JAVA
Design Patterns

By
Mr Sriman
Real Time Expert

DURGA SOFTWARE SOLUTIONS

Design Patterns

Core & J2EE Design Patterns

Mr. Sriman

T. Ganesh 9704303534

As part of this material we cover several design patterns that are part of GOF as well as J2EE Design Patterns

Contents

Introduction

1 Introduction

A developer will build software/an application to meet/solve the requirements of an enterprise or a business firm using some programming language. While developing the applications they might use any programming language of their chose like c, c++ or Java etc.

These programming languages provides API's to the developers in building the components, but they never document the best practices, bad practices or design considerations that a developer needs to follow.

A developer while working; needs to understand more than just an API. They need to understand issues like the following

- What are best practices?
- What are the bad practices?
- What are the common recurring problems and proven solutions to these problems?
- How is code refactored from a bad practice to a better one (typically described by pattern)?

This is what a pattern exactly does. It helps you in identifying the recurring problems and provides a pre-built solution that can be applied at its best in solving those problems.

The first efforts in documenting the problem and their solutions have been done in 1970's by Christopher Alexander. He is a civil engineer and architect, has document various patterns in his area in several books.

The software community subsequently adopted the idea of pattern based on his work. Patterns in the software were popularized by the book Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (also known as the Gang of Four, or GOF).

In addition the experts in the J2EE community also documented design patterns based on their experience in solving various problems which they encounter while designing/working on J2EE projects. As these patterns closely coupled with various tiers of J2EE these are also referred J2EE Design patterns.

1.1 What is a Pattern?

Patterns are about documenting a solution for a well know (recurring) problem in a particular context. It can also be defined as recurring **solution** to a **problem** in a **context.** Let me elaborate the things. First, what is a context? A context is an environment, surroundings or situations under which something exits. What is a problem? A problem is something that needs to be resolved. Solution? It is the answer to the problem in a context that helps resolve the issue.

As defined by experts each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution.

1.2 Pattern identification

When we face a recurring problem and resolve it, they first document its characteristics using the pattern template. These documented patterns are called candidate patterns. These candidate patterns will not be added to the pattern catalog. Rather they observe and document these problems and their solutions across multiple projects.

When a new problem is encountered, rather than documenting it immediately, we try to identify weather this problem and a solution has been already available existing pattern catalogs. If not based on their relevance and the context these candidate patterns will be turned into the standard patterns.

1.3 Pattern Template

A pattern template contains many sections describing the various aspects related to the pattern. Every pattern will be given a name to refer it and communicate about it with other people in the community.

In general a J2EE pattern template may contain the following sections:

a) **Problem**: Describes the issues that a developer has faced.

b) **Forces**: List of reasons that makes the developer forces to use that pattern, justification for using the pattern.

c) **Solution**: Describes briefly what can be done to solve the problem.

   I. Structure: Diagrams describing the basic structure of the solution.

   II. Strategies: Provides code snippets showing how to implement it

d) **Consequences**: Describes result of using that pattern. Pros and cons of using it.

e) **Related patterns**: This section lists other related patterns and their brief description around it.

1.4 Pattern Catalog

Patterns, builds recurring solution for a problem in a context. As they could be multiple problems, we ended up in having several patterns to resolve. Even they are multiple problems; based on their nature we can classify them into groups. For e.g. all these patterns are used to solve the problems that encounter in a presentation-tier are called presentation-tier catalog patterns.

But there is no particular process in place to identify or classify them in groups.

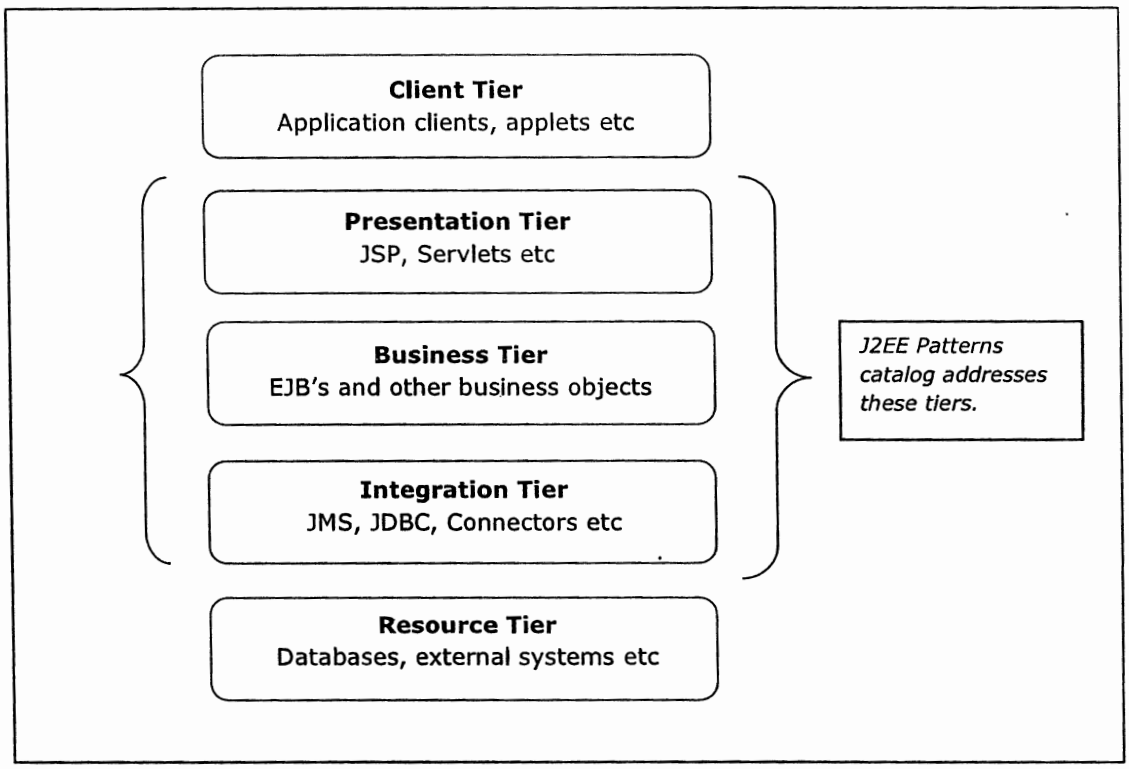
1.4.1 GOF Pattern catalog

When it comes to GOF patterns they grouped together related patterns and came up with catalog of patters as shown below.

Creational Patterns It provides guidelines to instantiate a single or group of objects so, called Creational Patterns	➢ Abstract Factory ➢ Builder ➢ Factory Method ➢ Prototype ➢ Singleton
Structural Patterns It provides a manner to define relationship between classes	➢ Adapter ➢ Bridge ➢ Composite ➢ Decorator ➢ Façade ➢ Flyweight ➢ Proxy
Behavioral Patterns It defines communication between classes and objects	➢ Chain of Responsibility ➢ Command ➢ Interpreter ➢ Iterator ➢ Mediator ➢ Memento ➢ Observer ➢ State ➢ Strategy ➢ Template Method ➢ Visitor

1.4.2 J2EE Pattern catalog

J2EE Platform/applications are multitiered system; we view the system in terms of tiers. A tier is a logical partition of related concerns. Tiers are used for separation of concerns. Each Tier handles its unique responsibility in the system. Each tier is a logical separation and has minimal dependency with other tiers. So, if we look into a J2EE Application it can be viewed as five several tiers as follows.



Five Tier Model of logical separation of concerns into tiers.

In J2EE the patterns are divided according to the functionality and the J2EE pattern catalog contains the following patterns.

Presentation Tier This tier contains all the presentation tier logic required to service the clients that access the system	➢ Intercepting Filter ➢ Front Controller ➢ Context Object ➢ Application Controller ➢ View Helper ➢ Composite View ➢ Service to worker ➢ Dispatcher View
Business Tier This tier provides business service required by the application clients	➢ Business Delegate ➢ Service Locator ➢ Session Façade ➢ Application Service ➢ Business Object ➢ Composite Entity ➢ Transfer Object ➢ Transfer Object Assembler ➢ Value List Handler
Integration Tier This tier is responsible for communicating with external resources and systems such as data stores etc.	➢ Data Access Object ➢ Service Activator ➢ Domain Store ➢ Web Service Broker

This material addresses few design patterns of GOF and few of J2EE Design patterns. Let's proceed in understanding one pattern after another.

Core/GOF Patterns

DURGA SOFTWARE SOLUTIONS, 23/3RT, Near Andhra Bank Opp: Reebok Show room, S.R Nagar, Cell. 9246212143. Phone: 040-64512786

2 Core/GOF Design Patterns

We are going to discuss Core/GOF patterns in the following sections.

2.1 Singleton

A singleton design pattern is the most popular and the older design pattern within the design pattern catalog. If you 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.

2.1.1 How to create a class as singleton

If you want to make a class as singleton you need to do the following

a) Declare the constructor of the class as private:- When we 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)

b) Declare a static method: - As you made the constructor of the class as private, no one from outside the class can call the constructor to create the object. 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 (which is not possible). So to call this method without the object, declare this method as static method.

As this method contains only the logic for creating the object, this method is also called as factory method (static factory method).

c) 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? That's where when you 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.

Let's take a look at sample piece of code to understand it better.

```
package com.sp.pattern;

public class DateUtil {
 // declare a static member of the same class-type in the class
 private static DateUtil instance;

 // construct is declared as private
 private DateUtil() {
        // no-op
 }

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

We can write the above piece of code in various other ways and there are many ways to improve it. Let's try to take a look at those in the below sections.

2.1.1 Eager Initialization

In the above piece of code we are instantiating the instance on the first call to the getInstance() method. Instead of delaying the instantiation till we call the method, we can instantiate it eagerly much before when class is loaded into the memory as shown below.

```
package com.sp.pattern;

public class DateUtil {
        // declare a 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

        // construct is declared as private
        private DateUtil() {
                // no-op
        }

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

2.1.2 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 one instances of the class. To avoid this we can even declare that method as synchronized.

```
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 conditional 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 as we can. To fix this we can have double level check for the condition as shown below.

```
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;
        .. contd
}
```

2.1.3 Static block Initialization

If you have idea around static blocks concept in java, we can use the same concept to instantiate the singleton class as shown below.

```
package com.sp.pattern;

public class DateUtil {
        // declare a static member of the same class-type in the class
        private static DateUtil instance;

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

        // construct is declared as private
        private DateUtil() {
                // no-op
        }

        // 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 before the hand.

It seems like with the above ways we understood the best possible ways 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-serialization process will creates as many numbers of objects for singleton class which avoids the rule of singleton.

Let's take a look at the below piece of code which is trying to serialize and de-serialize our singleton class.

```
package com.sp.test;

import com.sp.pattern.DateUtil;

public class SPTest {

  public static void main(String[] args) throws FileNotFoundException,
                IOException, ClassNotFoundException {
        DateUtil du1 = DateUtil.getInstance();

        ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(
                        new File("d:\\dateUtil.ser")));
        oos.writeObject(du1);

        DateUtil du2 = null;

        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(
                        new File("d:\\dateUtil.ser")));
        du2 = (DateUtil) ois.readObject();

        System.out.println("du1 == du2 : ? " + (du1 == du2));
        // 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 the singleton class. The de-serialization process will call readResolve() method 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.

Below is the complete fragment of code showing how to fix the above described problem.

```
package com.sp.pattern;

import java.io.Serializable;

public class DateUtil implements Serializable {
  // declare a static member of the same class-type in the class
  private static volatile DateUtil instance;

  // construct is declared as private
  private DateUtil() {
        // no-op
  }

  // 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) {
                          instance = new DateUtil();
                    }
              }
        }
        return instance;
  }

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

2.1.4 Override clone method and throw CloneNotSupportedException

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 cone() method and throw exception as shown below.

```
package com.sp.pattern;

import java.io.Serializable;

public class DateUtil implements Serializable, Cloneable {
	// declare a static member of the same class-type in the class
	private static volatile DateUtil instance;

	// construct is declared as private
	private DateUtil() {
		// no-op
	}

	// 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) {
					instance = new DateUtil();
				}
			}
		}
		return instance;
	}

	protected Object readResolve() {
		return instance;
	}

	@Override
	protected Object clone() throws CloneNotSupportedException {
		throw new CloneNotSupportedException();
	}
}
```

2.1.5 When to use singleton

With the above it seems like we understood the best possible way of writing a singleton class. Let's try to understand in which circumstances we need to go for a singleton class.

a) When a class has absolutely zero state. The methods of the class are not using any of the state of the class; rather the outcome of execution of the method depends on the parameter values with which you called the method. In such case you can declare that class as singleton.

b) When a class has some state and it has some methods. The methods of the class are using the state of the class. But the state the class contains is

completely read-only, which means if we create any number of objects of the class all those objects are going to represent the same state. So the outcome of the method execution doesn't depends on the state of the class rather would depend on the values with which you called the method. So, we can make such kind of classes also as singleton.

c) When a class has some state, and it has some methods. The methods are using the state of the class. The state the class is not read-only rather the state is a sharable state, which means every other class in my application should see the same state of the object. In such cases we don't need to create multiple objects, rather one instance of the class can be shared across multiple class in the application.

But in this case the state the class is holding is a common state, we need to synchronize the read and write access to the class by making the methods of the class as synchronized to avoid concurrency issues.

2.2 Factory

Not all objects in the java can be created out of new operator. Few objects may be created out of new; 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 drivers 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 abstracted 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.

Let's consider one more example, You have a pizza shop which sells pizza.

```
package com.fp.pattern;

public class Pizza {
  private int productCode;
  private float price;

  public void prepare() {
    System.out.println("preparing...");
  }

  public void bake() {
    System.out.println("baking...");
  }

  public void cut() {
    System.out.println("cutting...");
  }

   //setters and getters
  // toString()
}
```

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

```
package com.fp.pattern;

public class SpicyPizza extends Pizza {
  protected String spicyType;

  //setters and getters
  // toString()

}
```

```
package com.fp.pattern;

public class CheesePizza extends Pizza {
  private String cheeseType;

  // setters and getters
  // toString()
}
```

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

```
package com.fp.pattern;

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

        if (type.equals("normal")) {
              p = new Pizza();
        } else if (type.equals("cheese")) {
              p = new CheesePizza();
        } else if (type.equals("spicy")) {
              p = new SpicyPizza();
        }

        p.prepare();
        p.bake();
        p.cut();

        return p;
  }
}
```

Now demand for adding more pizza types came and we added TomotoPizza, 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 as shown below.

```
package com.fp.pattern;

public class PizzaFactory {
  public static Pizza createPizza(String type) {
        Pizza p = null;

        if (type.equals("normal")) {
              p = new Pizza();
        } else if (type.equals("cheese")) {
              p = new CheesePizza();
        } else if (type.equals("spicy")) {
              p = new SpicyPizza();
        }

        return p;
  }
}
```

Modified PizzaStore

```
package com.fp.pattern;

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

        p = PizzaFactory.createPizza(type);
        // call factory method to get the pizza

        p.prepare();
        p.bake();
        p.cut();

        return p;
  }
}
```

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

2.3 Factory Method

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 com.fm.pattern;

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 com.fm.pattern;

public class AltoCar extends Car {

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

}
```

```
package com.fm.pattern;

public class WagnorCar extends Car {

  @Override
  public void drive() {
        System.out.println("Driving waganor");
  }

}
```

These car's will be manufactured across various maruthi manufacturing units across india. Every manufacturing unit may not manufacture all the car models. For e.g. GujaratMaruthi manufacturing unit only manufactures alto cars and PuneMaruthi manufacturing unit only manufactures Waganor and Swift as shown below.

```
package com.fm.pattern;

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

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

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

        return car;
  }
}
```

```
package com.fm.pattern;

public class GujaratWorkshop {
  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 mechanisms 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 as shown below.

```
package com.fm.pattern;

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

        car = createCar(type);

        // assembling the car
        System.out.println("attach wheels");
        System.out.println("fit seats");
        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.

2.4 Abstract Factory

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 created two different factories. DBDAOFactory and XMLDAOFactory, these factories takes care of creating the Dao's of their type as shown below.

```
package com.af.pattern;

public class XMLEmpDao extends Dao {
  public void save() {
        System.out.println("Employee has been written to XML File");
  }
}
```

```
package com.af.pattern;

public class DBEmpDao extends Dao{
  public void save() {
        System.out.println("Employee has been written to table");
  }
}
```

```
package com.af.pattern;

public class DBDepDao extends Dao{
  public void save() {
        System.out.println("Department has been written to table");
  }
}
```

```
package com.af.pattern;

public class XMLDepDao  extends Dao{
  public void save() {
        System.out.println("Department has been written to XML File");
  }
}
```

```
package com.af.pattern;

public abstract class Dao {
  public abstract void save();
}
```

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

```
package com.af.pattern;

public class XMLDaoFactory extends DaoFactory {

  @Override
  public Dao createDao(String type) {
        Dao dao = null;

        if (type.equals("emp")) {
              dao = new XMLEmpDao();
        } else if (type.equals("dep")) {
              dao = new XMLDepDao();
        }
        return dao;
  }

}
```

```
package com.af.pattern;

public class DBDaoFactory extends DaoFactory {

  @Override
  public Dao createDao(String type) {
        Dao dao = null;

        if (type.equals("emp")) {
              dao = new DBEmpDao();
        } else if (type.equals("dep")) {
              dao = new DBDepDao();
        }
        return dao;
  }

}
```

```
package com.af.pattern;

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 com.af.test;

import com.af.pattern.Dao;
import com.af.pattern.DaoFactory;
import com.af.pattern.DaoMaker;

public class AFTest {

 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.

2.4.1 Difference between AbstractFactory and FactoryMethod

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

2.5 Template Method

Template method design pattern is a behavioral design pattern of the Gang of Four 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 as shown below.

```
package com.tm.pattern;

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 com.tm.pattern;

public class XMLDataRenderer extends DataRenderer {

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

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

}
```

```
package com.tm.pattern;

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 XMLDataRenderer or TextDataRenderer and has to call the render() method which delegates the call to readData() and processData() as per the algorithm to render it as shown below.

```
package com.tm.test;

import com.tm.pattern.DataRenderer;
import com.tm.pattern.XMLDataRenderer;

public class TMTest {
  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()
```

2.5.1 Points to remember

- 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 subclasses do need to re-write the same algorithm.

2.6 Adapter

Adapter 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 converters 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 temperate in that 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 Adapter class which takes the zipCode and maps to an city and passes this city name as input in finding the temperature at that location as shown below.

```
package com.adap.pattern;

public interface IWeatherFinder {
  public int find(String city);
}
```

```
package com.adap.pattern;

public class WeatherFinder implements IWeatherFinder {

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

}
```

The above class contains the logic for finding the temperate 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 adapter class as shown below.

```
package com.adap.pattern;

public class WeatherAdapter {
  public int findTemparature(int zipCode) {
        // maps this zipcode to city name
        String city = null;

        // acutally looks into a source (db or file)
        if (zipCode == 5353) {
              city = "hyderabad";
        }
        IWeatherFinder finder = new WeatherFinder();
        return finder.find(city);
  }
}
```

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

```
package com.adap.pattern;

public class WeatherWidget {

  public void showTemperature(int zipCode) {
        WeatherAdapter wa = new WeatherAdapter();

        System.out.println(wa.findTemparature(zipCode));
  }
}
```

Even adapter 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.

2.7 Flyweight

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 com.fw.pattern;

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

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

```
package com.fw.pattern;

public class Circle implements IShape {
  private String label;
  private int radius;
  private String fillColor;
  private String lineColor;

  public Circle() {
        label = "circle";
  }
  // setters & getters

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

}
```

```
package com.fw.pattern;

public class Rectangle implements IShape {
  private String label;
  private int length;
  private int breath;
  private String fillStyle;

  public Rectangle() {
        label = "rectangle";
  }
  // setters & 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 breadth to draw these shapes as shown below.

```
package com.fw.pattern;
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();
              }
        }
  }
}
```

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 shape 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 cricles 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 identify 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 com.fw.pattern;

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

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

```
package com.fw.pattern;

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 com.fw.pattern;

public class Rectangle extends IShape{
 private String label;

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

 @Override
 public void draw(int length, int breadth, String fillStyle) {
        System.out.println("drawing " + label + " with length : " + length
                        + " breath : " + breadth + " 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. The code fragment for the factory has been shown below.

```
package com.fw.pattern;

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 com.fw.pattern;

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");
              }
        }
  }
}
```

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

2.7.1 Key Points

a) If the object overhead is 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

b) Remove the extrinsic (non-sharable) state of the class and pass it as arguments to the parameters

c) Create a factory through which we can reuse the objects that creating new

d) The clients must use factory instead of creating the objects out of new operator

2.8 Command

Command design pattern is the one Behavioral Design pattern from Gang Of 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

a) Command:- It is an interface with execute method. It acts as a contract.

b) Client:- Client instantiates an concrete command object and associates it with a receiver

c) Invoker: - He instructs the command to perform an action.

d) Concrete Command: - Associates a binding between receiver and action.

e) 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 Controll.

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 wherein 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 com.cmd.pattern;

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

Implementing this interface we have two commands, PowerOn and PowerOff shown below.

```
package com.cmd.pattern;

//command, encapsulated with reciever 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 com.cmd.pattern;

// command, encapsulated with reciever to perform action
public class PowerOff implements Command {
  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 com.cmd.pattern;

// 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 handle that action.

```
package com.cmd.pattern;

public class RemoteControl {
  private Command command;

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

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

Finally client is the person who issues a command on the invoker.

```
package com.cmd.test;

import com.cmd.pattern.PowerOff;
import com.cmd.pattern.PowerOn;
import com.cmd.pattern.RemoteControl;
import com.cmd.pattern.Television;

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();

        // command setup with receiver
        PowerOff offCmd = new PowerOff(television);
        control.setCommand(onCmd);
        control.pressButton();
  }
}
```

Ex Ice cream + nuts

2.9 Decorator

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 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.

2.9.1 Key Points

a) Add and remove additional functionalities or responsibilities to an object dynamically at runtime, without affecting the other objects.

b) Usually these decorators are designed on component interface, so you can select the Object that has to be decorated at runtime

c) 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:

a) Component: - Pizza is the base interface.

b) Concrete component: - NormalPizza is the concrete implementation of the Pizza interface.

c) Decorator:- is the abstract class who holds the reference of the component and also implements from the component interface

d) Concrete Decorator: - Who implements from the abstract decorator and add additional responsibilities to the Concrete component.

```
package com.dec.pattern;

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

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

```
package com.dec.pattern;

public class NormalPizza implements Pizza {

	@Override
	public void bake() {
		System.out.println("Backing 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 Decorator implementing from Pizza and has reference of pizza to add toppings.

```
package com.dec.pattern;

public abstract 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 topping shown below.

```
package com.dec.pattern;

public class TomatoPizzaDecorator extends PizzaDecorator {

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

  @Override
  public void bake() {
        super.bake();
        addTomatoTopping();
  }

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

}
```

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

```
package com.dec.test;

import com.dec.pattern.NormalPizza;
import com.dec.pattern.Pizza;
import com.dec.pattern.TomatoPizzaDecorator;

public class Shop {

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

}
```

Presentation-Tier Patterns

3 Presentation-Tier patterns

These are the patterns that deal with presentation-tier aspects of the application. They receive request from the clients and delegates to other classes to perform business logic and forwards the request to present the view to the client.

Let's us try to understand various patterns around these.

3.1 Intercepting Filter

The presentation-tier request handling receives many different types of requests, which require various types of processing. Some requests are simply forwarded to the appropriate target component, while other request must be modified, audited, or uncompressed before being further processed.

Preprocessing and post-processing of a client request and response are required.

3.1.1 Where to use and benefits

- Logging and authentication
- Enhance security
- Add additional function to existing web application
- Decorate main process
- Debug
- Pre-processing and post-processing for specific clients
- Uncompress incoming request
- Convert input encoding schema
- Is the client's IP address from a trusted network?
- Do we support the browser type of the client?
- Does the client have a valid session?
- Being added or removed transparently or declaratively and triggered automatically
- Improve reusability
- Deployment-time composability
- Each filter is loosely coupled
- Inefficient for information sharing

3.1.2 Structure

1. Filter Manager: - The filter manager manages filter processing. It creates the FilterChain with appropriate filters in an order and initiates the processing.

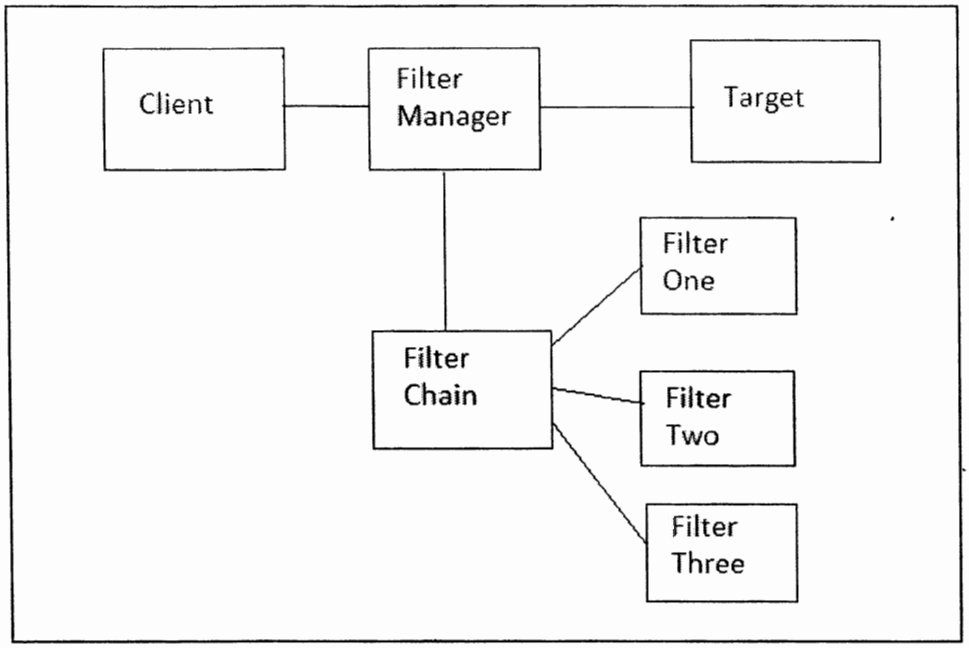
2. FilterChain: - It is the collection of independent filters

3. Filter: - These are the filters that are mapped to the target. The FilterChain coordinates their processing.

4. Target: - The target is the end resource request for processing by the client.

5. Client: - He is the one who sends the request for the target.

Structure



In a classic solution it consists of series of if-else-if checks, and it any of the series of checks fails will abort the request. Nested-If-else-if is the standard strategy which might leads to duplicate code copied across the components and may cause maintenance issues.

The solution for this is to provide flexible and configurable filters which can be added or removed via configuration with no code changes. A component is invoked with in the flow once all the filters in flow are processed.

Let's take an example to understand it better. We want the user to browse our application only using Firefox and if he uses any other browser in accessing the application we need to display an "Un-supported browser" message not allowing him to access it.

Browser detection logic is something that should be applied to not one request of the application; it has to be applied across all the requests that are coming into the application. If we write this logic as part of servlet in our application, then we end up in writing this logic in every servlet of the application which leads to code duplication and maintenance issues. Instead if we code a filter and apply to all the request of the application, we can apply this across all the request of the application. These are flexible and driven through configuration and at any point of time we can remove or add a new filter easily by changing configuration.

Below code fragment depicts the same.

```
package com.iw.filter;

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;
import javax.servlet.http.HttpServletResponse;

public class UserAgentFilter implements Filter {
	private Logger logger;
	private FilterConfig fConfig;

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

	@Override
	public void doFilter(ServletRequest request, ServletResponse response,
			FilterChain chain) throws IOException, ServletException {
		String userAgent = null;

		// request
		userAgent = ((HttpServletRequest) request).getHeader("User-Agent");
		logger.info("User Agent : " + userAgent);

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

		// response
	}

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

```
}
package com.iw.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;

public class HomeServlet extends HttpServlet {

	@Override
	protected void service(HttpServletRequest request,
			HttpServletResponse response) throws ServletException,
IOException {
		request.setAttribute("loggedInUser", "John");
		RequestDispatcher dispatcher =
request.getRequestDispatcher("home.jsp");
		dispatcher.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="listEmp.do">
			Logged In ${loggedInUser}
			<input type="hidden" name="token" value="${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

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	id="WebApp_ID" version="2.5">
	<display-name>InterceptingWeb</display-name>
	<filter>
		<filter-name>UserAgent</filter-name>
		<filter-class>com.iw.filter.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>com.iw.servlet.HomeServlet</servlet-class>
	</servlet>
	<servlet-mapping>
		<servlet-name>HomeServlet</servlet-name>
		<url-pattern>/home.do</url-pattern>
	</servlet-mapping>
</web-app>
```

3.2 View Helper

You want to separate view from processing logic. Developers often mingle business logic around view presentation logic to display dynamic business data to users.

When we wrote presentation logic, business logic and formatting logic together, as presentation-tier often tend to change, maintaining these would be difficult. This yields in poor separation of roles among web developers and software developers.

Forces

- Embedding business logic in the view promotes copy-paste of reuse. This causes maintenance problems and bugs as the logic has been duplicated across different views.
- It is desirable have a clean separation of roles between web developers and software developers
- One view is commonly used to respond to a particular business requirement.

Solution

There are multiple strategies in implementing the view component.

- A servlet itself can directly render the view as part of its logic, but this ends up in mixing the controller or business logic with presentation logic. Always it is recommended to separate presentation from business or controller logic. As presentation logic always tend to change, the web developers often need to modify the code in Servlets which creates a level of confusion in development.
- Using a jsp as a view strategy, If you want to do a template base rendering always the other option is to use jsp as a presentation. Even it is recommended to write view as part of jsp, sometimes you may end-up in embedding scriplet code to evaluate decision in displaying the view, which is not recommended. For example you may write some helpers classes in scriplets in displaying the views as shown below.

```
<jsp:useBean id="adminHelper" scope="request"
class="com.vh.pattern.AdminHelper"/>
<HTML>
 <HEAD>
        <BODY>
              <%if(adminHelper.isRoleAdmin()){
                    <%—display this content --%>
              }
        </BODY>
 </HEAD>
</HTML>
```

- Another option of presenting the view to the user is using a Custom Tag Helper. Using these view helpers promotes clear separation of the view from the business processing in an application.

  Using a custom tag library involves more upfront work than does using the JavaBean helper strategy. As working on Custom Tag development is treated often complicated when compared with JavaBean helper strategy.

  Below is the code snippet that helps you in understanding how to write custom tag as a view helper.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://presentationtier/tags/viewhelper" prefix="pt"%>
<!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>Presentation Tier</title>
      </head>
      <body>
            <pt:viewStudent id="10"></pt:viewStudent>
      </body>
</html>
```

```
package com.ct.view.tags;

import java.io.IOException;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;

public class ViewStudTag extends TagSupport {
	private String id;

	@Override
	public int doStartTag() throws JspException {
		JspWriter out = null;

		out = pageContext.getOut();
		try {
			out.print("Student details : " + id);
		} catch (IOException e) {
			e.printStackTrace();
		}

		return SKIP_BODY;
	}

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

}
```

```
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
	<tlib-version>1.0</tlib-version>
	<jsp-version>1.2</jsp-version>
	<short-name>c</short-name>
	<uri>http://presentationtier/tags/viewhelper</uri>
	<display-name>Presentation Tier pattern</display-name>
	<description>Custom Tag helper</description>

	<tag>
		<name>viewStudent</name>
		<tag-class>com.ct.view.tags.ViewStudTag</tag-class>
		<body-content>scriptless</body-content>
		<attribute>
			<name>id</name>
			<required>true</required>
			<type>java.lang.String</type>
		</attribute>
	</tag>
</taglib>
```

Advantages

- Improves application partitioning, reuse and maintainability
- Improves role separation

3.3 Front Controller

The presentation-tier request handling mechanism must control and coordinate processing each user request. Such control mechanism might be in either centralized or de-centralized manner.

The system requires a centralized access point for presentation-tier request handling to support the integration of system services, content retrieval and view management and navigation. When the user access the view directly without going through a centralized mechanism two problems occurs.

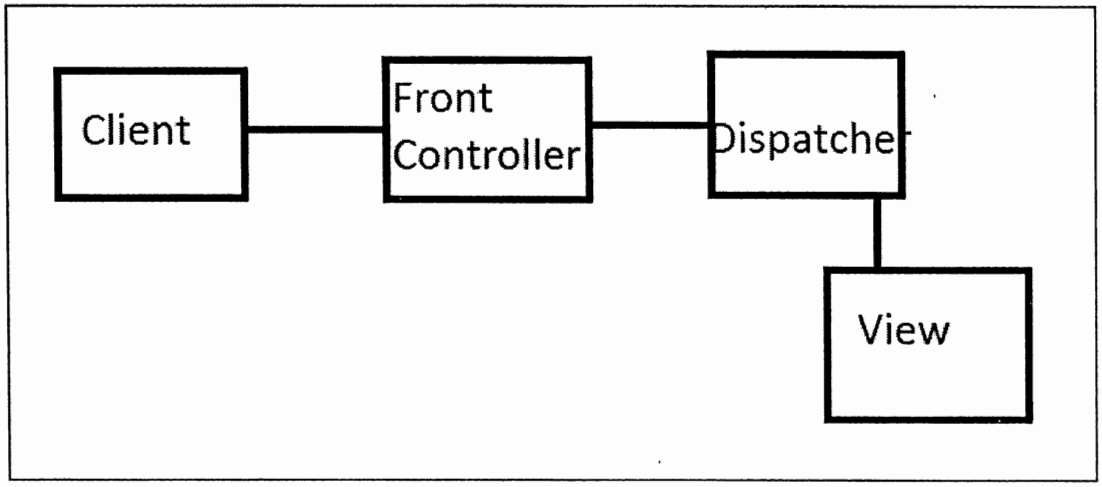
- Each view is required to provide its own system services, often results in duplicate code.
- View navigation is left to views which make the system difficult to understand.

Solution

Use a controller as the initial point of contact for handling a request. The controller manages to provide system services like security, authentication and authorization before delegating business processing, managing the choice of appropriate view and handling errors.

The main advantage of going for a front controller is reducing the amount of code that has to be duplicated across various views as part of the application. Typically a Controller coordinates with dispatcher component. Dispatchers are responsible for view management and navigation.

Structure



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 com.fc.command;

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

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

```
package com.fc.command;

import com.fc.valueobject.StudVO;

public class StudentViewCommand implements Command {

	@Override
	public String execute(HttpServletRequest request,
			HttpServletResponse response) {
		String page = null;
		StudVO studVO = null;

		// query data using delegate and dao populate in Value Object
		studVO = new StudVO(1, "John");

		request.setAttribute("stud", studVO);
		page = "showStud";

		return page;
	}

}
```

```
package com.fc.controller;

import java.io.IOException;

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

import com.fc.command.Command;
import com.fc.helper.CommandHelper;
import com.fc.helper.Dispatcher;

public class FrontController extends HttpServlet {

	@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 com.fc.helper;

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

```
package com.fc.helper;

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

	private String mapPageToView(String page) {
		if (page.equals("showStud")) {
			return "viewStud.jsp";
		}

		return null;
	}
}
```

```
package com.fc.valueobject;

public class StudVO implements Serializable {
	private int id;
	private String name;

	public StudVO(int id, String name) {
		super();
		this.id = id;
		this.name = name;
	}
	// setters and getters
}
```

viewStud.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<jsp:useBean id="stud" scope="request" beanName="studVO"
type="com.fc.valueobject.StudVO"/>
<!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="stud"/><br/>
			Name : <jsp:getProperty property="name" name="stud"/>
		</p>
	</body>
</html>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	id="WebApp_ID" version="2.5">
	<display-name>FrontControllerWeb</display-name>
	<servlet>
		<servlet-name>FrontController</servlet-name>
		<servlet-class>com.fc.controller.FrontController</servlet-class>
	</servlet>
	<servlet-mapping>
		<servlet-name>FrontController</servlet-name>
		<url-pattern>*.do</url-pattern>
	</servlet-mapping>
	<welcome-file-list>
		<welcome-file>index.jsp</welcome-file>
	</welcome-file-list>
</web-app>
```

3.4 Composite View

Web pages present different types of content from various sources. There in-turn comprises of multiple sub views in displaying a page. These pages will be coded and maintained by different skill set developers who are experts in those areas.

The problem here is some parts of the presentation will be repetitive. Rather than constructing the complete page with various atomic views, pages are being built with embedding the entire code in itself.

- The atomic portions of the view content changes frequently
- Multiple composite views might use similar subviews. Such as employee table may have to be displayed in different pages with some additional text decorating around.
- Layout changes are more difficult to manage
- Embedding frequently changing template text directly into views potentially affects the productivity and administration of system.

Use composite views that are composed of multiple atomic subviews. Each component of the template may be included dynamically into the whole and the layout of the page may be managed independently of the content.

This solution can be applied by inclusive and substitution of modular dynamic and static template fragments. It promotes the reuse of atomic portions of the view. For example in a web page we will have a common header, footer and navigation pane that has to be displayed across all the pages of the application and only the body the page is going to be changed.

In such a case rather than coding the header, footer and navigation in every view we can create them as atomic view and can be included as part of the whole view by using inclusive and substitutions dynamically.

Another benefit of using it is a web developer can prototype of the layout of a site, allowing the java developers to substitute the portions of the page with actual content at the time of development.

Participants and Responsibilities

Composite View: A composite view is an aggregation of multiple sub views.

View Manager: The view manager manages to include fragments into the composite view.

Included View: It is the sub-atomic view that is being included as part of the composite view.

Strategies

1) Jsp page View Strategy

2) Servlet View Strategy

3) JavaBean View management strategy

In this we embed fragments of the view by using the helper class, he assist in implementing the custom control logic of what has to be included as part of the jsp page.

```
<%@page import="com.cv.pattern.ContentHelper%>
<% ContentHelper ch = new ContentHelper(request);%>
<table>
   <% if(ch.hasSideNav() {%>
     <!—display side navigation →
   <%}%>
</table>
```

4) Standard Tag view management Strategy

a. Composite view at Translation-time content inclusion

```
<table border=1 valign="top" cellpadding="2%"
  width="100%">
  <tr>
    <td><%@ file="header.html" %> </td>
  </tr>
  <tr>
    <td><%@ file="body.html" %> </td>
  </tr>
</table>
```

b. Composite view at Runtime-time content inclusion

```
<table border=1 valign="top" cellpadding="2%"
 width="100%">
   <tr>
      <td><jsp:include page="header.jsp"
        flush="true"/> </td>
   </tr>
   <tr>
      <td><jsp:include page="body.jsp"
        flush="true"/> </td>
   </tr>
</table>
```

Advantages

- Improves modularity and reuse
- Reduces manageability

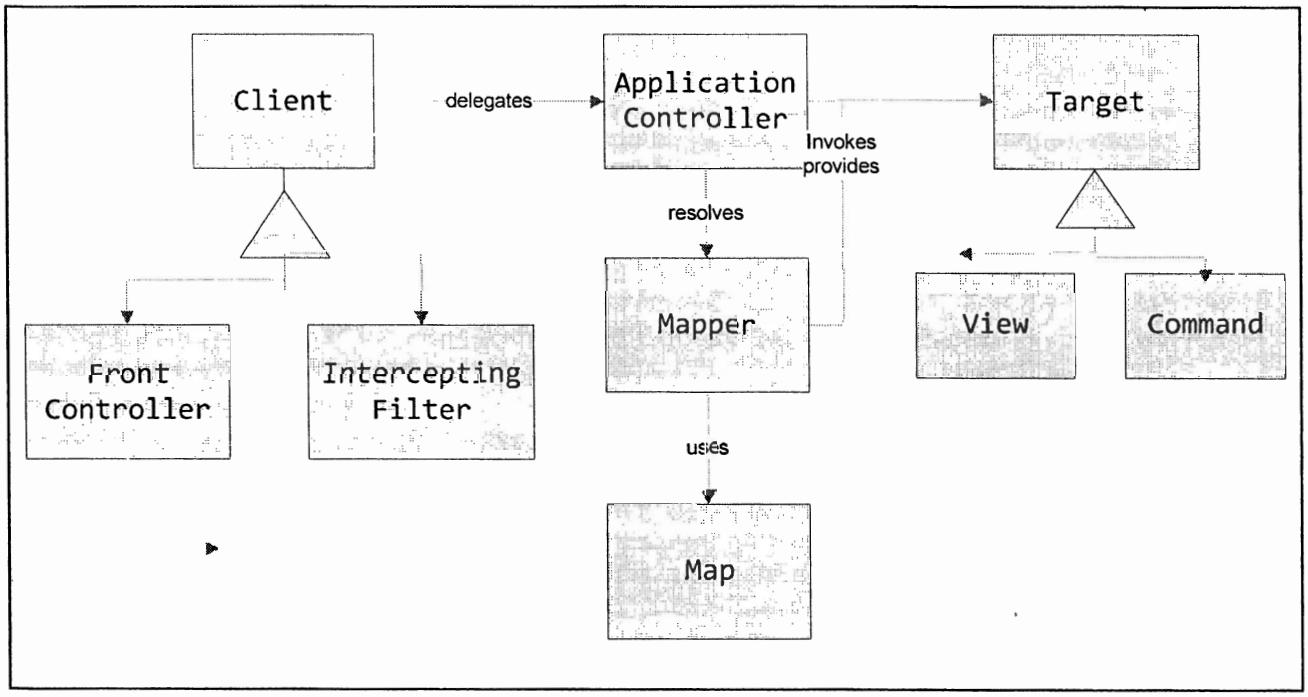
3.5 Application Controller

In the earlier section we discussed about Front controller, which applies system services or plumbing code centrally to all the requests of the application. Apart from this he will even do the action management and view management.

But if Front controller has been assigned more responsibilities, then it might go out of controller, rather we share few of the responsibilities of front controller to Application Controller.

If you want centralize the action and view management, you want to improve the request handling or want to apply application specific processing. You want to promote modularity. You want to improve the testability of the application, without a web container, all there are answered by Application Controller.

Structure



As per the above diagram, when the request has been sent by the client, first the request will be received be received by front controller and forwards to the Intercepting filter to apply pre and post processing. Front controller therefore forwards the request to Application controller for further processing. Here the application controller takes the help of Mapper to map the input values to an Command (similar to ActionForm in struts) and sends this as an input to Target (Action class in struts) for performing the action. Once the Target has performed the action it will returns the view that has to be presented to the client. The view will be taken as input by Application Controller and maps to a Page to render it to the user.

Even sometimes the application controller will be designed as protocol agnostic. In such a case mapping of the protocol specific request to the command will be taken care by front controller. This helps in testing your code independent of protocol.

Advantages

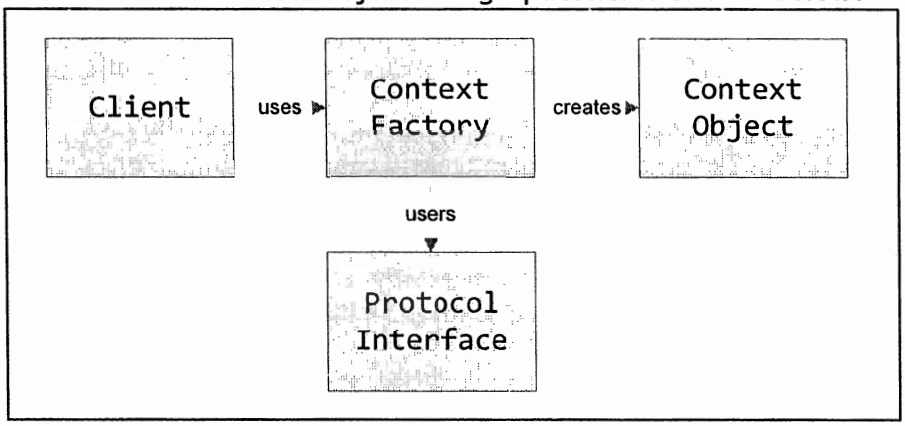
- Improves modularity
- Improves testability
- Improves extensibility

We will consider an example to understand it better with an combination of Context Object design pattern.

3.6 Context Object

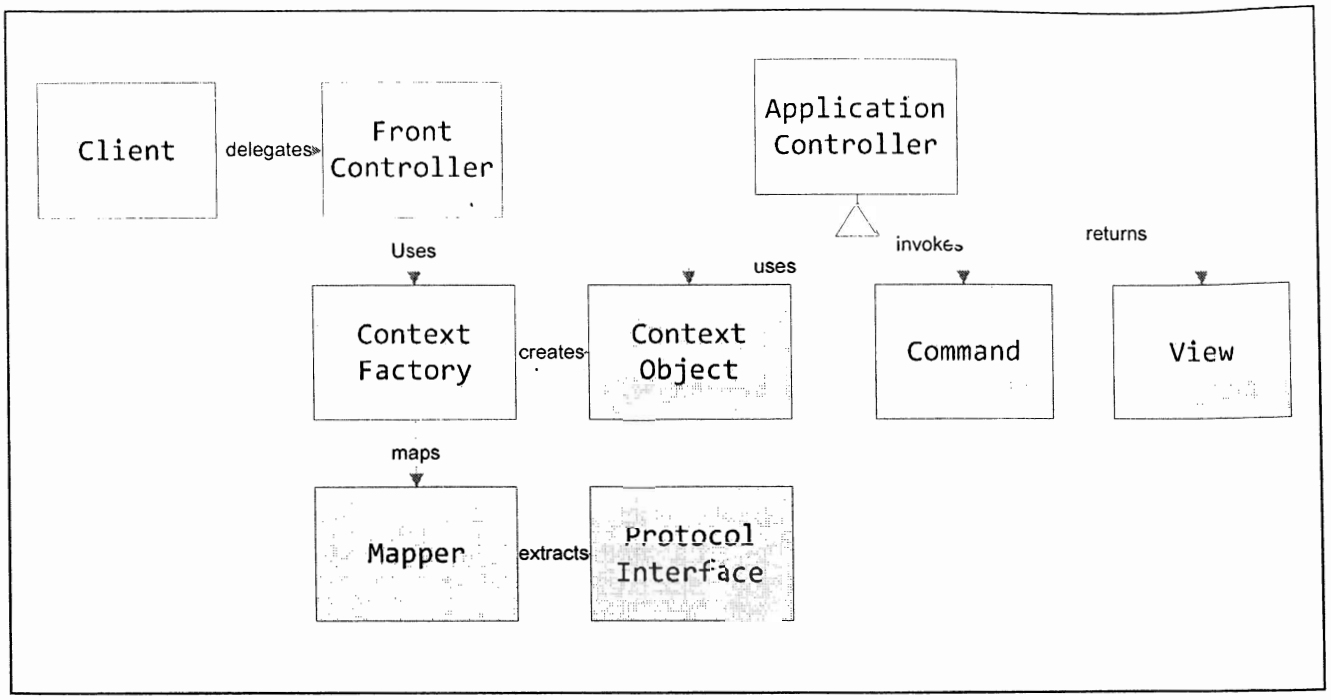
We want our application code to be independent of protocol, we want our main application logic to be free from protocol or web server and want to test it independent of protocol, and then we can go for Context Object.

Typical structure of the Context Object design pattern is shown below.



In the above diagram when the client sends the request, it will be received by Front controller, it turn might take the help of ContextFactory, it will extracts the data from protocol specific interface and populates the data into Context Object by creating it.

The below flow depicts the combination of Application Controller and Context Object together.



As we want our application controller to be independent of protocol, the mapping of Protocol specific interface to an context object will be done by front controller using the Context Factory. Now the Context Object will be passed as input to the Application Controller to perform the action and view management.

Now application controller identifies the Command (Action class in struts) to process the request and delegates/dispatches the view to the client that is being returned by Command.

Following code depicts the above flow.

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</title>
        </head>
        <body>
                <a href="viewStudent.do?id=10">Show Student Information</a>
        </body>
</html>
```

studdetail.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<jsp:useBean id="stud" scope="request" type="com.ac.valueobject.StudVO" />
<!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:green; font-size: x-large;">
            id : <jsp:getProperty property="id" name="stud"/><br/>
            Name: <jsp:getProperty property="name" name="stud"/>
        </p>
    </body>
</html>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">
    <display-name>ApplicationControllerWeb</display-name>
    <servlet>
        <servlet-name>front</servlet-name>
        <servlet-class>com.ac.controller.FrontController</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>front</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
</web-app>
```

```
package com.ac.valueobject;

public class StudVO {
	private String id;
	private String name;

	// constructor
	// setters and getters

}
```

```
package com.ac.command;

import com.ac.context.RequestContext;

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

```
package com.ac.command;

import com.ac.context.RequestContext;
import com.ac.valueobject.StudVO;

public class ViewStudentCommand implements Command {

	@Override
	public String execute(RequestContext requestContext) {
		String id = null;
		StudVO studVO = null;

		id = requestContext.getParameter("id");
		// call delegate and dao
		studVO = new StudVO(id, "Rama");

		requestContext.setAttribute("stud", studVO);

		return "showStudent";
	}
}
```

```
package com.ac.context;

import java.util.Map;

import javax.servlet.http.HttpServletRequest;

import com.ac.mapper.HttpRequestMapper;

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 com.ac.context;

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 com.ac.controller;

import java.io.IOException;

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

import com.ac.context.ContextFactory;
import com.ac.context.RequestContext;
import com.ac.helper.Dispatcher;

public class FrontController extends HttpServlet {

	@Override
	protected void service(HttpServletRequest request,
			HttpServletResponse response) throws ServletException,
IOException {
		String view = null;
		String page = 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 com.ac.controller;

import com.ac.command.Command;
import com.ac.context.RequestContext;
import com.ac.helper.CommandHelper;

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

		commandHelper = new CommandHelper();
		command =
commandHelper.getCommand(requestContext.getRequestUri());

		view = command.execute(requestContext);

		return view;
	}

	public String mapViewToPage(String view) {
		String page = null;

		if (view != null) {
			if (view.equals("showStudent")) {
				page = "studdetail.jsp";
			}
		}

		return page;
	}
}
```

```
package com.ac.helper;
import com.ac.command.Command;
import com.ac.command.ViewStudentCommand;
public class CommandHelper {
	public Command getCommand(String uri) {
		Command command = null;

		if (uri != null) {
			if (uri.contains("viewStudent.do")) {
				command = new ViewStudentCommand();
			}
		}
		return command;
	}
}
```

```
package com.ac.helper;

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 com.ac.mapper;

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));
                }
        }
}
```

3.7 Value Object

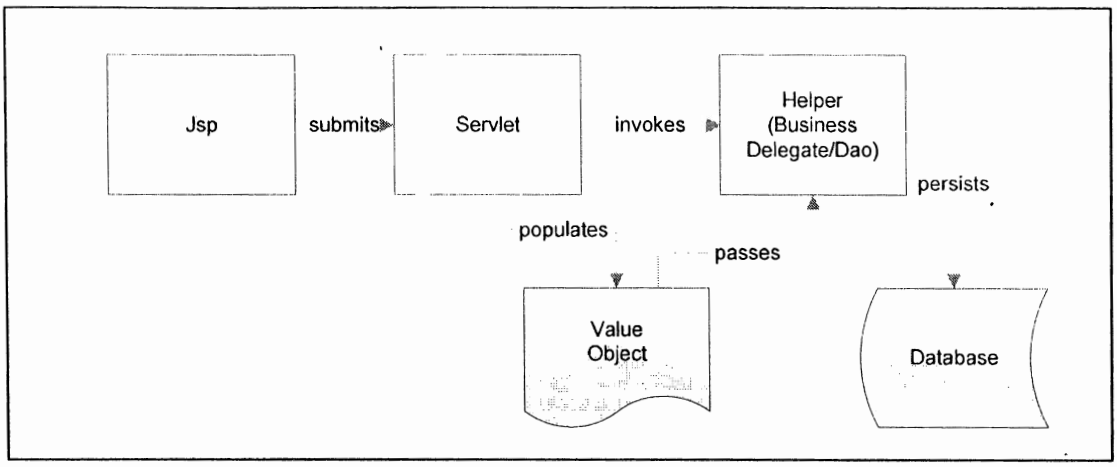
When the client has submitted the request from a jsp page, always the request will come to the servlet. The servlet can read the page submitted values using the request.getParameter(). Once all those values are retrieved by the servlet, it will not persist the data into database, rather it has to pass down these values to helper classes (Business delegate, Dao) to persist the data.

If a Jsp page has submitted 10 fields from its form, the servlet can read these values by request.getParamter() but it will not be nice to pass these 10 fields as 10 parameters in calling the methods of business delegate. Rather we can wrap these 10 submitted values into a pojo class and can pass down it as an input calling the business delegate method.

So, this pojo class which has attributes with setters/getters implementing serializable, used for transferring the data between the layers of the application or used for presenting the data to the client is called value object.

We will present an example on the value object while discussing about Business delegate.

Note: If this value object is used for transferring the data over the network between the client and the server, then it acts as transfer object.



Business & Integration-Tier Patterns

4 Business and Integration Tier patterns

4.1 Data Access Object (DAO)

DAO stands for Data access Object. In a J2EE Application, at some point we might need to read/write data into underlying persistence storage. To access/store data into the persistency storage we might need to use some API, so our high level business classes should not be exposed to the low level (legacy or third party api) in performing the persistence storage.

We want to separate presentation-tier/business-tier components from persistency. Let's say today our application is storing/accessing the data from relational database, after some time it want to persist the data into an Object oriented data base/ legacy system/ file storage, if our presentation-tier/business-tier components are performing the persistency, then any changes to the storage technology will effects these components and we might need to re-write the entire code in them.

To resolve the above said problem we need to go for DAO. DAO is also a pojo class like any other class in java. The main difference is DAO contains only the logic for Data access/storage. It can contain any number of methods like save, update, delete, get, find etc. All those methods will take some data/return some data by performing some persistency specific operations, those methods are not supposed to contain any other logic apart from persistence; even a simple business calculation is also generally not encouraged.

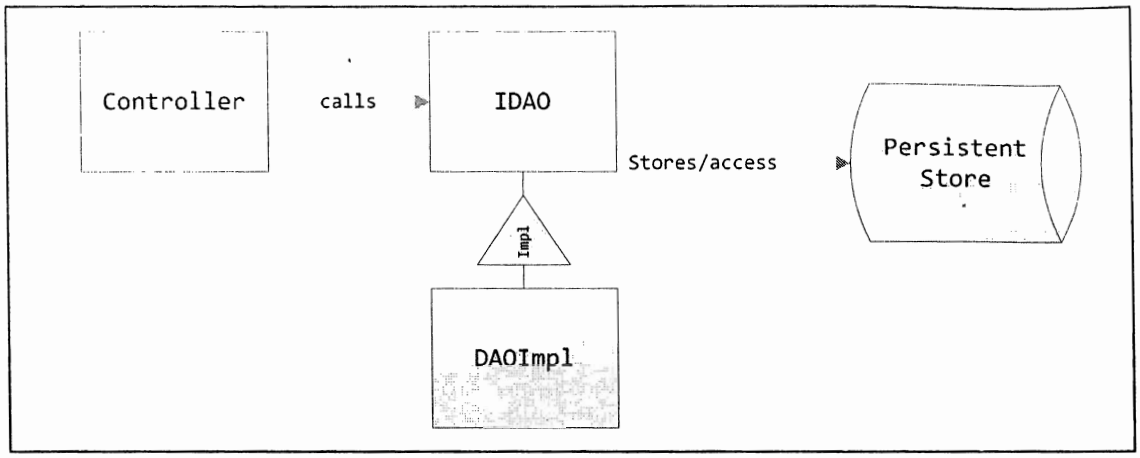
The data we pass to the DAO should be in persist-able format or the data it is returning would be in persistency storage specific format. DAO never performs any conversions/casting the data for storage or while retrieval.

If we change from one persistency storage technology to other we just need to re-write the code in the DAO, we don't need to touch other classes as part of the application.

Benefits of using Dao

- If Jsp's or beans or helper classes wants to retrieve the data from database they can use DAO.
- Persistency storage API's will change from persistency storage technology to technology, if all the classes in the application tries to perform the persistency, all the classes across the application will be coupled with those API (which are not uniform) and any changes to technology need to re-write the entire application
- Always we don't want to expose our presentation-tier components from persistency. Changes in persistency should affect only those classes.

Flow diagram depicting the usage of DAO



Following piece of code depicts the usage of DAO.

```
package com.dao.util;

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 sqe) {
        logger.throwing("ConnectionFactory",
              "Exception thrown by ConnectionFactory", sqe);
        throw sqe;
      } catch (ClassNotFoundException e) {
        logger.throwing("ConnectionFactory",
              "Not able to load driver class", e);
        throw e;
      }
      return con;
  }
}
```

```
package com.dao.pattern;
import com.dao.util.ConnectionFactory;
public class EmpDao {
	// sql queries
	private final String SQL_INSERT_EMP = "INSERT INTO
STUDENT(STUDENT_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", "hr",
"hr");
			pstmt = con.prepareStatement(SQL_INSERT_EMP);
			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 load driver", e);
			throw e;
		} finally {
			if (con != null) {
				if (isException == true) {
					con.rollback();
				} else {
					con.commit();
				}
				con.close();
			}
			if (pstmt != null) {
				pstmt.close();
			}
		}
	}
}
```

```
package com.dao.pattern;

import java.sql.SQLException;

public class EmpController {
        public void addEmp(int id, String name) throws SQLException,
                        ClassNotFoundException {
                EmpDao dao = new EmpDao();
                dao.insert(id, name);
        }
}
```

```
package com.dao.test;

import java.sql.SQLException;

import com.dao.pattern.EmpController;

public class DaoTest {
        public static void main(String[] args) throws SQLException,
                        ClassNotFoundException {
                EmpController ec = new EmpController();
                ec.addEmp(5, "Ravana");
        }
}
```

4.2 Business Delegate

We want to always separate presentation-tier components from business/persistency-tier specific components. We want to have clear separation of roles. We want to promote separation of concerns, answering all the above business delegate comes into picture.

Always the incoming request from a client-tier will be received by presentation-tier component, like a servlet. The servlet itself should never do persistency, if it does then any changes to the underlying storage technology will effects servlet, rather it should call a separate class designed for performing the persistency which is nothing but DAO.

Servlet can call the DAO? If not who has to call a DAO. If servlet calls the DAO then presentation-tier components will gets tightly coupled with persistence-tier components, to separate presentation-tier from persistency we use business delegate it between. Business delegate helps in three situations as described below.

1) Always a DAO class contains the persistency logic, if we observe carefully the persistency logic in the dao class is surrounded between try catch block, with SQLException as shown below.

```
class EmployeeDao {
    void save(EmployeeBO empBO) {
        Connection con = null;
        Statement stmt = null;
        try {
            Class.forName(driverClassName);
            con = DriverManager.getConnection(url, un, pwd);
            stmt = con.createStatement();
            // do some operation
        }catch(SqlException sqe) {
            throw sqe;
        }
    }
}
```

When we execute the above piece of code, due to some database failure if the logic doesn't work it will throw the sql exception. When the dao method is encountering the exception, does the method has to catch it and should suppress the exception and return a success value to the callee or it should re-throw the same exception back to the callee indicating he reason for the failure.

Always we should never return a success response to the callee rather we should throw the exception indicating the reason for the failure. But the sql exception you are throwing here is database neutral exception or database specific exception. An SqlException contains database specific error

information. For e.g.. SqlErrorCode and SqlErrorMesages. There are database specific values that are populated by jdbc api before throwing that exception. If it is an oracle database against which you are performing the operation then the SqlExceptions, SqlErrorCode field will contains a value something like ORA121 which is Oracle database specific error code.

Now if DAO is throwing the same SqlException to the Servlet, the Servlet has to re-throw the exception to the client or should not. We should never display ugly stacktrace to the end user. My Servlet should catch that exception and based on the exception information it has to display an appropriate error page to the user (displaying an friendly error information).

Let's say if it is ORA121, we may display one error page and it is ORA122 we may display different error page.

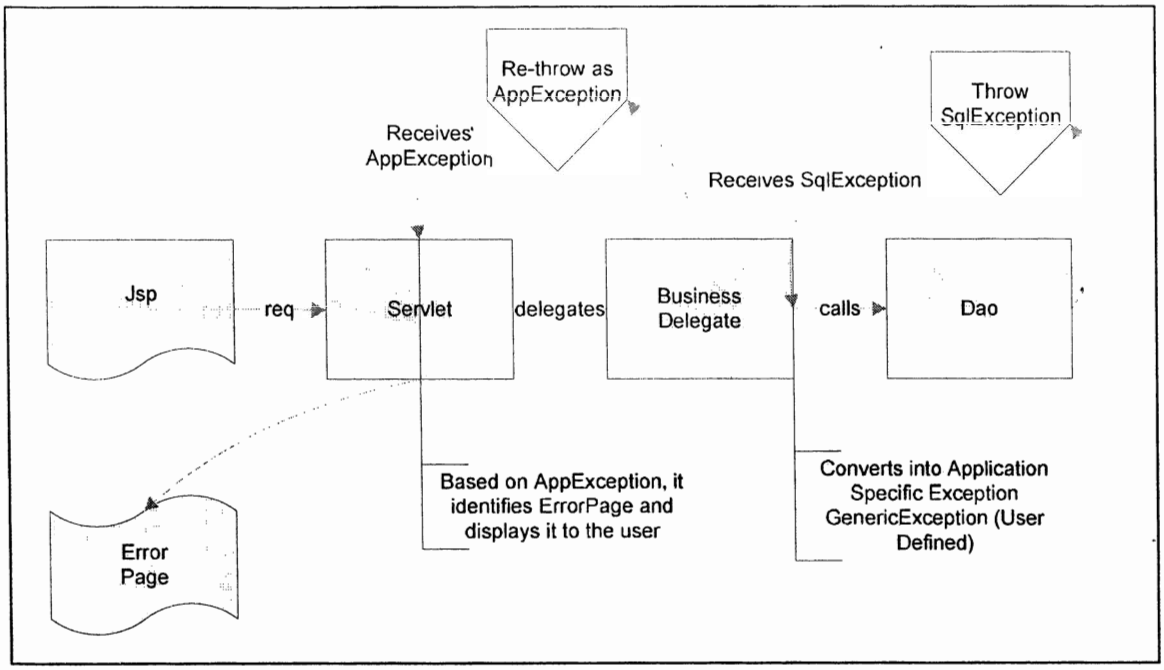
Now if we change the database from Oracle to MS Sql Server, still the Dao throws the SqlException only but it will not contains error information specific to Oracle rather error codes and messages will be specific to MS SqlServer.

With this I not only has to modify the changes in Dao, I need to even modify the servlet as well as it got exposed to database specific information. A change in persistency should not affect presentation, but now it demands to re-write/modify the presentation (Servlet).

To avoid this my Dao should not throw SqlException rather it should convert that SqlException into User/Application specific (User Defined exception). But Dao's are never designed to perform anything apart from persistency. This is where Delegate comes into picture.

Dao's throw SqlException which will be caught by Delegate and will be converted into Application Specific exceptions and throw back to servlets. If database changes I just need to modify my Dao and Delegate both belongs to business-tier. I don't have to re-write my servlet.

Following diagram depicts the same.



2) The second scenario where we use Business delegate is as follows.

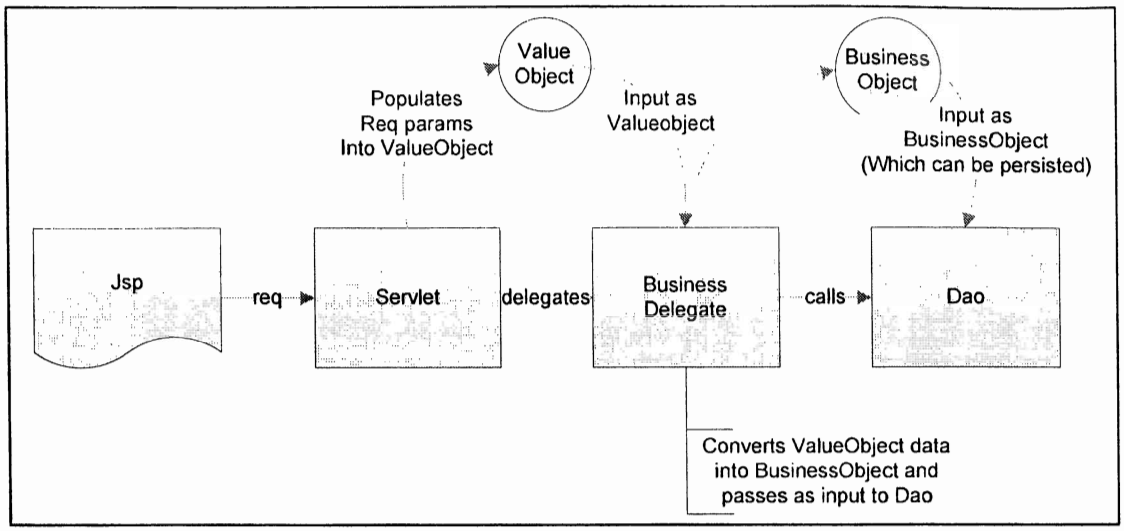
When we submit a request from Jsp page, it comes to servlet. Now servlet will tries to read the values that user has submitted as part of Jsp page through request.getParameter("paramname"); Ones he has read those values, he tries to pass these values to the next layer to perform the business logic.

If the user submit 10 input controls as part of Jsp page, the servlet will retrieves 10 values, but it will not pass these 10 values as 10 parameters to the Dao class method to perform persistency, rather it wraps into an object and will pass it as an input to other class.

This object into which it wraps the request parameters values is called value object. Value object always carries the values in presentation-tier specific format. It indicates the values we collected from req.getParam are in String format, will the servlet has to convert and should populate into Value Object and should set the same String values. Never the servlet should convert, if it converts again if the datatype of the column changed in database again the code has to be modified in Servlet. Instead the valueobject contains the attributes in String type only and will be passed as input to Dao. Can the Dao can take the value object data and can insert into database.

No, as those are string type the dao has to convert those values into database column specific types for persistency. But dao will not do anything apart from persistency, then who has to convert presentation-tier specific values into business-tier persistable format. Here comes business delegate.

Business delegate takes valueobject as input from servlet and converts those values into Business Object. This is another java bean which holds the values in persistable format. It passes this business object as input to Dao which performs the persistency. The following diagram shows the same.



3) Third usecase is in managing transactions. We perform any persistency operations by applying transactions.

In a Jdbc application we manage the transactions by writing the below code.

```
Class EmployeeDao {
    void save(EmployeeBO empBO) {
        boolean flag = false;
        Connection con = null;
        Statement stmt = null;
        try {
            con = DriverManager.getConnection();
            con.setAutocommit(false);
            // do something
        }catch(SqlException sqe) {
            flag = true;
        }finally {
            if(flag == true) { // indicates exception
                con.rollback();
            }else {
                Con.commit();
            }
        }
    }
}
```

We initially setAutocommit as false, and perform the operation, if any exception comes in the finally block we are going to rollback else we are going to commit the transaction. Let's say we want to add a new employee to the

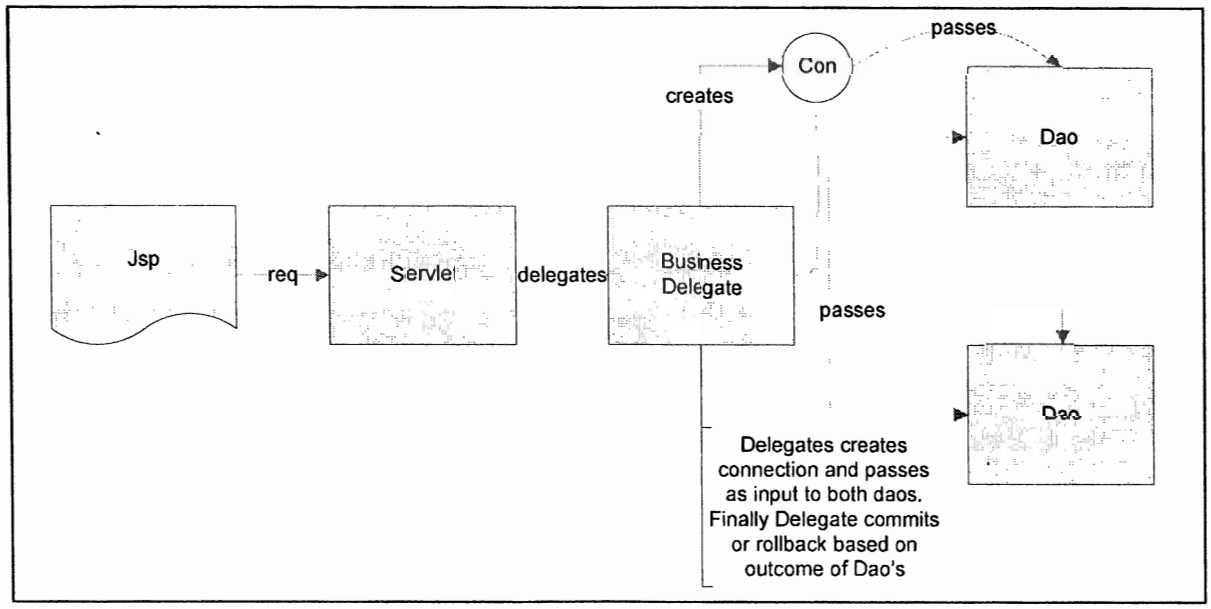
organization, as part of adding an employee we need to insert the data into two tables Employee and EmployeeDepartment indicating the employee works for which department.

Now inserting data into Employee and EmployeeDepartment has to happen as part of single transaction. If we have two Dao's performing the saving of employee information into their tables, then both the Dao's will commit the data independently irrespective of others. It means both the operations will not come under one transaction.

To hold both the operations as one transaction, the business delegate will creates the connection and passes it as input to both the dao's. Now the Dao's will uses the connection and performs the database operation, These Dao's will not close or commit the connection rather returns or throws exception based on the execution.

Now if business delegate receives an success outcome from both dao's it will commit the connection otherwise roll backs the connection in finally block.

The below diagram shows the same.



The following code sample will shows all the three scenarios we discussed in the above section

regstud.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>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"%>
<%@page isErrorPage="true" %>
<!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>Register Successfully</title>
	</head>
	<body>
		<p style="color:green;">
			Student ${studentId} register successfully....
		</p>
	</body>
</html>
```

```
package com.sms.bo;

import java.io.Serializable;
public class StudBO implements Serializable {
	private int id;
	private String name;

	//setters and getters
}
```

```
package com.sms.bo;

import java.io.Serializable;

public class StudentCourseBO implements Serializable {
        private int studentId;
        private int courseId;

        // setters and getters

}
```

db.properties (com.sms.common – package)

```
db.driverClassName=oracle.jdbc.driver.OracleDriver
db.url=jdbc:oracle:thin:@localhost:1521:xe
db.user=hr
db.password=hr
```

```
package com.sms.dao;

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

import com.sms.bo.StudBO;

public interface StudentDao {
        void insert(StudBO studBO, Connection con) throws SQLException,
ClassNotFoundException;
}
```

```
package com.sms.dao;

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

import com.sms.bo.StudentCourseBO;

public interface StudentCourseDao {
        void insert(StudentCourseBO studentCourseBO, Connection con) throws
SQLException,
                        ClassNotFoundException;
}
```

```
package com.sms.dao;

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

import com.sms.bo.StudBO;
import com.sms.util.ConnectionFactory;

public class StudentDaoImpl implements StudentDao {
	private static Logger logger;
	private final String SQL_INS_STUDENT = "INSERT INTO STUDENT(STUDENT_ID,
NAME) VALUES(?,?)";

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

	@Override
	public void insert(StudBO studBO, Connection con) throws SQLException,
			ClassNotFoundException {
		PreparedStatement pstmt = null;

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

```
package com.sms.dao;

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

import com.sms.bo.StudentCourseBO;
import com.sms.util.ConnectionFactory;

public class StudentCourseDaoImpl implements StudentCourseDao {
	private static Logger logger;
	private final String SQL_INSERT_STUD_COURSE = "INSERT INTO
STUDENTCOURSE(STUDENT_ID, COURSE_ID) VALUES(?,?)";

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

	@Override
	public void insert(StudentCourseBO studentCourseBO, Connection con)
			throws SQLException, ClassNotFoundException {
		PreparedStatement pstmt = null;

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

```
package com.sms.delegate;

import com.sms.util.GenericException;
import com.sms.vo.RegVO;

public interface RegistrationDelegate {
	void register(RegVO regVO) throws GenericException;
}
```

```
package com.sms.delegate;

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

import com.sms.bo.StudBO;
import com.sms.bo.StudentCourseBO;
import com.sms.dao.StudentCourseDao;
import com.sms.dao.StudentCourseDaoImpl;
import com.sms.dao.StudentDao;
import com.sms.dao.StudentDaoImpl;
import com.sms.util.ConnectionFactory;
import com.sms.util.GenericException;
import com.sms.vo.RegVO;

public class RegistrationDelegateImpl implements RegistrationDelegate {
        private static Logger logger;

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

        @Override
        public void register(RegVO regVO) throws GenericException {
                boolean isEx = false;
                StudBO studBO = null;
                Connection con = null;
                StudentCourseBO studentCourseBO = null;
                StudentDao studentDao = null;
                StudentCourseDao studentCourseDao = null;

                try {
                        con = ConnectionFactory.getConnection();
                        studBO = new StudBO();
                        studBO.setId((Integer.parseInt(regVO.getStudentId())));
                        studBO.setName(regVO.getName());

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

                        studentCourseBO = new StudentCourseBO();
                        studentCourseBO

        .setStudentId(Integer.parseInt(regVO.getStudentId()));
```

..... contd...

```
		studentCourseBO.setCourseId(Integer.parseInt(regVO.getCourseId()));
                studentCourseDao = new StudentCourseDaoImpl();
                studentCourseDao.insert(studentCourseBO, con);

            } catch (SQLException e) {
                isEx = true;
                logger.throwing("RegistrationDelegateImpl",
                        "Unable to register Student", e);
                throw new GenericException(e);
            } catch (ClassNotFoundException e) {
                isEx = true;
                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/close
connection", e);
                        throw new GenericException(e);
                    }
                }
            }
        }
}
```

```
package com.sms.servlet;

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

	@Override
	protected void service(HttpServletRequest request,
			HttpServletResponse response) throws ServletException,
IOException {
		boolean isEx = false;
		RequestDispatcher requestDispatcher = null;
		RegistrationDelegate registrationDelegate = null;
		String studentId = null;
		String name = null;
		String courseId = null;
		RegVO regVO = null;
		String page = null;

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

		regVO = new RegVO();
		regVO.setStudentId(studentId);
		regVO.setName(name);
		regVO.setCourseId(courseId);

		registrationDelegate = new RegistrationDelegateImpl();
		try {
			registrationDelegate.register(regVO);
			request.setAttribute("studentId", studentId);
		} catch (GenericException e) {
			isEx = true;
			logger.throwing("RegServlet", "Error occured during registration",
					e);
		}
		if (isEx) {
			page = "error.jsp";
		} else {
			page = "confirm.jsp";
		}
		requestDispatcher = request.getRequestDispatcher(page);
		requestDispatcher.forward(request, response);
	}
}
```

```
package com.sms.util;

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("com/sms/common/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.driverClassName"));
			con = DriverManager.getConnection(props.getProperty("db.url"),
					props.getProperty("db.user"),
					props.getProperty("db.password"));
			con.setAutoCommit(false);
		} catch (SQLException sqe) {
			logger.throwing("ConnectionFactory", "Unable to get connection",
					sqe);
			throw sqe;
		} catch (ClassNotFoundException e) {
			logger.throwing("ConnectionFactory", "Unable to load Driver class",
					e);
			throw e;
		}
		return con;
	}
}
```

```
package com.sms.util;

public class GenericException extends Throwable {

        public GenericException() {
                super();
        }

        public GenericException(String arg0, Throwable arg1) {
                super(arg0, arg1);
        }

        public GenericException(String arg0) {
                super(arg0);
        }

        public GenericException(Throwable arg0) {
                super(arg0);
        }

}
```

```
package com.sms.vo;

import java.io.Serializable;

public class RegVO implements Serializable {
        private String studentId;
        private String name;
        private String courseId;

        // setters and getters

}
```

4.3 Business Object

When we submit the Jsp page, the request comes to Servlet. The Servlet tries to read the Jsp submitted values using req.getParameter and these values are of String type in nature. Servlet wraps these string values into a java bean called ValueObject and passes as input to Business Delegate.

Now the Business delegate converts these String represented values in ValueObject into database persistable format and populates into another java bean called Business Object.

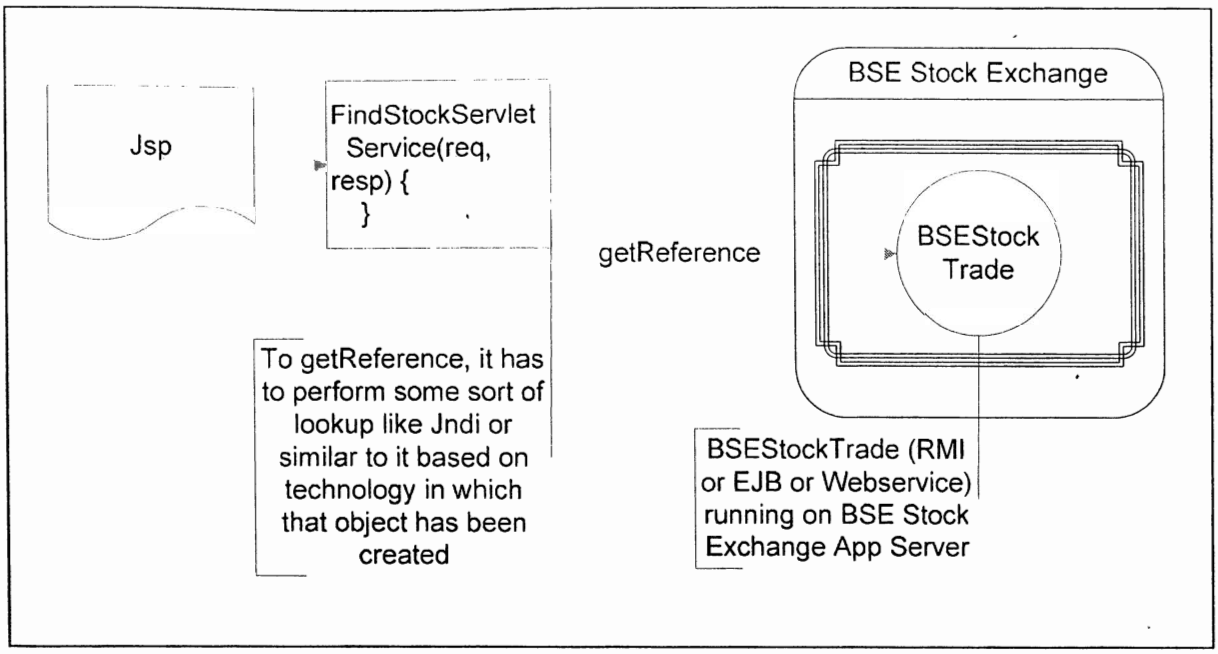
Business Object is another java bean which always contains the data in a persistable format in nature. Please refer to the previous example for how the valueobject data is converted into business object and its flow.

4.4 Service Locator

Service Locator design pattern is used for locating services that are needed as part of an application. Services could be a datasource object within the application server or could be external application business object which your application may be dependent on.

For example If you are displaying a stock price information as part of your application, the information about the stock will not be there with us, rather it will be available with BSE Stock exchange or NSE Stock exchange. Can our application can access the database of BSE/NSE Stock market, BSE/NSE stock exchange will not share their database information of their systems to us, as it is a security breach.

Instead those will expose some classes which contain functionality for getting the stock information (price). Those objects will be available and are running with BSE/NSE stock exchange servers and we need to talk to those objects by sending and receiving the data over the network. So, BSE/NSE stock exchange will expose those objects over the network to us using RMI or EJB or Web Service technologies.



If my class wants to talk to the BSE stock exchange provided classes it has to get the reference of the other application object. How to get the reference of other application object? It depends on in which technology the other application object has been created. Let's say if BSEStockTrade object is created using Ejb, then other application will exposes that object by binding it to the Jndi registry. So, we need to write the Jndi lookup code to get the reference of it.

Can our Servlet as part of our application can write the lookup code to get the reference of BSEStockTrade object, if it does we face the below problems

1) In our application we might have multiple classes, who might want to talk to BSEStockTrade object, so we need to write this lookup code across multiple classes in our application, this leads to code duplication.

2) The lookup code we write here will be environment specific lookup, which means the box on which the other object changes then we need to modify the code for lookup

3) The lookup code we are writing here is environment specific lookup, this means if the object is deployed and running on web logic server and is migrated from web logic to web sphere again our lookup code has to be modified to perform the lookup from different application server

4) The lookup code we have here is technology specific; it means if it is written in Ejb we need to perform Jndi lookup. If it is migrated from Ejb to RMI, again we need to modify the lookup code to perform Rmi lookup instead of Ejb. If it is migrated to webservice then the code for getting the reference of web service is altogether different from any of these.

So instead of servlet trying to get the reference other service objects, we have to write this code in a separate class called service locator. Always the service locator contains the locator method, it contains the logic for locating the services that are required for our application.

The advantages of service locator are as below.

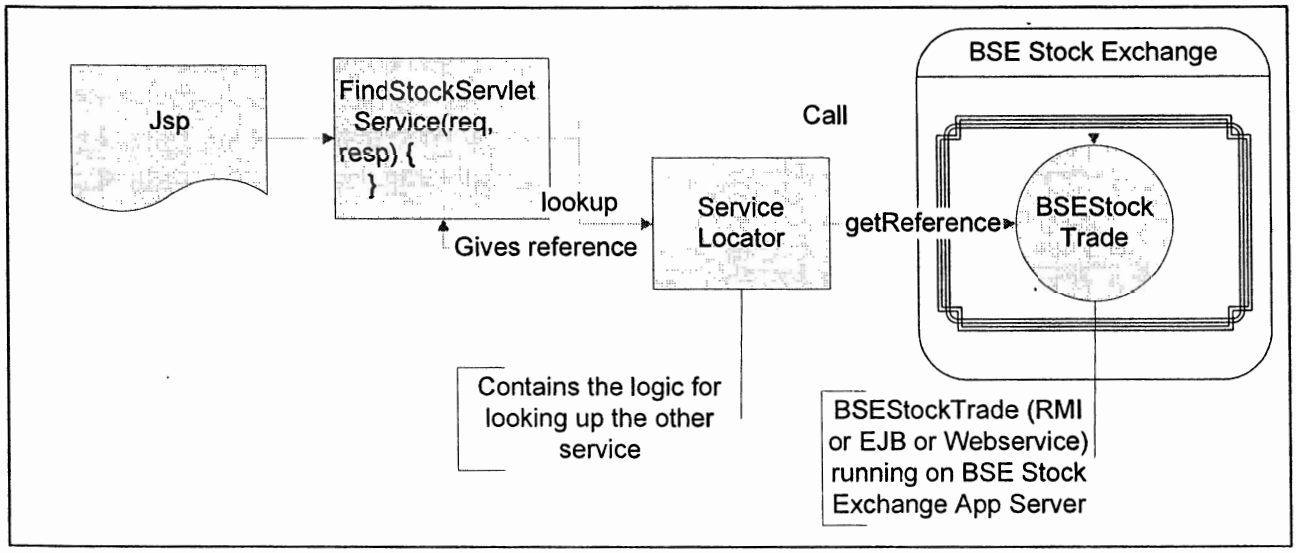
1) We don't need to duplicate the lookup logic across several classes rather we can write in service locator and all the other classes can call the service locator's method to get the reference.

2) If the environment/platform/technology changes, we don't need to modify the code across multiple classes rather we just need to modify the code at one single place.

3) Service Locator provides location transparency, which means none of the classes in our application knows how those services are being located, rather the classes feels that there are talking to normal objects that belongs to part of our application only.

4) Service Locator may even perform caching in looking up for services to optimize the performance

Below flow diagram depicts the usage of service locator



4.5 Session Façade

In some scenarios our application may have to talk to multiple services which belong to other application. The other application and its classes may not run on the local machine to which our application is belonging to, rather they might be physically separated and running over a network.

If our classes talks to multiple other classes or other applications there are lot of dis-advantages as described below.

1) Our classes are exposed to the finer details of other application classes which means we are tightly coupled with other business components

2) We need to understand the complete set of business components their methods and their dependencies which means we are exposed to the complexities of other applications

3) If the other application is running on a different machine to talk to those objects we need to make network calls several times this incurs bandwidth overhead and degrades the performance.

To overcome the above problems, we should never expose multiple services to the other applications rather we need to expose one high level service rather than multiple cross-grained services, so that the other applications can call only one network call to high level service (called as façade), which takes care of internally talking to multiple classes in providing the functionality.

The high level service which we are exposing here is called Session Façade which reduces the multiple network calls and optimizes the performance as well.

Below code snippet demonstrates the usage of Service Locator and Session Façade.

stockdetail.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>Stock Information</title>
      </head>
      <body>
            <p style="color:blue;">
                  Short Name : ${stock.shortName}<br/>
                  Price : ${stock.price}<br/>
                  Listed Date :${stock.listedDate}
            </p>
      </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="findStock">
			<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>
```

```
package com.sl.ext.bo;
import java.io.Serializable;
public class GetStock implements Serializable {
	private String shortName;
	// setters and getters
}
```

```
package com.sl.ext.bo;
import java.util.Date;
public class StockInfo {
	private int id;
	private float price;
	private Date listedDate;

	// setters and getters
}
```

```
package com.sl.ext.dto;

import java.util.Date;

public class SearchStock {
        private String shortName;
        private Date time;

        // setters and getters

}
```

```
package com.sl.ext.dto;

import java.util.Date;

public class Stock {
        private int id;
        private String shortName;
        private String companyName;
        private Date listedDate;
        private float price;

        // setters and getters

}
```

```
package com.sl.ext.bo;
import java.util.Date;
public class Quote {
        private int id;
        private Date time;

        // setters and getters
}
```

```
package com.sl.ext.bo;

public class StockDetail {
        private int id;
        private String shortName;
        private String companyName;

        // setters and getters

}
```

```
package com.sl.delegate;

import java.util.Date;

import com.sl.ext.dto.SearchStock;
import com.sl.ext.dto.Stock;
import com.sl.ext.facade.StockFacade;
import com.sl.util.StockFacadeLocator;
import com.sl.valueobject.SearchVO;
import com.sl.valueobject.StockVO;

public class StockDelegate {
	public StockVO search(SearchVO searchVO) {
		StockVO stockVO = null;
		SearchStock ss = null;
		Stock stock = null;
		StockFacade sf = null;
		StockFacadeLocator sfl = null;

		ss = new SearchStock();
		ss.setShortName(searchVO.getShortName());
		ss.setTime(new Date());
		sfl = new StockFacadeLocator();
		sf = sfl.getStockFacade();
		stock = sf.findStock(ss);

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

		return stockVO;
	}
}
```

```
package com.sl.ext.dto;

import java.util.Date;

public class Stock {
	private int id;
	private String shortName;
	private String companyName;
	private Date listedDate;
	private float price;

	// setters and getters
}
```

```
package com.sl.ext.facade;

import com.sl.ext.bo.GetStock;
import com.sl.ext.bo.Quote;
import com.sl.ext.bo.StockDetail;
import com.sl.ext.bo.StockInfo;
import com.sl.ext.dto.SearchStock;
import com.sl.ext.dto.Stock;
import com.sl.ext.service.BSEStockTrade;
import com.sl.ext.service.BSEStockTradeImpl;
import com.sl.ext.service.FindStockDetail;
import com.sl.ext.service.FindStockDetailImpl;

public class StockFacade {
        public Stock findStock(SearchStock searchStock) {
                FindStockDetail fsd = null;
                BSEStockTrade bst = null;
                Quote quote = null;
                StockInfo si = null;
                GetStock gs = null;
                StockDetail sd = null;
                Stock stock = null;

                gs = new GetStock();
                gs.setShortName(searchStock.getShortName());
                fsd = new FindStockDetailImpl();
                sd = fsd.findStock(gs);

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

                bst = new BSEStockTradeImpl();
                si = bst.getStockPrice(quote);

                stock = new Stock();
                stock.setId(si.getId());
                stock.setShortName(sd.getShortName());
                stock.setCompanyName(sd.getCompanyName());
                stock.setListedDate(si.getListedDate());
                stock.setPrice(si.getPrice());

                return stock;
        }
}
```

```
package com.sl.ext.service;

import com.sl.ext.bo.Quote;
import com.sl.ext.bo.StockInfo;

public interface BSEStockTrade {
	StockInfo getStockPrice(Quote quote);
}
```

```
package com.sl.ext.service;

import java.util.Date;

import com.sl.ext.bo.Quote;
import com.sl.ext.bo.StockInfo;

public class BSEStockTradeImpl implements BSEStockTrade {

	@Override
	public StockInfo getStockPrice(Quote quote) {
		float price = 0.0f;
		StockInfo si = null;

		if (quote != null) {
			if (quote.getId() == 1001) {
				price = 323.34f;
			} else if (quote.getId() == 1002) {
				price = 232.23f;
			}
		}
		si = new StockInfo();
		si.setId(quote.getId());
		si.setPrice(price);
		si.setListedDate(new Date());

		return si;
	}

}
```

```
package com.sl.ext.service;

import com.sl.ext.bo.GetStock;
import com.sl.ext.bo.StockDetail;

public interface FindStockDetail {
	StockDetail findStock(GetStock gs);
}
```

```
package com.sl.ext.service;

import com.sl.ext.bo.GetStock;
import com.sl.ext.bo.StockDetail;

public class FindStockDetailImpl implements FindStockDetail {

	@Override
	public StockDetail findStock(GetStock gs) {
		StockDetail sd = null;

		sd = new StockDetail();
		if (gs.getShortName().equals("ICICIBAN")) {
			sd.setId(1001);
			sd.setShortName(gs.getShortName());
			sd.setCompanyName("ICICI BANK Pvt Ltd");
		} else if (gs.getShortName().equals("CIPLA")) {
			sd.setId(1002);
			sd.setShortName(gs.getShortName());
			sd.setCompanyName("CIPLA INDIA Limited");
		}
		return sd;
	}

}
```

```
package com.sl.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 com.sl.delegate.StockDelegate;
import com.sl.valueobject.SearchVO;
import com.sl.valueobject.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("stockdetail.jsp");
		rd.forward(request, response);
	}

}
```

```
package com.sl.util;

import com.sl.ext.facade.StockFacade;

public class StockFacadeLocator {
        public StockFacade getStockFacade() {
                // look up logic
                return new StockFacade();
        }
}
```

```
package com.sl.valueobject;

import java.io.Serializable;

public class SearchVO implements Serializable {
        private String shortName;

        // setters and getters

}
```

```
package com.sl.valueobject;

import java.io.Serializable;

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

        // setters and getters
}
```