ORACLE Certified Professional Java SE 6 Programmer

Venu Kumaar.S

Hi Friends,

Just go through the following small story.. (taken from **"*You Can Win* "** - by Shiv Khera)

Once upon a time a very strong woodcutter asks for a job in a timber merchant, and he got it. The paid was really good and so were the work conditions. For that reason, the woodcutter was determined to do his best. His boss gave him an axe and showed him the area where he was supposed to work. The first day, the woodcutter brought 18 trees "Congratulations," the boss said. "Go on that way!" Very motivated for the boss' words, the woodcutter try harder the next day, but he only could bring 15 trees. The third day he try even harder, but he only could bring 10 trees. Day after day he was bringing less and less trees. The woodcutter thought that "I must be losing my strength". He went to the boss and apologized, saying that he could not understand what was going on.

The boss asked, "When was the last time you sharpened your axe?"

"Sharpen? I had no time to sharpen my axe. I have been very busy trying to cut trees.

If we are just busy in applying for jobs & work, when we will sharpen our skills to chase the job selection process?

My aim is to provide good and quality content to readers to understand easily. All contents are written and practically tested by me before publishing. If you have any query or questions regarding any article feel free to leave a comment or you can get in touch with me on venus.kumaar@gmail.com.

This is just a start, I have achieved till now is just 0.000001n% .

With Warm Regards

Venu Kumaar.S
venus.kumaar@gmail.com

Design patterns

Design patterns:

→ Which comes as Test solution for real time problems of application development. Design patterns are best practices to use s/w technologies effectively in project of application development.

→ Design Patterns are supporting code while developing s/w projects by using plain technologies.

→ The worst solution for real time problem of project or application development is called as "Anti-Pattern".

→ The organization ISO maintains both design pattern and anti-pattern.

→ Design patterns are implementation takes place in coding phase of project development. Generally PL/TL designs the task for developers specified the need and utilization of design patterns and developers will implement them while doing Task Completion.

→ There are 500+ design patterns in Java Environment.

→ Design patterns can be implemented by using many s/w technologies or programming languages.

→ Some important Design patterns are as follows:

JDK level:

- → Singleton java class
- → Synchronized singleton java class
- → Factory method/factory pattern
- → Abstract Factory
- → Template Method
- → Builder pattern
- → prototype pattern
- → Flyweight pattern
- → IOC(inversion of control) pattern
- → Adapter class
- → FastLine reader
- → VO(value object) class/DTO(Data Transfer) class

Web level

- → View Helper
- →**Composite** view
- → MVC1
- → MVC2
- → front controller
- → Intercepting filter
- → abstract controller

Integration layer Design patterns:

- → Session facade

→ message Facade

→ Service locator

→ Business delegate

Model Layer

→ DAO

→ Abstract DAO

I) JDK level:

1) Singleton java class:

Problem: Instead of creating multiple objects of java class having same data and wasting memory, degrading performance. It is recommended to create one object and use it for multiple times.

Solution: Use single java class.

The java class that allows us to create one object per jvm called as singleton java class.

→ The logger class of the log4j API is given as Singleton Java class

→ The Service locater class will be implemented as Single java class

Rules to implement singleton java class:

1) it must have only private constructors.
2) must have private static reference variable of same class.
3) override clone() to suppress cloning process.
4) must have public static factory method having the logic of Singleton(create and return only one object)

→ All these rules close all the doors of creating objects for java class and opens only one door to create object (i.e. Factory method where singleton logic is placed)

→ "The method of a class i.e. capable of creating and returning same or other class objects is called as Factory method".

Static member : This contains the instance of the singleton class.
Private constructor : This will prevent anybody else to instantiate the Singleton class.
Static public method : This provides the global point of access to the Singleton object and returns the instance to the client calling class.

```
//Singleton.java
class Singleton {
	// static reference variable
	private static Singleton n = null;

	// private constructor
	private Singleton() {
		System.out.println("Singleton: 0-param constructor (private)");
	}

	// static Factory method
	public static Singleton create() {
		// object creation having Singleton logic
		if (null == n){
			n = new Singleton();
		}
		return n;
	}// create()
```

```
}// class

//SingletonTest.java
public class SingletonTest {
	public static void main(String[] args) {
		Singleton t1 = Singleton.create();
		Singleton t2 = Singleton.create();
		Singleton t3 = Singleton.create();
		System.out.println(" t1 Object hashCode ....: " + t1.hashCode());
		System.out.println(" t2 Object hashCode ....: " + t2.hashCode());
		System.out.println(" t3 Object hashCode ....: " + t3.hashCode());

		// Singleton t4 = (Singleton) t2.clone();
		// System.out.println(" t4 Object hashCode ....: " + t4.hashCode());
	}
}
/*
OUTPUT:
Singleton: 0-param constructor (private)
t1 Object hashCode ....: 1671711
t2 Object hashCode ....: 1671711
t3 Object hashCode ....: 1671711*/
```

→*** **java.lang.Runtime** class is a pre-defined jdk level singleton java class

```
//SingleTon using enum

//SingleTonEnumTest.java
public class SingleTonEnumTest {

	public enum SingletonEnum {
		INSTANCE;
		public void doStuff() {
			System.out.println("Singleton using Enum");
		}
	}

	public static void main(String[] args) {
		System.out.println("\tHashCode..:"+SingletonEnum.INSTANCE.hashCode());
		SingletonEnum.INSTANCE.doStuff();
		System.out.println("\tHashCode..:"+SingletonEnum.INSTANCE.hashCode());
		SingletonEnum.INSTANCE.doStuff();
		System.out.println("\tHashCode..:"+SingletonEnum.INSTANCE.hashCode());

	}
}
/*
Output:
	HashCode..:1671711
Singleton using Enum
	HashCode..:1671711
Singleton using Enum
	HashCode..:1671711
*/
```

Singleton pattern is one of the most important pattern used commonly. This pattern is used when we want to restrict class to make only single object and that single object is used by all other classes.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

class JDBC {
	private static JDBC jdbc=null;
	private JDBC(){}
	public static JDBC getInstance()
	{
		if(jdbc==null)
		{
			jdbc=new JDBC();
		}
		return jdbc;
	}
	private static Connection getConnection() throws ClassNotFoundException,SQLException,Exception
	{
		Connection con=null;
		Class.forName("com.mysql.jdbc.Driver");
		con=DriverManager.getConnection("jdbc:mysql://localhost:3306/shalabh","root","root");
		return con;
	}
	public int insert(String name) throws ClassNotFoundException, SQLException, Exception{
		Connection con = null;
		PreparedStatement pstmt = null;
		int recordCount = 0;
		try {
				con = this.getConnection();
				pstmt = con.prepareStatement("insert into user(name) values(?)");
				pstmt.setString(1, name);
				recordCount = pstmt.executeUpdate();
		}
		finally {
				if(pstmt!=null)
				pstmt.close();
				if(con!=null)
				con.close();
		}
		return recordCount;
	}
	public void select(String name) throws ClassNotFoundException, SQLException, Exception{
		Connection con = null;
		PreparedStatement pstmt = null;
		ResultSet rs = null;
		try {
				con = this.getConnection();
				pstmt = con.prepareStatement("select * from user where name=?");
```

```
                        pstmt.setString(1, name);
                        rs = pstmt.executeQuery();
                        while(rs.next())
                        {
                                System.out.println("name = "+rs.getString("name"));
                        }
                }
                finally {
                        if(rs!=null)
                        rs.close();
                        if(pstmt!=null)
                        pstmt.close();
                        if(con!=null)
                        con.close();
                }
        }
}

public class SingletonPattern{
        public static void main(String[] args) throws ClassNotFoundException,SQLException{
                JDBC jdbc=JDBC.getInstance();
                try {
                        jdbc.insert("shalabh");
                        jdbc.select("shalabh");
                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
}
```

2) **Synchronized Singleton Java class:**

→ When multiple threads acts on single object then there is a possibility of getting multi-threading issues. This chance is also there while working with the methods on singleton java class.

→ To overcome this problem it is recommended to design Synchronized singleton java class. that means the static factory method and other user defined methods or that class must be taken as synchronized methods.

→ In the multi-threading environment to prevent each thread to create another instance of singleton object and thus creating concurrency issue we will need to use locking mechanism. This can be achieved by **synchronized** keyword. By using this synchronized keyword we prevent Thread2 or Thread3 to access the singleton instance while Thread1 inside the method crate().

```
//Singleton.java
class Singleton {
        // static reference variable
        private static Singleton n = null;

        // private constructor
        private Singleton() {
                System.out.println("Singleton: 0-param constructor (private)");
        }
```

```
	// static Factory method
	public static synchronized Singleton create() {
		// object creation having Singleton logic
		if (null == n){
			n = new Singleton();
		}
		return n;
	}// create()
}// class
```

So this means that every time the **create ()** is called it gives us an additional overhead. To prevent this expensive operation we will use double checked locking so that the synchronization happens only during the first call and we limit this expensive operation to happen only once. It is only required for:

```
//Singleton1.java
class Singleton1 {
	// static reference variable
	private static Singleton1 n = null;

	// private constructor
	private Singleton1() {
		System.out.println("Singleton: 0-param constructor (private)");
	}

	// static Factory method
	public static synchronized Singleton1 create() {
		// object creation having Singleton logic
		if (null == n) {
			synchronized (Singleton1.class) {
				if (null == n) {
					n = new Singleton1();
				}// if
			}// synchronized block
		}// if
		return n;
	}// create()
}// class

//SingletonTest1.java
public class SingletonTest1 {
	public static void main(String[] args) {
		Singleton1 t1 = Singleton1.create();
		Singleton1 t2 = Singleton1.create();factory pattern
		Singleton1 t3 = Singleton1.create();
		System.out.println(" t1 Object hashCode ....: " + t1.hashCode());
		System.out.println(" t2 Object hashCode ....: " + t2.hashCode());
		System.out.println(" t3 Object hashCode ....: " + t3.hashCode());

		// Singleton1 t4 = (Singleton1) t2.clone();
		// System.out.println(" t4 Object hashCode ....: " + t4.hashCode());
	}// main()
}// class

/*
Output:
	Singleton: 0-param constructor (private)
		t1 Object hashCode ....: 1671711
```

```
        t2 Object hashCode ....: 1671711
        t3 Object hashCode ....: 1671711
*/
```

3) Factory Method/Factory Pattern:

Factory pattern or Factory method pattern is a java class that is use to encapsulate object creation code. Suppose our super class has 'n' number of sub-classes and based on data provided, factory class return one of its sub-class object.

Problem: Using "new" keyword we can't create object with flexibility and by applying restrictions.
Solution: Use Factory method/Factory pattern

→ The method of a java class i.e. capable of constructing and returning its own class(or) other class object is called as "Fctory method".

→ There are two types of factory methods:
1) static factory method 2) Instance factory method

Ex: Static factory method

```
Thread t = Thread.currentThread();
Class c = Class.forName("Test");
Runtime rt = Runtime.getRuntime();
```

`Calendar cl = Calendar.getInstance();` → cl is not calender object, it is object of gregorian calendar class which is sub class of Calender class(abstract class).

→ Static factory method are useful create object of java class outside of its class that class contains only orivate constructors. These are also useful while designing singleton java classes.

→ **Prototype of static factory method:**
public static <class name/abstract class name/interface name> method(parameters);

→ **Examples on instance factory method:**

```
String s = new String("ok");
String s1 = s.concat("Hello"); // okHello

StringBuffer sb = new StringBuffer("hello");
String s2 = sb.substring(0, 3);// hell

Date d = new Date();
String s3 = d.toString();
```

→ If you want to create "**new**" object by using existing obj data and behavior then use instance factory method.
→ Factory method can return either its own class object or other class object or subclass object.
→ The object given by factory method represents certain process/logic i.e. executed in factory method.
→ Better not to design factory methods returning empty objects(objects containing default data)

Application: refer singleton program.

→ **Prototype of InstanceFactory method:**
public <class-name>/<interfacename>/<abstractclass name> method(-)

```
String s1 = new String("ok");
String s2 = s1.concat("hello"); //okhello
(String class is immutable class)


StringBuffer sb = new StringBuffer("ok");
sb.append("hello");
(StringBuffer class is mutable class)
```

→ Most of the predefined class & programmer developed java classes are generally mutable classes.

```
//Test.java
class Test {
	int a;
	float b;

	Test(int a, int b) {
		this.a = a;
		this.b = b;
	}// Test(-,-)

	public void modifyData() {
		a = a * a;
		b = b + b;
	}// modifyData()

	public void disp() {
		System.out.println("\ta = " + a + "  b = " + b);
	}// disp()
}// class

// MutableTest.java
public class MutableTest {
	public static void main(String args[]) {
		Test t = new Test(10, 20);
		System.out.println("Given Values are.....:");
		t.disp();
		System.out.println("t object hashcode..: " + t.hashCode());
		t.modifyData();
		System.out.println("\nAfter Modification the Values are.....:");
		t.disp();
		System.out.println("t object hashcode..: " + t.hashCode());
	}// main()
}// class

/*
Output:
	Given Values are.....:
		a = 10  b = 20.0
	t object hashcode..: 1671711

	After Modification the Values are.....:
```

```
            a = 100   b = 40.0
        t object hashcode..: 1671711
*/
```

Factory pattern or Factory method pattern is a java class that is use to encapsulate object creation code. Suppose our super class has 'n' number of sub-classes and based on data provided, factory class return one of its sub-class object

```
class Course{
        String courseName;
        String courseDuration;

        public String getCourseName()
        {
                return courseName;
        }
        public String getCourseDuration(){
                return courseDuration;
        }
}

class MBA extends Course{
        public MBA(String subject){
                System.out.println("This course related with "+subject);
        }
}
class MCA extends Course{
        public MCA(String subject){
                System.out.println("This course related with "+subject);
        }
}

public class CourseFactory {

        public Course getCourse(String courseName, String subject)
        {
                if(courseName.equals("MBA"))
                {
                        return new MBA(subject);
                }
                else if(courseName.equals("MCA")){
                        return new MCA(subject);
                }
                return null;
        }
        public static void main(String[] args) {
                CourseFactory factory=new CourseFactory();
                factory.getCourse("MBA","Management");
        }

}
```

→*** **Note**: When object data is modified if it is reflecting with the same object, then it is called

"mutable object" (its class is called mutable class);

→ Object data is modified, if it is reflecting with the current object, but reflecting in a newly created object and reflecting by returning new object, then that object is called as immutable object(it's class is called as immutable class).

String constant pool:

→ String constant pool maintains set of readily available objects. Due to this objects availability new objects will not be created they will be accessed from pool as needed.

→ To utilize this StringConstantPool the java.lang.String class is given as immutable class for remaining classes this king of constant pools are not available. so, they are given as mutable classes.

→ **To develop user-defined immutable java class:**

1) class must be final class
3) member variables must be taken as private, final variable
4) When data modification is required it should be done through factory methods having logic to return new objects.

```
//immutable Test.java
final class Test {
	private final int a;
	private final String b;

	// constructor
	public Test(int a, String b) {
		this.a = a;
		this.b = b;
		System.out.println("Test:2-param constructor");
	}// Test(-,-)

	// Write factory method when data modification is required
	public Test modifyData(int a, String b) {
		return new Test(a, b);
	}// modifyData(-,-)

	public Test mofifyA(int a) {
		return new Test(a, this.b);
	}// mofifyA(-)

	public Test modifyB(String b) {
		return new Test(this.a, b);
	}// modifyB(-)
		// display object data

	public void disp() {
		System.out.println("a = " + a + " b= " + b);
	}// disp()
}// class
```

→ Only final classes are not immutable classes.

Ex: System, StringBuffer, Integer classes are final classes, but they are not immutable classes.

```
//immutable Test.java
final class Test {
	private final int a;
	private final String b;

	// constructor
	public Test(int a, String b) {
		this.a = a;
		this.b = b;
		System.out.println("Test:2-param constructor");
	}// Test(-,-)

	// Write factory method when data modification is required
	public Test modifyData(int a, String b) {
		return new Test(a, b);
	}// modifyData(-,-)

	public Test modifyA(int a) {
		return new Test(a, this.b);
	}// mofifyA(-)

	public Test modifyB(String b) {
		return new Test(this.a, b);
	}// modifyB(-)
		// display object data

	public void disp() {
		System.out.println("a = " + a + " b= " + b);
	}// disp()
}// class

//ImmutableTest.java
public class ImmutableTest
{
public static void main(String args[])
 {
		Test t = new Test(10, "guru10");
		System.out.println("t object hashcode " + t.hashCode());
		t.disp();

		Test t1 = t.modifyData(20, "guru20");
		System.out.println("t1 object hashcode " + t1.hashCode());
		t1.disp();
		t.disp();

		Test t2 = t.modifyA(30);
		System.out.println("t2 object hashcode " + t2.hashCode());
		t2.disp();
		t.disp();

		Test t3 = t.modifyB("guru40");
		System.out.println("t3 object hashcode " + t3.hashCode());
		t3.disp();
		t.disp();
	}//main()
}//class
```

4) **Template method Design pattern:**

Problem: Task1 → a();
b();
c();
d();
x();

Here to complete the task we need to multiple methods by remembering method names and their sequence of invocation.

Solution: Use template method

```
public void myMethod() {
        a();
        b();
        c();
        d();
        x();
}
```

→ Task1: **myMethod()**;
now only one method need to be call to complete task.

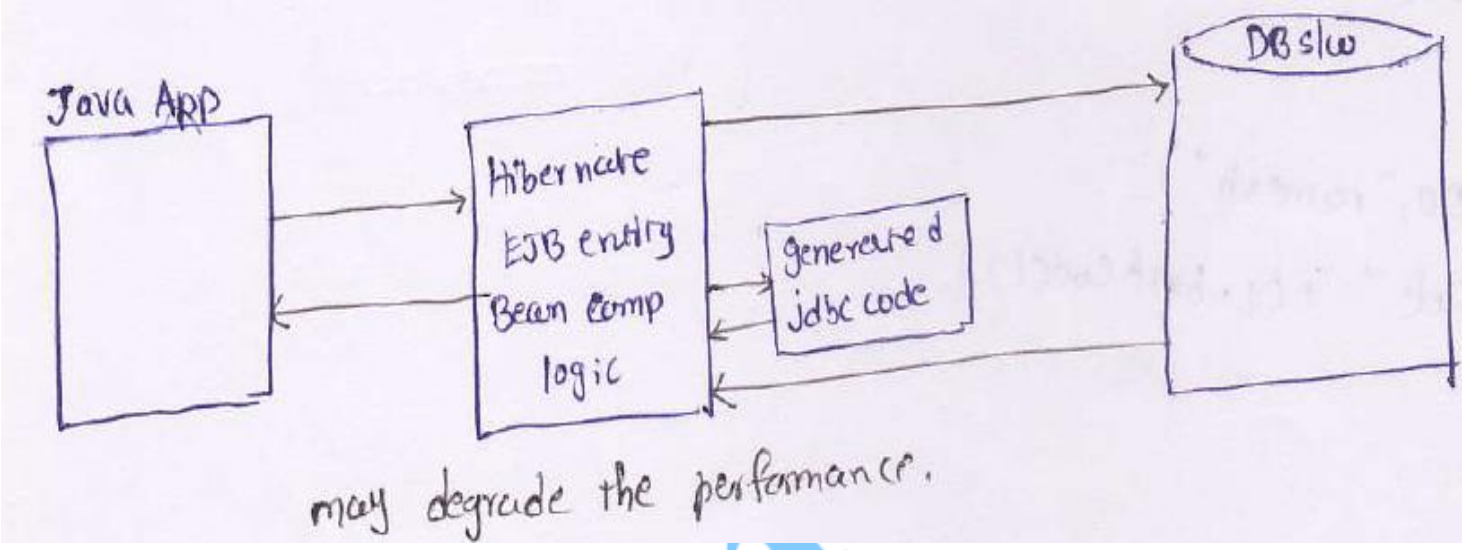
Ex:→ The process method of org.apache.struts.action.RequestProcessor class calls 19 processXxx() methods in a sequence to complete the task.

So, this method is called Template method.

→ **ActionServlet** should call all this 19 methods directly or **RequestProcess** class object to complete RequestProcess. But it **calls** only **one** method i.e. "**process()**".

5) **FastLine Reader:**

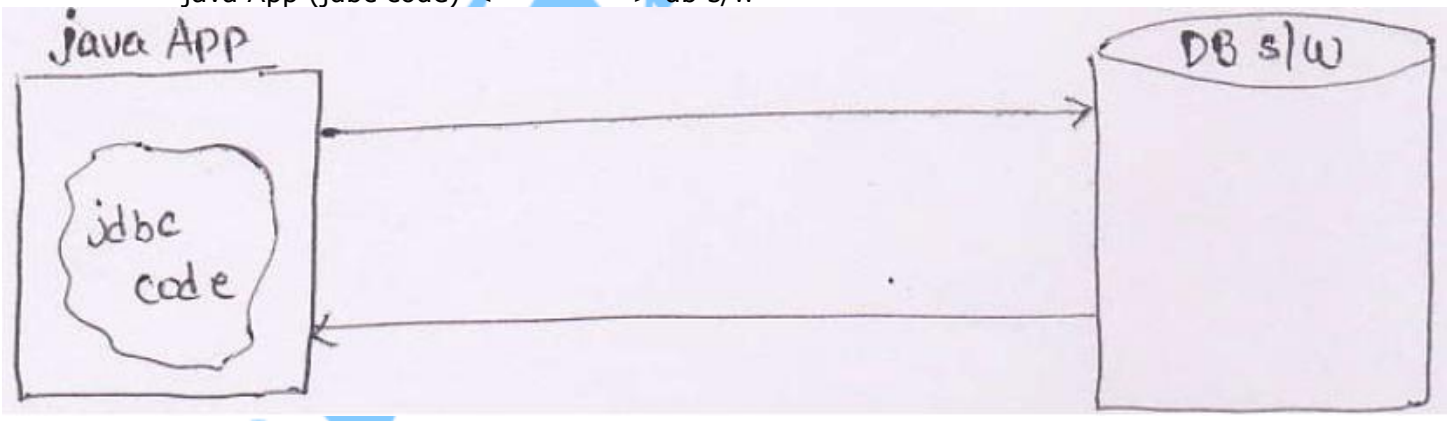
Problem: using high-end ORM s/ws (like hibernate) to develop persistence logic in our small and medium scale apps may degrade the performance because these ORM s/ws internally generate jdbc code to complete the persistence logic development. This may degrade the performance.



Solution: Use FastLine Reader design pattern:

→ Write jdbc code directly in your small and medium scale applications to interact with the db s/w to improve the performance.

java App (jdbc code) **<======>** db s/w



abstract : Hiding the implementation

6) Abstract Factory pattern:

Abstract factory pattern is one level higher than factory pattern. An abstract factory returns factories, and later on factory returns one of its sub-class based on data provided.

Problem: While working with the object create and returned by factory method object factory pattern we need to specify the class names of those objects.

Solution: Use abstract factory pattern which is extension of factory pattern.

→ Here, in the class names of objects returned by factory method/pattern to implement common interface or should extend from common class or abstract class.

→ abstract means hiding implementation. abstract factory hides the classnames of the object while receiving and utilizing those objects.

→ Entire jdbc specification jdbc drivers development and utilizing those drivers in our jdbc code is happending based on abstract factory designed pattern.

Connection con = DriverManager.getConnection("jdbc:odbc:oradsn","scott","tiger");

→ Here "con" is jdbc connection object that it is the object of jdbc driver supplied java class that implements java.sql.Connection.

→ Here we are working with jdbc "con" object without exposing and knowing its class name even though **DriverManager.getConnection(-,-,-)** returns different classes objects based on jdbc driver and jdbc url we use. This is nothing but abstract factory pattern implementation.

Example:

```
		//Common interface
//Abc.java
interface Abc {
	public void xyz();
}// interface Abc

// implemention of class1
// Test1.java
class Test1 implements Abc {
	public Test1() {
		System.out.println("Test1");
	}// Test1()

	public void xyz() {
		System.out.println("xyz of Test1");
	}// xyz()
}// class Test1

// Implementaion of class2
// Test2.java
class Test2 implements Abc {
	public Test2() {
		System.out.println("Test2");
	}// Test2()

	public void xyz() {
		System.out.println("xyz of Test2");
	}// xyz()
}// class Test2

class Demo {
	public static Abc m1(String name) {
		// factory pattern
		if (name.equals("a")) {
			return new Test1();
		} else if (name.equals("b")) {
			return new Test2();
```

```
            } else {
                  return null;
            }
      }// m1(-)
}// class Demo

// client code
public class AbstractFactoryTest {
      public static void main(String[] args) {
            // abstract factory pattern behaviour
            Abc ab1 = Demo.m1(args[0]);
            ab1.xyz();
      }// main()
}// class AbstractFactoryTest
```

→ javac AbstractFactoryTest.java

→ java AbstractFactoryTest a

→ java AbstractFactoryTest b

→ For related information and related example on refer singleton example.

7) **Prototype:**

As a name suggest Prototype means making a clone that is a new object is created by cloning an existing one. The prototype pattern can be a useful way of creating copies of objects. If the cost of creating a new object is expensive and resource intensive, we prototype the object.

Problem: Creating new object from scratch level having the existing object data has initial data is quire time consuming process.

Solution: Use cloning to create new object where new object contains existing object data has initial data and constructor will not be executed during object creation.

→ When objects are created "new" keyword constructors will be executed for objects data initialization

→ When objects are created to cloning constructors will not be executed. Because there is no need of objects initializing new objects that are created to cloning(as they old existing object data as initial data)

.

→ Working with cloning is nothing but working with prototype designed pattern.

```
//Demo.java
class Demo implements java.lang.Cloneable {
      int a, b;

      public Demo(int x, int y) {
            System.out.println("Demo:2-param constructor");
            a = x;
            b = y;
      }// Demo(-,-)

      public Demo() {
            System.out.println("Demo:0-param constructor");
```

```
	}// Demo()

	public Object myClone() throws Exception {
		Object obj = super.clone(); // performs cloning current invoking object
		return obj;
	}// myClone()

	public void disp() {
		System.out.println("\ta=" + a + "b=" + b);
	}// disp()
}// class Demo

// CloneableTest.java
public class CloneableTest {
	public static void main(String args[]) throws Exception {
		// create First object
		Demo d1 = new Demo(10, 20);
		System.out.println("d1 hashcode....:" + d1.hashCode());
		d1.disp();

		// perform cloning
		Demo d2 = (Demo) d1.myClone();
		System.out.println("d2 hshCode...:" + d2.hashCode());
		d2.disp();
	}// main()
}// class ClonableTest
```

→ Compared to normal object creation the object creation through cloning takes less time because there is no constructor execution.

→ When objects are created through cloning and deserialization the constructors will not be executed.

```
	interface Prototype {

	public Prototype doClone();

}
class Person implements Prototype {

	String name;

	public Person(String name) {
		this.name = name;
	}

	@Override
	public Prototype doClone() {
		return new Person(name);
	}

	public String toString() {
		return "This person name is  " + name;
	}
```

```
}

class Cat implements Prototype {

	String sound;

	public Cat(String sound) {
		this.sound = sound;
	}

	@Override
	public Prototype doClone() {
		return new Cat(sound);
	}

	public String toString() {
		return "This cat make " + sound+" sound";
	}
}


public class Test {

	public static void main(String[] args) {

		Person person1 = new Person("shalabh");
		System.out.println("person 1:" + person1);
		Person person2 = (Person) person1.doClone();
		System.out.println("person 2:" + person2);

		Cat cat1 = new Cat("meon!");
		System.out.println("cat 1:" + cat1);
		Cat cat2 = (Cat) cat1.doClone();
		System.out.println("cat 2:" + cat2);

	}

}
```

8) **Builder Design pattern:**

Builder pattern is used to build complex object from simple objects. In other words, with the builder pattern, the same object construction process can be used to create different objects. A Builder is an interface (or abstract class) that is implemented (or extended) by Concrete Builders.

Problem: construction of complex objects directly by taking one complex class it not recommended process.

Solution: construct complex objects from multiple simple objects step-by-step as needed. This is called builder design pattern. Which improves the re-usability of multiple individual objects by constructing the complex objects.

Program:

```
//BuilderTest.java
class Burger {
	public int price() {
		return 25;
	}// price()
}// class Burger

class Fries {
	public int price() {
		return 15;
	}// price()
}// class Fries

class Drink {
	public int price() {
		return 30;
	}// price()
}// class Drink

class MealBuilder {
	public int calcPrice() {
		// builder pattern logic
		return new Burger().price() + new Fries.price() + new Drink.price();
	}// calcPrice()
}// class MealBuilder

class SnackBuilder {
	public int clacPrice() {
		// builder pattern logic
		return new Fries().price() + new Drink.price();
	}// clacPrice()
}// class SnackBuilder

public class BuilderTest {
	public static void main(String args[]) {
		// compelx object1
		MealBuilder mb = new MealBuilder();
		int val = mb.calcPrice();
		System.out.println("The meal price is " + val);

		// compelx object1
		SnackBuilder sb = new SnackBuilder();
		int val1 = sb.calcPrice();
		System.out.println("The snacks price is " + val1);
	}// main()
}// class BuilderTest
```

→ The ActionMapping object of Struts1.x environment is created based on BulterDesign Pattern.

→ The request, response objects of Servlet programming will be created by container based on Builder design pattern.

9) Adapter class:

Problem: when class of the application directly implements interface it as provide implementation for all the methods of that interface even-though it is not interested to provide implementation for certain

methods.

```
interface Xyz {
	public void a();

	public void b();

	public void c();
}// interface Xyz

class Test implements Xyz {
	public void a() {
		----;
		----;
	}// a()

	public void b() {
		----;
		----;
	}// b()

	public void c() // null method
	{
	}// c()
}// class Test
```

→ Test is not interested to provide implementation for method c(), but it is forced to define it as null method definition.

Solution: Take adapter class implementing interface and provide null method definitions for interface methods. So, the class of Application can override its own choice method by extending from Adapater class.

```
		interface Xyz {
	public void a();

	public void b();

	public void c();
}// interface Xyz

abstract class MyAdapter implements Xyz {
	public void a() {
	}

	public void b() {
	}

	public void c() {
	}
}// abstract class

class Test extends MyAdapter {
	public void a() {
		----;
		----;
```

```
	}// a()

	public void b() {
		----;
		----;
	}// b()
}// class Test

class Test1 extends MyAdapter {
	public void c() {
		----;
		----;
	}// c()
}// //class Test1
```

→ Generally the Adapter classes will be taken as abstract classes, because these classes doesn't contain serious logics (contains null methods definitions). This helps to provide direct instantiation for Adapter class.

→ java.awt.evetn.WindowAdapter is an adapter class for 3 event listener interfaces like windowListener, WindowStateListener, WindowFocusListener

→ javax.Servlet.GenericServlet class is partially an adapter class for javax.servlet.Servlet interface.

→ **Where did you use interfaces and Adapter classes in your projects**?

→ PL designs API specification of the project having rules(method declarations) and guidelines(method definitions(concrete methods)) to programmer. This process we uses abstract classes supplied both rules and guidelines and we uses interfaces to supply only rules(method declaration).

Sun micro system gives API specification(like jdbc and servlet etc..) to vendor companies having rules and guidlines to develop s/w's (like jdbc drivers and containers). In this process abstract classes will be used. supply both rules and guidelines and interfaces will be used supplied just rules.

→ While developing adapter classes abstract classes will be taken. while developing business components Service interfaces or java interface.

→ Provide special run-time capabilities to object to JRE marker interfaces are requires.

→ May ordinary java class as special component must be extended from special classes, abstract classes are ites must implements special interface.

10) **VO/DTO class:**

→ **Problem**: ResultSet object is not Serializable object. so, we can't send it directly over the network.

→ **Solution**:

Approach1: Use RowSets instead of ResultSets (all RowSets are Serializable objects by default)

Approach2: Copy data of ResultSet onject to colletion framework datastructure and send that datastructure over the network. (All Colletion framework Datastructures are Serializable objects by default).

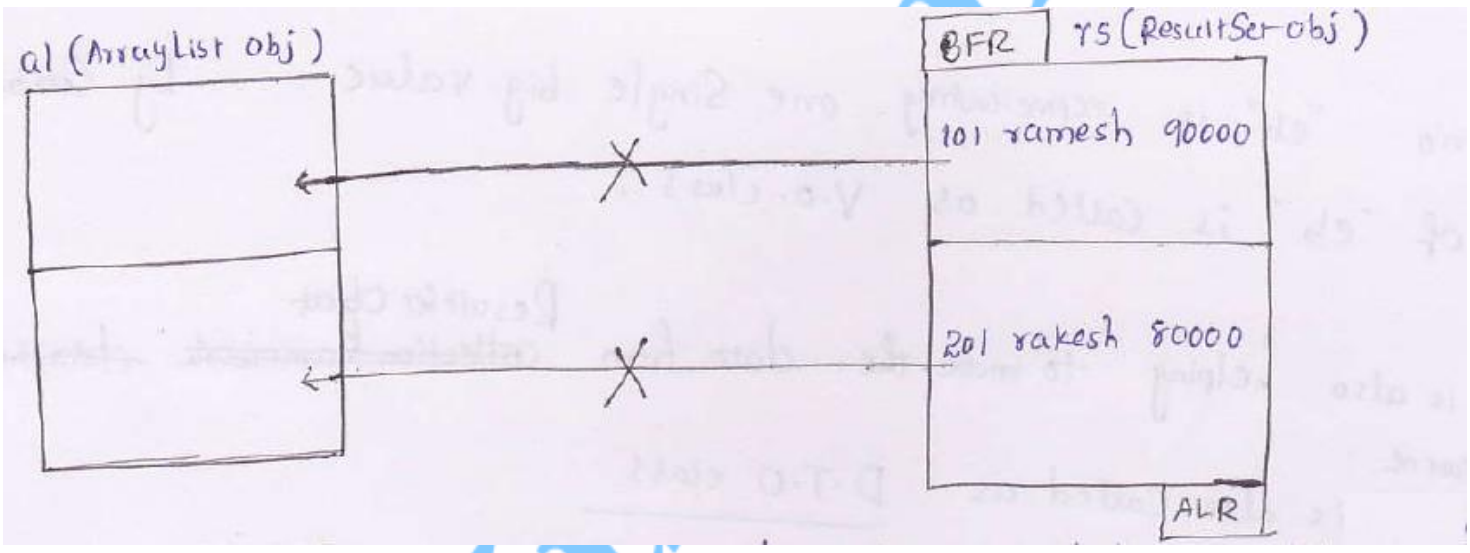
→****Note:** since most of jdbc driver are not supporting RowSets so use Approach2 to solve the above problem.

→ After moving data to data structure if you are looking to perform Simultaneous Read operations then use non-synchronize data structure like ArrayList, HashMap for better performance.

→ After moving data to collection framework data structure if you are looking to perform both read and write operations simultaneously then use Synchronized data structure like vector for thread safety.

→ In the above senario we prefer working with ArrayList Datastructure.

→ Understanding the problem related to copying ResultSet to ArrayList:



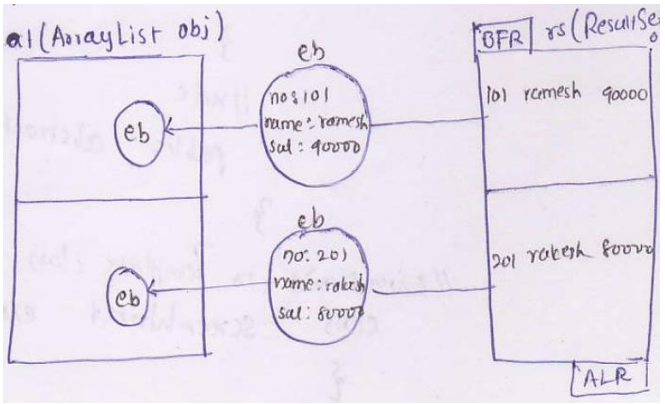
→ Each record of ResultSet contains multiple values including multiple objects. Each element of ArrayList allows only one object. so, we can't ove each record of ResultSet directly to each element of ArrayList.

→ To make the above operation possible create multiple objects for user-defined class having the data of multiple records and add these objects to multiple elements of ArrayList. In this process this user defined java class is called as VO/DTO class.

Example:
Vo/DTO class(java bean):

```
//EmpBean.java

public class EmpBean implements java.io.Serializable {
	private int no;
	private String name;

	/**
	 * @return the no
	 */
	public int getNo() {
		return no;
	}

	/**
```

```
     * @param no
     *            the no to set
     */
    public void setNo(int no) {
        this.no = no;
    }

    /**
     * @return the name
     */
    public String getName() {
        return name;
    }

    /**
     * @param name
     *            the name to set
     */
    public void setName(String name) {
        this.name = name;
    }
}// class

//Logic to copy ResultSet object records to ArrayList

ResultSet rs=st.executeQery("select * from emp");
    ArrayList al=new ArrayList();
    while(rs.next()){
    //copy each record to one emp bean obj
        EmpBean eb=new EmpBean();
        eb.setNo(rs.getInt(1));
        eb.setName(rs.getString(2));
    //add each EmpBean object to ArrayList
        al.add(eb);
    }//while
```



→ In the above scenario "eb" is representing one single big value by combining multiple values.So, the class fof "eb" is called as VO class.

Similarly "eb" is also helping to move the data from ResultSet object to ArrayList elements. So, the class of "eb", is also called as DTO class.

→***Note: we need to implement above design pattern only by working with jdbc. This implementation is not required working with Spring, Hibernate think of they directly return List data structure.

11) Template Design Pattern:

→ **Problem**: Developing multiple classes of same category from scratch level kills the re usability and increases the burden on the programmers.

→ **Solution**: Use Template class providing model for programmers having rules and guidelines so, programers can develop their classes based on this template class quite easily.

Template classes are generally abstract classes.

Example: //TemplateTest.java

```
import java.io.*;

//Template class
abstract class Writer1 {
	// guideline
	public int sum(int a, int b) {
		return a + b;
	}

	// rule
	public abstract void showResult(int res);
}

// Extention1 to Template class
class ScreenWriter extends Writer1 {
	public void showResult(int res) {
		System.out.println(res);
	}
}

// Extension2 to template class
class FileWriter1 extends Writer1 {
	public void showResult(int res) {
		try {
			String s1 = String.valueOf(res);
			BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(
					new FileOutputStream("abc.txt")));
			bw.write(s1);
			bw.flush();
			bw.close();
		}// try
		catch (Exception e) {
			e.printStackTrace();
		}// catch
	}// showResult
}// class

public class TemplateTest {
	public static void main(String[] args) throws Exception {
		ScreenWriter sc = new ScreenWriter();
		int res = sc.sum(10, 20);
		sc.showResult(res);

		FileWriter1 fw = new FileWriter1();
		res = fw.sum(10, 20);
		fw.showResult(res);

	}// main
}// class

/*
 Output:
	30
*/
```

12) IOC/Dependency Injection:

→ **Problem**:

Dependency lookup: Dependency lookup of resource is getting its dependent value by searching for them explicitly is called as Dependency lookup. In Dependency lookup the resource has to pull the value explicitly before utilizing them.

Ex: **1.** Student gets materiel only when he demands for it.
2. The way DataSource object is gathered from Registry s/w through JNDI code.

→ **Solution**: Dependency Injection/IOC

Dependency Injection: If underlying container/framework/server/run-time environment dynamically assigns values to resource then it is called as "Dependency Injection".

→ In DI underlying server/container.... pushes the values to resource. So, the resource need not to spend additional time to gathered the values.

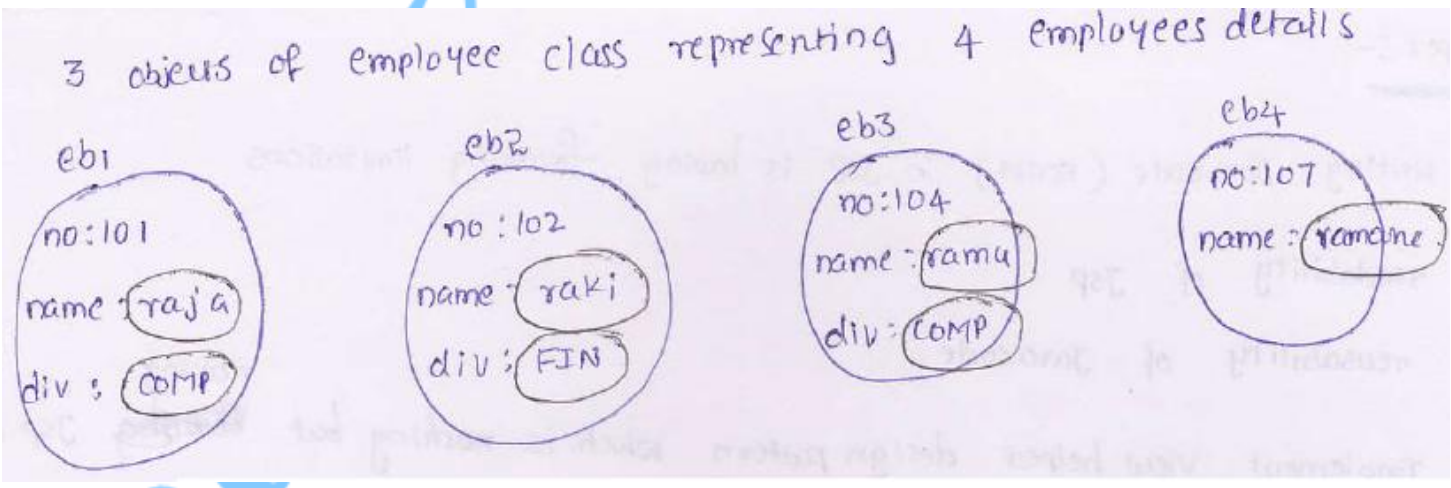
Ex: 1) the way JVM executes constructors automatically to initialize the object when object is created.
2) The way ActionServlet writes form data to form bean.
3) The way Spring container performs setter, constructor and interface injections on spring bean.

→ The way object is initialize through constructor execution comes under Dependency Injection.

→ The way object is assigned with data through method calls comes under dependency lookup.

13) **Fly weight design pattern**

Problem:

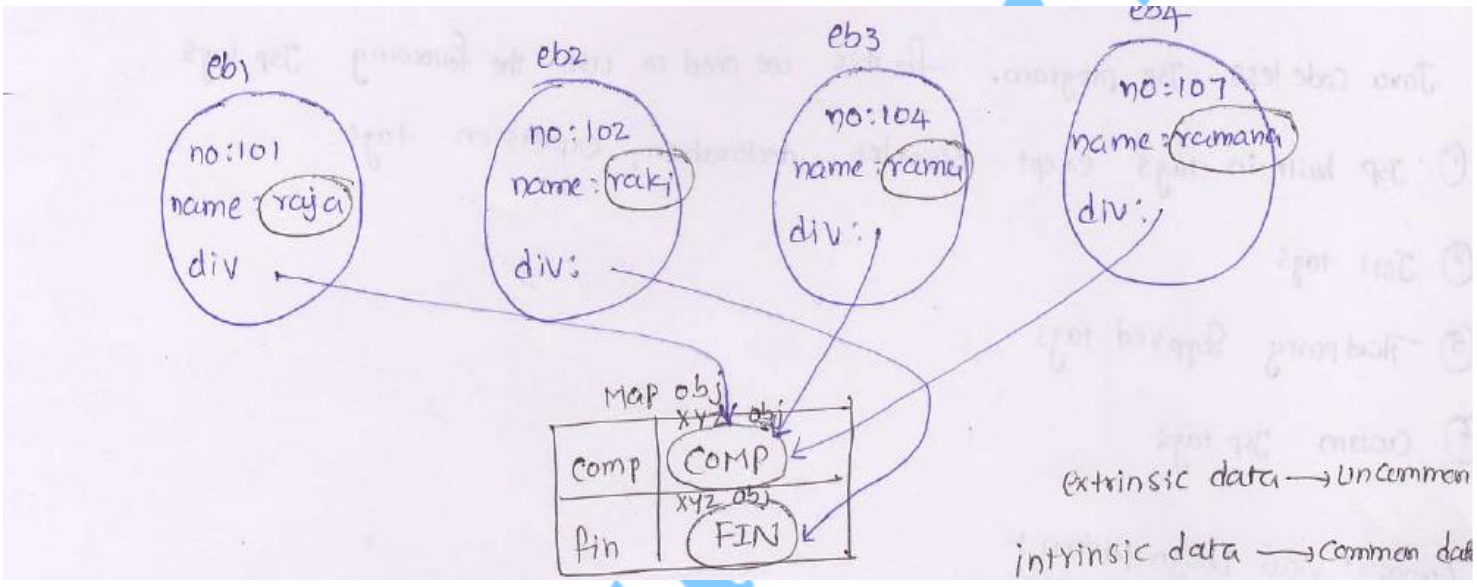


When you create multiple objects for a class having different values still there is a possibility of having some same, common data/values in certain properties of these multiple objects. instead object think about allocating common memory and using multiple objects.

The common data of multiple objects created for a class is called intrinsic data. Similarly the data that is specific to each object is called extrinsic data.

In the above diagram no name values in every object comes under extrinsic data and "div" comes under intrinsic data.

Solution: Store extrinsic data in every object and intrinsic data in common memory(common object) and use it in shared every object through Flyweight Design pattern implementation.



For example application on Flyweight Design pattern,

II) Web level design pattern:

1)View Helper:

Problem: Writing java code(script) in jsp is having following limitations.

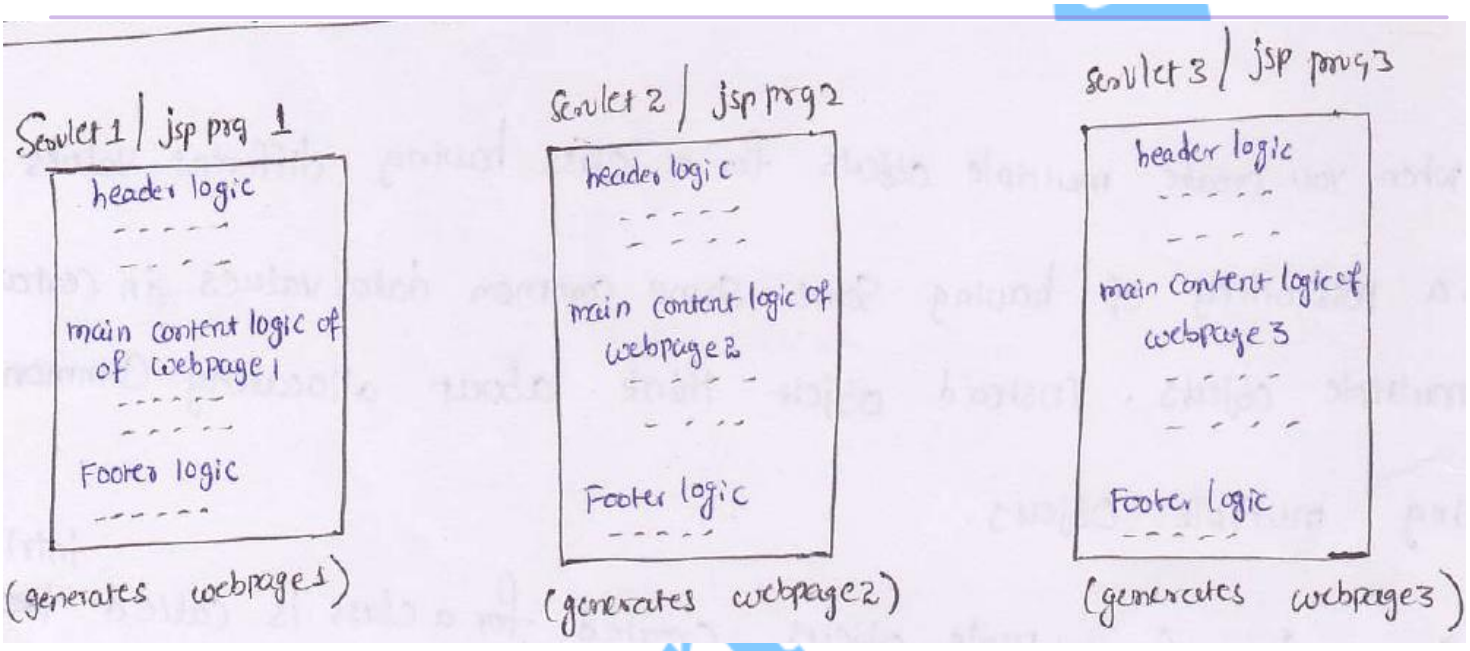
1) kills the readability of jsp
2) kills the re-usability of java code.

Solution: Implement view helper design pattern which is nothing but making jsp program as java codeless jsp program. For this we need to use the following jsp tags.

1) jsp built-in tags except Scriptlet, declaration, expression tags.
2) Jstl tags
3) Third party supplied tags
4) custom jsp tags.

2) Composit view Design pattern:

Problem:

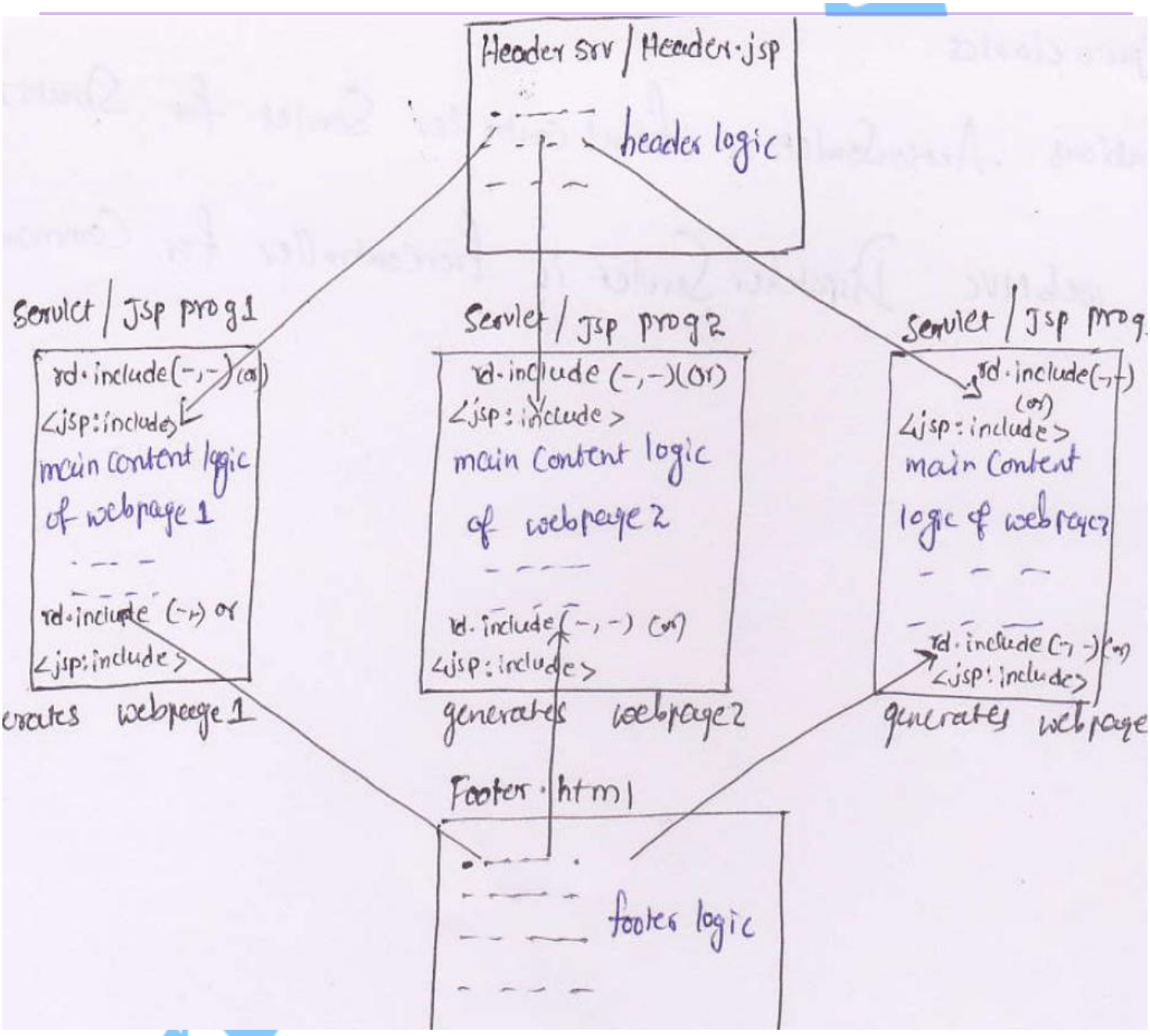


Here the common logic of multiple web resources program are not reusable logic (like header, footer logics) because they are hard coded in every web resource program.

→ the multiple web pages of web site contain same header and footer contents, but main content of each web page is different.

Solution: make the common logics as reusable logics by keeping them in separate web resource programs and implude their output. The main web resource programs its generate web pages for this we need to use "rd.include(-,-)"/<jsp:include>.

This whole process is called implementation of composite view design pattern.



→***Note: Here common logics (header and footer logics) are reusable logics

3) **Front controller Design pattern:**

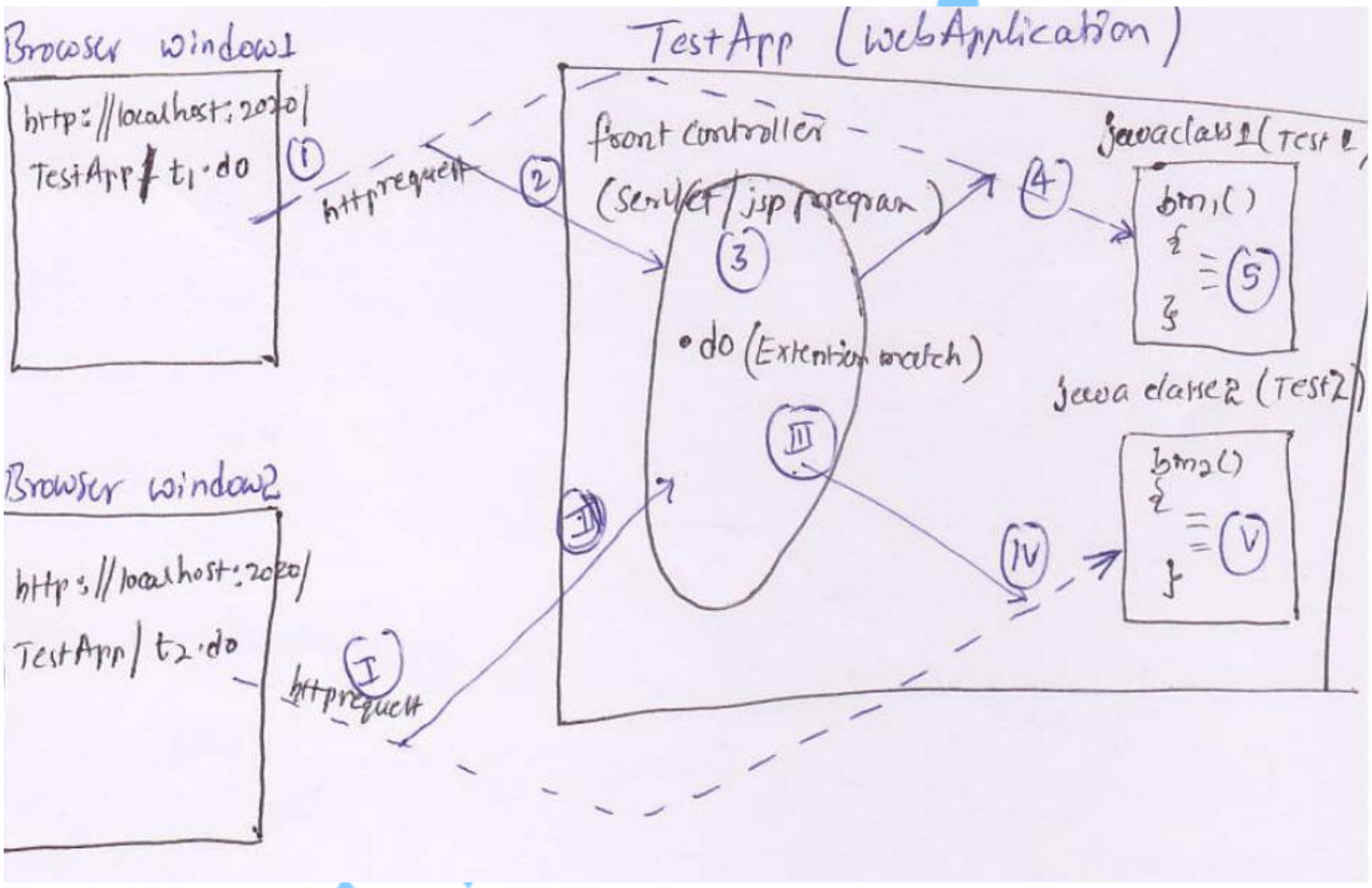
Problem: Ordinary java classes that are acting as web resource programs of web application can't take the client generated http request directly.

→***Note: only servlet/jsp programs of java web application can take http request directly.

Solution: develop special servlet/jsp programs as front controller trap all http requests given by

clients and to pass them to appropriate java classes.

Note: Front controller servlet/jsp program must be configured "web.xml" either by using extension match (or) directly match "url-pattern".



→***Note: Front controller is a special web resource program(servlet/jsp program) which acts as entry and exist point for all request that are given to other web resource programs of web applicaion which are java classes.

Ex: In Struts 1.x applications ActionServlet is front controller Servlet for Struts Action class (java classes).

In Spring web MVC DispatcherServlet is front controller for command controller classes(java classes).

web.xml:

configure FrontSrv program in web.xml file with *.do, *.do url pattern.

→***Note: Front controller servlet/jsp program must be configured in web.xml file either with directory match url pattern or with extension match url pattern.

web.xml

```
<web-app>
```

```
	<servlet>
		<servlet-name>F</servlet-name>
		<servlet-class>Frontsrv</servlet-class>
	</servlet>
	<servlet-mapping>
		<servlet-name>F</servlet-name>
		<url-pattern>*.do</url-pattern>
	</servlet-mapping>
</web-app>

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

@SuppressWarnings("serial")
public class FrontSrv extends HttpServlet {
	public void doGet(HttpServletRequest req, HttpServletResponse res)
			throws ServletException, IOException {
		// generate settings
		PrintWriter pw = res.getWriter();
		res.setContentType("text/html");
		// take the request and pass the request to appropriate java class
		int result = 0;
		if (req.getServletPath().equals("/t1.do")) {
			Test1 t1 = new Test1();
			result = t1.bm1(10, 20);
		}// if
		else if (req.getServletPath().equals("/t2.do")) {
			Test2 t2 = new Test2();
			result = t2.bm2(10, 20);
		}// else
			// display the result
		pw.println("<br><b> the Result is ....: " + result);
	}// doGet()

	public void doPost(HttpServletRequest req, HttpServletResponse res) {
		try {
			doGet(req, res);
		} catch (ServletException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}// doPost
}// class

class Test1 {
	public int bm1(int a, int b) {
		return a + b;
	}
}

class Test2 {
	public int bm2(int a, int b) {
		return a + b;
```

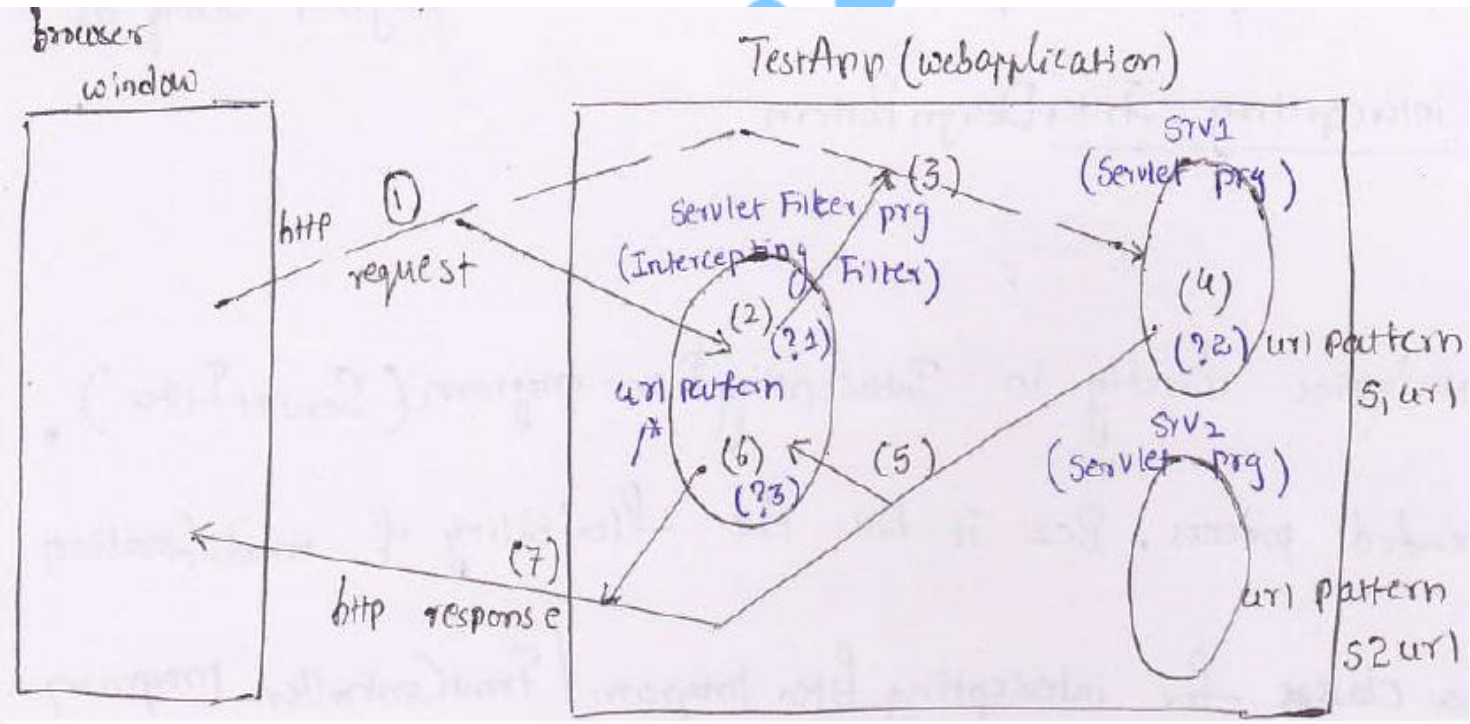
```
        }
}
```

4) **Intercepting Filter:** Intercepting Filter the special web resource program of web application that is capable of trapping the request & response of other web resource programs is called as "Intercepting Filter".

Problem: Keeping the common and global pre-request processing logic and post response generation logic in every main web resource program of web application kills the re-usability of the code.

Solution: Implement intercepting filter design patter which is nothing but Servlet Filter program.

→ Servlet Filter program execute common pre-request processing logic by trapping request. similarly execute common post-response generation logic by trapping the response of other web resource program



(?1) represents pre-request processing logic
(?2) represents main-request processing logic
(?3) represents post-response generation logic.

→ Servlet filet is nothing but implementing intercepting Filter Design pattern. Generally, we place the following pre-request processing logic in ServletFilter program.

1) Authentication logic
2) Authorization logic
3) Logging logic

→ We place the following post-response generation logics

1) Compression logic (It takes the output & compression the logic)
2) Transformation logic and etc...

→***Note: Front-controller traps the request to pass them to ordinary java classes. Intercepting filter traps the request going to Servlet programs, jsp programs and java classes.

A sample Servlet Filter Development:

MyFilter.java

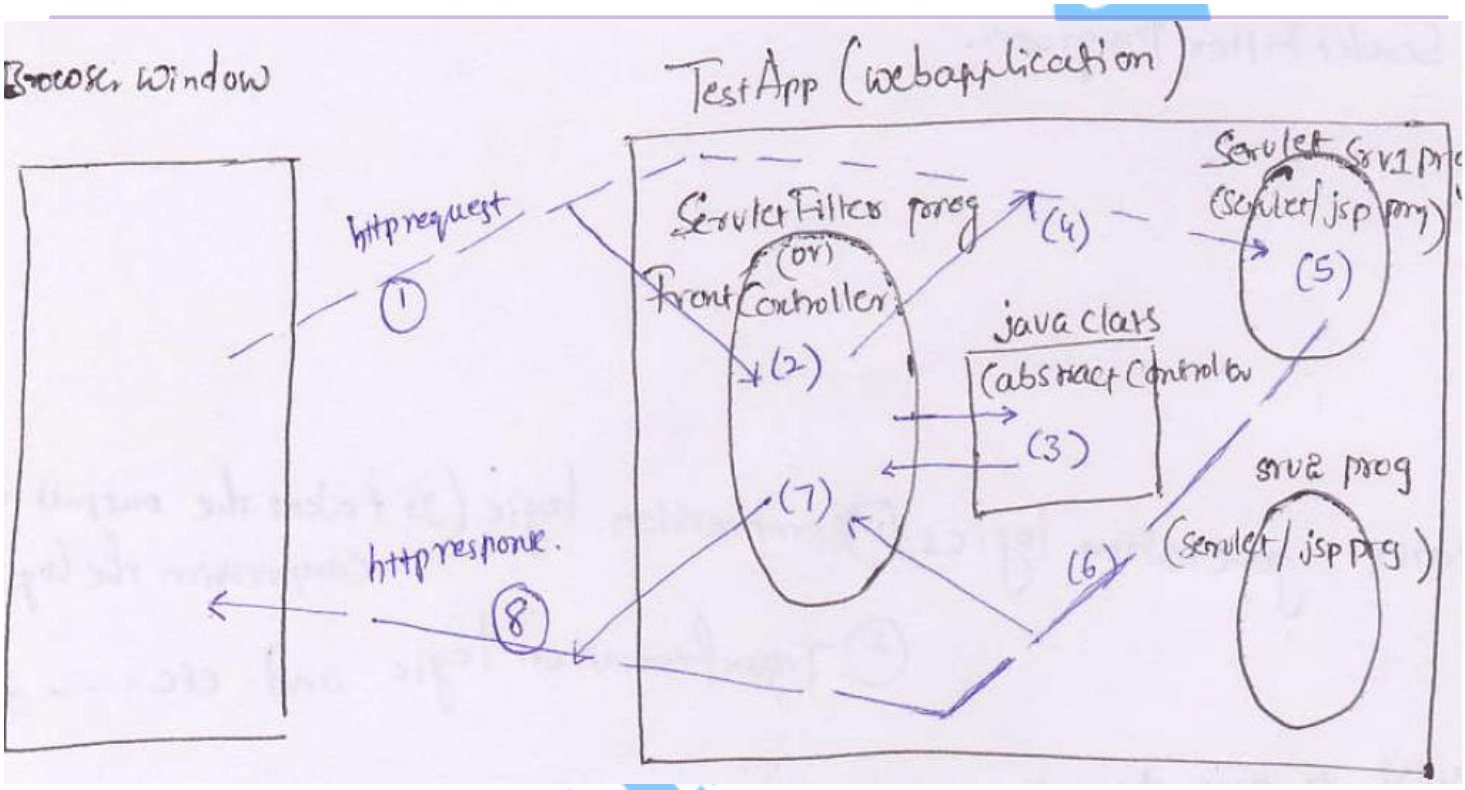
```
public class MyFilter implements Filter
{
public void init(FilterConfig fg)
{
----;// initialization logic
}//init()
public voud doFilter(ServletRequest req, ServletResponse res, filterchain fc)
{
----;
----;//pre-request processing logic
fc.doFilter(req,res);
----;
----;//post-response generation logic
}//doFilter()
public void destroy()
{
-----;
-----;//uninitialization logic;
}//destroy()
}//class
```

→ In struts 2.x application FilterDispatcher is a pre-defined ServletFilter program acting as Controller by implementing "intercepting Filter Design pattern".

5) **Abstract Controller:**

Problem:Hard coding main logics directly in Intercepting Filter program(ServletFilter), Frontcontroller is not recommended process. because it kills the flexibility of modification

Solution: Develop Helper java class for intercepting filter program/FrontController program having main logics get the flexibility of modification. This Helper class is nothing but Abstract Controller.



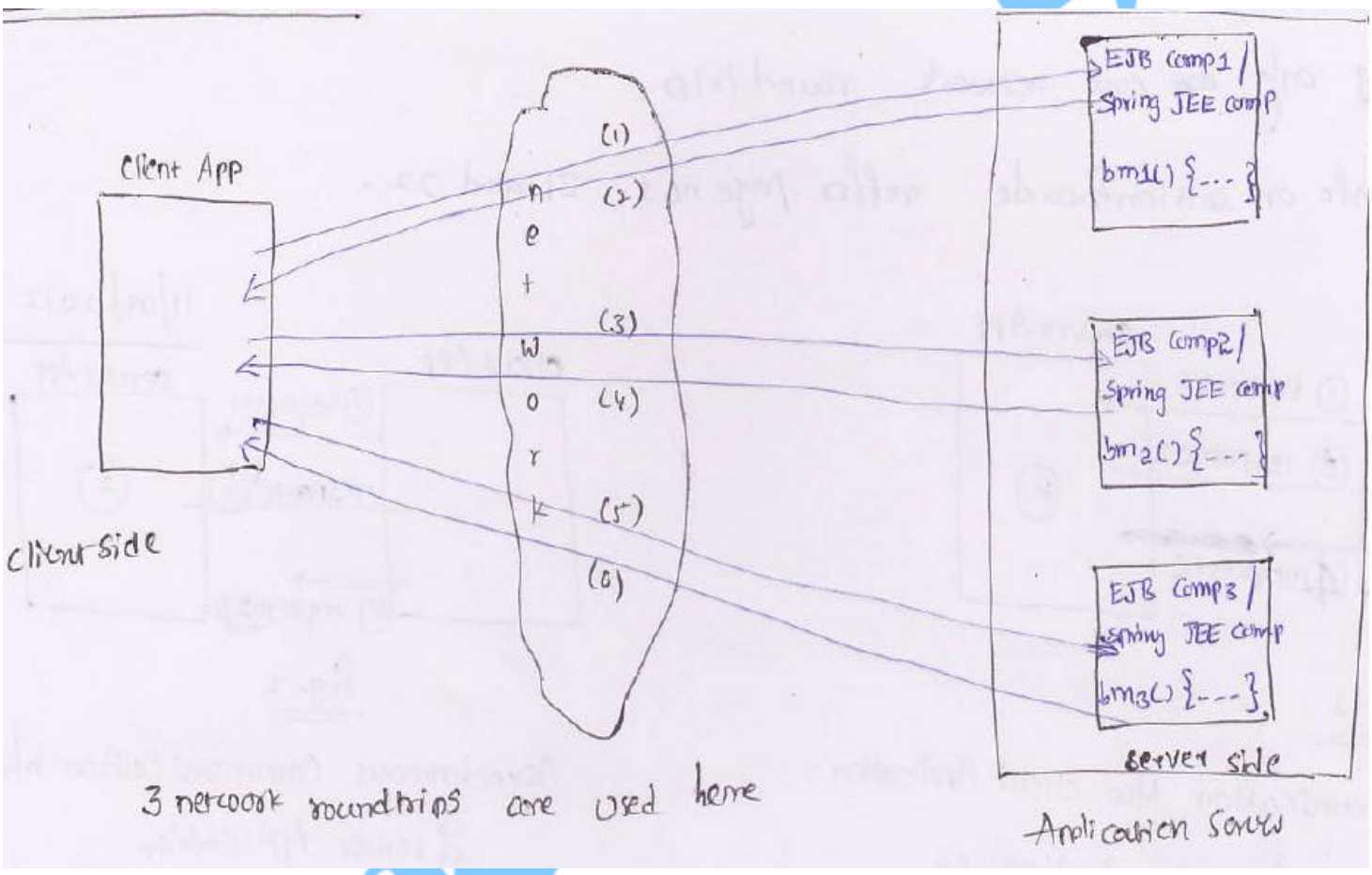
→ AbstractController class is very useful to customize the logics of front-controller program or IntercetingFilter program being from outside of those programs.

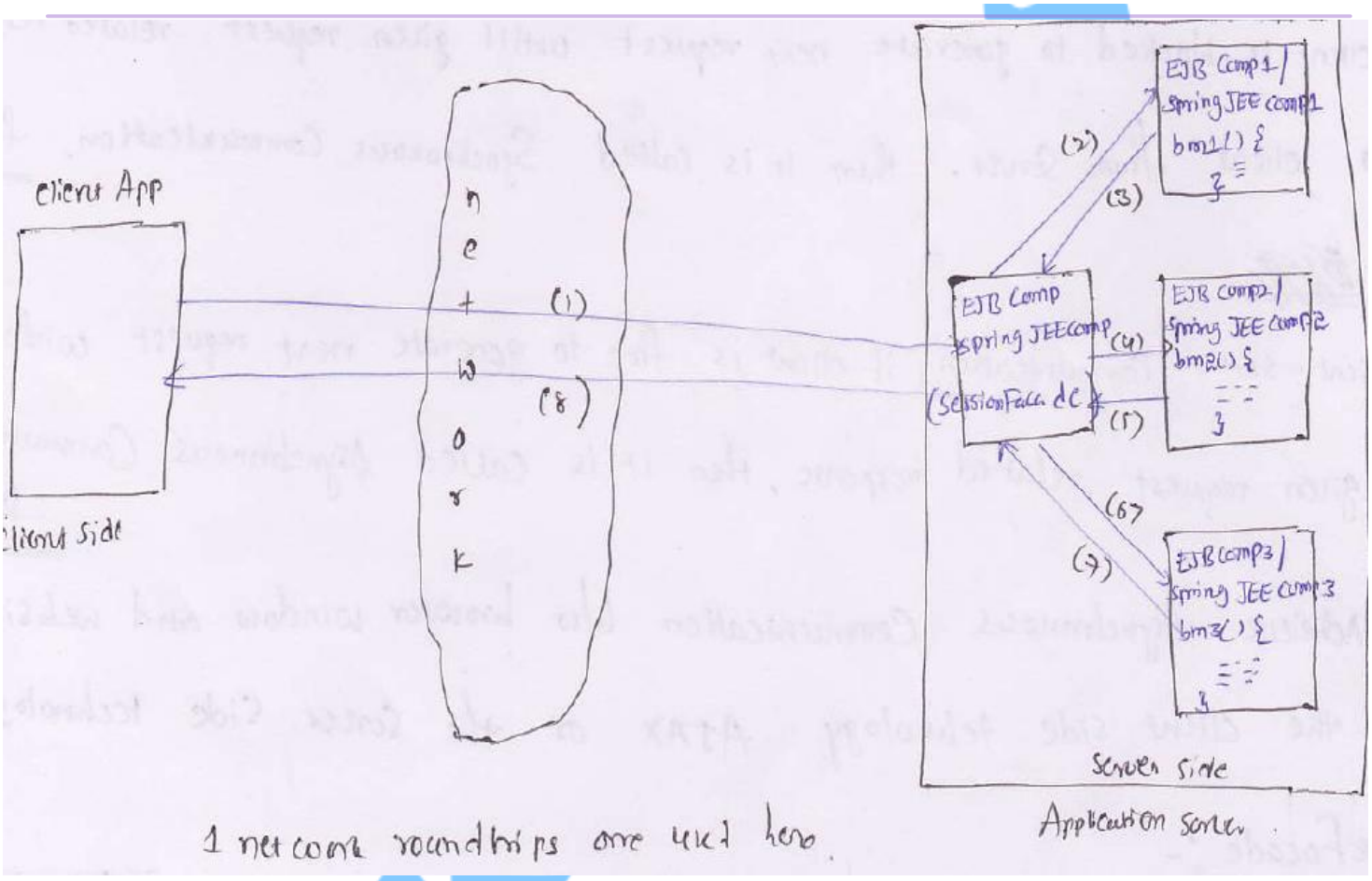
→ In struts1.x environment RequestProcessor class is Abstract controller for the Front controller ActionServlet. Similarly in Struts2.x environment Interceptors are the abstract controllers for Intercepting Filter/Controller called FilterDispatcher.

III) Integration Layers Design Pattern:

1) Session Facade Design Pattern:

Venu Kumaar.S





Problem: If remote client directly talks with multiple business components there is a change of getting multiple network round trips between client and business component as shown in above diagram I.

Solution: To reduce the network round trips the above I scenario take one dummy business component at server side to receive request from client and make that component interacting with other business component as shown above diagram II. Here the dummy business component is called as "SessionFacade". SessionFacade doesn't contain any serious business logic. But it reduce network round trips when client wants to interacting with multiple business component.

→ In the above diagram II client application talking with multiple business component by just utilizing only one network round trip.

→ for related

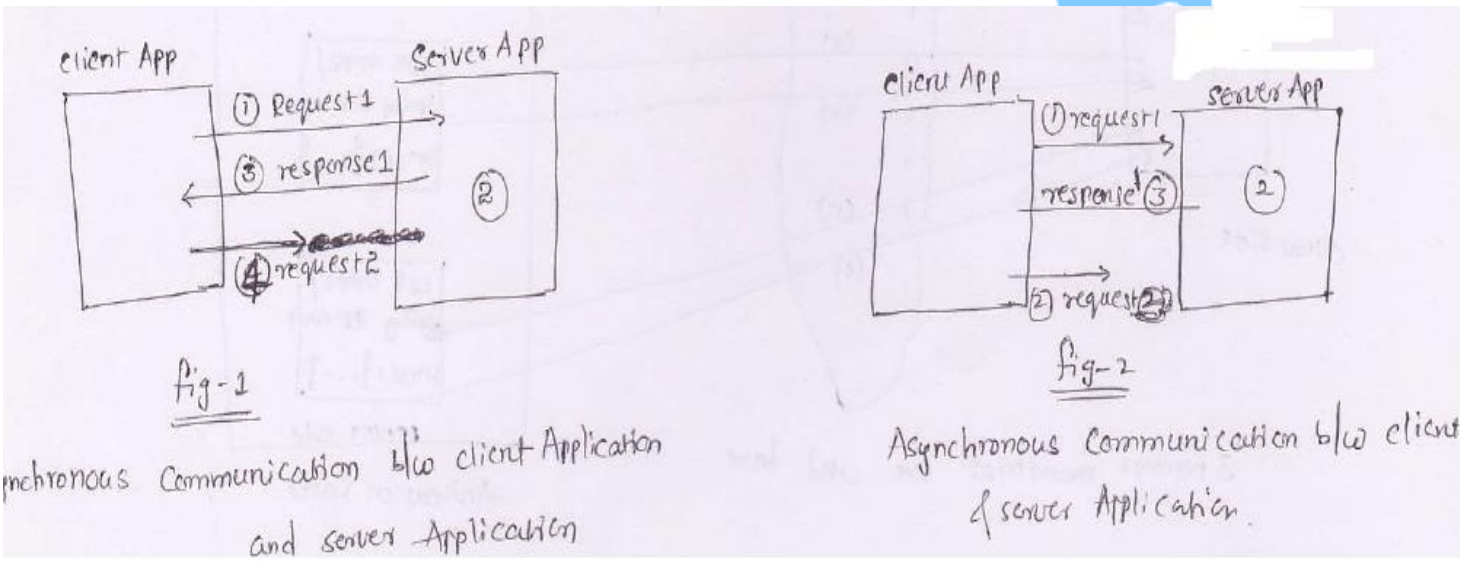


fig-1

nchronous Communication b/w client Application and server Application

fig-2

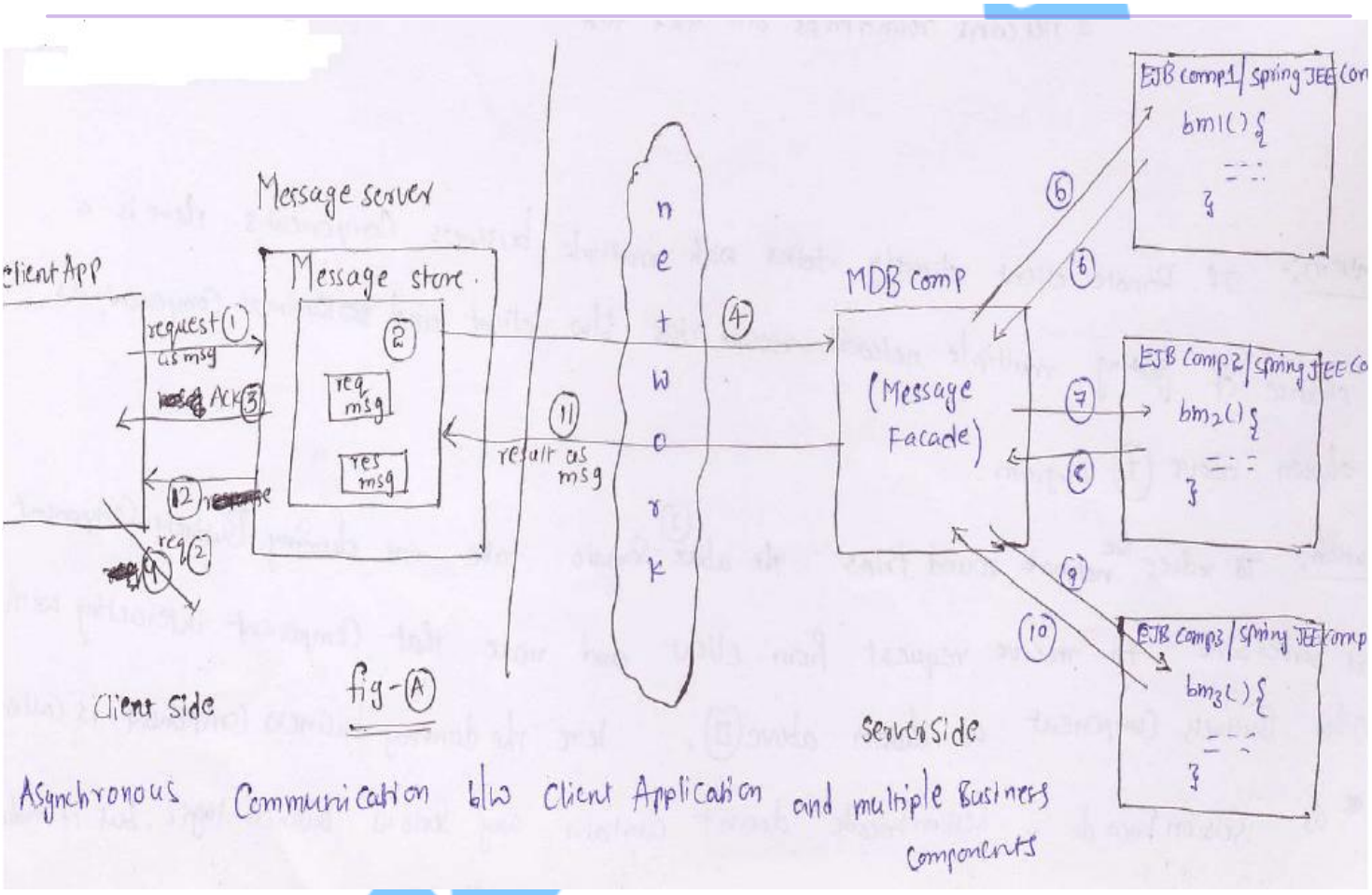
Asynchronous Communication b/w client & server Application.

→ If client is blocked to generate next request until given request related response comes back to client from server, then it is called Synchronous communication. fig -I

→ In client-server communication, if client is free to generate next request without waiting for given request related response, then it is called Asynchronous communication. Fig-II.

→ To achieve Asynchronous communication between browser window and website use either the client side technology AJAX or the server side technology portlets.

2) MessageFacade:



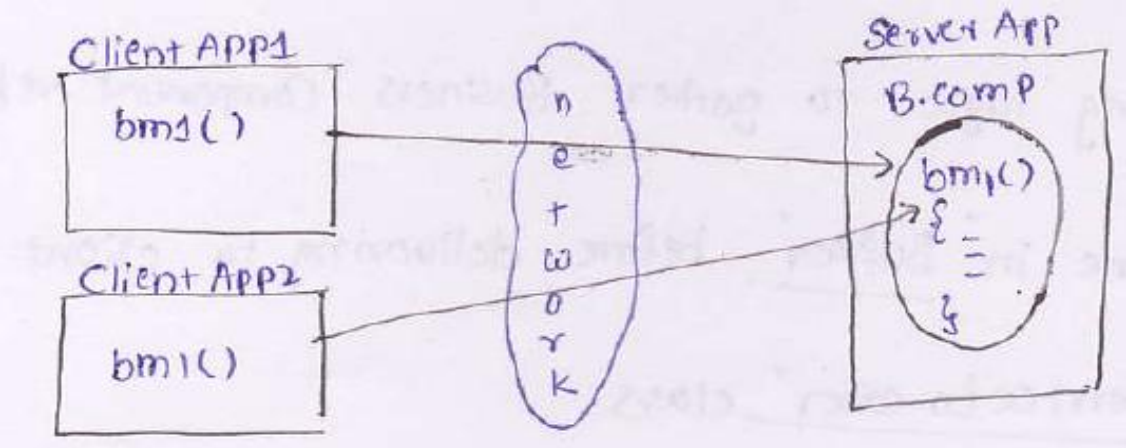
Problem: The implementation of SessionFacade design pattern reduces network round trips between client and business components, but that communication is synchronous communication. so the client is has to wait to generate next request until the result is received from multiple business components.

Solution: Use JMS and MDB support to make communication between client and business components more asynchronous as shown in diagram Fig-A

→ DAO factory is a factory returning one or other DAO class object based on the data that is supplied. This Factory class will be used in other resources of the project to get their choice DAO class object and to perform persistence operations by using that DAO class object.

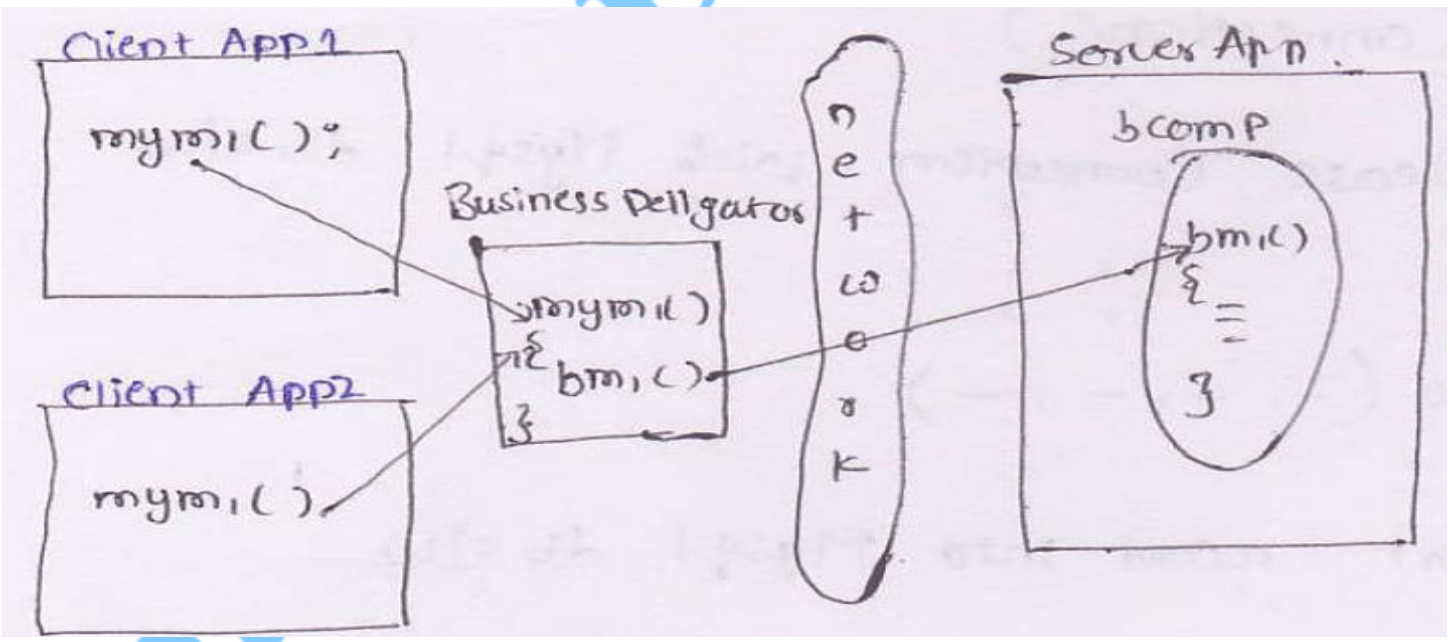
3) Business Delegate:

Problem: Business in client applications calls business methods of business components available in server application directly we don't get flexibility of modification. If business method details are changed in future.



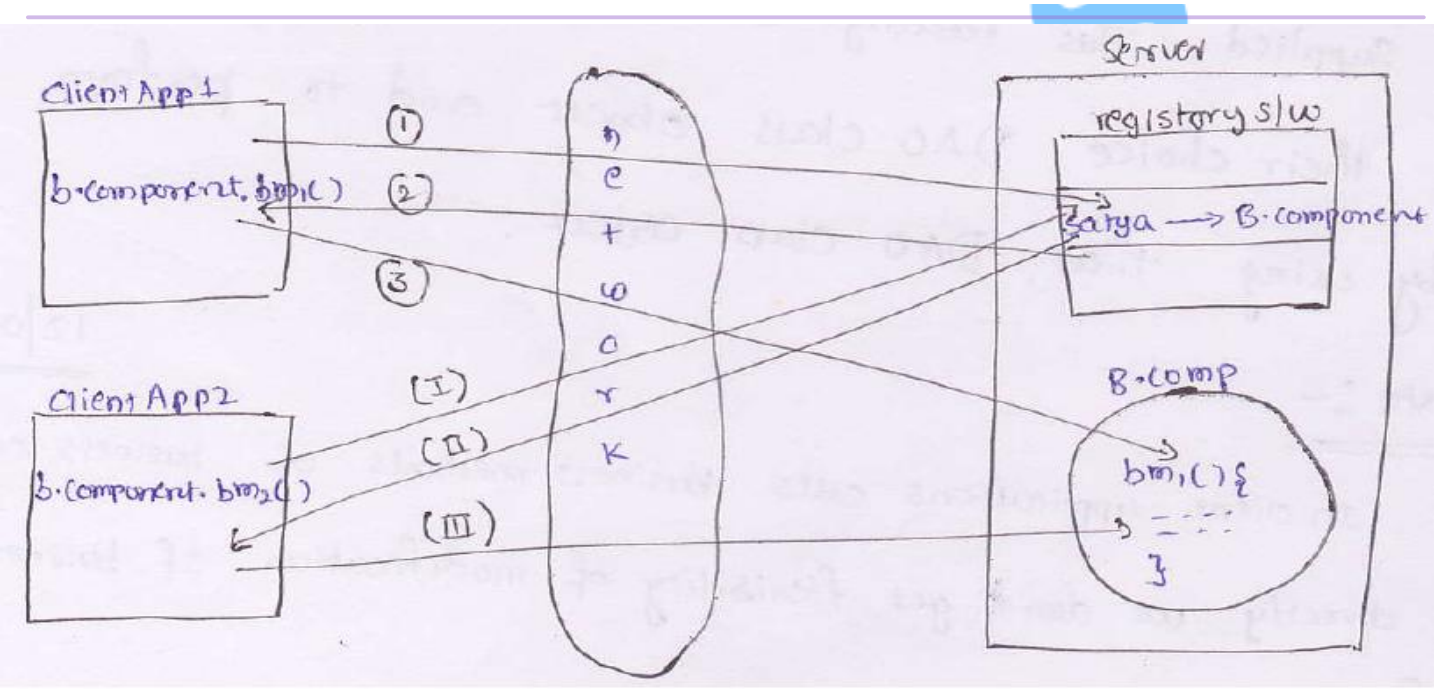
Solution: Developed helper class for client Application having the logics to call the business method of ServerApplication. So, any modifications are there in the details of business method can perform them Business Delegation class and there is no necessary of disturbing client applications.

→ In one angle it calls struts Action class as business delegate when it is having logic calling the business methods of Model layer EJB components or Spring applications.



4) ServiceLocator:

Problem: To call Business methods of Business components belonging to server the client Applications must gather business component references from registry s/w using JNDI code. If every client application separately gathers this business component reference from registry that may increase more network round trips.

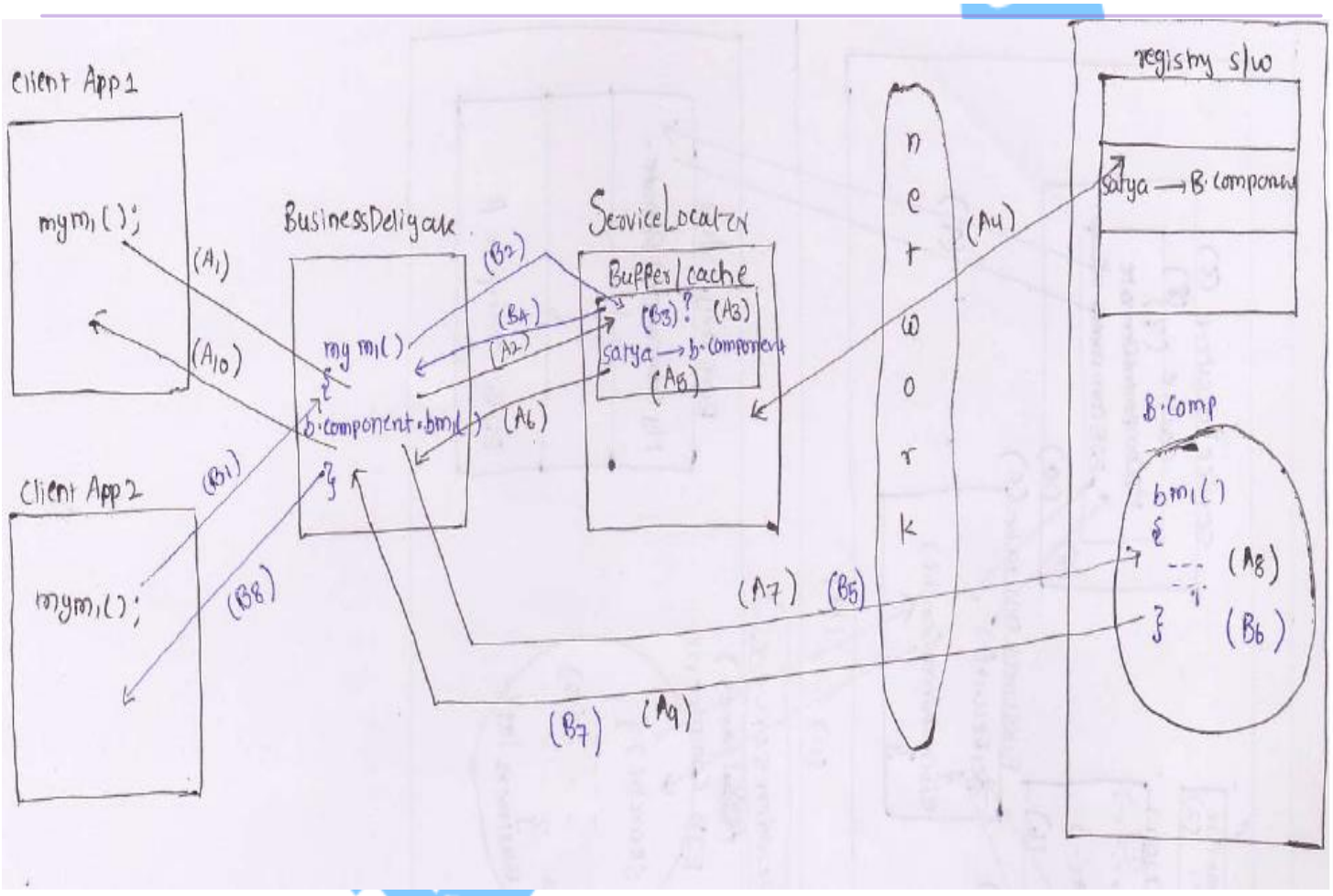


Solution: Develop Helper class having logic to gather business component reference from registry of server and to keep that one in "buffer" before delivering to client Applications. this helper class is called "ServiceLocater" class.

→ We generally develop this ServiceLocate class as Singleton java class to avoid multiple buffers to creations.

→ We generally used ServiceLocate class along with Business Delegate class.

→ Here Service Locator responsible to gather and give business object/component reference and Business Delegate responsible to call business methods by using business object reference and to pass the results to client application.



→ In the above diagram A1 to A10 indicates client applicaion1 flow of execution where business object reference is gathered from registry and registered in the Buffer of Service Locator before getting use.

→ B1 to B8 indicates client applicaion2, flow of execution where the business component reference is gathered from Buffer or cache of service locator call the business methods.

IV) Model Layer

DAO(Data Access Object):

Problem: Mixing-up persistence logic with other logics of application(Specially with business logic) does not give flexibility of modification for persistence logic when database s/w is changed or its details are changed.

Solution: Implement DAO design pattern. It is a java class that separates persistence logic from other logics of the application development and provides flexibility of modification.

DAO class contains the following logics:

1) Logic to establish the connection.
2) Logic to release the connection
3) Logic to perform CRUD operations on table as per project requirement.

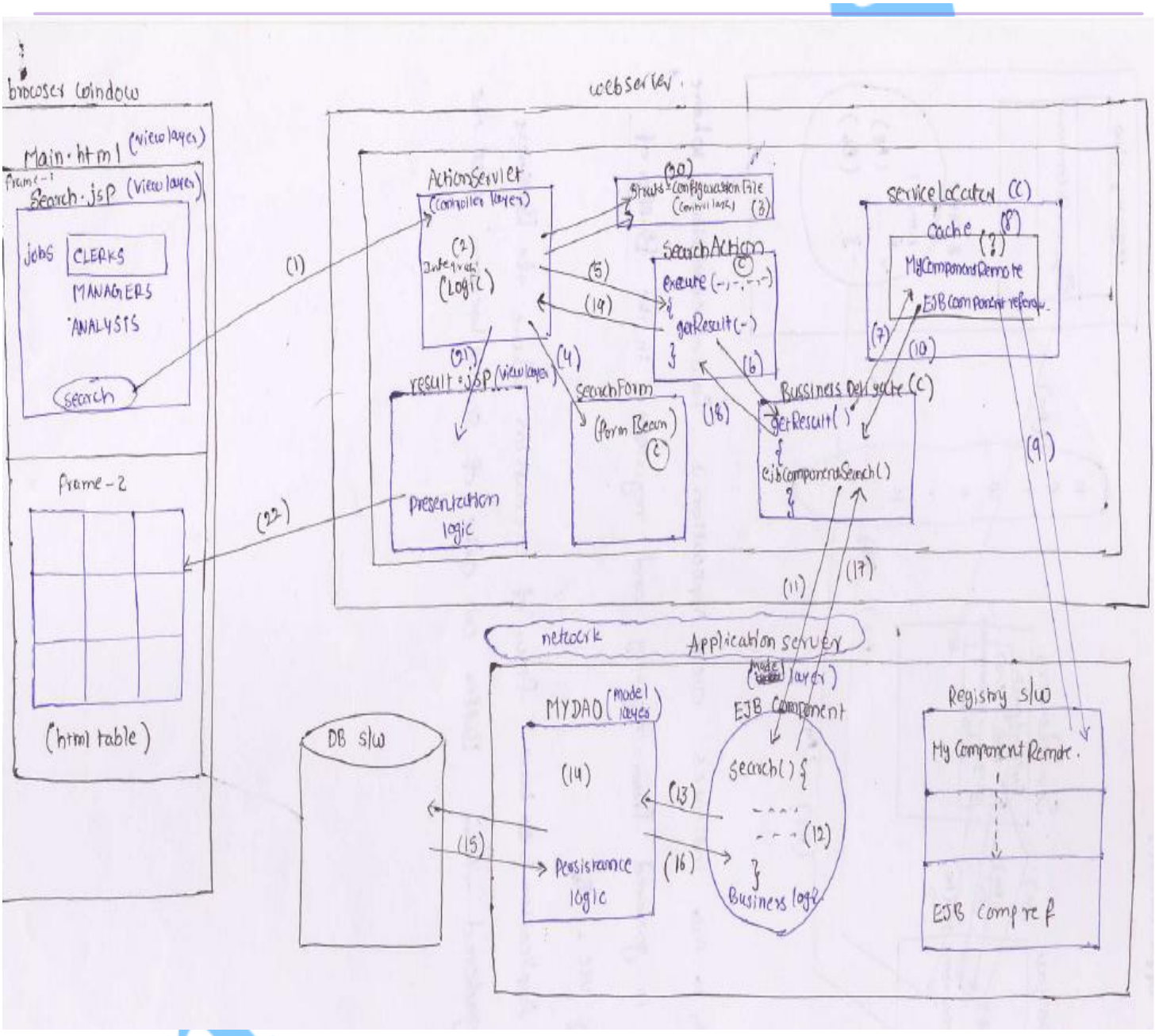
→***Note: In one project we can have wither one DAO class or multiple DAO classes.
To develop persistence logics in DAO class we can use any persistence technologies like JDBC, Hibernate & etc...

DAO Factory: It is extension of DAO, Factory, patterns which contains the ability to return one db s/w specific DAO class object based on the data that is supplied. This is useful when project deals with multiple db s/ws performing some persistence operations on multiple database softwares.

```
public class MyDaoFactory
{
//factory method
public static DAO getDAO(String name)
{
if (name.equals("oracle"))
 return OracleDAO();
 else if(name.equals("MySql"))
 return MySqlDAO();
 else
 return null;
}//getDAO()
}//class

abstract class DAO
{
public abstract void makeConnection();
public abstract void releaseConnection();
public abstract int insertInfo(-,-,-,-);
public abstract int updateInfo(-,-,-,-);
}//class

//DAO class for Oracle
class OracleDAO extends DAO
{
public abstract void makeConnection()
{
-----;
-----; //logic to establish connection with oracle db s/w
}
public abstract void releaseConnection()
{
-----;
-----; //logic to release connection with oracle db s/w
}
public abstract int insertInfo(-,-,-,-)
{
-----;
-----; //logic to insert record into Oracle table
}
public abstract int updateInfo(-,-,-,-)
{
-----;
-----; //logic to update record into Oracle table
}
```

```
}//class

//DAO class for MySql
class MySqlDAO extends DAO
{
public abstract void makeConnection()
{
-----;
-----; //logic to establish connection with MySql db s/w
}
public abstract void releaseConnection()
{
-----;
-----; //logic to release connection with MySql db s/w
}
public abstract int insertInfo(-,-,-,-)
{
-----;
-----; //logic to insert record into MySql table
}
public abstract int updateInfo(-,-,-,-)
{
-----;
-----; //logic to update record into MySql table
}
}//class
```

→ **MINI project by using Struts, EJB and JDBC technologies having the implementation of multiple design patterns:**



with respect to diagram:

1) From page submits the request by selecting one item of select box.
2) As a controller ActionServlet traps and takes the request
3) ActionServlet uses the entries of Struts-configuration file decide form bean and Action class to process the request.
4) ActionServlet writes the form data to Form Bean class object
5) ActionServlet calls the execute() of Action class.
6) This execute() passes the request to BusinessDeligate class
7) & 8) BusinessDeligate uses ServiceLocater class and checks the availability of EJB component reference in Buffer or cache.
9) since EJB component reference is not available in Buffer it will be gathered from the registry s/w of Applicaion Server and will be registered with the Cache or Buffer of ServiceLocater.

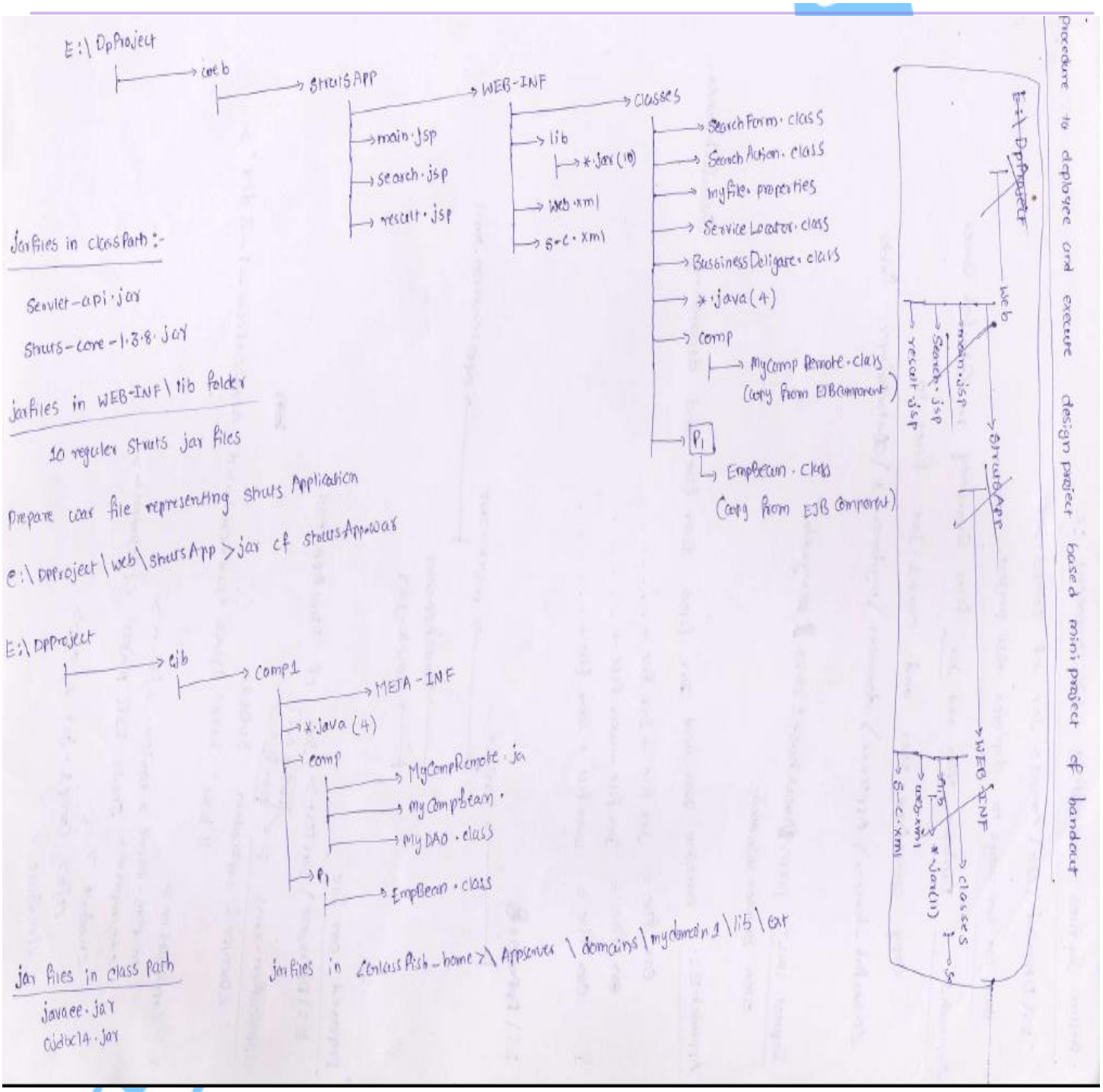
10) Service Locator passes the EJB component reference to Business Delegate class
11) Business Delegate class uses that EJB component reference and calls the Business method of EJB component
12), 13), 14), 15) &16) this business method takes the support of DAO class to send and executes SQL select query db table and gets the records into ArrayList object.
17) Business method of EJB component passes the result(ArrayList object) to business Delegate class
18) Business Delegate class passes the result to execute().
19)execute() keep the result in request attribute and transfer the control to ActionServlet
20)&21) ActionServlet uses the entries of Struts-configuration file get the result page and pass the control to Result.jsp.
22) Result.jsp formats the results by using presentation logic and sends that result to 2nd frame of web page as HTML table content.

Explicitly implemented Design patterns of the above project:

1) MVC2 m⟶ ejb component v⟶jsp programs c⟶ ActinoServlet
2) DAO
3)DTO/VO class
4)BusinessDeligate
5)ServiceLocator
6)Singleton java class ⟶ required while implementing ServiceLocator
7)Factory method ⟶ required while implementing ServiceLocator
8)View Helper Design Pattern.

Some Implicit DesignPatterns of the above project:

1) Front controller(ActionServlet)
2) Dependency Injection/IOC ⟶ The way ActionServlet writes FormData to FormBean class obj
3) AbstractController(The pre-defined helper class to ActionServlet which is nothing but Request processor class)
4) DTO/CO class(FormBean class)
5) Template method(The process(-,-) of Request processor)


E:\DpProject
web
StrutsApp
WEB-INF
classes
main.jsp
search.jsp
result.jsp
lib
*.jar (10)
web.xml
s-c.xml
SearchForm.class
SearchAction.class
myfile.properties
ServiceLocator.class
BussinessDeligate.class
*.java (4)
comp
MyCompRemote.class
Copy from EJBComponent
P1
EmpBean.class
(Copy from EJB Component)
Jarfiles in classPath :-
Servlet-api.jar
Struts-core-1.3.8.jar
jarfiles in WEB-INF\lib folder
10 regular Struts jar files
Prepare war file representing Struts Application
e:\DPproject\web\StrutsApp>jar cf StrutsApp.war
E:\DPproject
ejb
comp1
META-INF
*.java (4)
comp
MyCompRemote.ja
myCompBean.
MyDAO.class
P1
EmpBean.class
jar files in class path
javaee.jar
ojdbc14.jar
jarfiles in <GlassFish-home>\Appserver\domains\mydomain1\lib\ext
Procedure to deploye and execute design project based mini project of handout
E:\DPproject
web
result.jsp
search.jsp
main.jsp
StrutsApp
WEB-INF
s-c.xml
web.xml
*.jar(11)
classes

Prepare jar files representing EJB Component :-

E:\DPproject\ejb\comp1> jar cf comp1.jar

there are two ways to deployee this project.

Approach 1 :- Deployee War and jar files Seperately in Glassfish Server

Copy StrutsApp.war and comp1.jar files to

<Glassfish_home>\AppServer\domains\mydomain1\autodeployee folder

Request URL: http://localhost:8080/mywebapp

Open browser window

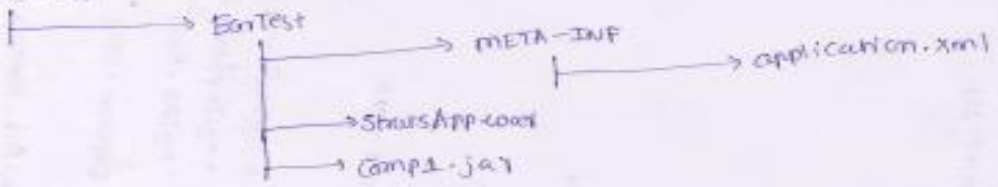
Approach 2 :- Combine War and Jar files Ear file and deployeeing Glassfish Server.

ear file = jar file + jar file + ...

ear file = jar file + war file + ...

Ear file = war file + war file + ...

E:\DpProject



To prepared ear file.

E:\DpProject\EarTest> jar cf FinalApp.ear

application.xml :- (DOCTYPE for ear file)

<!DOCTYPE application PUBLIC

"-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN" http://java.sun.com/dtd/application_1_3.dtd">

<application>

<display-name> EarTest </display-name>

<description> Struts EJB project </description>

<module>

<ejb> comp1.jar </ejb>

Types of Design Patterns:

There are mainly three types of design patterns which are further divided into design sections.

- **Creational Patterns**
  - Factory Pattern
  - Abstract Factory Pattern
  - Singleton Pattern
  - Builder Pattern
  - Prototype pattern
- **Structural Patterns**
  - Adapter pattern
  - Bridge pattern
  - Composite pattern
  - Decorator pattern
  - Façade pattern
  - Flyweight pattern
  - Proxy pattern : If creation of object is expensive, its creation can be postponed till the very need arises and till then, a simple object can represent it. Like instead of sending image we send a URL link, as it save time and relative cost of transferring image.

```
interface Image {
   void display();
}

class RealImage implements Image {

        private String fileName;

        public RealImage(String fileName){
          this.fileName = fileName;
          loadFromDisk(fileName);
        }

        @Override
        public void display() {
          System.out.println("Displaying " + fileName);
        }
```

```
        private void loadFromDisk(String fileName){
          System.out.println("Loading " + fileName);
        }
      }
class ProxyImage implements Image{

        private RealImage realImage;
        private String fileName;

        public ProxyImage(String fileName){
          this.fileName = fileName;
        }

        @Override
        public void display() {
          if(realImage == null){
            realImage = new RealImage(fileName);
          }
          realImage.display();
        }
      }

public class ProxyPatternDemo {

        public static void main(String[] args) {
          Image image = new ProxyImage("test_10mb.jpg");

          //image will be loaded from disk
          image.display();
          System.out.println("");
          //image will not be loaded from disk
          image.display();
        }
      }
```

- **Behavioral Patterns**
  - Command Pattern
  - Observer pattern
  - Iterator pattern
  - Strategy pattern
  - Template pattern
  - Visitor pattern