Karrasankar158@gmail.com

Design Pattern

Core/GOF Design Patterns

1. Singleton Design Pattern
2. Factory Design Pattern
3. Factory Method Design Pattern
4. Abstract Factory Design Pattern
5. Template Method Design Pattern
6. Adaptor Design Pattern
7. Command Design Pattern
8. Decorator Design Pattern
9. FlyWeight Design Pattern

Presentation Tier Patterns

1. Intercepting Filter Design Pattern
2. Front Controller Design Pattern
3. Application Controller Design Pattern
4. Context Object Design Pattern
5. Value Object Design Pattern

Business & Integration Tier Design Patterns

1. DAO (Data Access Object) Design Pattern
2. Business Delegate Design Pattern
3. Business Object Design Pattern
4. Service Locator Design Pattern
5. Session Facade Design Pattern

Core/GOF Design Patterns

1. Singleton Design Pattern

Singleton Design Pattern

If we create a class as singleton, an application allows only one instance of that class. Generally we create a class as singleton when we want global point of access to that instance.

1. **Declare the constructor of the class as private :**
   Other classes in the application cannot create the object of the class directly(stops allowing more instances).

2. **Declare a static method :**
   The methods of the same class can call the constructor to create objects, so in this method I can write the code to check and return only one object of the class.
   But if this method is a member method, to call it we need an object of the class(). So to call this method without the object, declare this object as static method.

3. **Declare a static member of the same class-type in the class :**
   In the above static method, we need to write the code for

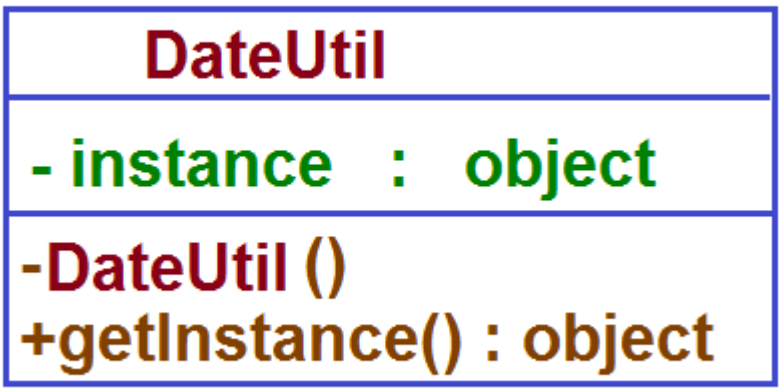
returning only one instance of the class.

How do you track whether an object for that class already exists ?

When you will create a first object you will assign it to a member variable. So in the next call to the method, you just return the same object which you stored in the member variable.

But member variables cannot be used in a static method, so you need to declare that variable as static variable to hold the reference of the same class.

The UML representation of the singleton :



Points to remember :

- DateUtil() is a private constructor
- instance is a static variable declared with in the class
- getInstance() is a static method, acts as a factory method to create the object of the class.

```
package com.sinleton;

public class DateUtil {
```

```
 //declare static member of the same class-type in the
class
 private static DateUtil instance;

 //constructor is declared as private
 private DateUtil(){ }

 //declare a static method to create only one instance
 //(static factory method)
 public static DateUtil getInstance(){
  if(instance==null){
        instance=new DateUtil();
  }
 return instance;
 }//getInstance

}
```

we can write the above piece of code in various other ways and there are many ways to implement it.

1. Eager Initialization :

```
package com.sinleton;

public class DateUtil {

 //declare static member of the same class-type in the
class

 private static DateUtil instance=new DateUtil();
 //instantiating the instance attribute when the class
is loaded

 //constructor is declared as private
 private DateUtil(){ }

 //declare a static method to create only one instance
 //(static factory method)
 public static DateUtil getInstance(){
   return instance;
 }

}
```

2. Static block Initialization :

```
package com.sinleton;

public class DateUtil {

 //declare static member of the same as class-type
 private static DateUtil instance;


//static block execute once when the class is loaded
 static{
       instance=new DateUtil();
 }


 //constructor is declared as private
 private DateUtil(){ }

 //declare a static method to create only one instance
 //(static factory method)
 public static DateUtil getInstance(){
   return instance;
 }

}
```

But the problem with above code is even you don't need the object also it will be instantiated.

3. Lazy Initialization :

- In most of the cases it is recommended to delay the instantiation process until the object is needed.
- To achieve this we can delay the creational process till the first call the getInstance() method.
- But the problem with this is in a multi-threaded environment when more than one thread are executing at the same time, it might end-up in creating more than instances of the class.
- To avoid this we can even declare that method as synchronized.

```
private static DateUtil instance;

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

Instead of making the whole method as synchronized, it is enough to enclose only the condition check in synchronized block.

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

- Again we have a problem with the above piece of code, after the first call to the getInstance(), in the next calls to the method will check for instance==null check.
- While doing this check, it acquires the lock to verify the condition which is not required.
- Acquiring and releasing locks are quiet costly and we should avoid as much we can. To fix this we can have double level check for the condition as shown below.

```
public class DateUtil {
private static DateUtil instance;

private DateUtil(){ }

public static DateUtil getInstance(){
if(instance==null){
 synchronized(DateUtil.class){
       //double check
 if(instance==null){
  instance=new DateUtil();
```

```
 }
}
}
return instance;
}

}
```

It is recommended to declare the static member instance as volatile to avoid problems in a multi-threaded environment.

```
public class DateUtil {
private static volatile DateUtil instance;

private DateUtil(){ }

public static DateUtil getInstance(){
if(instance==null){
 synchronized(DateUtil.class){
       //double check
 if(instance==null){
  instance=new DateUtil();
 }
}
}
return instance;
}

}
```

It seems like with the above way we understand the best possible way of creating a singleton class.

Do you still see any drawbacks in the above piece of code ?

Yes, when we serialize and de-serialize a singleton class, the de-serialize process will creates as many number of objects for singleton class which avoids the rules of singleton.

```
import java.io.Serializable;

public class DateUtil implements Serializable{
private static DateUtil instance;
```

```
private DateUtil(){ }

public static DateUtil getInstance(){
if(instance==null){
 synchronized(DateUtil.class){
       //double check
 if(instance==null){
  instance=new DateUtil();
 }
}
}
return instance;
}

}
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SingletonTest{
public static void main(String[] args) throws
FileNotFoundException, IOException,
ClassNotFoundException {

DateUtil d1=DateUtil.getInstance();

ObjectOutputStream oos=new ObjectOutputStream(new
FileOutputStream(new File("e:\\dateUtil.ser")));
oos.writeObject(d1);

ObjectInputStream ois=new ObjectInputStream(new
FileInputStream(new File("e:\\dateUtil.ser")));
DateUtil d2=(DateUtil)ois.readObject();

System.out.println("d1==d2 : ? "+(d1==d2));
 //the above statement returns false

}
}
```

So, how to avoid creating more than one objects of the singleton class even we serialize and de-serialize also.

That's where we need to write **readResolve()** method as part of singleton class.
The de-serialization process will call readResolve() on a class to read the byte stream to build the object.
If we write this method and can return the same instance of the class, we can avoid creating more than one object even in case of serialization as well.

```
package design;

import java.io.Serializable;

public class DateUtil implements Serializable{

 //declare static member of the same class-type in the
class
 private static volatile DateUtil instance;

 //constructor is declared as private
 private DateUtil(){ }

 //declare a static method to create only one instance
(static factory method)
 public static DateUtil getInstance(){

 if(instance==null){
  synchronized (DateUtil.class) {
   if(instance==null){    //double-check
    instance=new DateUtil();
   }
  }
 }
 return instance;
 }//getInstance

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

```
}
package design;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SingletonTest{
public static void main(String[] args) throws
FileNotFoundException, IOException,
ClassNotFoundException {

DateUtil d1=DateUtil.getInstance();

ObjectOutputStream oos=new ObjectOutputStream(new
FileOutputStream(new File("e:\\dateUtil.ser")));
oos.writeObject(d1);

ObjectInputStream ois=new ObjectInputStream(new
FileInputStream(new File("e:\\dateUtil.ser")));
DateUtil d2=(DateUtil)ois.readObject();

System.out.println("d1==d2 : ? "+(d1==d2));
 //the above statement returns true

}
}
```

Override clone() method and throw CloneNotSupportedExcepton :

- In order not to allow singleton class to be cloneable from created objects, it is even recommended to implement your class from Cloneable interface and override clone() method.
  Inside this method we should throw CloneNotSupportedException to avoid cloning of the object.

- If you observe carefully clone() method is protected method in object class which cannot be visible outside the class unless we override.
- The why do i need to implement from Cloneable and should throw exception in clone() method.

- For e.g. if someone might write a class implementing Cloneable interface and calling super.clone() method in it.
- If your singleton class extends from the other class, as the clone() method has been overridden in the base class you can use that method in cloning your object.

- So to avoid such kind of instances, it is always said to be recommended to override the clone() method and throws exceptions.

```
package design;

import java.io.Serializable;

public class DateUtil implements
Serializable,Cloneable{

 //declare static member of the same class-type in the
class
 private static volatile DateUtil instance;

 //constructor is declared as private
 private DateUtil(){ }

 //declare a static method to create only one instance
(static factory method)
 public static DateUtil getInstance(){

 if(instance==null){
  synchronized (DateUtil.class) {
   if(instance==null){    //double-check
    instance=new DateUtil();
   }
  }
```

```
 }
 return instance;
 }//getInstance

protected Object readResolve(){
 return instance;
}

@Override
protected Object clone() throws
CloneNotSupportedException{
      throw new CloneNotSupportedException();
      //super.clone();
      //return instance;
}

}
package design;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SingletonTest{
public static void main(String[] args) throws
FileNotFoundException, IOException,
ClassNotFoundException, CloneNotSupportedException {

DateUtil d1=DateUtil.getInstance();

ObjectOutputStream oos=new ObjectOutputStream(new
FileOutputStream(new File("e:\\dateUtil.ser")));
oos.writeObject(d1);

ObjectInputStream ois=new ObjectInputStream(new
FileInputStream(new File("e:\\dateUtil.ser")));
DateUtil d2=(DateUtil)ois.readObject();

System.out.println("d1==d2 : ? "+(d1==d2));
 //the above statement returns true
```

```
DateUtil d3=(DateUtil) d1.clone();
System.out.println("d1==d3 : ? "+(d1==d3));
 //the above statement returns true
}
}
```

Factory Design Pattern

- Not all objects in the java can be created out of new operator. Few objects may be created out of view; few may have to be created by calling a static factory method on the class(singleton) others may have to be created by passing other object as reference while creating etc.
- If I want a car, do I need to know how to create a car ? If I try to manufacture my own car, it takes ages to complete it. Instead I can go to a factory that is proficient in manufacturing car to get a car.
- The main advantage of going for factories is it abstracts the creational process of other class. For e.g let's take an example of Jdbc, Connection is an interface in jdbc api, respective vendor driver will have the implementation of Connection interface. Oracle driver will provide an implementation class for Connection interface and MS Sql server will provides its own implementation for Connection interface.
- If we try to create an Object of Connection interface, can we find what is the implementation class for the Connection interface provided by that vendor and can instantiate ? Practically if will not be possible for us to remember various vendor provided implementation class names.

- So, jdbc has provided an factory class called DriverManager, instead of we finding the implementation class for Connection interface, if we go to DriverManager and call a method getConnection(), he will takes care of finding the implementation class based on url we provided and instantiates the appropriate vendor implementation of Connection interface.
- In the above case we are subtracted from creation on Connection implementation object rather if we just go to DriverManager and call the getConnection() method he takes care of creating the implementation of connection object, this is the main advantage of going for factories.
- Every factory class has a method; it contains the logic for creating the object of another class, so it is called factory method. Generally these methods will be declared as static to let you call without creating the object of factory.

An UML representation of Factory pattern :

```
package design;

public interface Pizza{
      void prepare();
      void bake();
      void cut();
}
```

In the shop we sell various types of Pizza's like CheesePizza and ChickenPizza etc., So, we have several sub-classes from Pizza class to sell different types of pizza.

```
package design;
```

```
public class CheesePizza implements Pizza{

@Override
public void prepare() {
 System.out.println("preparing cheese pizza...");
}

@Override
public void bake() {
 System.out.println("baking cheese pizza...");
}

@Override
public void cut() {
 System.out.println("cutting cheese pizza...");
}

}
package design;

public class ChickenPizza implements Pizza{

@Override
public void prepare() {
 System.out.println("prepare chicken pizza...");
}

@Override
public void bake() {
      System.out.println("baking chicken pizza...");
}

@Override
public void cut() {
      System.out.println("cutting chicken pizza...");
}

}
```

Now in the PizzaStore class if someone orders a Pizza we need to check which type of pizza and needs to create it.

```
package design;

public class PizzaStore{
public Pizza orderPizza(String type){
```

```
 Pizza p=null;
 if(type.equals("cheese")){
  p=new CheesePizza();
 }else if(type.equals("chicken")){
  p=new ChickenPizza();
 }
 p.prepare();
 p.bake();
 p.cut();

 return p;

}
}
```

- Now demand for adding more pizza types came and we add TomatoPizza, CornPizza. Again we need to modify the code inside the orderPizza() method of PizzaStore to add or remove more pizza types, which involves the modify the code in PizzaStore class.

- May be some other classes in our application also want create a pizza, so the above piece of logic has to be written in another class. Apart from this if PizzaStore has to sell pizza it needs to know the implementation class of all Pizza types and how to instantiate them as well. Is there any better way handling this ?

- Yes, that's where factory comes into picture. Instead of writing the logic for creating various types of pizza's in PizzaStore we can separate it and write in a factory class called PizzaFactory. Now if PizzaStore wants a pizza, it doesn't need to know which is the implementation class for creating a pizza rather it can call a factory method on the PizzaFactory class to get a pizza.

```
package design;

public class PizzaFactory {
```

```
public static Pizza cteratePizza(String type) {
 Pizza p=null;

 if(type.equals("cheese")){
  p=new CheesePizza();
 }else if(type.equals("chicken")){
  p=new ChickenPizza();
 }

 return p;
}

}
```

Modified PizzaStore

```
package design;

public class PizzaStore{
public Pizza orderPizza(String type){
 Pizza p=null;

 p=PizzaFactory.cteratePizza(type);
  //call factory method to get the pizza
 p.prepare();
 p.bake();
 p.cut();

 return p;

}
}
```

In the above we are able to abstract to abstract the creation of pizza from PizzaStore, without knowing how to create a pizza, PizzaStote can sell the pizza easily.

Factory Method Design Pattern

- Factory method is used for creating the object for family of related classes with in the hierarchy.

- Let's consider for example Maruthi is manufacturing car's. It manufactures several types of car's.
- It has several manufacturing units in which the cars are being manufactured.

```
package design;

public abstract class Car {
      protected int engineNo;
      protected String color;

      //setters and getters
      //toString()

 //every car has its own driving style
 public abstract void drive();
}
package design;

public class AltoCar extends Car{

@Override
public void drive() {
 System.out.println("driving alto car");
}

}
package design;

public class WagnorCar extends Car{

@Override
public void drive() {
 System.out.println("driving wagnor car");
}

}
```

- These car's will be manufactured across various maruthi manufacturing units across india. Every manufacturing unit may not manufacture all the models.

- For e.g. GujaratMaruthi manufacturing unit only manufactures alto cars and PuneMaruthi manufacturing unit only manufactures Waganor and Swift.

```
package design;

public class PuneWorkshop{

public Car createCar(String type){
      Car car=null;

 if(type.equals("wagnor")){
       car=new WagnorCar();
 }

  //creating and assembly
 System.out.println("assemble engine");

return car;
}

}
package design;

public class GujratWorkshop{

public Car createCar(String type){
      Car car=null;

 if(type.equals("alto")){
       car=new AltoCar();
 }

  //creating and assembly
 System.out.println("attach wheels");
 System.out.println("fit seats");
 System.out.println("assemble engine");

return car;
}

}
```

- The problem in the above example is after creating the car, every manufacturing unit is using their own mechanism of assembling the car.
- Few are attaching wheels, seats and engine. Few others are forgetting to attach wheels and seats and delivering the car with engine.
- So in the above we want to bring up some quality control and standardization of assembling a car.
  Create a MaruthiWorkshop which takes care of standardizing the process of assembling the car.

```
package design;

public abstract class MaruthiWorkshop{

public Car assembly(String type){
      Car car=null;

  car=createCar(type);

  //creating and assembly
 System.out.println("attach wheels");
 System.out.println("fit seals");
 System.out.println("assemble engine");

return car;
}

protected abstract Car createCar(String type);

}
```

- The advantage with the above approach is every Workshop will take care of manufacturing the cars. But the complete control of assembly and delivering the car will be done across the workshops in the same manner as it is controlled by the super class.

- And now our factory method createCar in the above class can create objects of only the sub-classes of car class only(in the hierarchy).

  Even we add more car's in future also the assembling and delivering of those car's will not get affected.

  More over the code deals with super type, so it can work with any user-defined ConcreateCar types.

Abstract Factory Design Pattern

- Abstract Factory can be treated as a super factory or a factory of factories.
  Using factory design pattern we abstract the creation process of another class.
  Using the Abstract factory pattern we abstract the creation of family of classes.

- Let's understand it by taking an example. We have several Dao's classes to persist the data. Like EmpDao, DeptDao etc., in-turn these Dao's can persist the data into a Database or into an XML file.
  So we have now the Dao's as DBEmpDao, DBDeptDao and XMLEMPDao, XMLDeptDao.

- To create the objects of Dao's of related types we have create 2 different factories.
  DBDaoFactory and XMLDaoFactory, these factories takes care of creating the Dao's of their type.

```
package design;
```

```
public abstract class Dao{

 public abstract void save();

}
package design;

public class XmlEmpDao extends Dao{

@Override
public void save() {
 System.out.println("Employee has been written to XML
file");
}

}
package design;

public class DBEmpDao extends Dao{

@Override
public void save() {
 System.out.println("Employee has been written to
table");
}

}
package design;

public class XMLDeptDao extends Dao{

@Override
public void save() {
 System.out.println("Department has been written to XML
File");
}

}
package design;

public class DBDeptDao extends Dao{

@Override
public void save() {
```

```
 System.out.println("Department has been written to
table");
}

}
```

Now we have XmlDaoFactory and DBDaoFactory which creates Dao objects of their type.

```
package design;

public abstract class DaoFactory {
      public abstract Dao createDao(String string) ;
}
package design;

public class XMLDaoFactory extends DaoFactory{

public Dao createDao(String type){
 Dao dao=null;
 if(type.equals("emp")){
  dao=new XmlEmpDao();
 }else if(type.equals("dept")){
       dao=new XMLDeptDao();
 }
 return dao;
}

}
package design;

public class DBDaoFactory extends DaoFactory{

public Dao createDao(String type){
 Dao dao=null;
 if(type.equals("emp")){
  dao=new DBEmpDao();
 }else if(type.equals("dept")){
       dao=new DBDeptDao();
 }
 return dao;
}

}
```

```
package design;

public class DaoMaker {

public static DaoFactory make(String factoryType){
 DaoFactory df=null;
 if(factoryType.equals("xml")){
   df=new XMLDaoFactory();
 }else if(factoryType.equals("db")){
   df=new DBDaoFactory();
 }
 return df;
}

}
```

- now the DaoMaker will takes care of instantiating the appropriate factory to work with family of Dao's.
- For any application it is important to use all Dao's that belongs to same type.
- So our DaoMaker enforces this rule by encouraging you to get one type of factory from which you can Dao's.

For example if an application wants a Dao object it has to do the following.

```
package design;

public class AbstractFactoryTest {

public static void main(String[] args){
 DaoFactory daoFactory=null;
 Dao dao=null;

 daoFactory=DaoMaker.make("xml");
 dao=daoFactory.createDao("emp");
 dao.save();
}
```

```
}
```

- In the above example the client is using "xml" family of Dao's to perform operations.
- If we want to switch from "xml" to "db" he don't need to make lot of modifications as he is dealing with DaoFactory abstract class, he can easily switch between any of the implementation of DaoFactory by calling **DaoMaker.make("db")**.

- Hence Abstract Factory pattern helps you in enforcing your application to use related classes across the application.

The UML Diagram for the AbstractFactory :

Difference between AbstractFactory and FactoryMethod :

- Abstract Factory pattern delegates the responsibility of object instantiation to another object via composition.
- Factory Method pattern uses inheritance and relies on subclasses to handle the desired object instantiation.

Template Method Design Pattern

- Template method design pattern is a behavioral pattern of the Gang Of Four(GOF) design patterns catalog.
- In this pattern we have a base template method; it defines an algorithm with some abstract steps.
- These steps have to be implemented by sub-classes.

- For example I have a class called DataRenderer this class is responsible to rendering the data to output console.
- But to render the data first we need to read and process it.
- So we have a methods like readData() and processData().
- But these methods will be declared as abstract as there are multiple sources from which you can read the data and multiple ways we can process it.
- But to render the data you need to read and process it, so the algorithm for render() is fixed which is read and process but how to read, from where to read and how to process is left to the sub-classes to handle.

```
package design;

public abstract class DataRenderer{

 //algorithm is fixed
 public void render(){
       String data=null;
       String pData=null;

       data=readData();
       pData=processData(data);

       System.out.println(pData);
 }

 public abstract String readData();
 public abstract String processData(String data);
}
package design;

public class XMLDataRenderer extends DataRenderer {

 @Override
 public String readData() {
  return "xml data";
 }

 @Override
```

```
 public String processData(String data) {
  return "processed "+data;
 }

}
package design;

public class TextDataRenderer extends DataRenderer {

 @Override
 public String readData() {
  return "text data";
 }

 @Override
 public String processData(String data) {
  return "processed "+data;
 }

}
```

- In the above example the responsibilities of reading and processing has been left to sub-classes.
- render() method remains same calling the methods of your sub-classes.

Now if a client wants to reader the data, he/she has to create the object of XMLDataRender or TextDataRender and has to call the render() method which delegates the call to readData() and processData() as per the algorithm to render it.

```
package design;

public class TemplateMethodTest {

public static void main(String[] args){
 DataRenderer renderer=new XMLDataRenderer();
 renderer.render();
}

}
```

The Template method we declare in the base class cannot be overridden as the algorithm is fixed, and the sub-classes should not change the behavior of it we need to declare it as final.

public final void render()

The UML representation of the pattern :

Key points :

- The template method in the super class calls the methods of the sub-classes, instead the sub-classes calls the template method of the super class.
- Template methods are techniques for code reuse because with this you can standardize the algorithm and defer the specific implementations to the sub-classes. Again the sub-classes do need to re-write the same algorithm.

Adaptor Design Pattern

- Adaptor pattern helps you in converting interfaces of a class into another interface clients expects.
- We can apply this to our real world examples also. The best example for this is an AC power adapter.
- When we buy some imported items from other countries, they come with sockets of different model, which cannot be plugged into our plugs.
- To make them compatible, we attach converts in between so that they can be plugged-in.

- Let's take an example to understand better. We have an algorithm; it takes the name of the city and returns the temperature in the city.
- But the user inputs the zipCode of the location he lives rather than the city name.
- So to find the temperature our algorithm will not accept zipCode rather expects the city name.

In this case we can write an Adaptor class with takes the zipCode and maps to a city and passes this city name as input in finding the temperature at that location.

```
package design;

public interface IWeatherFinder {

 public int find(String city);

}
package design;

public class WeatherFinder implements IWeatherFinder{

 @Override
 public int find(String city) {
  return 22;
 }

}
```

The above class contains the logic for finding the temperature in a city.
Now my client instead of having name of the city, he has zipCode for which we need to find the temperature.

In this case the interface the client expected is different from the actual algorithm which has been designed.
To fix this problem we need to write one adaptor class.

```
package design;

public class WeatherAdaptor {

 public int findTemperature(int zipCode) {

       //maps this zipCode to city name
       String city=null;

       //actually looks into a source (db or file)
       if(zipCode==506001){
            city="hyderabad";
       }

   IWeatherFinder finder=new WeatherFinder();
   return finder.find(city);
 }

}
```

Now the client can find the temperature by passing the zipCode,
where the zipCode will be mapped to the city by adaptor and talks to respective class to find the temperature.

```
package design;

public class WeatherWidget {

 public void showTemparature(int zipCode){
  WeatherAdaptor wa=new WeatherAdaptor();
  System.out.println(wa.findTemperature(zipCode));
 }

}
```

Even adaptor pattern makes in-compatible interfaces compatible,
it introduces one more level of in-direction the code which makes it complicated to understand and sometimes tough to debug.

Command Design Pattern

- Command design pattern is the one Behavioral Design pattern from Gang Of Four Design Patterns.
- It is used to encapsulate a request as an Object and pass to an invoker.
- Invoker doesn't know how to service the request but uses encapsulated command object to perform the action.

Typically in a command design pattern there are five actors involved there are as follows :

1. **Command** : It is an interface with execute method. It acts as a contract.
2. **Client** : Client instantiates an concrete command object and associates it with a receiver.
3. **Invoker** : He instructs the command to perform an action.
4. **Concrete Command** : Associates a binding between receiver and action.
5. **Receiver** : It is the object that knows the actual steps to perform the action.

- Let's consider an example to understand it. For example PowerOn and PowerOff are the commands, to turn on/off the Television. These commands are received by the Television.

- You will issue these commands using the remote controller who acts as an Invoker. Client is the person who uses this remote Control.

- The advantage of this is invoker is decoupled by the action performed by the receiver.
- The Invoker has no knowledge of the receiver.The Invoker issues a command where in the command performs the action on a receiver.
- The Invoker doesn't know the details of the action being performed. So, changes to the receiver action don't affect the invoker's action.

Here is the code snippet explaining the same, The interface acts as a core contract for commands.

```
package design;

public interface Command {
     public void execute();
}
```

Implementing this interface we have 2 commands, PowerOn and PowerOff .

```
package design;

  //command, encapsulated with receiver to perform
action
public class PowerOn implements Command{

       //receiver on who command performs the action
      private Television television;

      public PowerOn(Television television) {
            this.television=television;
      }
```

```
  @Override
  public void execute() {
    television.on();
  }

}
```

```
package design;

  //command, encapsulated with receiver to perform
action
public class PowerOff implements Command{

       //receiver on who command performs the action
      private Television television;

      public PowerOff(Television television) {
            this.television=television;
      }

  @Override
  public void execute() {
    television.off();
  }

}
```

In the above Television is the receiver on who the command is issuing the action.

```
package design;

 //receiver (he knows how to perform the action)
public class Television {

  public void on() {
      System.out.println("Television switcher on ...");
  }

  public void off() {
      System.out.println("Television turning off ...");
  }
```

```
}
```

Now remote control is the invoker who can issue several commands and command triggers an action on the receiver who knows how to to handle that action.

```
package design;

public class RemoteControl {

	private Command command;

	public RemoteControl(Command command){
		this.command=command;
	}

	public void pressButton(){
		command.execute();
	}

}
```

(OR)

```
package design;

public class RemoteControl {

	private Command command;

	public RemoteControl(Command command){
		this.command=command;
	}

	public RemoteControl() { }

	public void pressButton(){
		command.execute();
	}

	public void setCommand(PowerOn onCmd) {
		command=onCmd;
	}
```

```
	public void setCommand(PowerOff offCmd) {
		command=offCmd;
	}

}
```

Finally Client is the Person who issues a command on the Invoker.

```
package design;

public class Person {

 public static void main(String[] args){

	 //Invoker
	RemoteControl control=new RemoteControl();

	//receiver
	Television television=new Television();


	//command setup with receiver
	PowerOn onCmd=new PowerOn(television);
	control.setCommand(onCmd);
	control.pressButton();

	PowerOff offCmd=new PowerOff(television);
	control.setCommand(offCmd);
	control.pressButton();
 }

}
```

output :

```
Television switcher on ...
Television turning off ...
```

The UML representation of Command design pattern :

Decorator Design Pattern

- Decorator is one of the widely used structural patterns.
- It adds dynamically the functionality to an object at runtime without affecting the other objects.
- It adds additional responsibilities to an object by wrapping it. So it is also called as wrapper.

- For example, Pizza is the object which is object which is already baked and consider as a base object.
- As requested by the customer we might need to add some additional toppings on it like cheese or tomato on it, important is only for the object customer requested without effecting other Pizza's.
- You can consider these toppings as additional responsibilities added by the decorator.

Key Points :

- Add and remove additional functionalities or responsibilities to an object dynamically at runtime, without affecting the other objects.
- Usually these decorators are designed on component interface, so you can select the object that has to be decorated at runtime.
- Sometime adding an additional responsibility to a class may not be possible by sub-classing it. Only available way of achieving it is using decorator.

Following are the participants of the Decorator design pattern :

1. **Component** : Pizza is the base interface.
2. **Concrete component** : NormalPizza is the concrete implementation of the Pizza interface.
3. **Decorator** : is the abstract class who holds the reference of the component and also implements from the component interface.
4. **Concrete Decorator** : Who implements from the abstract decorator and add additional responsibilities to the Concrete component.

```
package design;

public interface Pizza {
   public void bake();
}
```

Now implementing the above interface we have a concrete class NormalPizza.

```
package design;

public class NormalPizza implements Pizza{

      @Override
      public void bake() {
        System.out.println("Booking normal pizza ...");

      }
}
```

Now we want to add some more toppings on the normal pizza. We don't want to modify all the pizza's rather one instance of the normal pizza we want to decorate.

So create Abstract Decoder implementing from Pizza and has reference of Pizza to add toppings.

```
package design;

public class PizzaDecorator implements Pizza {
      private Pizza pizza;
```

```
	public PizzaDecorator(Pizza pizza) {
		this.pizza=pizza;
	}

	@Override
	public void bake() {
		pizza.bake();

	}
}
```

Now create an concrete decorator extends from Abstract Decorator to add Tomato as a toppings .

```
package design;

public class TomatoPizzaDecorator extends
PizzaDecorator {

	public TomatoPizzaDecorator(Pizza pizza) {
		super(pizza);
	}

	public void bake(){
		super.bake();
		addTomatoTapping();
	}

	public void addTomatoTapping() {
		System.out.println("Tomato topping added
...");
	}
}
```

If a PizzaShop wants a TomatoPizza rather modifying the Pizza, he can use TomatoPizzaDecorator as Pizza which decorates the Normal Pizza and serves.

```
package design;

public class Shop {

public static void main(String[] args){
  Pizza pizza=new TomatoPizzaDecorator(new
NormalPizza());
```

```
  pizza.bake();
}

}
```

The UML representation of Decorator Design Pattern :

FlyWeight Design Pattern

- The FlyWeight is a structural design pattern. In flyweight pattern, instead of creating large number of similar objects, those are reused to save memory.
- This pattern is especially useful when memory is a key concerned.

- For e.g. smart mobile comes with applications. Let's consider it has an application which is similar to paint application.
- A user can draw as many shapes in it like circles, triangles and squares etc.,

- Mobiles are the small devices which come with limited set of resources; memory capacity in a smart phone is very less and should use it efficiently.

- In this case if we try to represent one object for every shape that user draws in the app, the entire mobile memory will be filled up with these objects and makes your mobile run quickly out of memory.

Let's understand this by taking a sample code here.

I have an interface IShape which represents several shapes which I have to draw.

```
package design;

public interface IShape {

 void draw();

}
```

Now I have Circle & Rectangle as implementation classes for the above interface.

```
package design;

public class Circle implements IShape{

	private String label;
	private int radius;
	private String fillColor;
	private String lineColor;

	public Circle() {
		label="circle";
	}

	//setters and getters

	@Override
	public void draw() {
	  System.out.println("drawing "+label+" with
radius : "+radius+" fillColor : "+fillColor+" lineColor
: "+lineColor);
	}
}
```

```
package design;

public class Rectangle implements IShape {

	private String label;
	private int length;
	private int breath;
	private String fillStyle;

	public Rectangle() {
		label="rectangle";
	}

	//setters and getters

	@Override
	public void draw() {
	  System.out.println("drawing "+label+" with
length : "+length+" breath : "+breath+" fillStyle :
"+fillStyle);
	}
}
```

Now in my mobile application I want to 100 circles and rectangles.
To do this I need to create 100 circles/rectangle objects,
I need to set the properties like radius,
length and breath to draw these shapes as shown below.

```
package design;

public class PaintApp {

 public void render(int noOfShapes){
   IShape[] shapes=new IShape[noOfShapes+1];

  for(int i=1;i<=noOfShapes;i++){
     if(i%2==0.0f){
          shapes[i]=new Circle();
          ((Circle) shapes[i]).setRadius(i);
          ((Circle) shapes[i]).setLineColor("red");
          ((Circle) shapes[i]).setFillColor("white");
          shapes[i].draw();
```

```
      }else {
            shapes[i]=new Rectangle();
            ((Rectangle) shapes[i]).setLength(i+i);
            ((Rectangle) shapes[i]).setBreath(i*i);
            ((Rectangle)
shapes[i]).setFillStyle("dotted");
            shapes[i].draw();
      }
  }
 }//render
}
```

- Let's say I want to draw 1000 shapes, do I need to create 1000 shape implemented class objects.
- Creating 1000 shape objects consumes more amount of memory and yields in performance issues.

- Now think to draw 1000 shapes do we need to create really 1000 shapes objects.
  If we observe here our circle or rectangle objects contains attributes.
- These attributes represent state of the class.
  Out of which few attributes are common across all the circles/rectangles we draw.
- For e.g. the label of a circle will be "circle" even we draw 1000 circles also, similarly the label of the rectangle will also be "rectangle".
- This type of attributes or state contained in a class is called intrinsic state, which can be shared across the objects of the class.
- But if we look at the radius, lineColor, fillColor or fillStyle these are the values based on which we need to draw the shape, which are going to change from one circle/rectangle to other.
  This type of attributes or state in a class is called extrinsic state (non-sharable).

- So, flyweight pattern encourages developers to identity intrinsic and extrinsic state in an object, and recommends passing extrinsic state as dynamic values while calling the methods rather storing it as state.
- This makes object to be reusable, making it to draw several shapes with less number of objects.

- Designing an object down to the lowest levels of granularity makes the flexible.
- But makes it more overweight and performance gets effected.
- Let's re-design our classes based on the recommendations provided by flyweight to again optimal utilization of memory.

First separate the extrinsic data and pass them as parameters to the methods.

```
package design;

public abstract class IShape {

 public void draw(int radius,String fillColor,String
lineColor){
       //no-op
 }

 public void draw(int length,int breath,String
fillStyle){
       //no-op
 }

}
package design;

public class Circle extends IShape{

      private String label;

      public Circle() {
```

```
            label="circle";
      }

      @Override
      public void draw(int radius,String
fillColor,String lineColor) {
        System.out.println("drawing "+label+" with
radius : "+radius+" fillColor : "+fillColor+" lineColor
: "+lineColor);
      }

}
package design;

public class Rectangle extends IShape {

      private String label;

      public Rectangle() {
            label="rectangle";
      }

      @Override
      public void draw(int length,int breath,String
fillStyle) {
        System.out.println("drawing "+label+" with
length : "+length+" breath : "+breath+" fillStyle :
"+fillStyle);
      }

}
```

- Now to draw 1000 circles/rectangles we don't need to use 1000 shape objects rather than we can reuse the same object to draw any number.
- In order to reuse the objects we need a factory. The factory class here stores the objects and allows us to track the objects to reuse.
- It contains a map of key as shapeType and an object for that shape.

If we want to draw a circle rather than creating an object for circle, we can go to the factory and ask for a circle,
it checks and create/return the existing object in case one exists.

```
package design;

import java.util.HashMap;
import java.util.Map;

public class ShapeFactory {

 private volatile static Map<String,IShape> shapes;

 static{
       shapes=new HashMap<String,IShape>();
 }

 public synchronized static IShape getShape(String
type){
   IShape shape=null;

   //if exists return the existing object
   if(shapes.containsKey(type)){
         shape=shapes.get(type);
   }else{
        //if shape not found create and store
       if(type.equals("circle")){
             shape=new Circle();
       }else if(type.equals("rectangle")){
             shape=new Rectangle();
       }
       shapes.put(type, shape);
   }


   return shape;
 }

}
```

Now in my mobile app rather than creating shape objects we can request for the objects from factory,

which is going to either create one new or returns the existing for if already an object for the shape exists as shown below.

```
package design;

public class PaintApp {

 public void render(int noOfShapes){
   IShape shape=null;

  for(int i=1;i<=noOfShapes;i++){
      if(i%2==0.0f){
            shape=ShapeFactory.getShape("circle");
            shape.draw(i, "red", "white");
      }else {
            shape=ShapeFactory.getShape("rectangle");
            shape.draw(i+i, i*i, "dotted");
      }
  }//for

 }//render


 public static void main(String[] args){
       PaintApp pa=new PaintApp();
       pa.render(2);
 }

}
```

output :

```
drawing rectangle with length : 2 breath : 1 fillStyle
: dotted
drawing circle with radius : 2 fillColor : red
lineColor : white
```

If we see the advantage here with only 2 objects of the shape class we can manage to draw lakhs of shapes also.
We can achieve higher performance with little amount of memory.

The UML representation of FlyWeight Design Pattern :

Key Points :

- If the object overhead ia an issue, where need to reduce the memory footprint. With little amount of design changes we should be able to achieve this, but client should be notified with the impact.
- Remove the extrinsic (non-sharable) state of the class and pass it as arguments to the parameters.
- Create a factory through which we can reuse the objects that creating new.
- The clients must use factory instead of creating the objects out of new operator.

Presentation Tier Patterns

1. Intercepting Filter Design Pattern
2. Front Controller Design Pattern
3. Application Controller Design Pattern
4. Context Object Design Pattern
5. Value Object Design Pattern

Intercepting Filter Design Pattern

```
package dp.fcdp;

import java.io.IOException;
```

```
import java.util.logging.Logger;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;


public class UserAgentFilter implements Filter {

 private Logger logger;
 private FilterConfig config;

@Override
public void init(FilterConfig config) throws
ServletException {
      this.config=config;
      logger=Logger.getLogger("logfile");
}

@Override
public void doFilter(ServletRequest request,
ServletResponse response, FilterChain chain)

throws IOException, ServletException {
 String userAgent=null;
 userAgent=((HttpServletRequest)
request).getHeader("User-Agent");
 logger.info("User Agent : "+userAgent);


 if(userAgent.contains("Mozilla")) {
  chain.doFilter(request, response);
 }else{
  RequestDispatcher
rd=request.getRequestDispatcher("badBrowser.jsp");
  rd.forward(request, response);
 }

}
```

```
@Override
public void destroy() {
 logger=null;
}

}
package dp.fcdp;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HomeServlet extends HttpServlet {

 private static final long serialVersionUID = 1L;

@Override
protected void service(HttpServletRequest
request,HttpServletResponse response)
               throws ServletException, IOException{
 request.setAttribute("loggedInUser", " john");
 RequestDispatcher
rd=request.getRequestDispatcher("home.jsp");
 rd.forward(request, response);
 }
}
```

home.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
```

```
<title>Home Page</title>
</head>
<body>
<form action="empList.do">
LoggedIn ${loggedInUser}
 <input type="hidden" name="token" value="${token}" />
</form>

</body>
</html>
```

badBrowser.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Browser not supported</title>
</head>
<body>
 <p style="color: red;">Browser not Supported  use
Firefox </p>
</body>
</html>
```

web.xml

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
id="WebApp_ID" version="3.0">

<filter>
 <filter-name>UserAgent</filter-name>
```

```
 <filter-class>dp.fcdp.UserAgentFilter</filter-class>
</filter>
<filter-mapping>
 <filter-name>UserAgent</filter-name>
 <url-pattern>/*</url-pattern>
 <dispatcher>REQUEST</dispatcher>
</filter-mapping>

<servlet>
 <servlet-name>HomeServlet</servlet-name>
 <servlet-class>dp.fcdp.HomeServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>HomeServlet</servlet-name>
 <url-pattern>/home.do</url-pattern>
</servlet-mapping>

</web-app>
```

index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Home</title>
</head>
<body>
<a href="/javaT/home.do">Click Here</a>
</body>
</html>
```

output :

```
http://localhost:8001/javaT/
LoggedIn john
```

Front Controller Design Pattern

index.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">

<title>Home Page</title>
</head>
<body>

<a href="StudentView.do">Show Student</a>

</body>
</html>
package design;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public interface Command {
 String execute(HttpServletRequest
request,HttpServletResponse response);
}
package design;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


public class StudentViewCommand implements Command {

@Override
public String execute(HttpServletRequest request,
HttpServletResponse response) {

 String page=null;
 StudentVO studentVo=null;
```

```
 //query data using delegate and dao populate in Value
Object
 studentVo=new StudentVO(1,"Ashok");

 request.setAttribute("student", studentVo);
 page="showStudent";

 return page;
}


}
package design;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FrontController extends HttpServlet {

 private static final long serialVersionUID = 1L;

@Override
protected void service(HttpServletRequest request,
HttpServletResponse response)
                                                 throws
ServletException, IOException{

 String page=null;
 Command command=null;
 String requestUri=null;
 CommandHelper helper=null;
 Dispatcher dispatcher=null;

 requestUri=request.getRequestURI();
 helper=new CommandHelper();
 System.out.println("Request URI : "+requestUri);
 command=helper.getCommand(requestUri);

 page=command.execute(request, response);

 dispatcher=new Dispatcher();
```

```
 dispatcher.dispatch(request, response, page);
 }
}
package design;

public class CommandHelper {

 public Command getCommand(String uri) {
   if(uri.contains("StudentView.do")){
         return new StudentViewCommand();
   }

   return null;
 }

}
package design;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Dispatcher {

 public void dispatch(HttpServletRequest request,
HttpServletResponse response, String page) {

 RequestDispatcher dispatcher=null;
 if(page!=null){

dispatcher=request.getRequestDispatcher(mapPageToView(p
age));
   try{
        dispatcher.forward(request, response);
   }catch(ServletException e){
         e.printStackTrace();
   }catch (IOException e) {
         e.printStackTrace();
   }
 }//if

 }
```

```
private String mapPageToView(String page) {

  if(page.equals("showStudent")){
       return "viewStudent.jsp";
  }

  return null;
}

}
package design;

public class StudentVO {

 private int id;
 private String name;

 public StudentVO(int id, String name){
      this.id=id;
      this.name=name;
 }

 //setters and getters

public int getId() {
      return id;
}
public void setId(int id) {
      this.id = id;
}

public String getName() {
      return name;
}
public void setName(String name) {
      this.name = name;
}

}
```

viewStudent.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>

<jsp:useBean id="student" type="design.StudentVO"
scope="request" beanName="studentVO"/>


<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Student Information</title>
</head>
<body>

<p style="color: blue;font-size: large;font-family:
sans-serif;">
Student Id : <jsp:getProperty property="id"
name="student"/><br>
Student Name : <jsp:getProperty property="name"
name="student"/>
</p>

</body>
</html>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
id="WebApp_ID" version="3.0">

<servlet>
 <servlet-name>front</servlet-name>
 <servlet-class>design.FrontController</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>front</servlet-name>
 <url-pattern>*.do</url-pattern>
</servlet-mapping>

<welcome-file-list>
```

```
<welcome-file>index.jsp</welcome-file>
</welcome-file-list>

</web-app>
```

result :

```
http://localhost:8001/javaT/

http://localhost:8001/javaT/StudentView.do

browser :
Student Id : 1
Student Name : Ashok

console :
Request URI : /javaT/StudentView.do
```

Application Controller Design Pattern

index.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">

<title>Index Page</title>
</head>
<body>

<a href="studentView.do?id=10">Show Student
Information</a>

</body>
</html>
```

viewStudent.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>

<jsp:useBean id="student" type="design.StudentVO"
scope="request" />


<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Student Information</title>
</head>
<body>

<p style="color: blue;font-size:x-large; large;font-
family: sans-serif;">
Student Id : <jsp:getProperty property="id"
name="student"/><br>
Student Name : <jsp:getProperty property="name"
name="student"/>
</p>

</body>
</html>
```

web.xml

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
id="WebApp_ID" version="3.0">

<display-name>ApplicationControllerWeb</display-name>
<servlet>
 <servlet-name>front</servlet-name>
 <servlet-class>design.FrontController</servlet-class>
</servlet>
<servlet-mapping>
```

```
 <servlet-name>front</servlet-name>
 <url-pattern>*.do</url-pattern>
</servlet-mapping>

<welcome-file-list>
<welcome-file>index.jsp</welcome-file>
</welcome-file-list>

</web-app>
package design;

public class StudentVO {

 private String id;
 private String name;

 //constructor
 public StudentVO(String id, String name){
      this.id=id;
      this.name=name;
 }

 //setters and getters

public String getId() {
      return id;
}
public void setId(String id) {
      this.id = id;
}

public String getName() {
      return name;
}
public void setName(String name) {
      this.name = name;
}

}
package design;

public interface Command {
 String execute(RequestContext requestContext);
}
package design;
```

```
public class StudentViewCommand implements Command {

@Override
public String execute(RequestContext requestContext) {

 String id=null;
 StudentVO studentVo=null;

 id=requestContext.getParameter("id");

 //call delegate and dao
 studentVo=new StudentVO(id,"Ashok");

 requestContext.setAttribute("student", studentVo);

 return "showStudent";
}

}
package design;

import java.util.Map;

import javax.servlet.http.HttpServletRequest;

public class ContextFactory {

public RequestContext
getContextObject(HttpServletRequest request){
  Map<String, String[]> requestMap=null;
  RequestContext requestContext=null;
  HttpRequestMapper requestMapper=null;

  requestMapper=new HttpRequestMapper();
  requestMap=requestMapper.extract(request);
  requestContext=new
RequestContext(request.getRequestURI(),requestMap);

return requestContext;
}

public void bindContextObject(HttpServletRequest
request,RequestContext requestContext){
 HttpRequestMapper requestMapper=null;
```

```
 requestMapper=new HttpRequestMapper();
 requestMapper.bind(request,
requestContext.getResponseMap());
}

}
package design;

import java.util.HashMap;
import java.util.Map;

public class RequestContext {

  private String requestUri;
  private Map<String, String[]> requestMap;
  private Map<String, Object> responseMap;

  public RequestContext(){
        requestUri=null;
        requestMap=new HashMap<String, String[]>();
        responseMap=new HashMap<String, Object>();
  }

  public RequestContext(String requestUri, Map<String,
String[]> requestMap) {
        this.requestUri=requestUri;
        this.requestMap=requestMap;
        this.responseMap=new HashMap<String, Object>();
  }

  public String getParameter(String param) {
   String[] val=null;
   if(param != null){
         val=requestMap.get(param);
   }
  return val[0];
  }

  public void setAttribute(String param, Object value)
{
       responseMap.put(param, value);
  }

  public String getRequestUri() {
```

```
        return requestUri;
    }

    public Map<String, String[]> getRequestMap() {
        return requestMap;
    }

    public Map<String, Object> getResponseMap() {
        return responseMap;
    }

}
package design;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FrontController extends HttpServlet {

 private static final long serialVersionUID = 1L;

@Override
protected void service(HttpServletRequest request,
HttpServletResponse response)
                                                    throws
ServletException, IOException{

 String page=null;
 String view=null;
 Dispatcher dispatcher=null;
 RequestContext requestContext=null;
 ContextFactory contextFactory=null;
 ApplicationController applicationController=null;

 // plubbing code (security, authorization)

 // extracting data from protocol
 contextFactory=new ContextFactory();

requestContext=contextFactory.getContextObject(request)
;
```

```
 applicationController=new ApplicationController();
 view=applicationController.process(requestContext);

 // binding data back to protocol
 contextFactory.bindContextObject(request,
requestContext);
 page=applicationController.mapViewToPage(view);

 dispatcher=new Dispatcher();
 dispatcher.dispatch(request, response, page);
 }
}
package design;

public class ApplicationController {

public String process(RequestContext requestContext) {
      String view=null;
      Command command=null;
      CommandHelper commandHelper=null;

      commandHelper=new CommandHelper();
      command=commandHelper.getCommand(requestContext.ge
tRequestUri());
      view=command.execute(requestContext);

return view;
}

public String mapViewToPage(String view) {
  String page=null;
  if(view !=null){
        page="viewStudent.jsp";
  }

return page;
}

}
package design;

public class CommandHelper {

 public Command getCommand(String uri) {
```

```
  Command command=null;
   if(uri.contains("studentView.do")){
         command= new StudentViewCommand();
   }

   return command;
 }

}
package design;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Dispatcher {

 public void dispatch(HttpServletRequest request,
HttpServletResponse response, String page) {

 RequestDispatcher rd=null;

  rd=request.getRequestDispatcher(page);
   try{
        rd.forward(request, response);
   }catch(ServletException e){
         e.printStackTrace();
   }catch (IOException e) {
         e.printStackTrace();
   }
 }

}
package design;

import java.util.Map;
import java.util.Set;

import javax.servlet.http.HttpServletRequest;

public class HttpRequestMapper {
```

```
public Map<String, String[]> extract(HttpServletRequest
request) {
      Map<String, String[]> requestMap=null;
      requestMap=request.getParameterMap();

return requestMap;
}

public void bind(HttpServletRequest request,
Map<String, Object> responseMap) {

 Set<String> keys=null;
 keys= responseMap.keySet();

 for(String param : keys){
   request.setAttribute(param, responseMap.get(param));
 }

}

}
```

output :

```
http://localhost:8001/javaT/

http://localhost:8001/javaT/studentView.do?id=10

browser :
Student Id : 10
Student Name : Ashok
```

Context Object Design Pattern

index.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">

<title>Index Page</title>
</head>
<body>

<a href="studentView.do?id=10">Show Student
Information</a>

</body>
</html>
```

viewStudent.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>

<jsp:useBean id="student" type="design.StudentVO"
scope="request" />


<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Student Information</title>
</head>
<body>

<p style="color: blue;font-size:x-large; large;font-
family: sans-serif;">
Student Id : <jsp:getProperty property="id"
name="student"/><br>
Student Name : <jsp:getProperty property="name"
name="student"/>
</p>
```

```
</body>
</html>
```

web.xml

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
id="WebApp_ID" version="3.0">

<display-name>ApplicationControllerWeb</display-name>
<servlet>
 <servlet-name>front</servlet-name>
 <servlet-class>design.FrontController</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>front</servlet-name>
 <url-pattern>*.do</url-pattern>
</servlet-mapping>

<welcome-file-list>
<welcome-file>index.jsp</welcome-file>
</welcome-file-list>

</web-app>
package design;

public class StudentVO {

 private String id;
 private String name;

 //constructor
 public StudentVO(String id, String name){
      this.id=id;
      this.name=name;
 }

 //setters and getters

public String getId() {
      return id;
}
public void setId(String id) {
```

```
		this.id = id;
}

public String getName() {
		return name;
}
public void setName(String name) {
		this.name = name;
}

}
package design;

public interface Command {
 String execute(RequestContext requestContext);
}
package design;

public class StudentViewCommand implements Command {

@Override
public String execute(RequestContext requestContext) {

 String id=null;
 StudentVO studentVo=null;

 id=requestContext.getParameter("id");

 //call delegate and dao
 studentVo=new StudentVO(id,"Ashok");

 requestContext.setAttribute("student", studentVo);

 return "showStudent";
}

}
package design;

import java.util.Map;

import javax.servlet.http.HttpServletRequest;

public class ContextFactory {
```

```
public RequestContext
getContextObject(HttpServletRequest request){
  Map<String, String[]> requestMap=null;
  RequestContext requestContext=null;
  HttpRequestMapper requestMapper=null;

  requestMapper=new HttpRequestMapper();
  requestMap=requestMapper.extract(request);
  requestContext=new
RequestContext(request.getRequestURI(),requestMap);

return requestContext;
}

public void bindContextObject(HttpServletRequest
request,RequestContext requestContext){
 HttpRequestMapper requestMapper=null;

 requestMapper=new HttpRequestMapper();
 requestMapper.bind(request,
requestContext.getResponseMap());
}

}
package design;

import java.util.HashMap;
import java.util.Map;

public class RequestContext {

  private String requestUri;
  private Map<String, String[]> requestMap;
  private Map<String, Object> responseMap;

  public RequestContext(){
        requestUri=null;
        requestMap=new HashMap<String, String[]>();
        responseMap=new HashMap<String, Object>();
  }

  public RequestContext(String requestUri, Map<String,
String[]> requestMap) {
        this.requestUri=requestUri;
        this.requestMap=requestMap;
```

```
        this.responseMap=new HashMap<String, Object>();
  }

  public String getParameter(String param) {
   String[] val=null;
   if(param != null){
         val=requestMap.get(param);
   }
  return val[0];
  }

  public void setAttribute(String param, Object value)
{
       responseMap.put(param, value);
  }

  public String getRequestUri() {
      return requestUri;
  }

  public Map<String, String[]> getRequestMap() {
      return requestMap;
  }

  public Map<String, Object> getResponseMap() {
      return responseMap;
  }

}
package design;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FrontController extends HttpServlet {

 private static final long serialVersionUID = 1L;

@Override
protected void service(HttpServletRequest request,
HttpServletResponse response)
```

```
                                                        throws
ServletException, IOException{

 String page=null;
 String view=null;
 Dispatcher dispatcher=null;
 RequestContext requestContext=null;
 ContextFactory contextFactory=null;
 ApplicationController applicationController=null;

 // plubbing code (security, authorization)

 // extracting data from protocol
 contextFactory=new ContextFactory();

requestContext=contextFactory.getContextObject(request)
;

 applicationController=new ApplicationController();
 view=applicationController.process(requestContext);

 // binding data back to protocol
 contextFactory.bindContextObject(request,
requestContext);
 page=applicationController.mapViewToPage(view);

 dispatcher=new Dispatcher();
 dispatcher.dispatch(request, response, page);
 }
}
package design;

public class ApplicationController {

public String process(RequestContext requestContext) {
      String view=null;
      Command command=null;
      CommandHelper commandHelper=null;

      commandHelper=new CommandHelper();
      command=commandHelper.getCommand(requestContext.ge
tRequestUri());
      view=command.execute(requestContext);

return view;
```

```
}

public String mapViewToPage(String view) {
  String page=null;
  if(view !=null){
        page="viewStudent.jsp";
  }

return page;
}

}
package design;

public class CommandHelper {

 public Command getCommand(String uri) {

  Command command=null;
   if(uri.contains("studentView.do")){
         command= new StudentViewCommand();
   }

   return command;
 }

}
package design;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Dispatcher {

 public void dispatch(HttpServletRequest request,
HttpServletResponse response, String page) {

 RequestDispatcher rd=null;

  rd=request.getRequestDispatcher(page);
   try{
```

```
        rd.forward(request, response);
   }catch(ServletException e){
         e.printStackTrace();
   }catch (IOException e) {
         e.printStackTrace();
   }
 }

}
package design;

import java.util.Map;
import java.util.Set;

import javax.servlet.http.HttpServletRequest;

public class HttpRequestMapper {

public Map<String, String[]> extract(HttpServletRequest
request) {
      Map<String, String[]> requestMap=null;
      requestMap=request.getParameterMap();

return requestMap;
}

public void bind(HttpServletRequest request,
Map<String, Object> responseMap) {

 Set<String> keys=null;
 keys= responseMap.keySet();

 for(String param : keys){
   request.setAttribute(param, responseMap.get(param));
 }

}

}
```

output :

```
http://localhost:8001/javaT/
```

```
http://localhost:8001/javaT/studentView.do?id=10

browser :
Student Id : 10
Student Name : Ashok
```

Business & Integration Tier Design Patterns

1. DAO (Data Access Object) Design Pattern
2. Business Delegate Design Pattern
3. Business Object Design Pattern
4. Service Locator Design Pattern
5. Session Facade Design Pattern

Data Access Object (DAO) Design Pattern

```
package design.dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.logging.Logger;

public class ConnectionFactory {

 private static Logger logger;

static{
      logger=Logger.getLogger("logfile");
}

public static Connection getConnection(final String
driverClassName,final String url,final String
userName,final String password) throws SQLException,
ClassNotFoundException {
 Connection con=null;
```

```
 try{
  Class.forName(driverClassName);

con=DriverManager.getConnection(url,userName,password);
  con.setAutoCommit(false);
 }catch(SQLException e){
   logger.throwing("ConnectionFactory", "Exception
thrown by ConnectionFactory", e);
   throw e;
 }catch(ClassNotFoundException e){
   logger.throwing("ConnectionFactory", "Not able to
load driver class", e);
   throw e;
 }

return con;
}

}
package design.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.logging.Logger;

public class EmpDao {

  //SQL queries
 private final String INSERT_QUERY="insert into
emp(emp_id,name) values(?,?)";
 private Logger logger;

 public EmpDao(){
  logger=Logger.getLogger("logfile");
 }

public void insert(int id, String name) throws
SQLException, ClassNotFoundException {

 boolean isException=false;
 Connection con=null;
 PreparedStatement pstmt=null;
```

```
 try{

con=ConnectionFactory.getConnection("oracle.jdbc.driver
.OracleDriver","jdbc:oracle:thin:@localhost:1521:xe","s
ystem","tiger");
  pstmt=con.prepareStatement(INSERT_QUERY);
  pstmt.setInt(1, id);
  pstmt.setString(2, name);
  pstmt.executeUpdate();
 }catch(SQLException e){
       isException=true;
       logger.throwing("StudentDao", "Unable to get
Connection", e);
       throw e;
 }catch(ClassNotFoundException e) {
       isException=true;
       logger.throwing("StudentDao", "Unable to get
Connection", e);
       throw e;
 }finally{

   if(con != null){
         if(isException==true){
               con.rollback();
         }else con.commit();

   con.close();
   }

 if(pstmt != null)
       pstmt.close();


 }//finally
}
}


/*
create table emp(emp_id number(20),name varchar2(15),
sal number(7,2));
*/
package design.dao;

import java.sql.SQLException;
```

```
public class EmpController {

public void addEmp(int id, String name) throws
ClassNotFoundException, SQLException {

      EmpDao dao=new EmpDao();
      dao.insert(id,name);
}

}
package design.dao;

import java.sql.SQLException;

public class DaoTest {

public static void main(String[] args) throws
ClassNotFoundException, SQLException {

 EmpController empController=new EmpController();
 empController.addEmp(9,"Ashok");

}

}
```

Business Delegate Design Pattern

index.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">

<title>Index Page</title>
</head>
```

```
<body>

<a href="/javaT/registerStudent.jsp">Show Student
Registration Information</a>

</body>
</html>
```

registerStudent.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Register Student</title>
</head>
<body>

<form action="register">
<table>
<tr><td>Student Id : </td><td> <input type="text"
name="studentId"></td></tr>
<tr><td> Name : </td><td> <input type="text"
name="name"></td></tr>
<tr><td>Course Id : </td><td> <input type="text"
name="courseId"></td></tr>
<tr><td colspan="2"> <input type="submit"
value="Register"></td></tr>
</table>
</form>

</body>
</html>
```

error.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
```

```
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Error !!!</title>
</head>
<body>

<p style="color:red;">
  Error while processing your request. Please contact
administration.
</p>

</body>
</html>
```

confirm.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Registration Successfully</title>
</head>
<body>

<p style="color:green;">
  Student ${studentId} register successfully ....
</p>

</body>
</html>
package business.delegate;

import java.io.Serializable;

public class StudentBO implements Serializable {

      private int id;
      private String name;

      //getters and setters
```

```
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
}
package business.delegate;

import java.io.Serializable;

public class StudentCourseBO implements Serializable {

	private int studentId;
	private int courseId;

	// setters and getters

	public int getStudentId() {
		return studentId;
	}
	public void setStudentId(int studentId) {
		this.studentId = studentId;
	}
	public int getCourseId() {
		return courseId;
	}
	public void setCourseId(int courseId) {
		this.courseId = courseId;
	}
}
```

db.properties

```
db.driverClassName = oracle.jdbc.driver.OracleDriver
db.url = jdbc:oracle:thin:@localhost:1521:xe
db.userName = system
db.password = tiger
package business.delegate;
```

```
import java.sql.Connection;
import java.sql.SQLException;

public interface StudentDao {
 void insert(StudentBO studentBO, Connection con)
throws SQLException;
}
package business.delegate;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.logging.Logger;

public class StudentDaoImpl implements StudentDao {

  //SQL queries
 private final String INSERT_QUERY="insert into
student(student_id,name) values(?,?)";
 private static Logger logger;

static {
  logger=Logger.getLogger("log_file");
}

@Override
public void insert(StudentBO studentBO, Connection con)
throws SQLException {

 PreparedStatement pstmt=null;

 try{
  pstmt=con.prepareStatement(INSERT_QUERY);
  pstmt.setInt(1, studentBO.getId());
  pstmt.setString(2, studentBO.getName());
  pstmt.executeUpdate();
 }catch(SQLException e){
       logger.throwing("StudentDao", "Unable to perform
insert", e);
       throw e;
 }finally{
     if(pstmt != null)
      pstmt.close();
 }
```

```
}
}


/*
  create table student(student_id number(20),name
varchar2(15));
 */
package business.delegate;

import java.sql.Connection;
import java.sql.SQLException;

public interface StudentCourseDao {
      void insert(StudentCourseBO studentCourseBO,
Connection con) throws SQLException;
}
package business.delegate;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.logging.Logger;

public class StudentCourseDaoImpl implements
StudentCourseDao {

  //SQL queries
 private final String INSERT_QUERY="insert into
course(student_id, course_id) values(?,?)";
 private static Logger logger;

static {
  logger=Logger.getLogger("log_file");
}

@Override
public void insert(StudentCourseBO studentCourseBO,
Connection con) throws SQLException {

 PreparedStatement pstmt=null;

 try{
  pstmt=con.prepareStatement(INSERT_QUERY);
```

```
  pstmt.setInt(1, studentCourseBO.getStudentId());
  pstmt.setInt(2, studentCourseBO.getCourseId());
  pstmt.executeUpdate();
 }catch(SQLException e){
       logger.throwing("StudentCourseDao", "Unable to
perform insert", e);
       throw e;
 }finally{
     if(pstmt != null)
      pstmt.close();
 }

}
}


/*
  create table course(student_id number(20), course_id
number(20));
 */
package business.delegate;

import java.sql.SQLException;

public interface RegistrationDelegate {
   void register(RegistrationVO registrationVO) throws
SQLException, ClassNotFoundException, GenericException;
}
package business.delegate;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.logging.Logger;

public class RegistrationDelegateImpl implements
RegistrationDelegate {

private static Logger logger;

static{
  logger=Logger.getLogger("log_file");
}

@Override
```

```
public void register(RegistrationVO registrationVO)
throws SQLException, ClassNotFoundException,
GenericException {
 boolean isEx=false;
 Connection con=null;
 StudentBO studentBO=null;
 StudentCourseBO studentCourseBO=null;
 StudentDao studentDao=null;
 StudentCourseDao studentCourseDao=null;

 try{
  con=ConnectionFactory.getConnection();

  studentBO=new StudentBO();

studentBO.setId(Integer.parseInt(registrationVO.getStud
entId()));
  studentBO.setName(registrationVO.getName());

  studentDao=new StudentDaoImpl();
  studentDao.insert(studentBO, con);

  studentCourseBO=new StudentCourseBO();

studentCourseBO.setStudentId(Integer.parseInt(registrat
ionVO.getStudentId()));

studentCourseBO.setCourseId(Integer.parseInt(registrati
onVO.getCourseId()));

  studentCourseDao=new StudentCourseDaoImpl();
  studentCourseDao.insert(studentCourseBO, con);

 }catch(SQLException e){
   logger.throwing("RegistrationDelegateImpl", "Unable
to register student", e);
   throw new GenericException(e);
 }catch (ClassNotFoundException e) {
   logger.throwing("RegistrationDelegateImpl", "Unable
to register student", e);
   throw new GenericException(e);
 }finally{
      if(con != null) {
          try{
               if(isEx)
```

```
                        con.rollback();
                    else
                        con.commit();
                con.close();
            }catch(SQLException e){

      logger.throwing("RegistrationDelegateImpl",
"Unable to commit/rollback/clode connection", e);
                throw new GenericException(e);
            }

       }//if

 }//finally


}

}
package business.delegate;

import java.io.IOException;
import java.sql.SQLException;
import java.util.logging.Logger;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class RegistrationServlet extends HttpServlet {
  private static Logger
logger=Logger.getLogger("log_file");

@Override
protected void service(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {

  boolean isEx=false;
  String studentId=null;
  String name=null;
  String courseId=null;
  String page=null;
```

```
  RegistrationVO registrationVO=null;
  RegistrationDelegate registrationDelegate=null;
  RequestDispatcher requestDispatcher=null;

  studentId=request.getParameter("studentId");
  name=request.getParameter("name");
  courseId=request.getParameter("courseId");

  registrationVO=new RegistrationVO();
  registrationVO.setStudentId(studentId);
  registrationVO.setName(name);
  registrationVO.setCourseId(courseId);

  registrationDelegate=new RegistrationDelegateImpl();

  try{
       registrationDelegate.register(registrationVO);
       request.setAttribute("studentId", studentId);
  }catch(GenericException | ClassNotFoundException |
SQLException e){
        isEx=true;
        logger.throwing("Registration Servlet", "Error
occured during registration", e);
  }

  if(isEx)
        page="error.jsp";
  else
        page="confirm.jsp";

  requestDispatcher=request.getRequestDispatcher(page);
  requestDispatcher.forward(request, response);

}

}
package business.delegate;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
import java.util.ResourceBundle;
import java.util.Set;
import java.util.logging.Logger;
```

```
public class ConnectionFactory {
 private static Properties props;
 private static Logger logger;

static{
 logger=Logger.getLogger("log_file");
 props=new Properties();
 ResourceBundle
rb=ResourceBundle.getBundle("business/delegate/db"); //
business.delegate.db
 Set<String> keys=rb.keySet();
 for(String key:keys){
       props.put(key, rb.getString(key));
 }
}

public static Connection getConnection() throws
SQLException, ClassNotFoundException {
 Connection con=null;

 try{
     Class.forName(props.getProperty("db.driverClassNam
e"));

con=DriverManager.getConnection(props.getProperty("db.u
rl"),props.getProperty("db.userName"),props.getProperty
("db.password"));
    con.setAutoCommit(false);
 }catch(SQLException e){
   logger.throwing("ConnectionFactory", "Unable to get
Connection", e);
  throw e ;
 }catch (Exception e) {
   logger.throwing("ConnectionFactory", "Unable to load
driver class", e);
   throw e ;
}
 return con;

}

}
package business.delegate;
```

```
public class GenericException extends Throwable {

public GenericException(){
	super();
}

public GenericException(String str, Throwable exp){
	super(str,exp);
}

public GenericException(String str){
	super(str);
}

public GenericException(Throwable exp){
	super(exp);
}

}
package business.delegate;

import java.io.Serializable;

public class RegistrationVO implements Serializable{

	private String studentId;
	private String name;
	private String courseId;

	// setters and getters

	public String getStudentId() {
		return studentId;
	}
	public void setStudentId(String studentId) {
		this.studentId = studentId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getCourseId() {
		return courseId;
```

```
	}
	public void setCourseId(String courseId) {
		this.courseId = courseId;
	}

}
```

web.xml

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
id="WebApp_ID" version="3.0">

<display-name>BusinessDelegate</display-name>
<servlet>
 <servlet-name>delegate</servlet-name>
 <servlet-
class>business.delegate.RegistrationServlet</servlet-
class>
</servlet>
<servlet-mapping>
 <servlet-name>delegate</servlet-name>
 <url-pattern>/register</url-pattern>
</servlet-mapping>

<welcome-file-list>
<welcome-file>index.jsp</welcome-file>
</welcome-file-list>

</web-app>
```

output :

```
http://localhost:8001/javaT/
(OR)
http://localhost:8001/javaT/register?studentId=123&name
=ashok&courseId=432

Student 123 register successfully ....

//add required jar files to project lib flder like
servlet-api.jar, ojdbc14.jar etc.,
```

Session Facade Design Pattern

index.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">

<title>Index Page</title>
</head>
<body>

<a href="/javaT/findStock.jsp">Find Stock
Information</a>

</body>
</html>
```

findStock.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Find Stock</title>
</head>
<body>
<form action="stock">
<table>
 <tr><td>Short Name : </td><td><input type="text"
name="shortName"/></td></tr>
 <tr><td colspan="2"><input type="submit"
value="Find"/></td></tr>
</table>
```

```
</form>
</body>
</html>
```

stockDetails.jsp

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<jsp:useBean id="stock"
type="session.facade.vo.StockVO" scope="request"/>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Stock Information</title>
</head>
<body>
<p style="color:blue">
 Short Name : ${stock.shortName} <br>
 Price : ${stock.price} <br>
 Listed Date : ${stock.listDate}
</p>
</body>
</html>
package session.facade.bo;

import java.io.Serializable;

public class GetStock implements Serializable{

 private String shortName;

 //setters and getters

public String getShortName() {
      return shortName;
}
```

```
public void setShortName(String shortName) {
      this.shortName = shortName;
}

}
package session.facade.bo;

import java.util.Date;

public class StockInfo {

  private int id;
  private float price;
  private Date listDate;

  //getters and setters

public int getId() {
      return id;
}
public void setId(int id) {
      this.id = id;
}
public float getPrice() {
      return price;
}
public void setPrice(float price) {
      this.price = price;
}
public Date getListDate() {
      return listDate;
}
public void setListDate(Date listDate) {
      this.listDate = listDate;
}

}
package session.facade.dto;

import java.util.Date;

public class SearchStock {

  private String shortName;
  private Date time;
```

```
  //getters and setters

public String getShortName() {
      return shortName;
}
public void setShortName(String shortName) {
      this.shortName = shortName;
}
public Date getTime() {
      return time;
}
public void setTime(Date time) {
      this.time = time;
}

}
package session.facade.dto;

import java.util.Date;

public class Stock {

  private int id;
  private String shortName;
  private String companyName;
  private Date listedDate;
  private float price;

  //getters and setters

public int getId() {
      return id;
}
public void setId(int id) {
      this.id = id;
}
public String getShortName() {
      return shortName;
}
public void setShortName(String shortName) {
      this.shortName = shortName;
}
public String getCompanyName() {
      return companyName;
```

```
}
public void setCompanyName(String companyName) {
	this.companyName = companyName;
}
public Date getListedDate() {
	return listedDate;
}
public void setListedDate(Date listedDate) {
	this.listedDate = listedDate;
}
public float getPrice() {
	return price;
}
public void setPrice(float price) {
	this.price = price;
}

}
package session.facade.bo;

import java.util.Date;

public class Quote {
 private int id;
 private Date time;

 //setters and getters

public int getId() {
	return id;
}
public void setId(int id) {
	this.id = id;
}
public Date getTime() {
	return time;
}
public void setTime(Date time) {
	this.time = time;
}

}
package session.facade.bo;

public class StockDetail {
```

```
  private int id;
  private String shortName;
  private String companyName;

  //getters and setters

public int getId() {
      return id;
}
public void setId(int id) {
      this.id = id;
}
public String getShortName() {
      return shortName;
}
public void setShortName(String shortName) {
      this.shortName = shortName;
}
public String getCompanyName() {
      return companyName;
}
public void setCompanyName(String companyName) {
      this.companyName = companyName;
}

}
package session.facade.delegate;

import java.util.Date;

import session.facade.StockFacade;
import session.facade.StockFacadeLocator;
import session.facade.dto.SearchStock;
import session.facade.dto.Stock;
import session.facade.vo.SearchVO;
import session.facade.vo.StockVO;

public class StockDelegate {

public StockVO search(SearchVO searchVO){
  StockVO stockVO=null;
  SearchStock searchStock=null;
  Stock stock=null;
  StockFacade stockFacade=null;
```

```
  StockFacadeLocator stockFacadeLocator=null;

   searchStock=new SearchStock();
   searchStock.setShortName(searchVO.getShortName());
   searchStock.setTime(new Date());

   stockFacadeLocator=new StockFacadeLocator();
   stockFacade=stockFacadeLocator.getStockFacade();
   stock=stockFacade.findStock(searchStock);

   stockVO=new StockVO();
   stockVO.setShortName(stock.getShortName());
   stockVO.setPrice(String.valueOf(stock.getPrice()));

stockVO.setListDate(stock.getListedDate().toString());

   return stockVO;
}

}
package session.facade;

import session.facade.bo.GetStock;
import session.facade.bo.Quote;
import session.facade.bo.StockDetail;
import session.facade.bo.StockInfo;
import session.facade.dto.SearchStock;
import session.facade.dto.Stock;
import session.facade.service.BSEStockTrade;
import session.facade.service.BSEStockTradeImpl;
import session.facade.service.FindStockDetail;
import session.facade.service.FindStockDetailImpl;

public class StockFacade {

public Stock findStock(SearchStock searchStock){

  FindStockDetail findStockDetail=null;
  BSEStockTrade bseStockTrade=null;
  Quote quote=null;
  StockInfo stockInfo=null;
  GetStock getStock=null;
  StockDetail stockDetail=null;
  Stock stock=null;
```

```
  getStock=new GetStock();
  getStock.setShortName(searchStock.getShortName());

  findStockDetail=new FindStockDetailImpl();
  stockDetail=findStockDetail.findStock(getStock);

  quote=new Quote();
  quote.setId(stockDetail.getId());
  quote.setTime(searchStock.getTime());

  bseStockTrade=new BSEStockTradeImpl();
  stockInfo=bseStockTrade.getStockPrice(quote);

  stock=new Stock();
  stock.setId(stockInfo.getId());
  stock.setShortName(stockDetail.getShortName());
  stock.setCompanyName(stockDetail.getCompanyName());
  stock.setListedDate(stockInfo.getListDate());
  stock.setPrice(stockInfo.getPrice());

  return stock;
}

}
package session.facade.service;

import session.facade.bo.Quote;
import session.facade.bo.StockInfo;

public interface BSEStockTrade {
   StockInfo getStockPrice(Quote quote);
}
package session.facade.service;

import java.util.Date;

import session.facade.bo.Quote;
import session.facade.bo.StockInfo;



public class BSEStockTradeImpl implements BSEStockTrade
{

@Override
```

```
public StockInfo getStockPrice(Quote quote) {
 float price=0.0f;
 StockInfo stockInfo=null;

 if(quote!=null){
      if(quote.getId()==1001){
            price=323.34f;
      }else if (quote.getId()==1002) {
            price=323.23f;
      }
 }

stockInfo=new StockInfo();
stockInfo.setId(quote.getId());
stockInfo.setPrice(price);
stockInfo.setListDate(new Date());

return stockInfo;
}

}
package session.facade.service;

import session.facade.bo.GetStock;
import session.facade.bo.StockDetail;

public interface FindStockDetail {
  StockDetail findStock(GetStock getStock);
}
package session.facade.service;

import session.facade.bo.GetStock;
import session.facade.bo.StockDetail;

public class FindStockDetailImpl implements
FindStockDetail {

@Override
public StockDetail findStock(GetStock getStock) {
  StockDetail stockDetail=null;
  stockDetail=new StockDetail();

  if(getStock.getShortName().equals("ICICIBAN")){
    stockDetail.setId(1001);
    stockDetail.setShortName(getStock.getShortName());
```

```
    stockDetail.setCompanyName("ICICI BANK Pvt Ltd");
  }else if(getStock.getShortName().equals("IBM")){
      stockDetail.setId(1002);
      stockDetail.setShortName(getStock.getShortName());
      stockDetail.setCompanyName("IBM INDIA Pvt Ltd");
  }

      return stockDetail;
}

}
package session.facade.servlet;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import session.facade.delegate.StockDelegate;
import session.facade.vo.SearchVO;
import session.facade.vo.StockVO;

public class FindStockServlet extends HttpServlet {

@Override
protected void service(HttpServletRequest
request,HttpServletResponse response)throws
ServletException, IOException{
      String shortName=null;
      SearchVO searchVO=null;
      StockVO stockVO=null;
      StockDelegate stockDelegate=null;

      shortName=request.getParameter("shortName");


      searchVO=new SearchVO();
      searchVO.setShortName(shortName);

      stockDelegate=new StockDelegate();
      stockVO=stockDelegate.search(searchVO);
```

```
      request.setAttribute("stock", stockVO);
      RequestDispatcher
rd=request.getRequestDispatcher("stockDetails.jsp");
      rd.forward(request, response);
}

}
package session.facade;

public class StockFacadeLocator {

 public StockFacade getStockFacade(){
       //look up logic
       return new StockFacade();
 }
}
package session.facade.vo;

import java.io.Serializable;

public class SearchVO implements Serializable{
 private String shortName;

 //getters and setters

public String getShortName() {
      return shortName;
}

public void setShortName(String shortName) {
      this.shortName = shortName;
}

}
package session.facade.vo;

import java.io.Serializable;

public class StockVO implements Serializable{
 private String shortName;
 private String listDate;
 private String price;

 //setters and getters
```

```
public String getShortName() {
	return shortName;
}
public void setShortName(String shortName) {
	this.shortName = shortName;
}
public String getListDate() {
	return listDate;
}
public void setListDate(String listDate) {
	this.listDate = listDate;
}
public String getPrice() {
	return price;
}
public void setPrice(String price) {
	this.price = price;
}

}
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
id="WebApp_ID" version="3.0">

<display-name>Session Facade</display-name>
<servlet>
 <servlet-name>facade</servlet-name>
 <servlet-
class>session.facade.servlet.FindStockServlet</servlet-
class>
</servlet>
<servlet-mapping>
 <servlet-name>facade</servlet-name>
 <url-pattern>/stock</url-pattern>
</servlet-mapping>

<welcome-file-list>
<welcome-file>index.jsp</welcome-file>
</welcome-file-list>

</web-app>
```

output :

```
http://localhost:8001/javaT/

(OR)

http://localhost:8001/javaT/stock?shortName=ICICIBAN
Short Name : ICICIBAN
Price : 323.34
Listed Date : Mon Mar 21 16:40:02 IST 2016

(OR)

http://localhost:8001/javaT/stock?shortName=IBM
Short Name : IBM
Price : 323.23
Listed Date : Mon Mar 21 16:41:33 IST 2016
```