DESIGN PATTERN

By SEKHAR SIR

[Thursday, May 29, 2014]

Recursive Problem:-

> If some problem occurs again and again in a particular context then we call it as a Recursive Problem.

> For example, if an audio player having support with MP2 files gets problem for MP3 files and having support for MP3 gets problem MP4. So it is a recursive problem.

> In a software application, for example a recursive problem will be transferring the data across layers.

Q. Why Design Patterns?

Ans:-

> Design patterns are used for solving recursive problems in a software application design.

> A design pattern is a description for how to solve a recursive problem.

> Design patterns are not a technology or a tool or a language or a platform or a framework.

> Design patterns are effective proven solutions for recursive problems.

Q. How many Design Patterns?

Ans:-

> Actually, there is no fixed count of no of design patterns because design patterns is not a package and not in the form of classes.

> SUN Microsystems constituted a group with four professional with the name of Gang of Four (GOF) to find effective solutions for the recursive problems.

> According to GOF, they found 23 design patterns as effective solutions for re-occurring problems.

> GOF divided java design patterns into 4 categories

(a) Creational Design Patterns:-
   (1) Singleton Pattern.
   (2) Factory Pattern
   (3) Abstract Factory Pattern
   (4) Builder Pattern
   (5) Prototype Pattern

(b) Structural Design Pattern
   (1) Adaptor Pattern
   (2) Proxy Pattern
   (3) Composite Pattern
   (4) Flyweight Pattern
   (5) Façade Pattern
   (6) Bridge Pattern
   (7) Decorator Pattern

(c) Behavioral Design Pattern
   (1) Template Method Pattern
   (2) Mediator Pattern
   (3) Chain of responsibility pattern
   (4) Strategy Pattern
   (5) Command Pattern
   (6) Visitor Pattern
   (7) Iterator Pattern

(d) J2EE Design Pattern
   (1) Intercepting Filter

(2) Front Controller
(3) Composite View
(4) View Helper
(5) Service Locator
(6) Business delegate
(7) Data Transfer Object
(8) Data Access Object
(9) Inversion of Control

[Friday, May 30, 2014]

toString() method:-

> toString() method is a method of java.lang.Object class.
> When we pass an object, as parameter to the output statement internally toString() method of that object will be called.
> If an object overrides toString() method then overridden toString() method will be called otherwise Object class toString(0 method will be called.
> If Object class toString() method is called then it returns classname@unsigned hexadecimal fromat of hashcode.

Example1:-

```
class Test
    {
        int a;
        Test(int a)
        {
        this.a=a;
        }
    }
class Main
    {
        public static void main(String args[])
        {
        Test test=new Test(10);
        System.out.println(test);
        }

    }
```

Output:-Test@4fD567

Example2:-

```
class Test
    {
        int a;
        Test(int a)
        {
        this.a=a;
        }
        public String toString()
        {
        return "Value : "+a;
        }
    }
    class Main
    {
        public static void main(String args[])
        {
        Test test=new Test(10);
        System.out.println(10);
        }
    }
```
Output:- Value:10

equals() and = = operator :-

- > In java Objects are compare in 2 ways.
  1. Identical comparison (= =)
  2. Meaningful comparison (equals())
- > Identical comparison means whether 2 references are referring a single object or not.
- > Meaningful comparison means whether 2 objects are equal according to the values or not.
- > In a class, if equals method is not overridden then there will be no difference between equals() method and = = operator. Because equals() methods of Object class internally use = = operator only.
- > If we assign an integer literal between the range of -128 to 127, then internally jvm creates a single object and makes other object as references to that single object.

  Example1:-
  ```
  Integer i1=100;
  Integer i2=100
  i1==i2 --> true
  i1.equals(i2) --> true
  ```
  Example2 :-
  ```
  Integer i1=200;
  Integer i2=200;
  i1==ii2 --> false
  i1.equals(i2) --> true
  ```
  Example3 :-
  ```
  Integer i1=129;
  Integer i2=159;
  i1==i2 --> true
  i1.equals(i2) --> false
  ```

[Tuesday, June 03, 2014]

- > If we want to compare two objects meaningfully then we must override equals() method in that class.
- > If we do not override equals() method, then super class (Object class) equals() method of Object class internally uses (= =) operator. So there is no difference between equals and = = operator in Object.

  Example:-
  ```
  class Test
  {
          int x;
          Test(int x)
          {
                  this.x=x;
          }
  }
  class Main
  ```

```
{
        public static void main(String args[])
        {
                Test t1= new Test();
                Test t2= new Test();
                if(t1==t2)
                {
                        System.out.println("t1 and t2 are identically equal");
                }
                if(t1.equals(t2))
                {
                        System.out.println("t1 and t2 are meaningfully equal");
                }
        }
}
```

Output: no output

- There is no output for the above example, because in Test class equals() method is not overridden. So equals and = = operator both are one and same.
- We can override equals() method of object class in Test class like the following.

```
@Override
public boolean equals(Object obj)
{
        if(obj instanceOf Test && (((Test)obj).x==this.x))
        return true;
        else
        return false;
}
```

- instanceOf operator checks an object at left side belongs to a class at right side or not.
- After overriding equals() method, the output of above example is t1 and t2 are meaningfully same.

Note:- public boolean equals(Test obj){----}--->overloading equals() method.
public booelan(Object obj){----}--->overriding equals() method

hashCode() method:-

- It is a native method of java.lang.Object class.
- In java, native methods are JVM dependent. It means their implementation exists in JVM.
- *If hashCode() method of Object class is not overridden by a class then hashCode() of object class returns an int value by converting memory address of an object into int value.*

[Wednesday, June 04, 2014]

- hashCode is not a memory address . It is an integer format of an object data.
- If hashCode() method is not overridden then the hashCode is an integer format of an object memory.
- In each class hashCode() method can be overridden. But the classes whose objects are going to be used as keys in a hash table and hast map must be overridden by hashCode() method.
- In all wrapper classes like Integer, Double, Float etc and String class hashCode() method is overridden.

hashCode() in String class:-

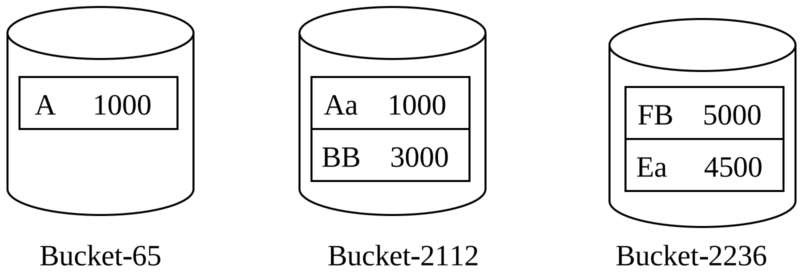
- In String class hashCode() method is overridden and the hashCode of a string is calculated by using the formula $s[0]*31^{n-1}+s[1]*31^{n-2}+\ldots\ldots\ldots\ldots+s[n-1]$.

- If two strings are equal according to equals() method then their hashCode() are definitely same.
  String str=new String("AB");
  Strign str2=new String("AB");
  str.hashCode(); ---->2081
  str2.hashCode();----->2081
- If two strings have same hashCode then they may or may not be equal according to equals() method
  String str=new String("Aa");
  Strign str2=new String("BB");
  str.hashCode(); ---->2112
  str2.hashCode();----->2112
- In the above example hashCode are same and the strings are unequal. So two unequal strings may produce either same hashCode or different hashCode.
- If two strings are unequal according to equals() method then there is no guarantee that they produce dustings hashCode.
  String str=new String("FB");
  Strign str2=new String("ab");
  str.hashCode(); ---->2236
  str2.hashCode();----->3105
- Here hashCode() are distinct and two strings are unequal.

Q. If two strings are equal according to equals() then hashCodes are same?
Ans:-Same
Q. If two strings hashCode is same then they are equal according to equals() method?
Ans:-May or May not
Q. If two strings are unequal according to equal method then they produce distinct hashCode?
Ans:-May or May not
Q. If two strings hashCode is different then they have unequal according to equals(0 mehod.
Ans:-Yes
[Thursday, June 05, 2014]
Q. When to override hashCode() and equals() method in a class?
Ans:-When we are working with hashing collections like Hashtable,HashSet and HashMap then the data will be stored internally in the form of hash buckets and each bucket has an identification number called hashCode

- While storing an element in a Hashtable and HashMap then first hashCode of he key is found and then that key value pair is going to be stored int that bucket.
- In order to find the hashCode internally hashCode() method of the key will be called.
  for example,
  Hashtable table=new Hashtable();
  table.put("A",1000);
  table.put("Aa",1000);
  table.put("BB",3000);
  table.put("FB",5000);
  table.put("Ea",4500);
  A----65
  Aa---2112
  BB---2112
  FB---2236

Ea---2236

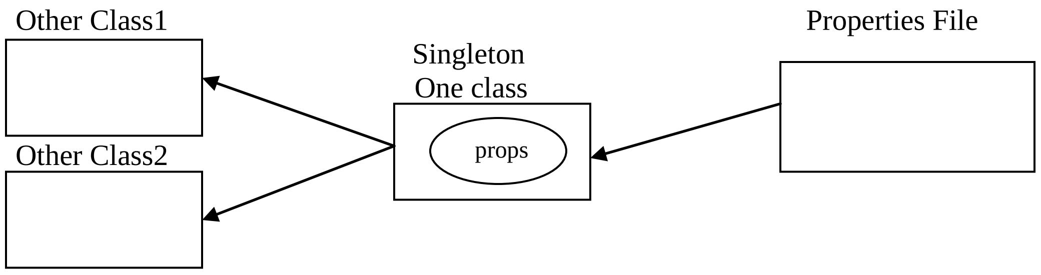


- > While retrieving the value of a key from a Hashtable or a HashMap, then first hashCode(0 method is called in the given key to find the right bucket and then equals() method is called to search for that key in that bucket.
- > While searching for a key in a Hashtable and HashMap then internally first hashCode() method is called after that equals() method is called.
- > So if we want to use any java classes object as key in Hashtable and HashMap collection then that hashCode() method and equals() method must be overridden.

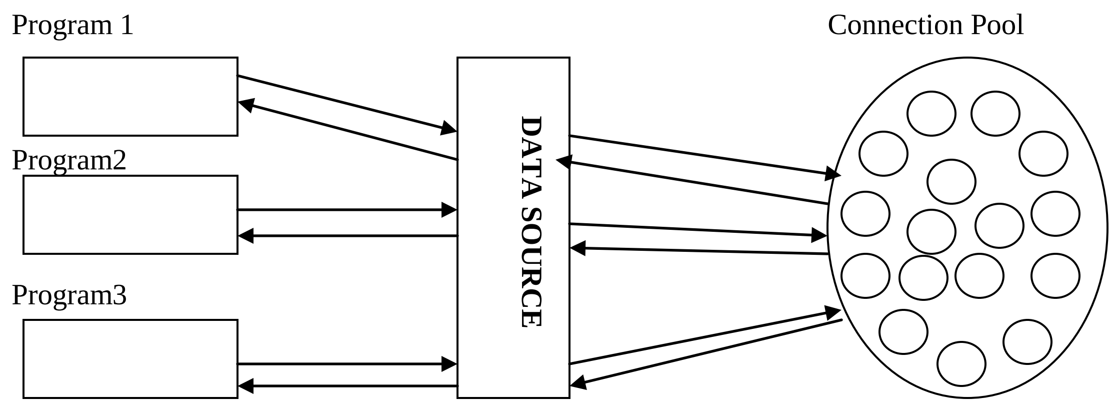
[Friday, June 06, 2014]

Creational Design Pattern:-

- > Creational Design Pattern deals with object creation.
- > Creational Design Pattern tries to create objects according to the requirement.
- > Creational Design Pattern hides object creation process from the client applications.
- > Singleton design pattern says those create one instance (object) of a class and provide a global access point for that instance.
- > Generally if a class has a fixed functionality and it does not differ for any no. of object created for that class then we make that class as Singleton class.
- > If multiple objects are created further same functionality then the memory for the entire remaining object will be canted. So to reduce the memory wastage, we make that class as Singleton.
  For example,
  Suppose, we create a class for reading a properties file data into properties file data into properties object and returning that properties class object to the other classes.
  Here the logic in a class is fixed and if multiple objects are created for that class then the memory will be wasted. So we make that class as Singleton class
- > The other classes we use the one instance of the class and reach the properties object from that class.



- > Another example we find in Real time is, while obtaining connection from connection pool by multiple programs, one data-source object is acting as Singleton object.



Different ways of making a class as Singleton:-

Approach1:-

- We can make a class as a static class. A static class is implicitly a Singleton class.
- To make a static class, we need to make all properties and methods as static methods and make the constructer of that class as private.

Approach2:-

- If we want to make a class as Singleton then object creation of that class should not allowed from outside of the class. To restrict an object creation of class from outside, we need to make constructer of that class as private.

[Saturday, June 07, 2014]

- We need to create one object of that class inside the class and then we need to define a public static factory method to return the one object to the other classes.

  For example,

```
public class OnlyOne
{
        private OnlyOne(){}
        public static OnlyOne getInstance()
        {
        return onlyone;
        }
}
class Simple
{
        public static void main(String args[])
        {
        OnlyOne onlyone=OnlyOne.getInstane();
        OnlyOne onlyone2=OnlyOne.getInstance();
        System.out.println(onlyone == onlyone2);
        System.out.println(OnlyOne.hashCode()==OnlyOne.hashCode());
        }
}
Output: true
        true
```

- In the above code, the one object of class created at class load time. It means the object is early created.
- The drawback of early creation is if the object is not used by any other classes then the object memory is wastage.

Approach3:-

> We can create the one object of the Singleton class inside the static factory method. So that we can create the one object lazily and hence we can avoid the memory wastage problem of the above approach.

> In a multi-threading environment, if two threads are simultaneously calling the static factory method, then two objects of Singleton class will be created. So the Singleton principle is violated.

> To solve the concurrent issue, we need to make the static factory method as a synchronized method.

> If a method is synchronized then only one thread is allowed to access it simultaneously. So that the object for the class will be created only for once.

```
public class OnlyOne
{
        private static OnlyOne onlyone=null;
        private OnlyOne(){}
        public synchronized static OnlyOne getInstance()
        {
                if(onlyone=null)
                        {
                                onlyone=new OnlyOne();
                        }
                        return onlyone;
        }
}
```

> The above Singleton class is an appropriate way of making the class as Singleton. But if another class clones an object of the Singleton class then one more object of Singleton class will be created. It means again Singleton principle is violated.

Approach4:-

> To make a java class as 100% Singleton class, we should not allow object cloning.

> In order to restrict the cloning, we should override clone() method in Singleton class by implementing that class from Cloneable interface.

```
public class OnlyOne implements Cloneable
{
        private static OnlyOne onlyone=null;
        private OnlyOne(){}
        public synchronized static OnlyOne getInstance()
        {
                if(onlyone=null)
                        {
                                onlyone=new OnlyOne();
                        }
                        return onlyone;
        }
        public Object clone() throws CloneNotSupportedException
        {
                throw new CloneNotSupportedException;
        }
}
```

[Monday, June 09, 2014]

Factory Design Pattern:-

> When we want to meet the following requirements then we apply factory design pattern.
 (1) We want to hide the object creation process.
 (2) We want to hide which subclass object is created of a class or interface.
 (3) We want to make object creation process as reusable for the entire application.

> Factory design pattern and factory method design pattern, both are one and same.

> The mostly used design pattern across multiple technologies and framework of java is the factory design pattern.

Example1:-

> In JDBC, we are obtaining a connection with a database by calling getConnection() method.

> Our application is a client application and our application does not need how and which connection class object is created, but it needs a connection object.

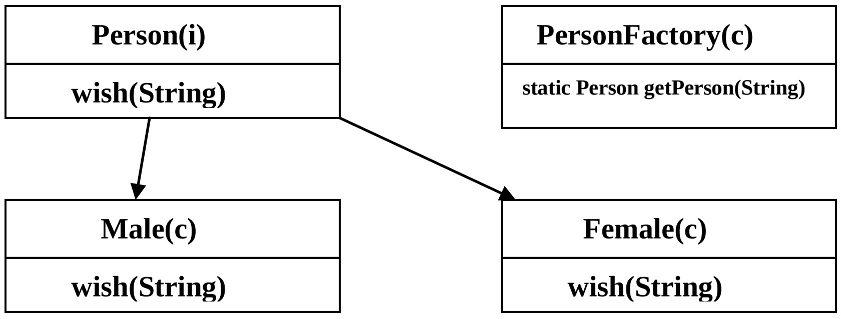
> So getConnection() method hides the object creation process and simply returns a connection object needed.
Connection con=DriverManger.getConneciton(url,uname,pwd);

> In the above statement factory design pattern is applied.

Example2:-

> In java.net package, there is a class is called URL and its openConnection() method returns an object of type URLConneciton, by hiding the creation process and also by hiding the details of which class object is created. So here also a factory design pattern.
URL url= new URL(String url);
URLConnection connection=url.openConnection();

The following example creates a static factory method to return a object of type Person to the client application.



```
C:\FactoryPattern\
public interface Person
{
void wish(String name);
}
//Male.java
class Male implements Person
{
        @Override
        public void wish(String name)
        {
        System.out.println("Welcome Mr. "+name);
        }
}
//Female.java
```

```
	public class Female implements Person
	{
		@Override
		public void wish(String name)
		{
		System.out.println("Welcome Miss/Mrs "+name);
		}
	}
//PersonFactory.java
public class PersonFactory
{
	public static Person getPerson(String type)
	{
	if(type.equals("M"))
		{
			return new Male();
		}
	else if(type.equals("F"))
		{
			return new Female();
		}
	else
			return null;
	}
}
//Client.java
class Client
{
	public static void main(String args[])
	{
	Person person=PersonFactory.getPerson(args[0]);
	person.wish("XYZ");
	}
}
```

Abstract Factory Design Pattern:-

> Abstract Factory is a factory of factories.

> If factory design pattern is again applied on a factory class, i.e. if we apply factory on factory then we will get Abstract Factory.

> We have to use this abstract factory design pattern, when we want to return one factory object among a family of factory objects.
For example, in JAX-P (Java API for XML parsing), a DocumentBuilderFactory is a factory for DocumentBuilder object. Again DocumentBuilder is a factory for Document object. So DocumentBuilderFactory is called abstract factory.

```
DocumetnBuilder builder=DocumentBuilderFactory.getInstance();
Document document=builder.getDocument();
```

[Tuesday, June 10, 2014]

> In the following example, we are applying factory on factory.

> In the example, AnimalAbstractFactory is a factory for producing AnimalFactory and AnimalFactory is a factory for producing Animal.

$$\begin{tikzcd}
 & \text{Animal(i)} \arrow[dl] \arrow[dr] & \\
\text{Elephant(c)} & & \text{Shark(c)}
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \text{Animal Factory(i)} \arrow[dl] \arrow[dr] & \\
\text{LandFactory(c)} & & \text{SeaFactory(c)}
\end{tikzcd}$$

AnimalAbstractFactory(c)

```
AbstractFactory
   ├──►Animal.java
   ├──►Elephant.java
   ├──►Shark.java
   ├──►AnimalFactory.java
   ├──►SeaFactory
   ├──►LandFactory
   ├──►AnimalAbstractFactory
   └──►Test.java
```

```
    //Animal.java
    public interface Animal
    {
            void breathe();
    }
    //Elephant.java
    public class Elephant implements Animal
    {
            public void breathe()
            {
                    System.out.println("Elephant is Breathing");
            }
    }
    //Shark.java
    public class Shark implements Animal
    {
            public void breathe()
            {
                    System.out.println("Shark is Breathing");
            }
    }
//AnimalFactory.java
public interface AnimalFactory
{
    Animal getAnimalInstance();
}
//LandFactory.java
public class LandFactory implements AnimalFactory
{
    public Animal getAnimalInstance()
            {
                    return new Elephant();
            }
}
//SeaFactory.java
public class SeaFactory implements AnimalFactory
{
    public Animal getAnimalInstance()
            {
                    return new Shark();
            }
}

//AnimalAbstractFactoy.java
public class AnimalAbstractFactory
{
    public static AnimalFactory
    getAnimalFactoryInstance(String type)
            {
                    if(type.equals("water"))
                            return new SeaFactory();
                    else
                            return new KLandFactory();
            }

}
//Test.java
public class Test
{
    public static void main(String args[])
            {
                    AnimalFactory af=AnimalAbstractFactory.getAnimalFactoryInstance(args[0]);
                    Animal a=af.gerAnimalInstance();
                    a.breathe();
            }
}
```

Compile and Run the application.

Prototype Design Pattern:-

> This design pattern is applies when an object creation for a class is a costly (time-taking) operation.

> This design pattern says that, instead of re creating a costly object of a class, create one object of the class and put it in cash and then return a clone of it to the clients whenever it is asked.

> While cloning an object a duplicate object of the original object is created by without invoking a constructer of the class.

> This prototype design pattern will help to improve the performance of an application.

> **Cloning an object is nothing but making an identical copy of an existing object.**

> After cloning, the original object and cloned object will have separate memories.

> If we want to clone any java class objects then that class must be implement java.lang.Cloneable interface and it is a marker interface.

> In a class, we should override clone() method of Object class.

For example:-

```
public class Test implements Cloneable
```

```
{
        public Object clone() throws cloneNotSupportedException
                {
                }
}
```

- > There 2 types of cloning
  (a) Shallow Cloning (b) Deep Cloning
- > In shallow cloning, only primitive properties of an object are cloned but its reference type is not cloned.
- > In deep cloning, both primitive types and also reference types of an object are closed.

[Wednesday, June 11, 2014]

- > In the following example, we are cloning Employee class object and it contains a reference of Department class.
- > We are applying shallow cloning, which means the original and cloned object of Employee will share a single object of department.

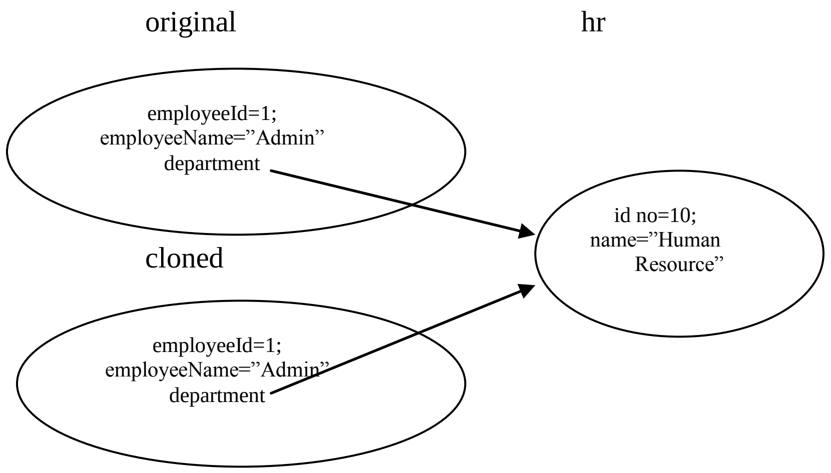
  Shallow Clone

  Employee.java
  Department.java
  TestCloning.java

```
        //Employee.java
        public class Employee implements Cloneable
        {
                private int employeeId;
                private String employeeName;
                private Department department;
                public Employee(int id, String name, Department dept)
                        {
                                this.employeeId=id;
                                this.employeeName=name;
                                this.department=dept;
                        }
                @Override
                public Object clone() throws CloneNotSupportedException
                {
                        return super.clone();
                }
        }
        //create setter and getters for the private properties.
//Department.java
public class department
{
        private int id;
        private stirng name;
        public department(int id, String name)
        {
                this.id=id;
                this.name=name;
        }
}
```

```
//create setters and getters
//TestCloning.java
public class TestCloning
{
        public static void main(String args[])throws CloneNotSupportedException
        {
                Department hr=new Department(10,"Human Resource");
                Employee original=new Employee(1,"Admin",hr);
                Employee cloned=(Employee)original.clone();
                //Let change the deparment name in cloned object and we will verify in original object
                cloned.getDepartment().setName("Finance");
                System.out.println(original.getDepartment().getName());
        }
}
```

- In the above client application, the original object and cloned object of Employee are sharing a single object (same object) of Department. So the changes made by cloned object on Department object will affect on original object also.
- The output of the above client application is Finance.



Deep Clone

- In case of deep cloning, along with primitive types reference types are also cloned. So changes made by original object on department will not affect on cloned object and vice versa.
- If we want to clone Employee object with deep cloning then the following changes are needed.

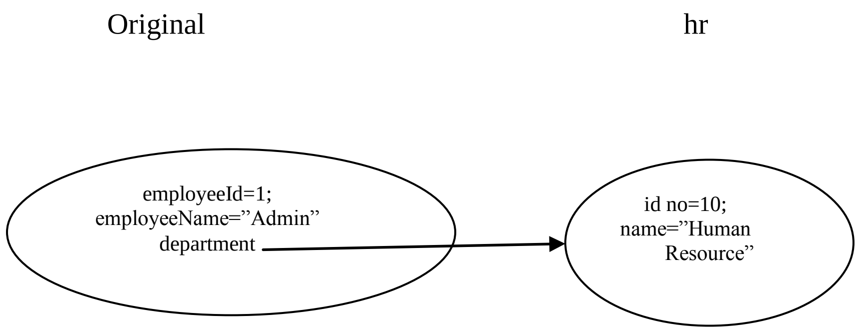
(1) Override clone method in Employee class like the following

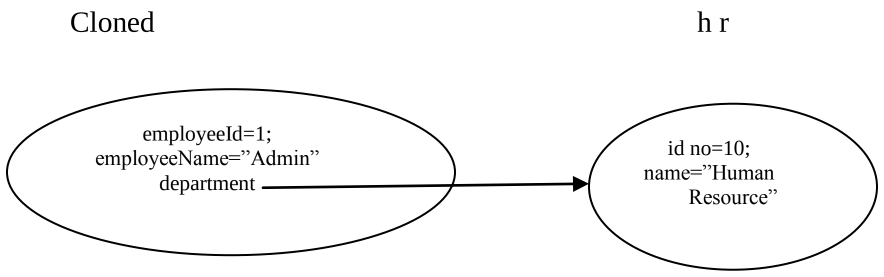
```
protected Object clone() throws CloneNotSupportedException
{
    Employee cloned= (Employee)super.clone();
    cloned.setDepartment((Department)cloned.getDeparment().clone());
    return cloned;
}
```

(2) Implement Department class from Cloneable interface and override clone method in Department class like the following.

```
@Override
protected Object clone() throws CloneNotSupportedException
{
```

```
            return super.clone();
        }
```





> In the following example we are applying prototype design pattern for objects of type Shape class.

> We are creating objects of type shape and putting in a Hashtable and a clone object of cached object is returned for the client.
In this example, Hashtable object is acting as chche

Prototype

- Shape.java
- Circle.java
- Square.java
- Rectangle.java
- ShapeCache.java
- Main.java

```
public abstract class Shape implements Cloneable
{
    private String id;
    protected String type;
    abstract void draw();
    public String getType()
    {
    return type;
    }
}
create getter and setter for id

public Object clone()
{
    Object clone=null;
    try
    {
```

```
    clone=super.clone();

    }catch(CloneNotSupportedException e)
    {
    e.printStackTrace()
    }
    return clone();
}
public class Circle extends Shape
{
    public Circle()
    {
            type="Cirlce";
    }
    @OVerride
    public void draw()
    {
            System.out.println("inside circle :: draw() method");
    }
}

public class Square extends Shape
{
    public Square()
    {
    type="Square";
    }
    @Override
    public void draw()
    {
    System.out.println("inside Square :: draw() method");
    }
}

public class Rectangle extends Shape
{
    public Rectangle()
    {
    type="Rectangle";
    }
    @Override
    public void draw()
    {
    System.out.println("inside Rectangle :: draw() method");
    }
}
import java.util.Hashtable;
public class ShapeCache
{
    private static Hashtable<String,Shape> ht=new Hashtable<String,Shape>();
    public static Shape getShape(String shapeId)
```

```
        {
        Shape cachedShape=ht.get(shapeID);
        return (Shape)cachedShape.clone();
        }
        public static void loadCache();
        {
        Circle cirlce=new Circle();
        circle.setId("1");
        ht.put(circle.getId(),cirlce);

        Square square=new Square();
        square.setId("2");
        ht.put(square.getId(),square);

        Rectangle rectangle=new Rectangle();
        rectangle.setId("3");
        ht.put(rectangle.getId(),rectangle);
        }
    }
    public class Main
    {
        public static void main(String args[])
        {
        ShapeCache.loadCache();

        Shape clonedShape=ShapeCache.getShape("1");
        System.out.println("Shape:"+clonedShape.getType());

        Shape clonedShape2=ShapeCache.getShape("2");
        System.out.println("Shape:"+clonedShape2.getType());

        Shape clonedShape3=ShapeCache.getShape("3");
        System.out.println("Shape:"+clonedShape3.getType());

        Shape cloneShape1=ShapeCache.getShape("1");
        System.out.println("Shape:"+clonedShape1.getType());

        System.out.println(clonedSahpe.hashCode());
        System.out.println(clonedSahpe1.hashCode());
        }
    }
```

[Thursday, June 12, 2014]

> This design pattern is used, to separate construction process of an object to re-use the same construction process for representing that object in different ways.

> This builder design pattern will make use of same construction process to represent an object in another different way in future.

> For example, we have a complex object are called House and its construction process will be same for different representations of the house. So we can separate the construction process and we can reuse it for multiple representations with the help of builder design pattern.

> For example, if we want wooden representation of House or Ice representation of house then same construction process to be follow for the different representations.

```
Builder
   ├──▶House.java
   ├──▶HouseBuilder.java
   ├──▶IglooHouseBuilder.java
   ├──▶TipiHouseBuilder.java
   └──▶BuliderSample.java
```

```
//House.java
public class House
{
        private String basement;
        private String structure;
        private String roof;
        private String interior;
        //Create setter methods of above
        public String toString()
                {
                        String
str="Basement:"+basement+"\n"+"Structure:"+structure+"\n"+"Roof:"+roof+"\n"+"Interion:"+int
erior;
                        return str;
                }
}
//HouseBuilder
public interface HouseBuilder
{
        public void buildBasement;
        public void buildStructure;
        public void buildRoof;
        public void buildInterior;
        public void constructHouse();
        public House getHouse();
}
```

```
//IglooHouseBuilder
public class IglooHouseBuilder implements HouseBuilder
{
        private House house;
        public IglooHouseBuilder()
        {
                this.house= new House();
        }
        pbulic void buildBasement()
        {
                house.setBasement("Ice Bars");
        }
        public void buildStructure()
        {
                house.setStructure("Ice Blocks");
        }
        public void buildInterior()
        {
                house.setInterior("Ice Carvings");
        }
        public void buildRoof()
        {
                house.setRoof("Ice Dome");
        }
        publi void costructHouse()
        {
        this.buildBasement();
        this.buildStructure();
        this.buildRoof();
        this.buildInterior();
        }
        public House getHouse()
        {
        return this.house;
        }
}
//TipiHouseBuilder
public class TipiHouseBuilder implements HouseBuilder
{
        private House house;
        public TipiHouseBuilder()
        {
                this.house= new House();
        }
        pbulic void buildBasement()
        {
                house.setBasement("Wooden Poles");
        }
        public void buildStructure()
        {
                house.setStructure("Wood and Ice");
        }
        public void buildInterior()
        {
                house.setInterior("Fire Wood");
        }
        public void buildRoof()
        {
                house.setRoof("Wood Skins");
        }
        publi void costructHouse()
        {
        this.buildBasement();
        this.buildStructure();
        this.buildRoof();
        this.buildInterior();
        }
        public House getHouse()
        {
        return this.house;
        }
}
//HouseBuilderFactory.java
public class HouseBuilderFactory
{
        public static HouseBuilder
        getHouseBuilderInstance(String type)
                {
                        if(type.equals("Igloo"))
                        return new IglooHouseBuilder();
                        else if(type.equals("Tipi"))
                        return new TipiHouseBuilder
                        else
                        return null;
                }
}
//BuilderSample.java
class BuilderSample
{
        public static void main(String args[])
                {
                HouseBuilder builder=HouseBuilderFactory.getHouseBuilderInstance(args[0]);
                builder.constructHouse();
                House house=builder.getHouse();
                System.out.println(house);
                }
}
```

Output:

```
C:\ java BuilderSample Igloo
Basement:       Ice Bars
Structure:      Ice Blocks
Roof:           Ice Dome
Interior:       Ice Carvings
```

House ------ Complex Object
HouseBuilder----Constructio process
TipiHouseBuilder----representation -1 of house
IglooHouseBuilder----representation-2 of house
Structural Design Pattern:-

> These categories of design patterns talks about aggregation and composition of object. It means they talks about HAS-A-Relationship between objects.
> In HAS-A Relationship, one object contains another object. The one object is called container object and another object is called contained object.
> For example, Library class object contains Book class object. So Library object is container object and book object is contained object.
> An Aggregation, a contained object can exists independently without container object also without a container object.
> I composition, a contained object cannot exist without a container object.

- For example, a Book object can exist without a Library. So there is aggregation between Library and Book object.
- For example, a Department object cannot exist without a College object. So there is a composition between College and Department object.

[Friday, June 13, 2014]

Proxy Design Pattern:-

- Proxy is a term indicates "in place of" or "on behalf of ".
- For example, ATM is a proxy for a bank.
- Proxy design pattern is used in the following cases.
  (1) When we want to efficiently manage the memory of expensive objects.
  (2) When we want to control the access to a real object.
  (3) When we want to represent an object resided at one jvm at another jvm.
- In web applications, a filter acts as a proxy for a servlet.
- In distributed applications with RMI, CORBA, EJB and Web-services, proxy objects are created for client and server objects of communication, for representing server object at client side and client object at server side.
- Proxy Objects are 2 types
  (1) Remote Proxy (2) Virtual Proxy
- If the proxy object is representing a real object belongs to same jvm then it is called virtual proxy.
- If proxy object is representing a real object of another jvm then we call it as remote proxy.

Flyweight Design Pattern:-

- While creating multiple objects of a class, if multiple objects contain some common data across objects and some uncommon data across objects then flyweight design pattern is applied.
- If memory allocated separately for common data across multiple objects of a class then more memory is consumed and it leads to poor performance.
- In order to reduce the memory consumption and also to improve the performance we need to separate common data into one object and then we need to share it across multiple objects. This is called a flyweight design pattern.
- ***** There is a difference between flyweight design pattern and static members. If we make common data as static members then the static members become common to all objects of the class but not for some objects of the class.
- According flyweight design pattern the common data is classed ***intrinsic data*** and uncommon data is called ***extrinsic data***.
- For example,

```
public class IdCard
{
        private String companyName;
        private String division;
        private in empno;
        private String empnae;
        ...............
}
```

  - The above class is for creating id cards for employees of the company working at each division.
  - For example, if there are 50 employees are working at division 1 then the company name and division are common data for 50 employees.
  - In order to reduce the memory consumption, we can separate company name and division into one object and we can share it for 50 employees.
  - If separated like the above statement then flyweight design pattern is applied.

- In the above Idcard class ***intrinsic*** data is companyName and division and ***extrinsic*** data is employeeNumber and employeeName.

[Saturday, June 14, 2014]

Flyweight Design Pattern:-

- Flyweight
    - Icard.java
    - IntrisicFactory
    - Main.java

```
public class Icard
{
    private stirng empname;
    private int empid;
    private Intrinsic intr;
    public Icare(String empname, int empid, Intrinsic intr)
    {
            this.empname=empname;
            this.empid=empid;
            this.intr=intr;
    }
    public void print()
    {
            System.out.println("Empname:"+empname);
            System.out.println("Empid:"+empid);
            System.out.println("Comp Name:"+intr.getCompName());
            System.out.println("Division"+intr.getDivision());
    }
}
public interface Intrinsic
{
    String getDivision();
    String getCompName();
}
import java.util.*;
public class IntrinsicFactory
{
    public static Map<String, Intrinsic> m=new HashMap<String,Intrinsic>();
    public static count;
    public static Intrinsic getIntrinsic(String division)
            {
                    if(m.containsKey(division))
                            {
                                    Object o=m.get(division);
                                    Intrinsic i=(Intrinsic)o;
                                    return i;
                    }
                    else
                    {
                            Intrinsic intr=new IntrinsicFactory.IntrinsicClass("SYZ Ltd",division);
                                    m.put(division,intr);
                                    return intr;
                            }
```

```
		}
		private static class IntrinsicClass implements Intrinsic
		{
			private String division;
			private String compName;
			private IntrinsicClass(String compName,String division)
			{
				this.compName=compName;
				this.division=division;
				count++;
				System.out.println("No of intrinsic objects: "+count);
			}
		@Override
		public String getDivision()
		{
			return division;
		}
		public String getCompName()
		{
			return compName;
		}
		}
}
class Main
{
   public static void main(String args[])
   {
		Icard ic1=new Icard("A",111,InstrincFactory.getInstrinc("COMP"));
		Icard ic2=new Icard("B",112,InstrincFactory.getInstrinc("COMP"));
		Icard ic3=new Icard("C",901,InstrincFactory.getInstrinc("FIN"));
		Icard ic4=new Icard("D",902,InstrincFactory.getInstrinc("COMP"));
		ic1.print();
		System.out.println("=====================");
		ic2.print();
		System.out.println("=====================");
		ic3.print();
		System.out.println("=====================");
		ic4.print();
		System.out.println("=====================");
   }
}
```

Compile and Execute.

```
No of Intrinsic Object=1;
No of intrinsic object=2;
Emp Nmae        :A
Comp Name       : XYZ Ltd
Division        :COMP
.
.
.
.
```

> While defining one class in another class, if we put static keyword for the inner class then it becomes as nested class.
> If we put an inner class or a nested class with private modifier then an object of that class cannot be created at outside of its outer class.
> In the above example code, the factory method of IntrinsicFactory class checks whether an intrinsic object of a division existing in Map object or not.
> If exists then returns the same intrinsic object and if not then creates a new intrinsic object puts it in Map and then returns it.
> The no of intrinsic object created in above example are two.

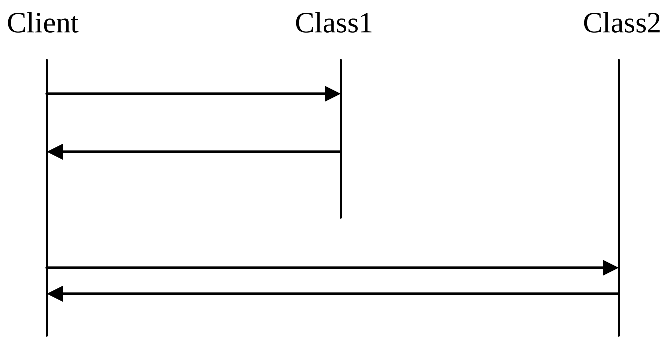
[Monday, June 16, 2014]

Façade Design Pattern:-

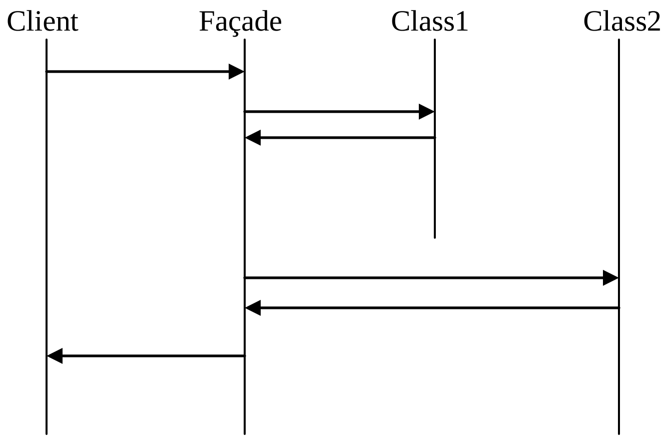
> When there are multiple sub-systems are needed to invoke from a client for an operation then façade design pattern is used.
> Façade pattern hides the complexity of calling multiple subsystem form a client, by providing a high level system.
> A client invokes a façade and façade invokes the multiple subsystems and finally returns the result back to the client.

Example1:-

If there is a requirement that the result of class1 of pack1 is needed as input for calling class2 of pack2, in order to get the desired final output then client has to call class1 and then followed by class2.

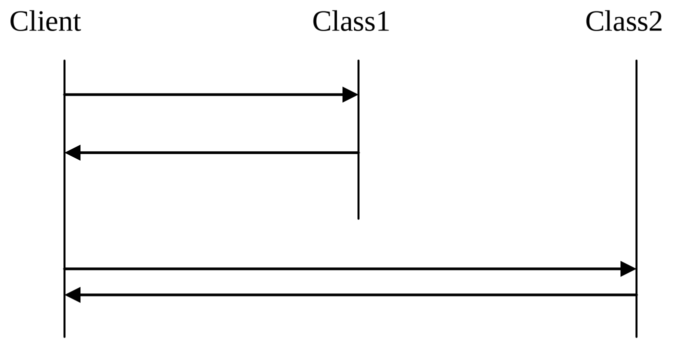


If façade is applied for the above diagram then it becomes like the following.

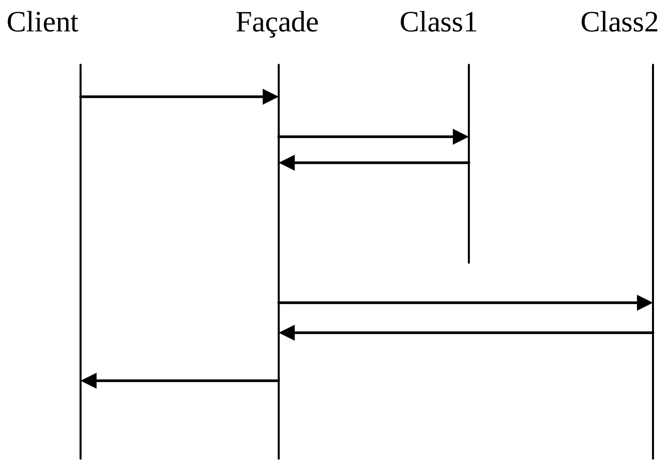


Example2:-

If you want to transfer the money from accout1 to account2 then the two subsystems to be invoked are withdraw from account1 and deposit to accoutn2.



> If façade pattern is applied then the result will be like the following.



Decorator Design Pattern:-

- This design pattern is used when we want to add some additional functionality to some objects of a class at runtime but not for all objects of a class.
- If additional behavior is added for a class is extending it from some super class then the additional behavior is added for all object of the class but not for some objects.

  Example1:-

  Using BufferedReader class object, we can add a special behavior at runtime for one Reader object. So BufferedReader is a decorator of Reader object.

  Example2:-

  Suppose we have a Window class and we have 3 objects for it. We want to add scrollbars for one window object but not for all window objects then we apply decorator design pattern.

  Window w1=new Window();
  Window w2=new Window();
  Window w3= new Window();
  ScrollBarsDecorator window= new ScrollBarsDecorator(w3);

  In the following example, we are applying Decorator design pattern for decoration 2 Dollar class objects among 4 objects of Dollar class.

```
C:\Decorator
    ├─►Currency.java
    ├─►Rupee.java
    ├─►Dollar.java
    ├─►Decorator.java
    ├─►AUSDDecorator.java
    ├─►SGDDecorator.java
    └─►Main.java
C:\Decorator
abstract class Currency1
{
        public String description;
        public Currency1()
        {
                description="unknown";
        }
        public String getDescription()
        {
                return description;
        }
        abstract dcouble cost(double value);
}
public class Rupee extend Currency1
{
        public Rupee()
        {
                description="Rupee";
        }
        public double cost(double value)
        {
                return value;
        }
}
public class Dollar extend Currency1
{
        public Dollar()
        {
                description="Dollar";
        }
        public double cost(double value)
        {
                return value;
        }
}
public class Decorator
{
        abstract String getDescription();
}
public class AUSDDecorator extends Decorator
{
		Currency1 currency1;
		public AUSDDecorator (Currency1 currency1)
		{
			this.currency1=currency1;
		}
		public String getDescription()
		{
			String str=currency1.getDescription()+"It's Australlin Dollar"
			return str;
		}
	}
	public class SGDDecorator extends Decorator
	{
		Currency1 currency1;
		public SGDDecorator (Currency1 currency1)
		{
			this.currency1=currency1;
		}
		public String getDescription()
		{
			String str=currency1.getDescription()+"It's Singapore Dollar"
			return str;
		}
	}
	public class Main
	{
		public class void main(String args[])
		{
		Currency1 c1=new Rupee();
		System.out.println(c1.getDescription());
		Currency1 c2=new Dollar();
		Currency1 c3=new Dollar();
		Currency1 c4=new Dollar();
		Currency1 c5=new Dollar();
		AUSDDecorator d1= new AUSDDecorator(c3);
		SGDDEcorator d1=new SGDDEcorator(c5);
		System.out.println(d1.getDescription());
		System.out.println(c2.getDescription());
		System.out.println(c4.getDescription());
		System.out.println(d2.getDescription());
		}
	}
```

[Tuesday, June 17, 2014]

Adapter Design Pattern:-

> This design pattern is used to make two incompatible objects as compatible.

> For example, a mobile charger need 5 volts power and a wall socket produce 240 volts of power. So both are incompatible. To make the two as compatible we use adapter.

> Another example is, a memory card and laptop are incompatible objects. To make them as compatible, we use a card-reader. This card reader is an adapter.

- In the following code we have two classes MP3Player and MP4Player both are two different classes and to make the two classes are compatible, we are using one adapter class called AudioAdapter

```
//MP3Player.java
public class MP3Player
{
        private AudioAdapter adapter;
        public void play(String type)
        {
                if(type.equale("mp3"))
                {
                //play
                }
                else if(type.equals("mp4"))
                {
                adapter = new AudioAdapter()
                adapter.paly();
                }
                else
                {
                System.out.println("Invalid Format");
                }
        }
}
```

```
//MP3Player.java
public class AudioAdapter
{
        private MP4Player player;
        public void play()
        {
                player=new MP4Player();
                player.play();
        }
}
//MP3Player.java
public class MP4Player
{
        public void paly()
        {
        //play
        }
}
```

Behavioral Design Pattern

Observer Design Pattern:-

- This design pattern is used, whenever we want to establish one-to-many communication between the objects.
- In this design pattern, whenever the state of one object is modified then it should update its new state with all registered observer objects.
- In this observer design pattern the one object is called subject or observable and many objects are called observers.

For example, a company is a subject and shareholders are observers. Whenever a company policy is modified then it will be updated to all shareholders of the company. So here observer design pattern is applied.
Another example is if a blog has multiple registered users then the changes made to the blog will be updated to all the registered users. Here blog is a subject and registered users are the observers. The design pattern applied here is observer design pattern.

> The following example, subject with 3 observers. When a state change occurred in subject then it is notified to all observers.

```
ObserverPattern
  ├──► Subject1.java
  ├──► SubjectImpl.java
  ├──► Observer1.java
  ├──► ObserverImpl.java
  └──► ObserverTest.java
```

```
public interface Subject1
{
        void setState(String state);
        String getState();
        void addObserver(Observer1 ob);
        void removerObserver(Observer ob);
        void print();
}
import java.util.*;
public class SubjectImpl emplements Subject1
{
        private String state ="Hello";
        private List al= new ArrayList();
        public void setState(String state)
        {
        this.state=state;
        notifyObservers();
        }
        public string getState()
        {
        return state;
        }
        public void addObserver(Observer1 o)
        {
        al.add(o);
        }
        public void removeObserver(Observer1 o)
        {
        al.remove(o);
        }
        public void notifyObservers()
        {
        Iterator it=al.iterator();
        while(it.hashNext())
        {
        Observer1 o1=(Observer1)it.next();
        o1.update(this);
        }
        }
        public void print()
        {
        System.out.println("Subject state is : "+getState());
        }
}
public interface Observer1
{
        public void update(Subject1 s);
        public void print();
}
public class ObserverImpl implements Observer1
{
        private String str1=null;
        public void update(Subject1 s)
        {
        str1=s.getState();
        }
        public void print()
        {
        System.out.println("Observer state is : "+str1);
        }
}
public class ObserverTest
{
        public static void main(String args[])
        {
        Subject1 s1=new SubjectImpl();
        Observer o1=new ObserverImpl();
        Observer o2=new ObserverImpl();
        Observer o3=new ObserverImpl();

        s1.add.Observer(o1);
        s1.add.Observer(o2);
        s1.add.Observer(o3);

        System.out.println("Default Values");

        s1.print();
        o1.print();
        o2.print();
        o3.print();

        s1.setState("New State");
        System.out.println("----------------");
        s1.print();
        o1.print();
        o2.print();
        o3.print();
        System.out.println("-----------------");
```

```
                s1.removeObserver(o3);

                s1.setState("New State2");
                s1.print();
                o1.print();
                o2.print();
                o3.print();

                }

        }
```
Output:-

[Wednesday, June 18, 2014]

Strategy Design Pattern:-

> Define a group of algorithms, encapsulate the data and make the algorithms are interchangeable in coding. This is called strategy design pattern.

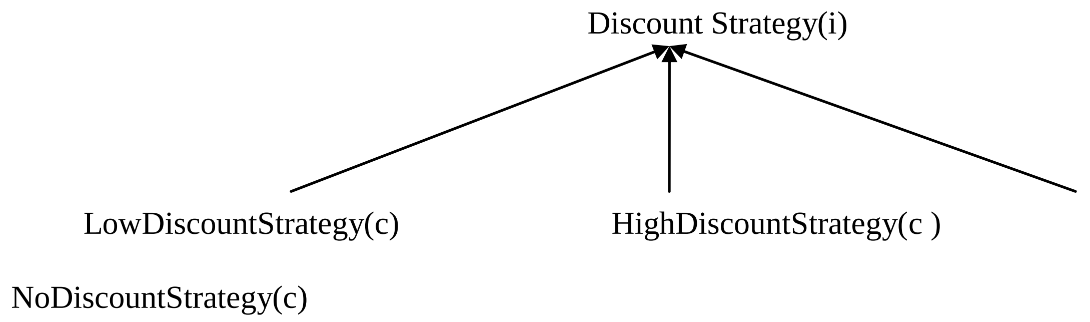
> For example, we have a group of sorting algorithms like bubble short, radix short, shell short etc. In a client application we can interchange one algorithm at runtime, by without making the changes to client application.

> Another example is, we have a group of files compressing algorithms like zip, rar or jar etc. In a client application we can interchange one algorithm with another at runtime by without making any changes to a client application.

> In hibernate, we have different inheritance strategies and we can replace one strategy with another, by without making any changes to the client application.

For example:-

> We are creating a client application called shopping mall and we are replacing the discount strategy from one to another by without making any changes to shopping mall class.



```
                public interface DiscountStrategy
                {
                        public double getFinalBill(int amount);
                }
                public class LowDiscontStrategy implements DiscountStrategy
                {
                        @Override
                        public double getFinalBill(int amount)
```

```
        {
        return (amount-(amount*0.1));
        }
}
public class HighDiscontStrategy implements DiscountStrategy
{
        @Override
        public double getFinalBill(int amount)
        {
        return (amount-(amount*0.5));
        }
}
public class NoDiscontStrategy implements DiscountStrategy
{
        @Override
        public double getFinalBill(int amount)
        {
        return amount;
        }
}
public class ShoppingMall
{
        private String customerName;
        private int amount;
        private DiscountStrategy strategy;

        public void setCustomerName(String customerName)
        {
        this.customerName=customerName;
        }
        public void setAmount(int amount)
        {
        this.amount=amount;
        }
        public void setStrategy(DiscountStrategy strategy)
        {
        this.strategy=strategy;
        }
        public String getCustomerName()
        {
        return customerName;
        }
        public int getAmount()
        {
        return amount;
        }
```

```
            public double getFinalBillAfterDiscount()
            {
            double finalAmount=strategy.getFinalBill(amount);
            return finalAount;
            }
        }
        public class Main
        {
        public static void main(String args[])
        {
            //Monday Customer
            ShoppingMall s1=new ShoppingMall();
            s1.setCustomerName("Monday Customer");
            s1.setAmount(1000);
            s1.setStrategy(new LowDiscountStrategy());
            System.out.println("Customer Name : "+s1.getCustomerName());
            System.out.println("Amount : "+s1.getAmount());
            System.out.println("Amount          after          Discount          :
        "+s1.getFinalBillAfterDiscount());

            //Thursday Customer
            ShoppingMall s2=new ShoppingMall();
            s2.setCustomerName("Thursday Customer");
            s2.setAmount(1000);
            s2.setStrategy(new HighDiscountStrategy());
            System.out.println("Customer Name : "+s2.getCustomerName());
            System.out.println("Amount : "+s2.getAmount());
            System.out.println("Amount          after          Discount          :
        "+s2.getFinalBillAfterDiscount());
        }
        }
```

Output:-

```
Customer Name:      Monday Customer
Amount:             1000
Amount after discount:      900

Customer Name:      Monday Customer
Amount:             1000
Amount after discount:      500
```

[Thursday, 19 ,JUNE, 2014]

Iterator Design Pattern:-

> The definition of iterator pattern in provide a unique way to access elements of an aggregate object by hiding underlying collection type used from a client application.

> In other words, iterator design pattern says that provide a common way to cycle through the elements of collection types like List/Set/Hashtable…etc by hiding the collection type used from a client application.

> In java, java.util.Iterator is used to implement iterator design pattern in an application.

- In the following example, we have SongsOfThe70s and SongsOfThe90s classes where one class has aggregate (collection) object as ArrayList and other class Hashtable.
- We have a client class called DiscJockey, it iterates the collection of both 70s and 90s classes, by without knowing about collection types used.

```
IteratorDesignPattern
        SongInfo.java
        SongIterator.java
        SongsOfThe70s.java
        SongsOfThe90s.java
        DiscJockey.java
        Main.java
public interface SongInfo
{
    String songName;
    int yearReleased;
    public SongInfo(String newSongName, int newYearReleased)
    {
    songName=newSongName;
    yearReleased=newYearReleased;
    }
    public String getSongName()
    {
    return songName;
    }
    public int getYearReleased()
    {
    return yearReleased;
    }
}
import java.util.Iterator;
public interface SongIterator
{
    public Iterator createIterator();
}
import java.util.ArrayList;
import java.util.Iterator;
public class SongsOfThe70s implements SongIterator
{
    ArrayList<SongInfo> bestSongs;
    public SongsOfThe70s()
    {
```

```
    bestSongs=new ArrayList<SongInfo>();
    addSong("Imagine", 1971);
    addSong("AmericanPie", 1971);
    addSong("I will servive", 1979);
    }
    public void addSong(String songName, int yearReleased)
    {
    SongInfo songInfo=new SongInfo(songName, yearReleased);
    bestSongs.add(songInfo);
    }
    public ArrayList<SongInfo> getBestSongs()
    {
    return bestSongs;
    }
    public Iterator createIterator()
    {
    return bestSongs.iterator();
    }
}
import java.util.Hashtable;
import java.util.Iterator;
public class SongsOfThe90s implements SongIterator
{
    Hashtable<Integer,SongInfo> bestSongs= new Hashtable<Integer,SongInfo>();
    int hashKey=0;
    public SongsOfThe90s()
    {
    addSong("LoosingMyReligion", 1991);
    addSong("Creep", 1993);
    addSong("WalkOnTheOcean", 1991);
    }
    public void addSong(String songName, int yearReleased)
    {
    SongInfo songInfo=new SongInfo(songName, yearReleased);
    bestSongs.put(hashKey,songInfo);
    hashKey++
    }
    public Hashtable<Integer,SongInfo> getBestSongs()
    {
    return bestSongs;
    }
    public Iterator createIterator()
    {
    return bestSongs.values().iterator();
    }
}
```

```
import java.util.Iterator;
public class DiscJockey
{
    SongIterator iter70sSongs;
    SongIterator iter90sSongs;
    public DiscJockey(SongIterator newSongs70s, SongIterator newSongs90s)
    {
    iter70sSongs=newSongs70s;
    iter90sSongs=newSongs90s;
    }
    public void showTheSongs()
    {
    Iterator Songs70s=iter70sSongs.createIterator();
    Iterator songs90s=iter90sSongs.createIterator();
    System.out.println("Songs of the 70s\n");
    printTheSongs(songs90s);
    System.out.println("Songs of the 90s\n");
    printTheSongs(songs90s);
    }
    public void pringTheSongs(Iterator iterator)
    {
    while(iterator.hashNext())
    {
    SongInfo songInfo = (SongInfo)iterator.next();
    System.out.println(songInfo.getSongName());
    System.out.println(songInfo.getYearReleased()+"\n");
    }
    }
}
public class Main
{
    public static void main(String args[])
    {
    SongsOfThe70s=new SongsOfThe70s;
    SongsOfThe90s=new SongsOfThe90s;
    DiscJockey dj=new DiscJockey(songs70s,songs90s);
    dj.showTheSongs();
    }
}
```

Output:

```
Imagine
1971
American Pie
1971
I will survive
1979
```

Creep
1993
Losing my Religion
1991

Chain of Responsibility design pattern:-

> This design pattern says that divides a request processing logic into multiple independent objects and makes the objects as child to process a request.
> If complete request process is implemented in single object then that object becomes a heavy weight object. So dividing into multiple independent objects and give a chance to participate a chain of object to participate in a request.
> In Chain of Responsibility , there will be a startup object for the request and from there, there will be a link to a next object in the child
For example, if we put multiple catch blocks to a try block then the multiple catches will form a chain of responsibility.
> When an exception occurs then the control reaches to the first catch block and from there, there be a link the other block.
> Another example of chain of responsibility is filters in web applications. A request will first reach to filter1 and filter1 contains a link to filter2. So the filter chaining is a chain of responsibility pattern.

[Friday, 20 June 2014]

```
ChainOfResponsibility
            Currency.java
            DispenseChain.java
            Dollar50Dispenser.java
            Dollar20Dispenser.java
            Dollar10Dispenser.java
            ATMDispenseChain.java
public class Currency
{
    private int amount;
    public Currency(int amt)
    {
    this.amount=amt;
    }
    public int getAmount()
    {
    return this.amount;
    }
}
public class DispenseChain
{
public void setNextChain(DispenseChain nextChain);
public void dispense(Currency cur);
}
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.chian.dispense(new Currency(remainder));
    }
    }
}
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.chian.dispense(new Currency(remainder));
    }
    }
}
public class  Dollar10Dispenser implements DispenseChain
{
    private DispenseChain chain;
    @Override
    public void setNextChain(DispenseChain 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");
        }
        }
    }
    import java.util.Scanner;
    public class ATMDispenserChain
    {
        private DispenseChain c1;
        public ATMDispenseChain()
        {
        this.c1=new Dollat50Dispenser();
        DispenseChain c2= new Dollar20Dispenser();
        DispenseChain c3= new Dollar10Dispenser();
        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 multiple of 10);
        return;
        }
        //process the request
        atmDispenser.c1.dispense(new Currency(amount));
        }
        }
    }
```

J2EE Design Pattern

Service Locator Design Pattern:-

- In calling services of remote object, the remote service objects are registered in a JNDI registry with some keys.

- A client application will obtain the service object from the registry and then it invokes the services of that object.
- When there is a requirement for multiple types obtaining a service object from registry then every time looking in the registry across network for a service object will degrade the performance of the application.
- In order to reduce the no. of times lookup in the JNDI registry, a service locator design pattern says that, maintain a cache at client side for the registry objects and return the object from local cache to the client, instead of going to the registry.
- The service locator design pattern will improve the performance of the application by reducing the no. of network calls to a JNDI registry.
  For example, in connection pooling technique at real-time, one utility class is created and it gets data source object from JNDI registry and maintains it in cache. The client applications will get the data source object from that local utility class. So this utility class is a service locator.

[Monday, 23 June 2014]

Data Transfer Object/value Object:
When there is a huge amount of values are need to be transferred from one layer to another layer of the project then this DTO pattern is used.

- The values are transferred across layers individually then layer to layer interactions are increased. So, it will affect on performance of an application.
- According to DTO, put the values into an object and then transfer all the values on a single trip from one layer to another layer. It will improve the performance of an application.
- In a project java beans are used as a DTO’s.

Data Access Object (DAO) Design pattern:
DAO is a design pattern used for making a loosely coupled nature between business logic and persistence logic.

- If persistence logic is also inserted there is a tide coupling between both logics.
- With tide coupling we got two drawbacks
  1. The persistence logic will become non-reusable.
  2. Changes to persistence logic need change to business logic also.
- To overcome the above two problems DAO design pattern is used.
- In case of DAO persistence logic is transfer to a separate class, so it becomes reusable.
- If any changes are made on persistence logic will not force to change business logic.
- A class in which we implement persistence logic is called a DAO class.

Front Controller Design Pattern:
In a web application development, if clients are directly allowed to access different files then in each file, there is a need of writing common logic to be executed for each request.

- If the same code is written to multiple files code redundancy occurs.
- In order to avoid the code redundancy, we need to put a centralized access point for all the requests. That centralized point is front controller.
- Front controllers are Servlets in the web applications.

- The benefit of front controller part is, redundancy will be reduced, there is a reusability and easy maintenance of the code.
- A front controller tracks of all records of the clients and applies common logic and then transfer the request to Dispatcher. That Dispatcher will find a suitable file for the request and transfer's the request to appropriate file.
- The Dispatcher is a helper to the front controller to front controller Servlet. We call this helper as application controller.
- In struts framework ActionServlet is a front controller RequestProcessor is application controller.
- The most common logic exist in front controller is
  1. Authentication and Authorization
  2. Session Management
  3. Login

View helper Design Pattern:

This design pattern specifies that, use helper classes to adopt a model data into a view.

- In real time applications, a model component contains business logic and it generates some output/result for a given request. A view component will add some presentation logic for the result of a model, to display/show on browser.
- If a view directly gets the data from the model then some java code is required for reading data produced by model.
- In order to avoid java code from jsp pages we take a helper class like a Java Bean or a tag handler class and we set model's result to those classes.
- In a jsp, instead of writing the java code we write tags and we retrieve the data and then we apply a presentation for the data and send to the browser.
- A Java Bean or tag handler class will act as view helper classes.

  ……………Diagram-2……………….

[Tuesday, 24 June 2014]

Intercepting filter design pattern:

Intercepting filter is a web layer related design pattern like front controller, view helper etc.

- When we want to attach some pre-processing logic to a request and post processing logic to a response, by without disturbing the core functionality of the request. Then intercepting filter pattern is used.
- Intercepting filter design pattern says that, define the pre-processing and post-processing logic into a separate class called filter and define the core request processing functionality in a Servlet and make both objects as independent objects.
- The advantage of this design pattern is additional services can be added to the core functionality or existing services can be removed by without making any changes to core processing functionality.
- Another advantage of this pattern is, we can make the pre and post processing logic as reusable for the request processing functionalities of multiple Servlet
- In intercepting design pattern, a FilterChain take care about executing one or more filter then followed by a target
- A FilterChain is created and filters are added to the chain by FilterManager.

Composite –view Design Pattern:

Composite-view is a view which is constructed by gathering response of multiple view pages.

- In composite view the multiple view pages are called sub-views.

Here a composite-view is constructed by gathering response of multiple subpages like header.jsp, body.jsp, fotter.jsp . These three pages are called sub-views.

- if project contains multiple composite views with similar structure and if any modifications are needed in the structure of the composite views then, each composite view should be modified individually.
- Modifying the structure in the each composite view of a project makes a burden on developer and it is a time taken process.
- In order to overcome the above problems, we got composite-view design pattern.
- In composite view design pattern the pattern says that, define structure of the composite views into a separate page and there attached it to all the composite views.
- The advance of this design pattern is, whenever a layout changes are needed then if once the layout page modified then it affects on all composite view pages.
- For example tiles framework in MVC applications is an example for composite view design pattern.

Dependency Injection/Inversion of Control:

- This design pattern is established a lose coupling between two collaborating objects.
- Loose coupling between objects by following POJI/POJO model and with dependency injection.
- In dependency injection the dependency objects are injected while an external entity like container.
- Injection dependency are three types
  1. Setter injection
  2. Construction injection
  3. Interface injection