets complicated when the grammar becomes more complicated and hence hard to maintain and provide efficiency.
- *java.util.Pattern* and subclasses of *java.text.Format* are some of the examples of interpreter pattern used in JDK.

10. Iterator Pattern

Iterator pattern in one of the behavioral pattern and it's used to provide a standard way to traverse through a group of Objects. Iterator pattern is widely used in Java Collection Framework where Iterator interface provides methods for traversing through a collection. According to GoF, iterator design pattern intent is:

> "Provides a way to access the elements of an aggregate object without exposing its underlying representation"

Iterator pattern is not only about traversing through a collection, we can provide different kind of iterators based on our requirements. Iterator pattern hides the actual implementation of traversal through the collection and client programs just use iterator methods.

Let's understand this pattern with a simple example. Suppose we have a list of Radio channels and the client program want to traverse through them one by one or based on the type of channel, for example some client programs are only interested in English channels and want to process only them, they don't want to process other types of channels.

So we can provide a collection of channels to the client and let them write the logic to traverse through the channels and decide whether to process them. But this solution has lots of issues such as client has to come up with the logic for traversal. We can't make sure that client logic is correct and if the number of client grows then it will become very hard to maintain.

Here we can use Iterator pattern and provide iteration based on type of channel. We should make sure that client program can access the list of channels only through the iterator.

The first part of implementation is to define the contract for our collection and iterator interfaces.

```
package com.journaldev.design.iterator;
```

```
public enum ChannelTypeEnum {

    ENGLISH, HINDI, FRENCH, ALL;
}
```

ChannelTypeEnum is **java enum** that defines all the different types of channels.

```
package com.journaldev.design.iterator;

public class Channel {

    private double frequency;
    private ChannelTypeEnum TYPE;

    public Channel(double freq, ChannelTypeEnum type){
        this.frequency=freq;
        this.TYPE=type;
    }

    public double getFrequency() {
        return frequency;
    }

    public ChannelTypeEnum getTYPE() {
        return TYPE;
    }

    @Override
    public String toString(){
        return "Frequency="+this.frequency+", Type="+this.TYPE;
    }

}
```

Channel is a simple POJO class that has attributes frequency and channel type.

```
package com.journaldev.design.iterator;


public interface ChannelCollection {

    public void addChannel(Channel c);
```

```
    public void removeChannel(Channel c);

    public ChannelIterator iterator(ChannelTypeEnum type);

}
```

ChannelCollection interface defines the contract for our collection class implementation. Notice that there are methods to add and remove a channel but there is no method that returns the list of channels and it has a method that returns the iterator for traversal. ChannelIterator interface defines following methods;

```
package com.journaldev.design.iterator;

public interface ChannelIterator {

    public boolean hasNext();

    public Channel next();
}
```

Now our base interface and core classes are ready, let's proceed with the implementation of collection class and iterator.

```
package com.journaldev.design.iterator;

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

public class ChannelCollectionImpl implements ChannelCollection {

    private List<Channel> channelsList;

    public ChannelCollectionImpl() {
        channelsList = new ArrayList<>();
    }

    public void addChannel(Channel c) {
        this.channelsList.add(c);
    }

    public void removeChannel(Channel c) {
        this.channelsList.remove(c);
    }
```

```
    @Override
    public ChannelIterator iterator(ChannelTypeEnum type) {
        return new ChannelIteratorImpl(type, this.channelsList);
    }

    private class ChannelIteratorImpl implements ChannelIterator {

        private ChannelTypeEnum type;
        private List<Channel> channels;
        private int position;

        public ChannelIteratorImpl(ChannelTypeEnum ty,
                List<Channel> channelsList) {
            this.type = ty;
            this.channels = channelsList;
        }

        @Override
        public boolean hasNext() {
            while (position < channels.size()) {
                Channel c = channels.get(position);
                if (c.getTYPE().equals(type) ||
type.equals(ChannelTypeEnum.ALL)) {
                    return true;
                } else
                    position++;
            }
            return false;
        }

        @Override
        public Channel next() {
            Channel c = channels.get(position);
            position++;
            return c;
        }

    }
}
```

Notice the **inner class** implementation of iterator interface so that the implementation can't be used by any other collection. Same approach is followed by collection classes also and all of them have inner class implementation of Iterator interface.

Let's write a simple test class to use our collection and iterator to traverse through the collection of channels based on type.

```
package com.journaldev.design.iterator;

public class IteratorPatternTest {

    public static void main(String[] args) {
        ChannelCollection channels = populateChannels();
        ChannelIterator baseIterator =
channels.iterator(ChannelTypeEnum.ALL);
        while (baseIterator.hasNext()) {
            Channel c = baseIterator.next();
            System.out.println(c.toString());
        }
        System.out.println("******");
        // Channel Type Iterator
        ChannelIterator englishIterator =
channels.iterator(ChannelTypeEnum.ENGLISH);
        while (englishIterator.hasNext()) {
            Channel c = englishIterator.next();
            System.out.println(c.toString());
        }
    }
 private static ChannelCollection populateChannels() {
        ChannelCollection channels = new ChannelCollectionImpl();
        channels.addChannel(new Channel(98.5,
ChannelTypeEnum.ENGLISH));
        channels.addChannel(new Channel(99.5, ChannelTypeEnum.HINDI));
        channels.addChannel(new Channel(100.5,
ChannelTypeEnum.FRENCH));
        channels.addChannel(new Channel(101.5,
ChannelTypeEnum.ENGLISH));
        channels.addChannel(new Channel(102.5, ChannelTypeEnum.HINDI));
        channels.addChannel(new Channel(103.5,
ChannelTypeEnum.FRENCH));
        channels.addChannel(new Channel(104.5,
ChannelTypeEnum.ENGLISH));
        channels.addChannel(new Channel(105.5, ChannelTypeEnum.HINDI));
        channels.addChannel(new Channel(106.5,
ChannelTypeEnum.FRENCH));
        return channels;
    }

}
```

When I run above program, it produces following output;

```
Frequency=98.5, Type=ENGLISH
Frequency=99.5, Type=HINDI
Frequency=100.5, Type=FRENCH
Frequency=101.5, Type=ENGLISH
Frequency=102.5, Type=HINDI
Frequency=103.5, Type=FRENCH
Frequency=104.5, Type=ENGLISH
Frequency=105.5, Type=HINDI
Frequency=106.5, Type=FRENCH
******
Frequency=98.5, Type=ENGLISH
Frequency=101.5, Type=ENGLISH
Frequency=104.5, Type=ENGLISH
```

A. Iterator Pattern in JDK

We all know that Collection framework Iterator is the best example of iterator pattern implementation but do you know that java.util.Scanner class also Implements Iterator interface. Read this post to learn about Java Scanner Class.

B. Important Points

- Iterator pattern is useful when you want to provide a standard way to iterate over a collection and hide the implementation logic from client program.
- The logic for iteration is embedded in the collection itself and it helps client program to iterate over them easily.

11. Memento Pattern

Memento pattern is one of the **behavioral design pattern**. Memento design pattern is used when we want to save the state of an object so that we can restore later on. Memento pattern is used to implement this in such a way that the saved state data of the object is not accessible outside of the object, this protects the integrity of saved state data.

Memento pattern is implemented with two objects – **Originator** and **Caretaker**. Originator is the object whose state needs to be saved and restored and it uses an inner class to save the state of Object. The inner class is called **Memento** and its private, so that it can't be accessed from other objects.

Caretaker is the helper class that is responsible for storing and restoring the Originator's state through Memento object. Since Memento is private to Originator, Caretaker can't access it and it's stored as a Object within the caretaker.

One of the best real life example is the text editors where we can save it's data anytime and use undo to restore it to previous saved state. We will implement the same feature and provide a utility where we can write and save contents to a File anytime and we can restore it to last saved state. For simplicity, I will not use any IO operations to write data into file.

A. Originator Class

```
package com.journaldev.design.memento;

public class FileWriterUtil {

    private String fileName;
    private StringBuilder content;

    public FileWriterUtil(String file){
        this.fileName=file;
        this.content=new StringBuilder();
    }
```

```
    @Override
    public String toString(){
        return this.content.toString();
    }

    public void write(String str){
        content.append(str);
    }

    public Memento save(){
        return new Memento(this.fileName,this.content);
    }

    public void undoToLastSave(Object obj){
        Memento memento = (Memento) obj;
        this.fileName= memento.fileName;
        this.content=memento.content;
    }


    private class Memento{
        private String fileName;
        private StringBuilder content;

        public Memento(String file, StringBuilder content){
            this.fileName=file;
            //notice the deep copy so that Memento and FileWriterUtil
content variables don't refer to same object
            this.content=new StringBuilder(content);
        }
    }
}
```

Notice the Memento inner class and implementation of save and undo methods. Now we can continue to implement Caretaker class.

B. Caretaker Class

```
package com.journaldev.design.memento;

public class FileWriterCaretaker {

    private Object obj;
```

```
    public void save(FileWriterUtil fileWriter){
        this.obj=fileWriter.save();
    }

    public void undo(FileWriterUtil fileWriter){
        fileWriter.undoToLastSave(obj);
    }
}
```

Notice that caretaker object contains the saved state in the form of Object, so it can't alter its data and also it has no knowledge of its structure.

C. Memento Test Class

Let's write a simple test program that will use our memento implementation.

```
package com.journaldev.design.memento;
 public class FileWriterClient {

    public static void main(String[] args) {
         FileWriterCaretaker caretaker = new FileWriterCaretaker();

        FileWriterUtil fileWriter = new FileWriterUtil("data.txt");
        fileWriter.write("First Set of Data\n");
        System.out.println(fileWriter+"\n\n");

        // lets save the file
        caretaker.save(fileWriter);
        //now write something else
        fileWriter.write("Second Set of Data\n");

        //checking file contents
        System.out.println(fileWriter+"\n\n");

        //lets undo to last save
        caretaker.undo(fileWriter);

        //checking file content again
        System.out.println(fileWriter+"\n\n");

    }
}
```

Output of above program is:

```
First Set of Data



First Set of Data
Second Set of Data



First Set of Data
```

The pattern is simple and easy to implement, one of the thing needs to take care is that Memento class should be accessible only to the Originator object. Also in client application, we should use caretaker object for saving and restoring the originator state.

Also if Originator object has properties that are not immutable, we should use deep copy or cloning to avoid data integrity issue like I have used in above example. We can use Serialization to achieve memento pattern implementation that is more generic rather than Memento pattern where every object needs to have its own Memento class implementation.

One of the drawback is that if Originator object is very huge then Memento object size will also be huge and use a lot of memory.

Copyright Notice

References

1. http://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-with-examples
2. http://www.journaldev.com/1392/factory-design-pattern-in-java
3. http://www.journaldev.com/1418/abstract-factory-design-pattern-in-java
4. http://www.journaldev.com/1425/builder-design-pattern-in-java
5. http://www.journaldev.com/1440/prototype-pattern-in-java
6. http://www.journaldev.com/1487/adapter-design-pattern-in-java-example-tutorial
7. http://www.journaldev.com/1535/composite-design-pattern-in-java-example-tutorial
8. http://www.journaldev.com/1572/proxy-design-pattern-in-java-example-tutorial
9. http://www.journaldev.com/1562/flyweight-pattern-in-java-example-tutorial
10. http://www.journaldev.com/1557/facade-pattern-in-java-example-tutorial
11. http://www.journaldev.com/1491/bridge-pattern-in-java-example-tutorial
12. http://www.journaldev.com/1540/decorator-pattern-in-java-example-tutorial
13. http://www.journaldev.com/1763/template-method-design-pattern-in-java
14. http://www.journaldev.com/1730/mediator-design-pattern-in-java-example-tutorial
15. http://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java-example-tutorial
16. http://www.journaldev.com/1739/observer-design-pattern-in-java-example-tutorial
17. http://www.journaldev.com/1754/strategy-design-pattern-in-java-example-tutorial
18. http://www.journaldev.com/1624/command-design-pattern-in-java-example-tutorial
19. http://www.journaldev.com/1751/state-design-pattern-in-java-example-tutorial
20. http://www.journaldev.com/1769/visitor-design-pattern-in-java-example-tutorial
21. http://www.journaldev.com/1635/interpreter-design-pattern-in-java-example-tutorial
22. http://www.journaldev.com/1716/iterator-design-pattern-in-java-example-tutorial
23. http://www.journaldev.com/1734/memento-design-pattern-in-java-example-tutorial
24. http://en.wikipedia.org/wiki/Design_pattern