JAVA DESIGN PATTERNS

Prepared By
Bhaskara Sai Chitturi

Java Design Patterns

Design patterns are general, reusable solutions to common problems encountered in software design. They represent best practices used by experienced object-oriented software developers. By using design patterns, developers can ensure that their code is modular, maintainable, and scalable.

Design patterns can be categorized into three main types:

Creational Patterns: These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. They abstract the instantiation process, making the system independent of how its objects are created, composed, and represented.

We have different types in creational patterns

i. Factory ii. Singleton iii. Abstract iv. Builder v. Prototype

Structural Patterns: These patterns focus on class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

We have different types of structural patterns

i. Adapter ii. Bridge iii. Composite iv. Decorator v. Facade vi. Flyweight vii. Proxy

Behavioral Patterns: These patterns are concerned with algorithms and the assignment of responsibilities between objects. They help in managing interactions and responsibilities between objects, making the design more flexible and adaptable.

We have different types of behavioral patterns

i. Chain of Responsibility ii. Command iii. Interpreter iv. Iterator v. Mediator vi. Memento vi. Observer viii. State ix. Strategy x. Template Method xi. Visitor

Now let's have a look on few important design patterns which we use regularly in the industry

Factory Design Pattern

Factory design provides an interface for creating objects of super class allowing child to alter the type of objects that will be created, It encapsulates the Object creation logic in a separate method it promotes loose coupling between creator and created object

Problem : Imagine you walk into a mobile store and ask for specific type of mobile phone, The Store has a factory that decides which type of a mobile phone (Android, IOS, Windows) to give you based on your requirement

Solution : Lets create a solution where we have a factory that produces different types of mobiles, here are the steps to implement

Product interface - mobile interface to create different mobiles (blueprint of mobile)

Concrete products - IOS, Android, Windows

Factory - mobile factory that can create different mobiles

Client - who wants to create different mobiles

mobileInterface.java

```java
package com.designpattern;

public interface mobile
{
   public void printMobile();
}
```

Android.java

```
package com.designpattern;

public class Android implements mobile
{
@Override
   public void printMobile()
   {
      System.out.println("Android is here");
   }
}
```

IOS.java

```
package com.designpattern;
public class IOS implements mobile
{
   @Override
   public void printMobile()
      {
      System.out.println("IOS is here");
      }
}
```

windows.java

```
package com.designpattern;

public class Windows implements mobile
{
   @Override
   public void printMobile()
   {
      System.out.println("windows lumia is  here");
   }

}
```

mobileFactory.java

```
package com.designpattern;

public class mobileFactory {
/* mobile factory will create different mobiles based on the id we are giving
while creating object */
   mobile m1;
   public mobileFactory(int id) {
      if(id==1) {
         m1=new Android();
      }
      else if(id==2) {
         m1=new IOS();
      }
      else if(id==3) {
         m1=new Windows();
      }
      else {
         m1=null;
      }
   }
   public mobile getMobile() {
      return m1;
   }
}
```

App.java

```
package com.designpattern;
public class App
{
   public static void main( String[] args )
   {
       /*creating mobile factory object to create 2nd type of mobile*/
      mobileFactory  n1=new mobileFactory(2);
      mobile m1=n1.getMobile();
      m1.printMobile();
   }
}
```

Builder Design Pattern

The Builder design pattern is a way to construct complex objects step by step. It separates the construction of an object its representation allowing the same construction process to create different representation

Problem : Imagine you want to build house, you need various part like the foundation, walls, roof and so on, you can hire a builder who knows how to construct each part step by step until the house is complete

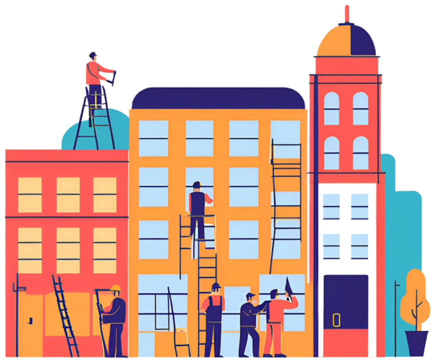
Solution : Steps to implement builder design pattern

House - Final product we want to build

House builder - This class knows how to build the house step by step

House Director - This class directs the building process

Myhouse - client code to create instance of everything



MyHouse.java

```
package demo.sample.designpattern;
public class MyHouse {
  public static void main(String args[]) {
    houseConstructor h1=new houseConstructor();
    houseDirector hd=new houseDirector(h1);
    house newhouse=hd.buildHouse();
    System.out.println(newhouse);
  }
}
```

HouseConstructor.java

```java
package demo.sample.designpattern;

public class houseConstructor {

    private house h;

    public houseConstructor() {
        this.h=new house();
    }
    public void buildhouseFoundation() {
        h.setHouseFoundationl("building foundation");
    }
    public void buildhousewalls() {
        h.setHouseWalls("building walls");
    }
    public void buildhouseroof() {
        h.setHouseRoof("creating roof");
    }
    public house getbuildhouse() {
        return h;
    }
}
```

HouseDirector.java

```java
package demo.sample.designpattern;

public class houseDirector {

    private houseConstructor hc;

    public houseDirector(houseConstructor h1) {
        this.hc=h1;
    }
    public house buildHouse(){
        hc.buildhouseFoundation();
        hc.buildhouseroof();
        hc.buildhousewalls();
        return hc.getbuildhouse();
    }
}
```

House.java

```
package demo.sample.designpattern;

public class house {
private String houseFoundationl;
private String houseWalls;
private String houseRoof;

public void setHouseFoundationl(String houseFoundationl) {
   this.houseFoundationl = houseFoundationl;
}
public void setHouseWalls(String houseWalls) {
   this.houseWalls = houseWalls;
}
public void setHouseRoof(String houseRoof) {
   this.houseRoof = houseRoof;
}
@Override
public String toString() {
   return "house [houseFoundationl=" + houseFoundationl + ", houseWalls=" +
houseWalls + ", houseRoof=" + houseRoof
         + "]";
}
}
```

Adapter Design pattern

The Adapter design pattern allows two incompatible interfaces to work together, it acts as a bridge between two objects enabling them to communicate and work together.

Problem : Imagine we have a new assessment system that requires a pen to write on a digital tablet. However we have an old pen that writes on paper we will use an adapter to make the old pen with new assessment system

Solution : Steps to implement adapter design pattern

Digitalpen.java - target interface to use

Paper pen - adaptee (used to build method for digital pen)

Penadapter - adapter (to create interface to convert)

Assessment - client

School - interface for the client

Digitalpen.java

```java
package demo.sample.designpattern;

public interface Digitalpen {
    public void write(String str);
}
```

paperpen.java

```java
package demo.sample.designpattern;

public class paperpen {

    public void mark(String str) {
        System.out.println(str);
    }

}
```

penAdapter.java

```
package demo.sample.designpattern;
public class penAdapter implements digtalpen {

    paperpen bp=new paperpen();

    @Override
    public void write(String str) {
        bp.mark(str);
    }
}
```

assessment.java

```
package demo.sample.designpattern;

public class assessment {

    private digitalpen p;

    public digitalpen getP() {
        return p;
    }

    public void setP(digitalpen p) {
        this.p = p;
    }

    public void writeAssessment(String str) {
        p.write(str);
    }
}
```

school.java

```
package demo.sample.designpattern;
public class school {
    public static void main(String args[]) {
    digitalpen bp=new penAdapter();
    assessment a1=new assessment();
    a1.setP(bp);
    a1.writeAssessment("Im writing assessment with pen");
    }
}
```

Composite Object design pattern

The composite design pattern allows you to compose objects into tree structures to represent part whole hierarchies, here we have two types of objects

1. Composite objects

    the object which has a children

2. Leaf objects

    the end node in the tree structure

Important rule : If we perform any operation on leaf node same operation should perform on composite object

Problem : Imagine we have a computer shop where the shopkeeper can add various products (like CPU, RAM, Monitor, etc.) to a computer. The shopkeeper should be able to treat individual products and groups of products uniformly. For example, adding a CPU or adding a Motherboard (which itself contains multiple products) should be handled in the same way.

Solutions : Steps to implement composite design pattern

Component: The interface for both individual and composite objects.

Leaf: Represents individual objects (e.g., Mouse, Keyboard).

Composite: Represents composite objects that can contain other components (e.g., Peripherals)

Shop: Client code that uses the composite pattern to manage and display the prices of computer parts.

Leaf.java

```java
class leaf implements component{

   int price;
   String name;

   public leaf(int price, String name) {
      super();
      this.price = price;
      this.name = name;
   }

   @Override
   public void showPrice() {
      System.out.println(name+" : "+price);
   }

}
```

composite.java

```java
class composite implements component{

   List<component> components = new ArrayList<>();
   String name;
   public composite(String name) {
      super();
      this.name = name;
   }
   public void addComponent(component c) {
      components.add(c);
   }
   @Override
   public void showPrice() {
      // TODO Auto-generated method stub
      System.out.println(name);
      for(component c: components) {
         c.showPrice();
      }
   }
}
```

shop.java

```java
package demo.sample.designpattern;
public class shop {

    public static void main(String[] args) {

        component mouse=new leaf(4000,"lenovo mouse");
        component keyboard=new leaf(5000,"lenovo keyboard");
        composite peri=new composite("peripharls");
        peri.addComponent(mouse);
        peri.addComponent(keyboard);
        peri.showPrice();

    }

}
```

Component.java

```java
package demo.sample.designpattern;
import java.util.*;

interface component{
    void showPrice();
}
```

Prototype design pattern

The prototype is used to create new objects copying an existing object known as prototype. Instead of creating new object from scratch you can clone the prototype, this pattern is useful when the creation of an object is complex or expensive

Problem : Imagine you have a book and you want to create multiple copies of this book

Solution : steps to implement prototype design pattern

Prototype pattern - an interface that declares method for cloning itself

Concrete pattern - a class that implements cloning method

Client - a class that uses the prototype to create new objects

Bookprototype.java

```
interface bookprototype{
   bookprototype cloneBook();

   void showBook();

}
```

Book.java

```
class book implements bookprototype{

    String bookName;
    String bookAuthor;

    public book(String bookName, String bookAuthor) {
        super();
        this.bookName = bookName;
        this.bookAuthor = bookAuthor;
    }

    public void showBook() {
        System.out.println(bookName+" by "+bookAuthor);
    }

    @Override
    public bookprototype cloneBook() {
      return new book (this.bookName,this.bookAuthor);
    }

}
```

library.java

```
package demo.sample.designpattern;

public class library {

    public static void main(String args[])
    {
        book b1=new book("Keerthi keeritalu","sulochana rani");

        bookprototype newbook1=b1.cloneBook();
        newbook1.showBook();
    }
}
```

Observer design pattern

The observer design pattern is used to create a one to many dependency between objects so that when one object changes state all the dependencies are notified and updated automatically, its often to implement distributed event handling systems

Problem : Imagine you subscribe to a Youtube channel, whenever the channel uploads a new video all subscribers get a notification about new video, the youtube channel notifies all its subscribe when there is an update

Solution : steps to implement

Channel - to upload a video can users subscribe to the channel

Subscriber - subscriber can get notified from here

Youtube - main class to create channel,subscribers

Subscriber.java

```
package demo.sample.designpattern;

public class subscriber {

    private String name;
    private channel ch=new channel();

    public subscriber(String name) {
        super();
        this.name = name;
    }

    public void update() {
        System.out.println("Hey "+name+" video uploaded with : "+ch.title);
    }

    public void subscribechannel(channel c1) {
        ch=c1;
    }

}
```

Subscriber.java

```java
package demo.sample.designpattern;

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

public class channel {

    String title;
    private List<subscriber> subs=new ArrayList<>();
    public void subscribe(subscriber c1) {
        subs.add(c1);
    }
    public void unsubscribe(subscriber c1) {
        subs.remove(c1);
    }
    public void notifysubs() {
        for(subscriber c:subs)
            c.update();
    }
    public void upload(String title) {
        this.title=title;
        notifysubs();
    }

}
```

youtube .java

```java
package demo.sample.designpattern;

public class youtube {

    public static void main(String args[]) {

        subscriber s1=new subscriber("bhaskara");
        channel codepen=new channel();
        codepen.subscribe(s1);
        s1.subscribechannel(codepen);
        codepen.upload("Java Spring Boot class");
    }
}
```