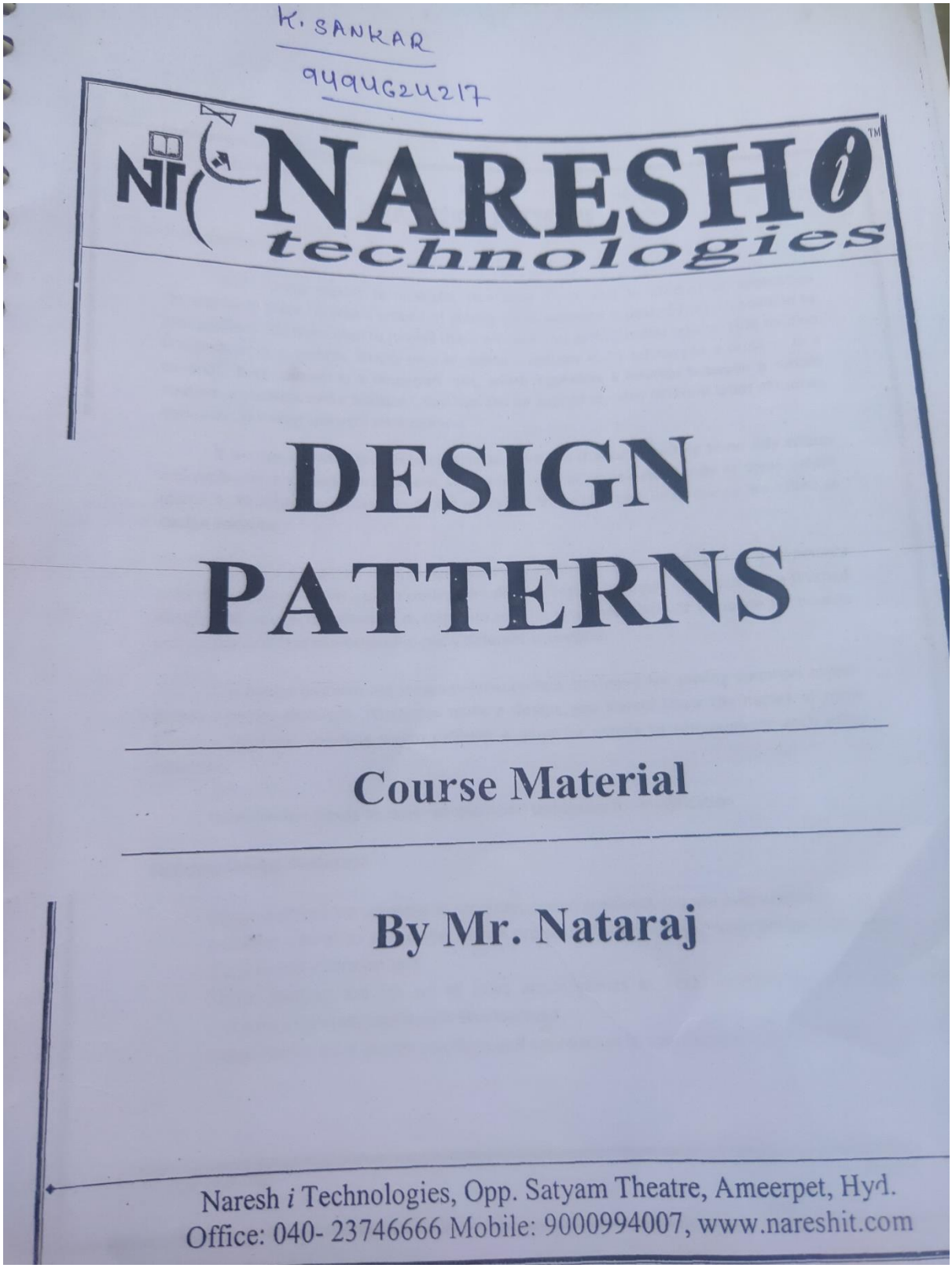

K. SANKAR
9949624217
NARESHi
technologies
DESIGN
PATTERNS
Course Material
By Mr. Nataraj
Naresh i Technologies, Opp. Satyam Theatre, Ameerpet, Hyd.
Office: 040- 23746666 Mobile: 9000994007, www.nareshit.com

JAVA DESIGN PATTERNS

Introduction

While using regular technologies (like Java ,C++,C etc) in product or application development there may be a chance of getting some problems repeatedly then a solution to that problem has been used to resolve those problems for getting better results. That solution is described as a pattern. Simply we can define a **pattern** as "A solution to a problem in a context". **Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution".** Patterns can be applied to many different areas of human endeavor, including software development.

If we use software technologies directly there is a chance of getting some side effects and problems in project development. We generally write some helper code or some helper resources to solve the problem in a best manner. This helper code or resources are called as **Design patterns.**

In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

The design patterns are language-independent strategies for solving common object-oriented design problems. When you make a design, you should know the names of some common solutions. Learning design patterns is good for people to communicate each other effectively.

Note: Design should be open for extension but closed for modification

Defining Design Patterns?

- ✓ Design patterns are solutions to recurring design problems you see over and over.
- ✓ A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it.
- ✓ Design patterns are the set of rules which comes as best solutions for recurring problems of project/application development.
- ✓ Design Patterns are proven solutions and approaches to specific problems.

What is Gang of Four?

One Day, Four Computer Scientist (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) met together, and had a big conversion about the Design Pattern. These gang of four people were called **Gang of Four.**

They divided total design Pattern into 3 part (Creational, Structural, Behavioral)
In General, There may be more 100+ design pattern. But they tried to find out the basic design patterns, from which other design patterns might have come. Finally they concluded 23 basic design Pattern. All other design pattern are extended from these patterns

The Gang of Four describes design patterns are **"descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."**

What is the need to use the design pattern?

The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use.

- Designing object-oriented software is hard and designing reusable object-oriented software is even harder - Erich Gamma.
- Learning design patterns speeds up your experience accumulation in OOA/OOD.
- Design patterns are the best practices to use software technologies more effectively in project development.
- A design pattern names, abstracts, and identifies key aspects of a common design structure that makes it useful for creating a reusable object-oriented design.
- Well-structured object-oriented systems have recurring patterns of classes and objects.

Difference between Design patterns and Anti Patterns

- The best solution for a recurring problems is **Design patterns** where as the worst solution for recurring problems is **Anti patterns.**

Note that the design patterns are not idioms or algorithms or components, just they are giving some hint to solve a problem effectively. Design patterns has no relation with designing phase of the project, they will be purely implemented in the development phase of the project.

Designing patterns can be implemented by using any programming language. Since Java is popular for large scale projects, we can see more utilization of design patterns in java.

History

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 the bad practices?
➔What are best 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 or GO4).**

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.

What is a Pattern/Design pattern?

Patterns are about documenting a solution for a well know (recurring) problem ina 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.

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.

Pattern Template/pattern Elements

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) **pattern name**: Having a concise, meaningful name for a pattern improves communication among developers

b) **Problem**: →What is the problem and context where we would use this pattern?
→What are the conditions that must be met before this pattern should be used?

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

d) **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

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

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

Classification of Design Patterns/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 Persistence -tier are called presentation-tier catalog patterns. But there is no particular process in place to identify or classify them in groups.

GOF Pattern catalog

According to GOF, all the Design Patterns can be classified into following category: Creational, Structural and Behavioral patterns.

1. Creational Patterns - Concern the process of object creation
2. Structural Patterns - Deal with the composition of classes and objects
3. Behavioral Patterns - Deal with the interaction of classes and objects

Creational Patterns: how an object can be created i.e. creational design patterns are the design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation (using **new** keyword) could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

All the creational patterns define the best possible way in which an object can be instantiated. These describes the best way to CREATE object instances. According to GOF, creational patterns can be categorized into five types.

1. Factory Pattern
2. Abstract Factory Pattern
3. Prototype Pattern
4. Builder Pattern
5. Singleton Pattern

and etc...

Structural Patterns: Structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities. Structural Patterns describe how objects and classes can be combined to form larger structures. According to GOF, Structured Pattern can be realized in the following patterns:

1. Adapter Pattern
2. Bridge Pattern
3. Composite Pattern
4. Decorator Pattern
5. Facade Pattern
6. Flyweight Pattern
7. Proxy Pattern

and etc..

Behavioral Patterns: Behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication i.e. prescribes the way objects interact with each other. They help make complex behavior manageable by specifying the responsibilities of objects and the ways they communicate with each other. The 11 behavioral patterns are:

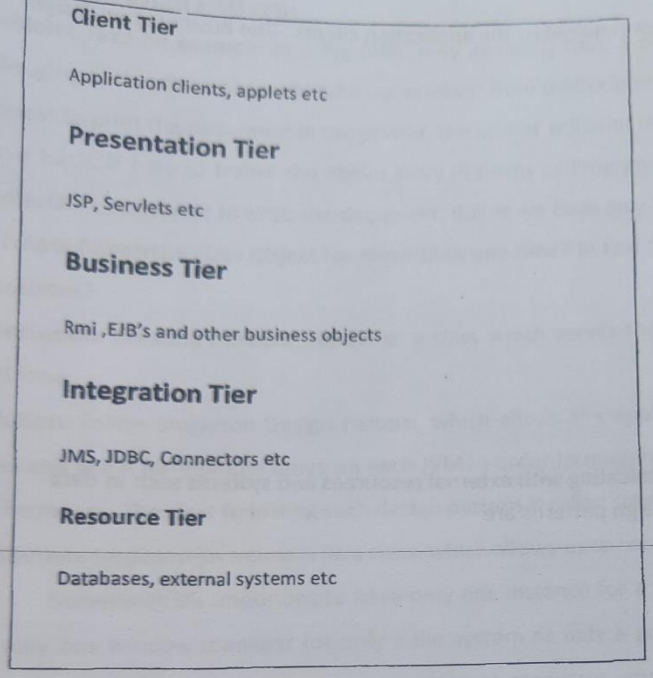
1. Chain of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Momento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern

J2EE/JEE patterns (Sun Ms JEE Patterns)

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 patterns

This tier contains all the presentation tier logic required to service the clients that access the system. The presentation tier design patterns are

- Intercepting Filter
- Front Controller
- Context Object
- Application Controller
- View Helper
- Composite View
- Service to worker
- Dispatcher View

Singleton Design Pattern:

Problem: Take an example of a Big MNC.They generally have 1 central printer for each floor.All associates belonging to that floor generally take the printout from that printer only. Even though 10 associates has given request to print the document in the printer, the printer will print the document one by one on First come first serve basis. If I try to frame the above story in terms of Programming, all associates will call the print() on PrinterUtil class object to print the document. But as we have only one Printer, should we allow the associates to create PrinterUtil class Object for more than one time? In real time, should we install one printer for each associates?

Conclusion: Creating multiple objects of a class which serves the same functionality is wastage of memory and time.

Solution: Follow Singleton Design Pattern, which allows the application to create only one object of a java class and use it for multiple times on each JVM in order to minimize the memory wastage and to increase the performance. The class following such design pattern is called singleton java class.

Definition: Singleton java class is java class, which allows us to create only one object per JVM.

Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager (or only a file system or only a print spooler). Usually singletons are used fo centralized management of internal or external resources and they provide a global point of access t themselves.

The singleton java class is used to encapsulate the creation of an object in order to maintain contr over it. This not only ensures only one object is created, but also allows **lazy instantiation** i.e. the instantiatic of object can be delayed until it is actually needed. This is especially beneficial if the constructor needs perform a costly operation, such as accessing a remote database.

Intent:

- ✓ Ensure that only one instance of a class is created.
- ✓ Provide a global point of access to the object.

Note: For a normal java class if programmer or container is creating only one object even though that class allows to create multiple objects then that java class is not singleton java class. According to this, then a java class of servlet program is not singleton java class. It is a normal java class for which servlet container creates only one object.

Business Tier Patterns

This tier provides business service required by the application clients , The Business tier design patterns are

- Business Delegate
- Service Locator
- Session Façade
- Application Service
- Business Object
- Composite Entity
- Transfer Object
- Transfer Object Assembler
- Value List Handler

Integration Tier patterns

This tier is responsible for communicating with external resources and systems such as data stores etc. The integration tier design patterns are

- Data Access Object
- Service Activator
- Domain Store
- Web Service Broker

- The singleton pattern should be used when there must be exactly one instance of a class, and when it must be accessible to clients from a global access point.
- If multiple applications of a project that are running from a single JVM wants to work with objects of java class having same data then it is recommended to make that java class as singleton java class. So that only one object will be allowed to create for that class and we can use that object for multiple tiles in multiple applications.

Ex:

I. In log4j environment, the **Logger** class is given as singleton java class.

II. java.lang.Runtime class is singleton java class

III. java.awt.Desktop is singleton java class

Rules To Develop Singleton Java Class:

1. Declare a private static reference variable to hold current class Object.This reference will hold null only for the first time,after then it will refer to the object forever(till JVM terminates).We will initialize this reference using static factory method as discussed in step 3.

```
classPrinterUtil {
privatestaticPrinterUtilinstance=null;
}
```

2. Declare all the constructor as private so that its object cannot be created from outside of the class using new keyword.

```
classPrinterUtil {
        privatestaticPrinterUtilinstance=null;
        privatePrinterUtil() {
                System.out.println("PrinterUtil()");
        }
```

3. Develop a static final factory method, which will return a new object only for the first time and the same object will be returned then after. Since we have only private constructor, we cannot use new keyword from outside of the program, we must declare this method as static, so that it can be

accessed directly using Class Name. Declare this final so that the child class will have no option to override and change the default behavior.

```
publicstatic finalPrinterUtilnewInstance(){
                if(instance==null)
instance=newPrinterUtil();
                returninstance;
}
```

4. Make Your Singleton class Reflection API proof.

We know that Reflection API can access the private variables,methods and constructors of the class, hence even if your constructor is private,we can still create the object of that class. To prevent this declare an instance boolean variable initially holding true. Change its value to false,immediately when constructor is called for the first time.Then after when even the constructor is called for 2nd time,it should throw SomeException saying object cannot be created for multiple times.

This approach also removes the Double Checking Problem in case of Multiple thread trying to create object at the same time,which we will discuss later.

```
publicclassPrinterUtil{
     privatestaticbooleanisNew=true;//1st Time-true,2nd Time-false,Used for Reflection
     API Proof,and Multi-Thread double check

     privatePrinterUtil() {
          //To prevent Reflection API  creating Multiple Objects
     if(isNew) {
          isNew=false;
          System.out.println("PrinterUtil()");
          }
    else  {
          thrownewInstantiationError("Cannot Create Multiple Object");
          }
    }
```

5. Make Your factory Method Thread Safety, so that Only one object is created even if more than 1 thread tries to call this method simultaneously. Declare the whole method as synchronized method, or use synchronized block

```
publicsynchronizedfinalstaticPrinterUtilgetInstance()
{
            if(instance==null)
instance=newPrinterUtil();
```

```
        returninstance;
    }
```

Instead of making the whole factory method as synchronized method, it is good to place only the condition check part in **synchronized** block.

```
publicstaticfinalPrinterUtilgetInstance()
{

    synchronized(PrinterUtil.class){
            if(instance==null){
                instance=newPrinterUtil();
            }
    }
    returninstance;
}
```

we have a problem with the above code, after the first call to the getInstance(), in the next calls to the same getInstance() method,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 must try to avoid them as much as we can. To solve this problem we can have double level checking (2 times null checking) for the condition as shown below.

```
publicstaticfinalPrinterUtilgetInstance()
    {
If(instance==null){ //1st null check
        synchronized(PrinterUtil.class){
            if(instance==null){
                instance=newPrinterUtil(); //2nd null check
            }
        }
    }
    returninstance;

}
```

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

```
public class PrinterUtil {
private static volatilePrinterUtil instance;
    .....
    .....
    }
```

Note: If you have used the Reflection Proof logic, then no need to worry about the 2nd null check. Because when you call the constructor for 2nd time, it will throw InstantiationError

6. Prevent Your Singleton Object from De-serialization. If you need your singleton object to send across the network, Your Singleton class must implement Serializable interface.But problem with this

approach is we can de-serialize it for N number of times,and each deserializationprocess will create a brand new object,which will violate the Singleton Design Pattern.

In order to prevent multiple object creation during deserialization process, override **readResolve()** and return the same object. readResolve() method is called internally in the process of deserialization. It is used to replace de-serialized object by your choice.

```
publicclassPrinterUtil{
      privatestaticPrinterUtil instance=null;

      privatePrinterUtil() {….}

      //Any Deserialization Process will give you the same Object
      protected Object  readResolve()
      {
            System.out.println("readResolve()");
            return instance;
      }

}
```

Note: Ignore this process if your class does not implement Serializable interface directly or indirectly. Indirectly means the super class or super interfaces has not implemented/extended Serializable interface.

7. Prevent Your singleton Object being Cloning.If your class is direct child of Object class,then I will suggest not to implementCloneable Interface, as there is no meaning of cloning the singleton object to produce duplicate objects out of it. Both are opposite to each other. However if Your class is the child of some other class or interface and that class or interface has implemented/extended Cloneable interface, then it is possible that somebody may clone your singleton class thereby creating many objects. We must prevent this as well.

   Override clone() in your singleton class and return the same old object. You may also throw CloneNotSupportedException.

```
publicclassPrinterUtil{
      privatestaticPrinterUtilp=null;

      privatePrinterUtil() {….}

      //Any Cloning Process will return you the Same Old object
      public Object clone() throwsCloneNotSupportedException
```

// ligel instance of singleton java class throw static block of that class is not recommended approach to follow. because it creates object of singleterm class the movement class is loaded if that object is not asked by any outsider of the class that object will be wasted.


```
        {
            throw new CloneNotSupportedException();
            //returnp; //If you want to return the same old Object
        }

    }
```

8. Inspite of All the above efforts, There is still a loop hole "The boss Reflection API".Using reflection API, the programmer can get access to private constructors, variables, and methods.We have to prevent this for Singleton Design Pattern.

   Declare a static instance variable to count how many times the object .For the first time when ever constructor is called, increment the count to 1.Next time when constructor is called, check if the value is one or not. If yes, throw InstantiationError

9. Use static-block or static definition. If you feel you don't want to use synchronized method or block but still want to achieve singleton behavior. You can use static-block or static definition to initialize the singleton java class object as follows.

```
publicclassPrinterUtil{
    privatestaticPrinterUtilp=new PrinterUtil();//static definition

    /* OR
    private static PrinterUtil p=null;
    static{
        p=new PrinterUtil();
        }
    */

    privatePrinterUtil() {

    }

    publicfinalstaticPrinterUtilgetInstance()
    {
        returninstance;
    }
}
```

Note: You have to take care of all the other problems except Multithreading.

This approach will create the Object even if you don't need them urgently (during class loading).This is not used so frequently in the industry.

//Serialization:-

-> the process of converting objects data into bits and bytes is called serialization. these bits & bytes can be written as files. (or) can be written over the N/w, who class is implements java.io.Serializable() is called serialization object.

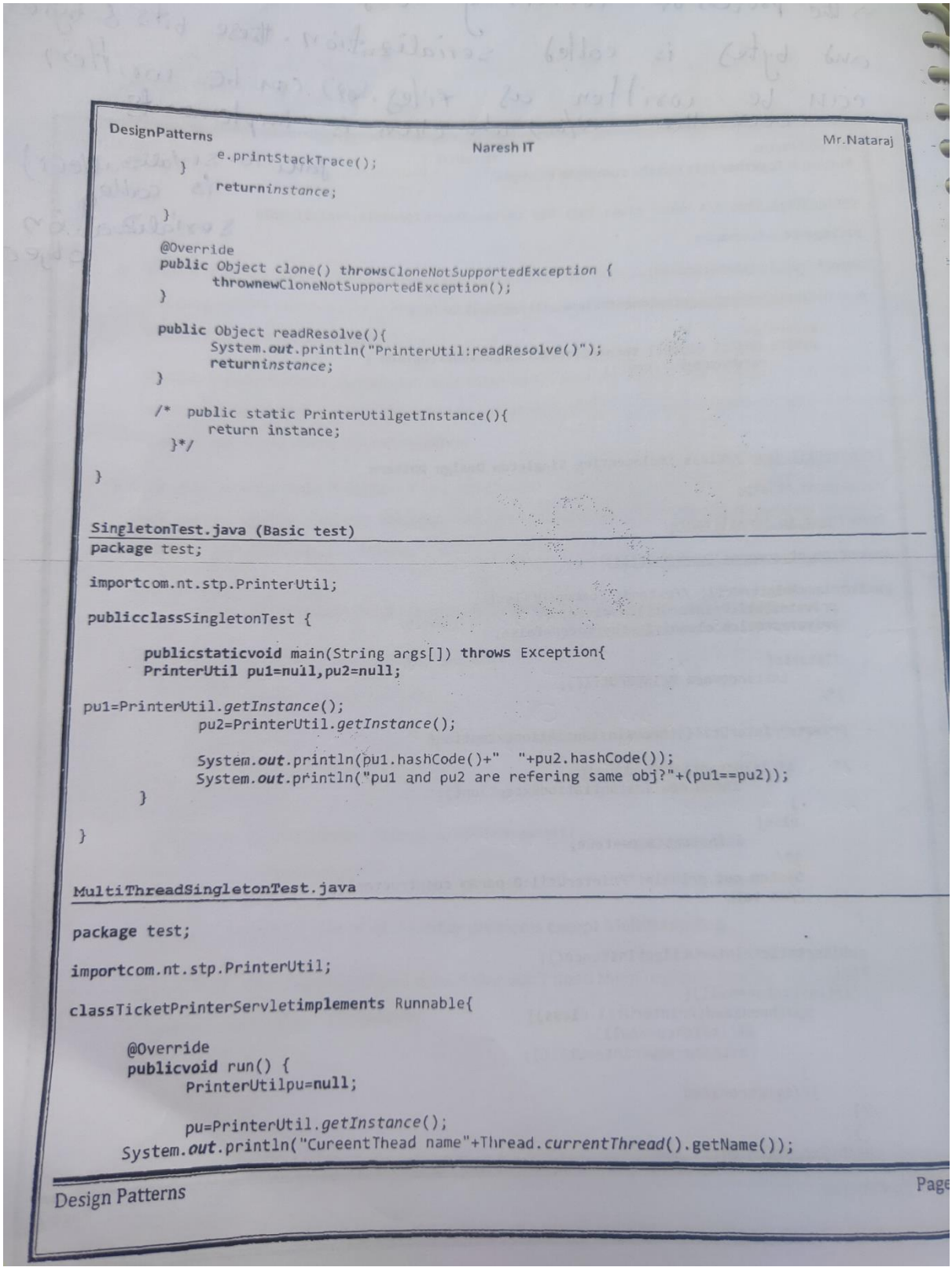
Putting it Together Lets see the complete Example

CommonsUtil.java //A super class that has implemented Cloneable,Serializable

```
package com.nt.commons;

import java.io.Serializable;

public class CommonsUtils implements Cloneable,Serializable {

        @Override
        public Object clone() throws CloneNotSupportedException {
                return super.clone();
        }

}
```

PrinterUtil.java //Class implementing Singleton Design Pattern

```
package com.nt.stp;

import java.io.Serializable;

import com.nt.commons.CommonsUtils;

public class PrinterUtil{ //extends CommonsUtils {
        private static PrinterUtil instance;
        private static boolean isInstantiated=false;

        /*static{
                instance=new PrinterUtil();
        }*/

        private PrinterUtil()throws InstantiationException{

        /*      if(isInstantiated==true){
                        throw new InstantiationException();
                }
                else{
                        isInstantiated=true;
                }*/
                System.out.println("PrinterUtil:0-param cosntructor");
                //no task
        }

        public static PrinterUtil getInstance(){
        try{
        //if(instance==null){
                synchronized(PrinterUtil.class){
                        if(instance==null){
                        instance=new PrinterUtil();
                }
                }//synchronized
        //}
          }
        catch(Exception e){
```



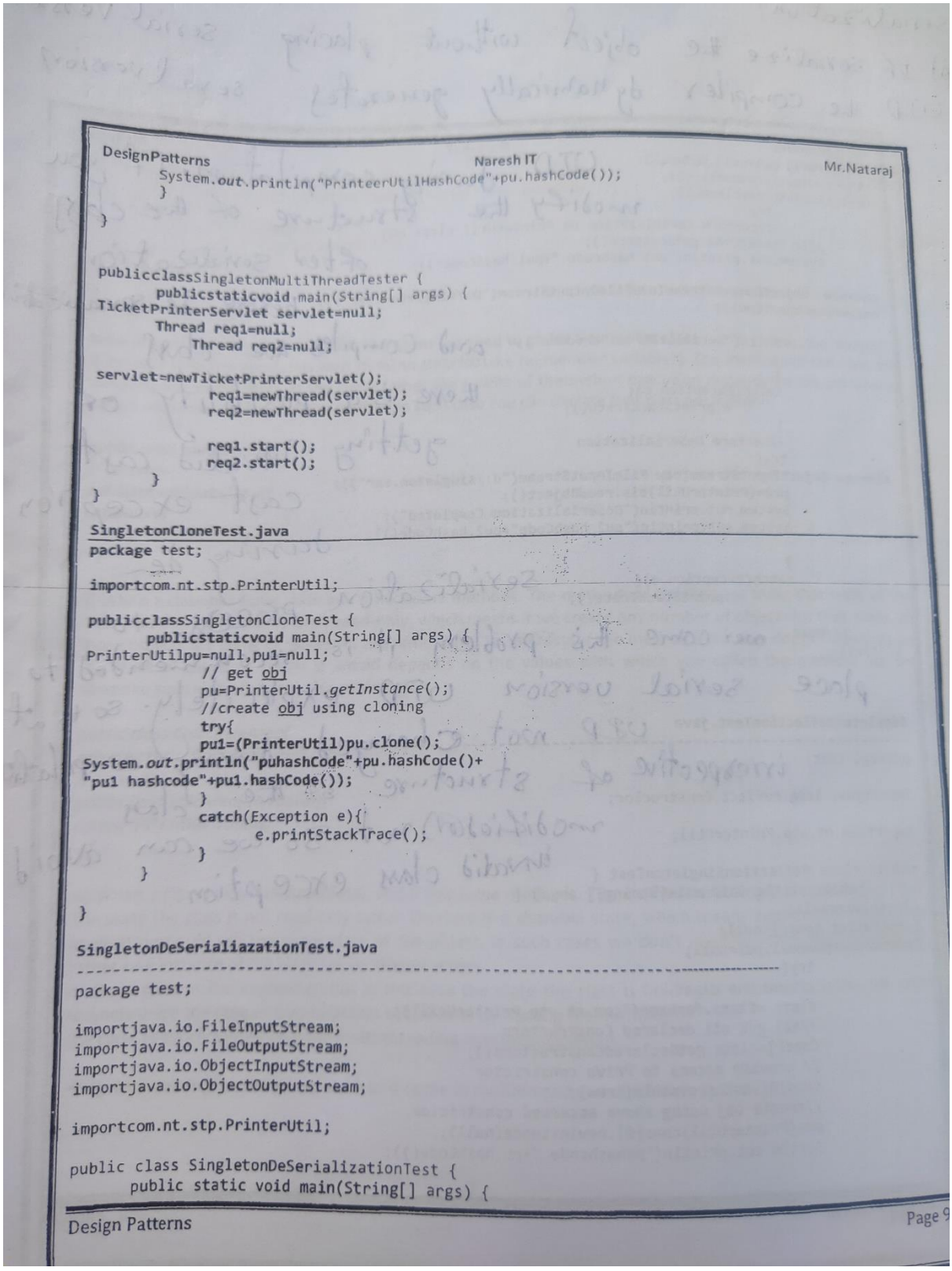

```
            e.printStackTrace();
        }
            returninstance;
    }
    @Override
    public Object clone() throwsCloneNotSupportedException {
        thrownewCloneNotSupportedException();
    }
    public Object readResolve(){
        System.out.println("PrinterUtil:readResolve()");
        returninstance;
    }
    /*  public static PrinterUtilgetInstance(){
        return instance;
      }*/
}
```

SingletonTest.java (Basic test)

```
package test;

importcom.nt.stp.PrinterUtil;

publicclassSingletonTest {

    publicstaticvoid main(String args[]) throws Exception{
    PrinterUtil pu1=null,pu2=null;

pu1=PrinterUtil.getInstance();
        pu2=PrinterUtil.getInstance();

        System.out.println(pu1.hashCode()+"  "+pu2.hashCode());
        System.out.println("pu1 and pu2 are refering same obj?"+(pu1==pu2));
    }

}
```

MultiThreadSingletonTest.java

```
package test;

importcom.nt.stp.PrinterUtil;

classTicketPrinterServletimplements Runnable{

    @Override
    publicvoid run() {
        PrinterUtilpu=null;

        pu=PrinterUtil.getInstance();
    System.out.println("CureentThead name"+Thread.currentThread().getName());
```



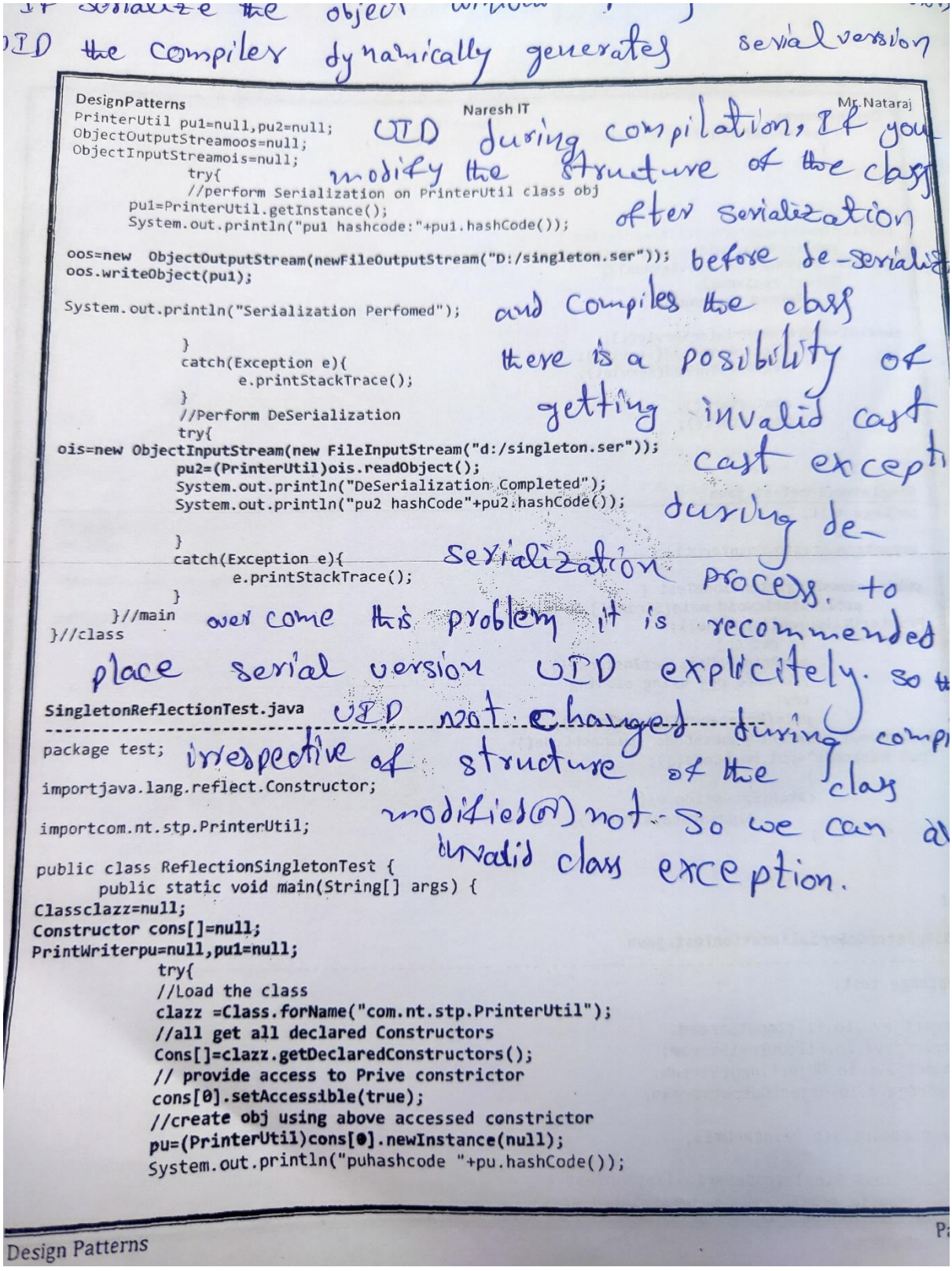

```
        System.out.println("PrinteerUtilHashCode"+pu.hashCode());
        }
}


publicclassSingletonMultiThreadTester {
        publicstaticvoid main(String[] args) {
TicketPrinterServlet servlet=null;
        Thread req1=null;
            Thread req2=null;

servlet=newTicketPrinterServlet();
             req1=newThread(servlet);
             req2=newThread(servlet);

             req1.start();
             req2.start();
        }
}
```

SingletonCloneTest.java

```
package test;

importcom.nt.stp.PrinterUtil;

publicclassSingletonCloneTest {
       publicstaticvoid main(String[] args) {
PrinterUtilpu=null,pu1=null;
             // get obj
             pu=PrinterUtil.getInstance();
             //create obj using cloning
             try{
             pu1=(PrinterUtil)pu.clone();
System.out.println("puhashCode"+pu.hashCode()+
"pu1 hashcode"+pu1.hashCode());
             }
             catch(Exception e){
                   e.printStackTrace();
             }
       }

}
```

SingletonDeSerialiazationTest.java

```
package test;

importjava.io.FileInputStream;
importjava.io.FileOutputStream;
importjava.io.ObjectInputStream;
importjava.io.ObjectOutputStream;

importcom.nt.stp.PrinterUtil;

public class SingletonDeSerializationTest {
      public static void main(String[] args) {
```



DID the compiler dynamically generates serial version UID during compilation, If you modify the structure of the class after serialization and compiles the class there is a posibility of getting invalid cast exception during de-serialization process. to over come this problem it is recommended place serial version UID explicitely. so the UID not changed during compilation irrespective of structure of the class modified(or) not. So we can avoid invalid class exception.

```
DesignPatterns                                    Naresh IT                    Mr.Nataraj
PrinterUtil pu1=null,pu2=null;
ObjectOutputStreamoos=null;
ObjectInputStreamois=null;
            try{
            //perform Serialization on PrinterUtil class obj
      pu1=PrinterUtil.getInstance();
      System.out.println("pu1 hashcode:"+pu1.hashCode());

oos=new ObjectOutputStream(newFileOutputStream("D:/singleton.ser"));
oos.writeObject(pu1);

System.out.println("Serialization Perfomed");

            }
            catch(Exception e){
                  e.printStackTrace();
            }
            //Perform DeSerialization
            try{
ois=new ObjectInputStream(new FileInputStream("d:/singleton.ser"));
            pu2=(PrinterUtil)ois.readObject();
            System.out.println("DeSerialization Completed");
            System.out.println("pu2 hashCode"+pu2.hashCode());

            }
            catch(Exception e){
                  e.printStackTrace();
            }
      }//main
}//class
```

SingletonReflectionTest.java

```
package test;

importjava.lang.reflect.Constructor;

importcom.nt.stp.PrinterUtil;

public class ReflectionSingletonTest {
      public static void main(String[] args) {
Classclazz=null;
Constructor cons[]=null;
PrintWriterpu=null,pu1=null;
            try{
            //Load the class
            clazz =Class.forName("com.nt.stp.PrinterUtil");
            //all get all declared Constructors
            Cons[]=clazz.getDeclaredConstructors();
            // provide access to Prive constrictor
            cons[0].setAccessible(true);
            //create obj using above accessed constrictor
            pu=(PrinterUtil)cons[0].newInstance(null);
            System.out.println("puhashcode "+pu.hashCode());
```

```
                pu1=PrinterUtil.getInstance();
                System.out.println("pu1 hashcode "+pu1.hashCode());
                }
                catch(Exception e ){
                      e.printStackTrace();
                }
        }
}
```

Now we try to understand in which situations we need to go for a singleton class.

i) When a class has absolutely zero state(no state/datai.e no member variables). The methods of the class are notusing any of the state of the class; rather the results of themethod execution depends on the parameter values with which you called the method.In such case you can declare that class as singleton.

```
public class MathDemo{
// no state
publicint add(intx,int y){
returnx+y;
    }
 }
```

ii) When a class has some state and it has some methods. The methods of theclass are using that state of the class. But that state iscompletely **read-only,** which means if we create any number of objects for that class, all those objects are going to represent the same state. So the result f the method execution doesn't depends on the state of the class rather it would depends on the values with which you called the method. So, we canmake such kind of classes also as singleton.

```
public class CalcCircleArea{
private static float final PI=3.14f;
// no state
public float calcArea(int radius){
return PI*radius*radius;
 }
```

iii) When a class has some state(data), and it has some methods. The methods areusing the state of the class. The state the class is not read-only rather thestate is a sharable state, which means every other class in my applicationshould see the same state of the object. In such cases we don't need tocreate 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 tosynchronize the read and write access to the class by making the methods of
the class as synchronized to avoid multi-threading concurrency issues.

→eg: Maintaining countries and state info cache in multithread env.. (like web applicaiton)

Factory Pattern

Sometimes, an Application (or framework) at runtime, cannot anticipate the class of object that it must create. The Application (or framework) may know that it has to instantiate classes, but it may only know about abstract classes (or interfaces), which it cannot instantiate. Thus the Application class may only know *when* it has to instantiate a new Object of a class, not *what kind of* subclass to create. Hence a class may want it's subclasses to specify the objects to be created.

Every time we can not create objects in the java using new operator/keyword . Few objects may be created using new, few may have to be created by calling a static factory method on the class (singleton) and others may have to be created by passing other object as reference while creating object and etc.. So it is better to have abstraction on this object creation process.

The main advantage of going for factory pattern is it abstracts the object creational process of the classes.

To have our own bike, we never try to manufacture our own Bike, becoz it takes lot of time and also very complex . Instead we can go to a Bike factory that is proficient in manufacturing bikes to get a bike.

The **factory pattern design pattern** handles the problems of object creation by defining a separate method for creating the objects, this methods optionally accept parameters defining for which class the object should be created, and returns the created object.

Problem: Creating object by knowing its creational process and dependencies is very complex and Creating multiple objects for multiple classes and utilizing one of them based on the user supplied data is wrong methodology, because the remaining objects become unnecessarily created objects.

Solution: Use Factory Pattern, here the method of class factory instantiates one of the several sub classes or other classes based on the data that is supplied by user (at runtime), that means objects for remaining sub classes/other classes will not be created.

A Simple Factory pattern returns an instance of one of several possible (sub) classes depending on the data provided to it. Usually all classes that it returns have a common parent class and common methods, but each performs a task differently and is optimized for different kinds of data.

Intent:

- ✓ Creates objects without exposing the instantiation logic to the client(Provides abstraction on object creation process).
- ✓ Refers to the newly created object through a common interface.

Application Areas:

- when class can't anticipate the class of objects it must create.
- when class want abstract the object creation process
- In programmer's language, you can use factory pattern where you have to create an object of any one of sub-classes or possible classes depending on the data provided.

eg:

In JDBC applications, **DriverManager.getConnection(-,-,-)** method logic is nothing but factory pattern logic because based on the argument values that are supplied, it creates and returns one of the JDBC driver class that implements the java.sql.Connection interface.

I. ***Connection con=Drivermanager.getConnection("jdbc:odbc:oradsn","<uname>","<pwd>");***

- *Here getConnection(-,-,-) call returns the object of Type-1 JDBC driver class i.e.* ***sun.jdbc.odbc.JdbcOdbcConnection*** *that implements java.sql.Connection interface.*

II. ***Connection con= Drivermanager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","<uname>","<pwd>");***

- *Here getConnection(_,_,_) call returns the object of Type-4 JDBC driver software (oracle thin driver) supplied class i.e.* ***oracle.jdbc.driver.OracleConnection*** *that implements java.sql.Connection interface.*

Car.java

```
package com.nt.fp;
public class Car {
private String model;
private String enggCC;

public void assembie(){
        System.out.println("Normal car Assembled");
}
public void roadTest(){
        System.out.println("Normal car tested");
}
public void deliver(){
        System.out.println("Normal Car deliverd");
 }
}
```

LuxoryCar.java

```
package com.nt.fp;
public class LuxoryCar extends Car {
        private String acType;
        @Override
        public void assemble() {
                System.out.println("Lurxory car Assembled");
        }
        @Override
        public void roadTest() {
         System.out.println("Luxory car roadTested");
        }
        @Override
        public void deliver() {
                System.out.println("Luxory car delivered");
        }
}
```

SportsCar.java

```
package com.nt.fp;
public class SportsCar extends Car {
private String power;
@Override
        public void assemble() {
         System.out.println("Sports car is Assembled");
        }
@Override
        public void roadTest() {
        System.out.println("Sports car is roadTested");
        }

@Override
        public void deliver() {
        System.out.println("Sports car is delivered");
        }
}
```

CarDealer.java

```
package test.problem;
import com.nt.fp.Car;
import com.nt.fp.LuxoryCar;
import com.nt.fp.SportsCar;
Problem:
```

```
public class CarDelear {
        public static void main(String[] args) {
                Car car=new Car();
                car.assemble();
                car.deliver();
                car.roadTest();
            Car car1=new LuxoryCar();
                car1.assemble();
                car1.deliver();
                car1.roadTest();
        }
}
```

- Here Dealer has to know object creation (manifacturing) and other processes to get and user the car. So take the Support of CarFactory as shown below to get Abstraction on Car object Creation and other Processes.

CarFactory.java

```
package test.solution;
import com.nt.fp.Car;
import com.nt.fp.LuxoryCar;
import com.nt.fp.SportsCar;
public class CarFactory {
        public static Car getCar(String type){
                Car car=null;
                if(type.equals("standard")){
                        car=new Car();
                }
                if(type.equals("luxory")){
                        car=new LuxoryCar();
                }
                if(type.equals("sports")){
                        car=new SportsCar();
                }
                car.assemble();
                car.roadTest();
                car.deliver();
                return car;
        }
}
```

CarDealer.java(improvised)

```
package test.solution;
import com.nt.fp.Car;
import com.nt.fp.LuxoryCar;
import com.nt.fp.SportsCar;
public class CarDelear {
        public static void main(String[] args) {
```

4

```
        Car car=CarFactory.getCar("standard");
        Car car1=CarFactory.getCar("luxory");
    }
}
```

Factory method Design pattern

Factory method is different from Factory mehod design pattern,Factory method just creates and returns the object whereas Factory method design pattern defines set of rules and guidelines for factories while objects for the related classes of same family.

Factory method design pattern is used for creating the objects for related classes with in the hierarchy belonging to same family .It defines set of rules and guidelines to create objects for same family classes.

For example Bajaj is manufacturing bikes. It manufactures several types of bikes and it has several Manufacturing units in which the bikes are being manufactured like ChennaiFactory, PuneFactory,NagpurFactory and etc.. and all these factories create same bajaj family bikies like pulsor,discover and etc.. if we do not provide rules and guidelines to these factories while creating same bajaj Family bikies then different factories will follow different methodologies while creating bikies which may lead quality issues like NagpurFactroy gives more quality bikes becoz it is following more standards and PuneFactory gives less quality bikes becoz it is following bit less standards. To overcome this problem we define set of rules and guidelines for all these factories to make factories creating bajaj bikies with same quality and standards

Problem:

Bike.java

```
package com.nt.fmp;

public abstract class Bike {
        private int id;
        private String enggCC;

        public abstract void drive();
}
```

Pulsor.java

```
package com.nt.fmp;

public class Pulsor extends Bike{
        private String pikupLevel;

        @Override
        public void drive() {
                System.out.println("Drivering Pulsor bike");
        }

}
```

Discover.java

```
package com.nt.fmp;

public class Discover extends Bike {
        private String mileage;

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

NagpurFactory.java

```
package com.nt.fmp;

public class NagpurFactory {

        public static void paint(){
                System.out.println("Painting Bike....");
        }
        public static void assemble(){
                System.out.println("Assembling Bike...");
        }
        pubiic static void test(){
                System.out.println("Testing Bike...");
        }

        public static Bike createBike(String type){
                Bike bike=null;
                if(type.equals("pulsor")){
                 bike=new Pulsor();
                 System.out.println("Creating Pulsor Bike");
                }
                else if(type.equals("discover")){
                        bike=new Discover();
                        System.out.println("Creating Discover Bike");
                }
                paint();
                assemble();
                return bike;
        }

}
```

ChennaiFactory.java

```
package com.nt.fmp;
```

```
public class ChennaiFactory {

        public static void paint(){
                System.out.println("Painting Bike....");
        }
        public static void assemble(){
                System.out.println("Assembling Bike...");
        }
        public static void test(){
                System.out.println("Testing Bike...");
        }

        public static Bike createBike(String type){
                Bike bike=null;
                if(type.equals("pulsor")){
                 bike=new Pulsor();
                 System.out.println("Creating Pulsor Bike");
                }
                else if(type.equals("discover")){
                        bike=new Discover();
                        System.out.println("Creating Discover Bike");
                }

                assemble();
                paint();
                test();
                return bike;
        }
}
```

NorthConsumer.java

```
package test.problem;
import com.nt.fmp.Bike;
import com.nt.fmp.NagpurFactory;

public class NorthCustomer {
        public static void main(String[] args) {
                Bike bike=null;
                   bike=NagpurFactory.createBike("pulsor");
                   bike.drive();
        }

}
```

SouthConsumer.java

```
package test.problem;

import com.nt.fmp.Bike;
import com.nt.fmp.ChennaiFactory;
```

```
public class SouthCustomer {
        public static void main(String[] args) {
                Bike bike=null;
                 bike=ChennaiFactory.createBike("pulsor");
                 bike.drive();
        }
}
```

Note : In the Above code ChennaiFactory is following more standards while creating Bajaj family bikes and NagpurFactory is not following same standards(no testing) ,So we get more quality bikes from Chennai Factory.To overcome this problem We take one common super class for both Factory classes defining Same set of rules and guidelines that factories must follow while creating Bajaj Family bikes ,due to this We can expect same quality bikes from both Factories .

So to bring up some quality control and standardization of manifacturing a bike ,create a BajajFactory class as super class to both factory classes which takes care of standardizing the process of manifacturing the bike as shown below.

Solution Code

BajajFactory.java

```
package com.nt.fmp;

public abstract class BajajFactory {
        public abstract void paint();
        public abstract void assemble();
        public abstract void test();
        public abstract Bike createBike(String type);

        public  Bike orderBike(String type){
                Bike bike=null;
                bike=createBike(type);
                paint();
                assemble();
                test();
                return bike;
        }
}
```

ChennaiFactory.java

```
package com.nt.fmp;
public class ChennaiFactory extends BajajFactory {

        public  void paint(){
                System.out.println("Painting Bike....");
```

```
    }
    public  void assemble(){
            System.out.println("Assembling Bike...");
    }
    public  void test(){
            System.out.println("Testing Bike...");
    }

    public  Bike createBike(String type){
            Bike bike=null;
            if(type.equals("pulsor")){
             bike=new Pulsor();
             System.out.println("Creating Pulsor Bike");
            }
            else if(type.equals("discover")){
                    bike=new Discover();
                    System.out.println("Creating Discover Bike");
            }

            return bike;
    }
}
```

NagpurFactory.java

```
package com.nt.fmp;
public class NagpurFactory extends BajajFactory {

    public  void paint(){
            System.out.println("Painting Bike....");
    }
    public  void assemble(){
            System.out.println("Assembling Bike...");
    }
    public  void test(){
            System.out.println("Testing Bike...");
    }

    public  Bike createBike(String type){
            Bike bike=null;
            if(type.equals("pulsor")){
             bike=new Pulsor();
             System.out.println("Creating Pulsor Bike");
            }
            else if(type.equals("discover")){
                    bike=new Discover();
                    System.out.println("Creating Discover Bike");
            }

            return bike;
    }
}
```

NorthConsumer.java

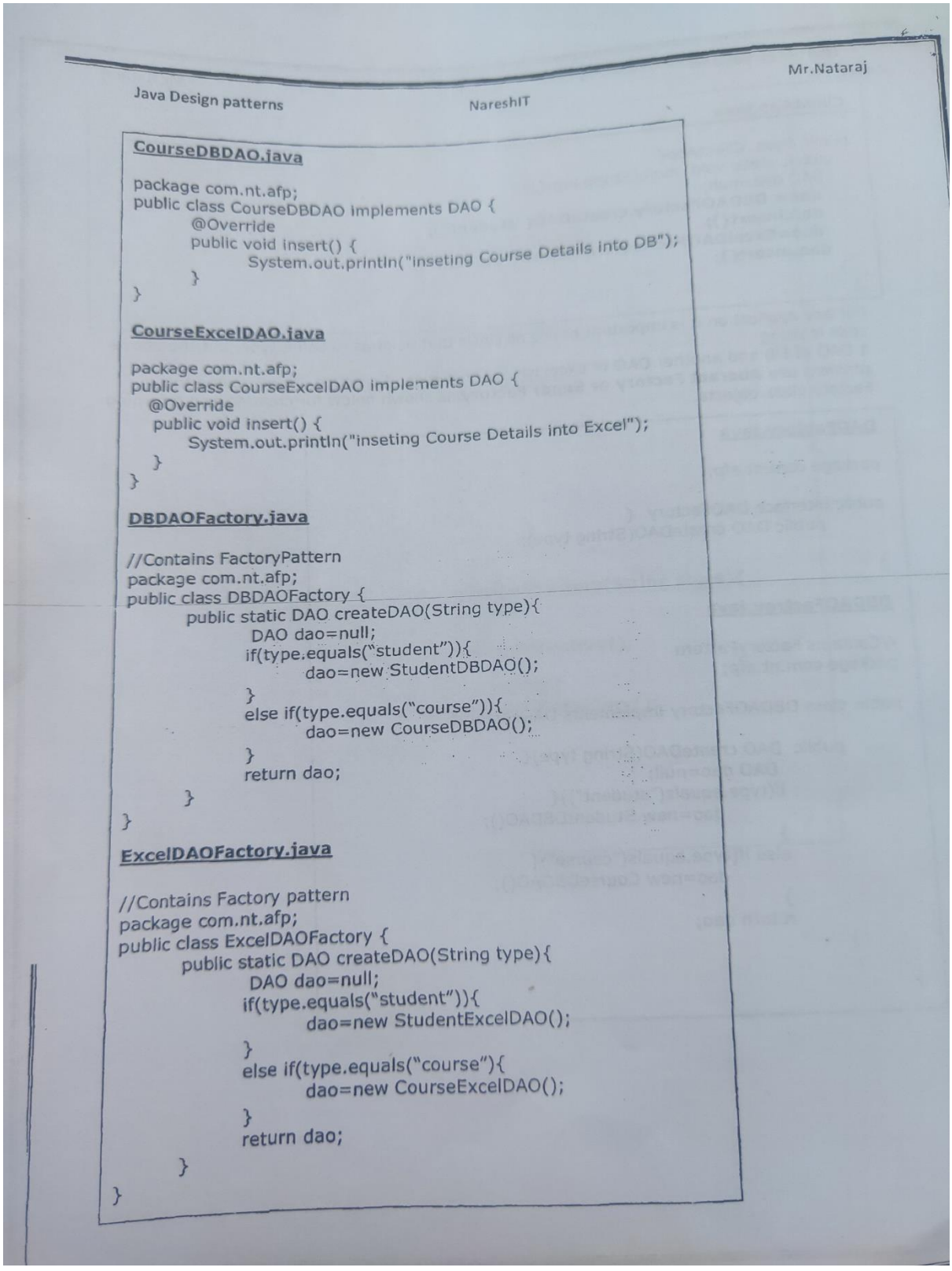
```
package test.solution;
import com.nt.fmp.BajajFactory;
import com.nt.fmp.Bike;
import com.nt.fmp.NagpurFactory;

public class NorthCustomer {
        public static void main(String[] args) {
                BajajFactory factory=null;
                 Bike bike=null;

                 factory=new NagpurFactory();
                 bike=factory.orderBike("pulsor");
                 bike.drive();
        }
}
```

SouthConsumer.java

```
package test.solution;
import com.nt.fmp.BajajFactory;
import com.nt.fmp.Bike;
import com.nt.fmp.ChennaiFactory;

public class SouthCustomer {

        public static void main(String[] args) {
                BajajFactory factory=null;
                 Bike bike=null;

                factory=new ChennaiFactory();
                 bike=factory.orderBike("pulsor");
                 bike.drive();
                 }
}
```

The advantage with the above approach is every Fatory will take care of manufacturing the bajaj family bikes, But the complete control of standards and processes to manufacture the car will be done across the multiple Factories in the same manner as it is controlled by the super class that contains factory method logic.

Now our factory method orderBike(-) in the above super class can create objects of only the sub-classes of Bike class only (i.e only bajaj family bikes). Even we add more bike models in future also the manufacturing and delivering of those bikes will not get affected. More over the code deals with super type, so it can work with any user-defined Concrete Car (sub class) types.



CourseDBDAO.java

```
package com.nt.afp;
public class CourseDBDAO implements DAO {
        @Override
        public void insert() {
                System.out.println("inseting Course Details into DB");
        }
}
```

CourseExcelDAO.java

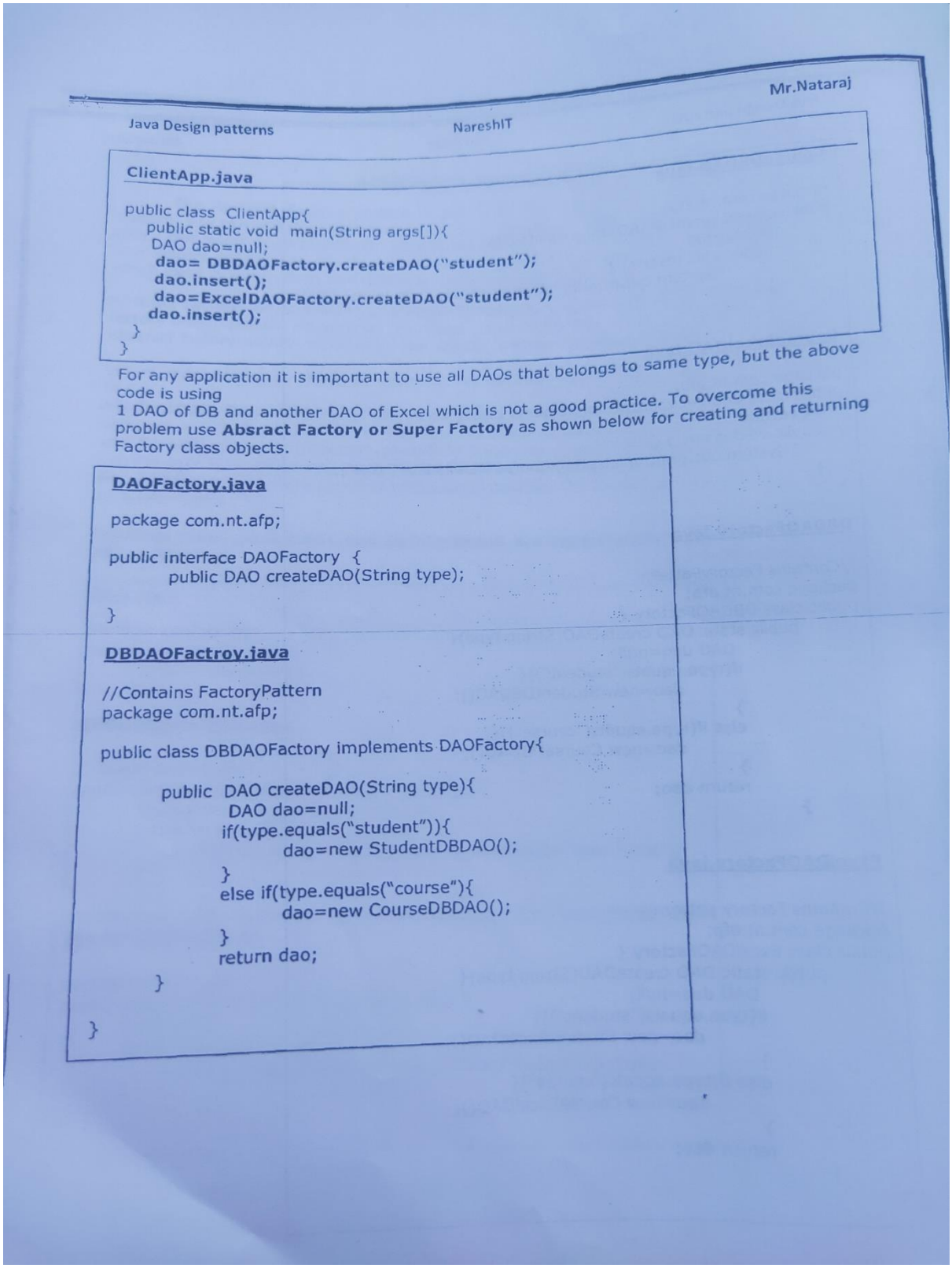
```
package com.nt.afp;
public class CourseExcelDAO implements DAO {
   @Override
    public void insert() {
        System.out.println("inseting Course Details into Excel");
    }
}
```

DBDAOFactory.java

```
//Contains FactoryPattern
package com.nt.afp;
public class DBDAOFactory {
        public static DAO createDAO(String type){
                 DAO dao=null;
                if(type.equals("student")){
                        dao=new StudentDBDAO();
                }
                else if(type.equals("course")){
                        dao=new CourseDBDAO();
                }
                return dao;
        }
}
```

ExcelDAOFactory.java

```
//Contains Factory pattern
package com.nt.afp;
public class ExcelDAOFactory {
        public static DAO createDAO(String type){
                 DAO dao=null;
                if(type.equals("student")){
                        dao=new StudentExcelDAO();
                }
                else if(type.equals("course"){
                        dao=new CourseExcelDAO();
                }
                return dao;
        }
}
```



ClientApp.java

```
public class  ClientApp{
    public static void  main(String args[]){
     DAO dao=null;
     dao= DBDAOFactory.createDAO("student");
     dao.insert();
     dao=ExcelDAOFactory.createDAO("student");
     dao.insert();
   }
}
```

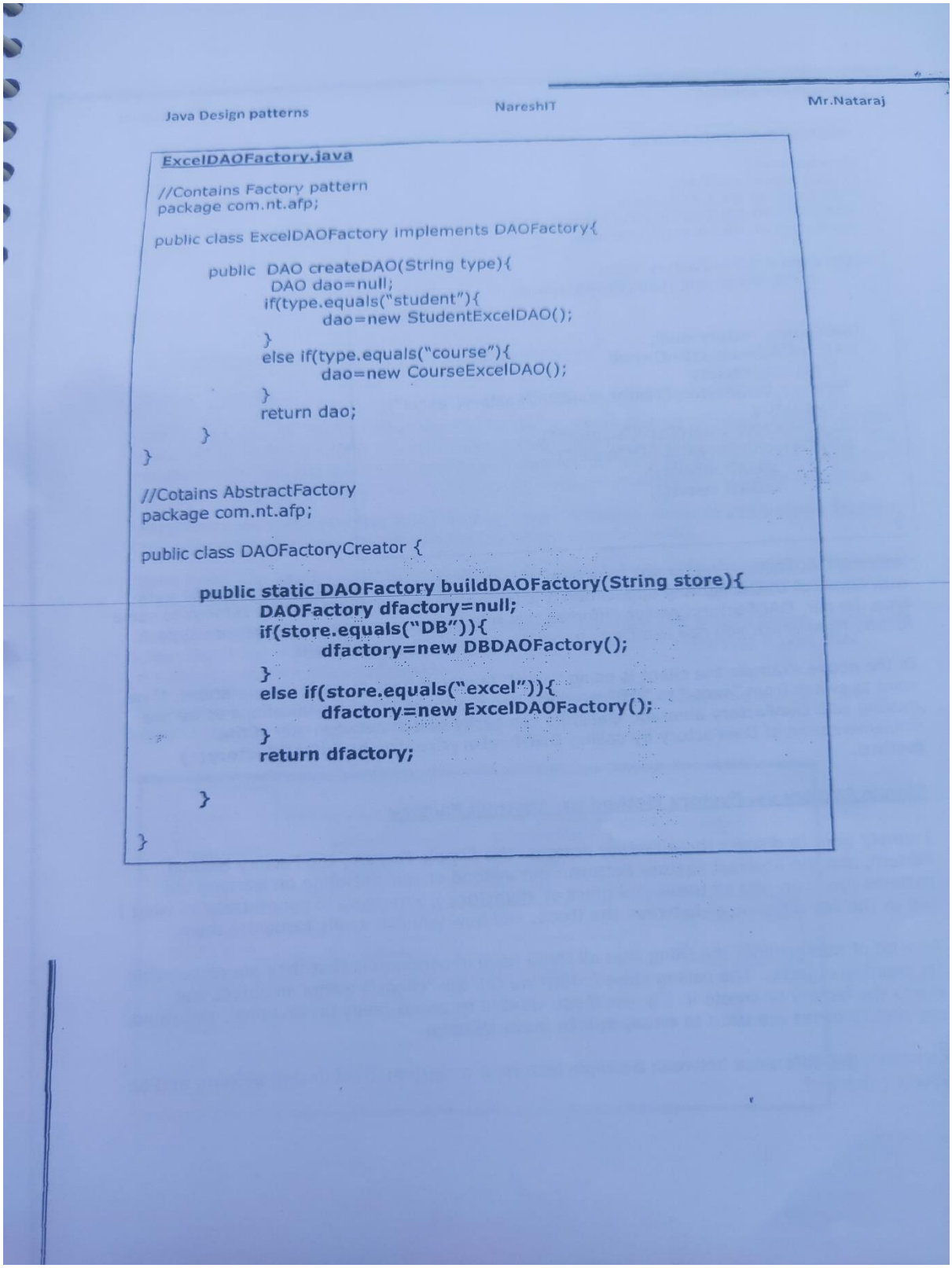
For any application it is important to use all DAOs that belongs to same type, but the above code is using
1 DAO of DB and another DAO of Excel which is not a good practice. To overcome this problem use **Absract Factory or Super Factory** as shown below for creating and returning Factory class objects.

DAOFactory.java

```
package com.nt.afp;

public interface DAOFactory  {
        public DAO createDAO(String type);

}
```

DBDAOFactroy.java

```
//Contains FactoryPattern
package com.nt.afp;

public class DBDAOFactory implements DAOFactory{

        public  DAO createDAO(String type){
                 DAO dao=null;
                if(type.equals("student")){
                        dao=new StudentDBDAO();
                }
                else if(type.equals("course"){
                        dao=new CourseDBDAO();
                }
                return dao;
        }

}
```



ExcelDAOFactory.java

```
//Contains Factory pattern
package com.nt.afp;

public class ExcelDAOFactory implements DAOFactory{

        public  DAO createDAO(String type){
                 DAO dao=null;
                if(type.equals("student"){
                        dao=new StudentExcelDAO();
                }
                else if(type.equals("course"){
                        dao=new CourseExcelDAO();
                }
                return dao;
        }
}

//Cotains AbstractFactory
package com.nt.afp;

public class DAOFactoryCreator {

        public static DAOFactory buildDAOFactory(String store){
                DAOFactory dfactory=null;
                if(store.equals("DB")){
                        dfactory=new DBDAOFactory();
                }
                else if(store.equals("excel")){
                        dfactory=new ExcelDAOFactory();
                }
                return dfactory;

        }

}
```

AbstractFactoryTest.java

```
package test;
import com.nt.afp.DAO;
import com.nt.afp.DAOFactory;
import com.nt.afp.DAOFactoryCreator;
import com.nt.afp.FactoryConstants;

public class AbstractFactoryTest {
        public static void main(String[] args)
{

   DAOFactory factory=null;
    DAO  stDAO=null,crDAO=null;
      // get DAOFactory
     factory= DAOFactoryCreator.buildDAOFactory("excel");
     // get DAOs
    stDAO=factory.createDAO("student");
     crDAO=factory.createDAO("excel");
                stDAO.insert();
                crDAO.insert();
       }
}
```

Now the DAOFactoryCreator will take care of instantiating the appropriate factory to work with family of DAOs. For any application it is important to use all DAOs that belongs to same type. So our DAOFactoryCreator enforces this rule by encouraging you to get one type of factory from which you can use Daos to perform persistence operations.

In the above example the client is using "excel" family of daos to perform operations. If we want to switch from "excel" to "DB" we don't need to make lot of modifications as we are dealing with DaoFactory abstract class, he can easily switch between any of the implementation of DaoFactory by calling **DAOFactoryCreator.buildDAOFactory(-)** method.

Simple Factory vs. Factory Method vs. Abstract Factory

I simply want to discuss three factory designs: the Simple Factory, the Factory Method Pattern, and the Abstract Factory Pattern. But instead of concentrating on learning the patterns (you can find all these and more at dofactory), I'm going to concentrate on what I see as the key differences between the three, and how you can easily recognize them.

As a bit of background, the thing that all three have in common is that they are responsible for creating objects. The calling class (which we call the "client") wants an object, but wants the factory to create it. I guess if you wanted to sound really professional, you could say that factories are used to ***encapsulate instantiation***.

So what's the difference between a simple factory, a factory method design pattern, and an abstract factory?

Factory Method

The official definition of the pattern is something like: ***a class which defers instantiation of an object to subclasses***. An important thing to note right away is that when we're discussing the factory pattern, we're not concentrating on the implementation of the factory in the client, but instead we're examining the manner in which objects are being created.

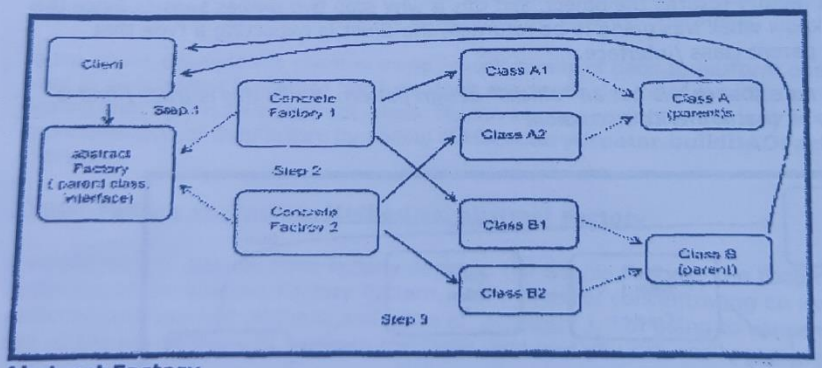
In this example the client doesn't have a direct reference to the classes that are creating the object, but instead has reference to the abstract "Creator". *(Just because the creator is abstract, doesn't mean this is the Abstract Factory!).* It is a **Factory Method** because the children of "Creator" are responsible for implementing the "Create" method. Another key point is that the creator is returning only one object. The object could be one of several types, but the types all inherit from the same parent class.

Step One: the client maintains a reference to the abstract Creator, but instantiates it with one of the subclasses. (i.e. Creator c = new ConcreteCreator1();)

Step Two: the Creator has an abstract method for creation of an object, which we'll call "Create". It's an abstract method which all child classes must implement. This abstract method also stipulates that the type that will be returned is the Parent Class or the Interface of the "product".

Step Three: the concrete creator creates the concrete object. In the case of Step One, this would be "Child Class A".

Step Four: the concrete object is returned to the client. Note that the client doesn't really know what the type of the object is, just that it is a child of the parent.



Abstract Factory

This is biggest pattern of the three. I also find that it is difficult to distinguish this pattern from the Factory Method at a casual glance. For instance, in the Factory Method, didn't we use an abstract Creator? Wouldn't that mean that the Factory Method I showed was a actually an Abstract Factory? The big difference is that by its own definition, an Abstract Factory is used to ***create a family of related products*** (Factory Method creates one product).

Step One: the client maintains a reference to an abstract Factory class, which all Factories must implement. The abstract Factory is instantiated with a concrete factory.

HireFreshers.java

```
public abstract class HireFreshers {
        public boolean conductAptitudeTest() {...} //1
        public boolean conductGroupDiscussion() {...}//2
        public boolean conductHR() {...}//5
        public abstract boolean conductTechnical();//3
        public abstract boolean conductSystemTest();//4
        public final boolean recruitmentProcess() {//Template method
      conductAptitudeTest();
      conductGroupDiscussion();
      conductTechnical();
      conductSystemTest();
      conductHR();
       }
}
```

2. Define the Child class of HireFreshers class,and override only the needed method with proper implementation.

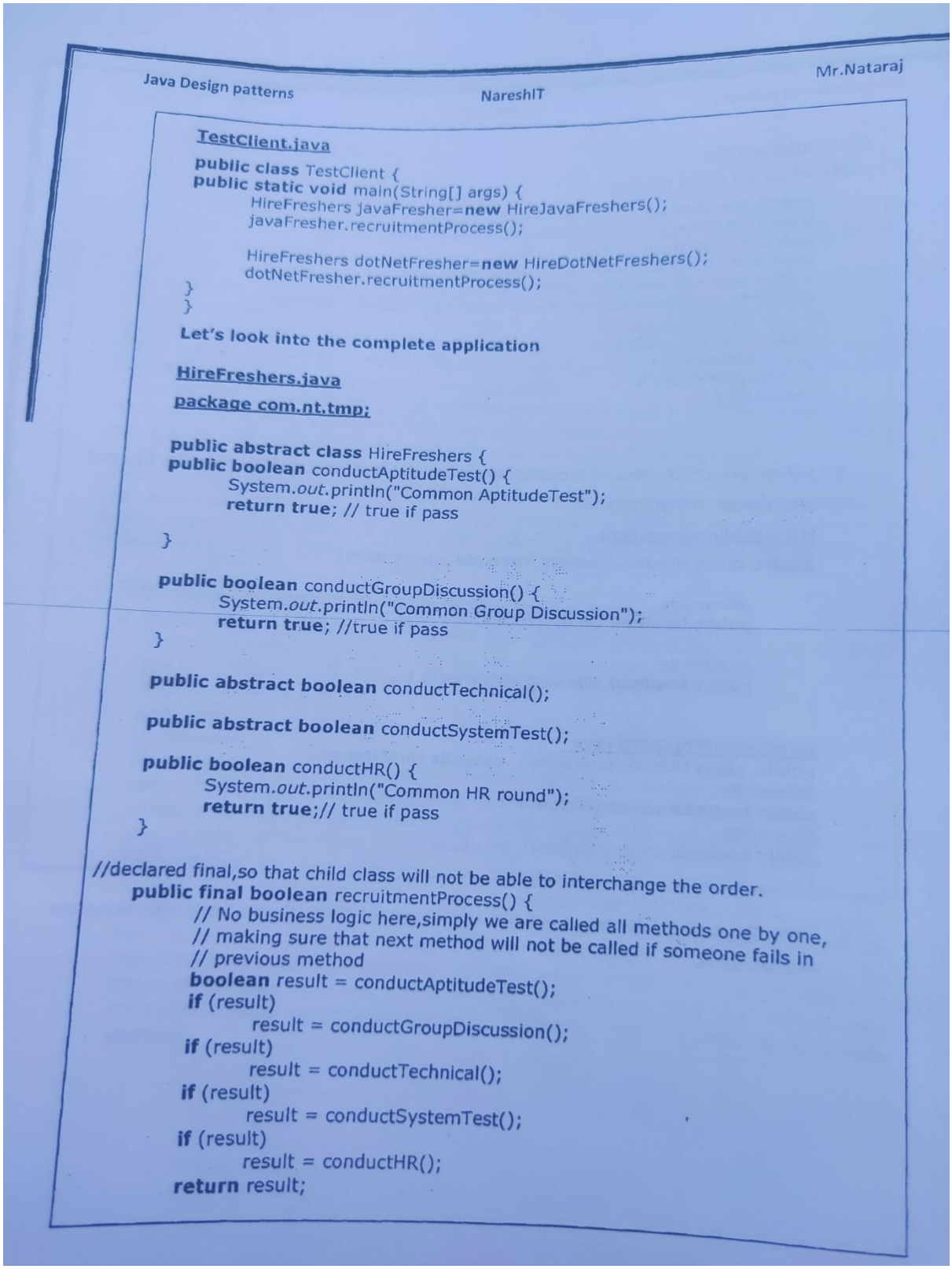
HireJavaFreshers.java

```
public  class HireJavaFreshers  extends HireFreshers{

        @Override
        public boolean conductTechnical() {..}

        @Override
        public boolean conductSystemTest() {... }
 }
```

HireDotNetFreshers.java

```
public  class HireDotNetFreshers  extends HireFreshers{
@Override
public boolean conductTechnical() {...}
@Override
public boolean conductSystemTest() {...}
}
```

3. From the client Application, create the object of Child Classes and call template method on it.



TestClient.java

```
public class TestClient {
public static void main(String[] args) {
        HireFreshers javaFresher=new HireJavaFreshers();
        javaFresher.recruitmentProcess();

        HireFreshers dotNetFresher=new HireDotNetFreshers();
        dotNetFresher.recruitmentProcess();
}
}
```

Let's look into the complete application

HireFreshers.java

```
package com.nt.tmp;

public abstract class HireFreshers {
public boolean conductAptitudeTest() {
        System.out.println("Common AptitudeTest");
        return true; // true if pass

}

public boolean conductGroupDiscussion() {
        System.out.println("Common Group Discussion");
        return true; //true if pass
}

public abstract boolean conductTechnical();

public abstract boolean conductSystemTest();

public boolean conductHR() {
        System.out.println("Common HR round");
        return true;// true if pass
}

//declared final,so that child class will not be able to interchange the order.
public final boolean recruitmentProcess() {
        // No business logic here,simply we are called all methods one by one,
        // making sure that next method will not be called if someone fails in
        // previous method
        boolean result = conductAptitudeTest();
        if (result)
                result = conductGroupDiscussion();
        if (result)
                result = conductTechnical();
        if (result)
                result = conductSystemTest();
        if (result)
                result = conductHR();
        return result;
```

```
    }
}
```

HireJavaFreshers.java

```
package com.nt.tmp;
public  class HireJavaFreshers  extends HireFreshers{

        @Override
        public boolean conductTechnical() {
                System.out.println("conduct Technical");
                return conductJavaTechnical();
        }

        @Override
        public boolean conductSystemTest() {
                System.out.println("conduct System Test");
                return conductJavaSystemTest();
        }

        private boolean conductJavaTechnical()
        {
                System.out.println("conduct Java Technical");
                return true;//false if failed
        }
        private boolean conductJavaSystemTest()
        {
                System.out.println("conduct Java System Test");
                return true;//false if failed
        }

}
```

HireDotNetFreshers.java

```
package com.nt.tmp;
public  class HireDotNetFreshers  extends HireFreshers{

        @Override
        public boolean conductTechnical() {
                System.out.println("conduct Technical");
                return conductDotNetTechnical();
        }

        @Override
        public boolean conductSystemTest() {
                System.out.println("conduct System Test");
                return conductDotNetSystemTest();
        }

        private boolean conductDotNetTechnical()
        {
```

```
        System.out.println("conduct DotNet Technical");
        return true;//false if failed
    }
    private boolean conductDotNetSystemTest()
    {
        System.out.println("conduct DotNet System Test");
        return true;//false if failed
    }

    }
```

TestClient.java

```
Package com.nt.test
public class TestClient {
public static void main(String[] args) {
HireFreshers javaFresher=null,dotNetFresher=null;
 boolean result1=false,result2=false;
    System.out.println("Hiring Java Freshers...");
       javaFresher=new HireJavaFreshers();
       result1=javaFresher.recruitmentProcess();

    if(result1)
     System.out.println("Congrats You are Selected");
    else
    System.out.println("Sorry Plz try again after 6 Months");

      System.out.println("\n Hiring DotNet Freshers...");

    dotNetFresher=new HireDotNetFreshers();

     result2=dotNetFresher.recruitmentProcess();
     if(result2)
      System.out.println("Congrats You are Selected");
     else
   System.out.println("Sorry Plz try again after 6 Months");
     }
}
```

Builder Design Pattern

The builder pattern is a creational design pattern used to assemble complex objects. With the builder pattern, the same object construction process can be used to create different objects. The builder has 4 main parts: a **Builder, Concrete Builders, a Director, and a Product.**

A Builder is an interface (or abstract class) that is implemented (or extended) by Concrete Builders. The Builder interface sets forth the actions (methods) involved in assembling a Product object. It also has a method for retrieving the Product object (ie, getProduct()). The Product object is the object that gets assembled in the builder pattern.

Concrete Builders implement the Builder interface (or extend the Builder abstract class). A Concrete Builder is responsible for creating and assembling a Product object. Different Concrete Builders create and assemble Product objects differently.

A Director object is responsible for constructing a Product. It does this via the Builder interface to a Concrete Builder. It constructs a Product via the various Builder methods.

There are various uses of the builder pattern. For one, if we'd like the construction process to remain the same but we'd like to create a different type of Product, we can create a new Concrete Builder and pass this to the same Director. If we'd like to alter the construction process, we can modify the Director to use a different construction process.

Consider a process of house construction. The process needs several sub-processes like

- **create Basement**
- **create roof**
- **create interior**
- **create structure**

In order to create your dream home, you must execute this sub-process in a particular order. If you mismatch the order, You may build a grave.

Basement->structure->roof->interior

Problem: To build the house, you will have to take different experts. A single person may not be good at all the processes, and we certainly don't need a grave.If we think programmatically, then a House object creation is divided into multiple sub-object

creation, and then putting all together, our beautiful house will be ready.

Solution: Use Builder Design Pattern, which will simplify a complex Object creation Process, which will be easy to maintain and extend. The same creation process will create different implementation class object.

Note: GOF says,

"Separate the construction of a complex object from its representation so that the same construction process can create different representations" i.e. Builder Design

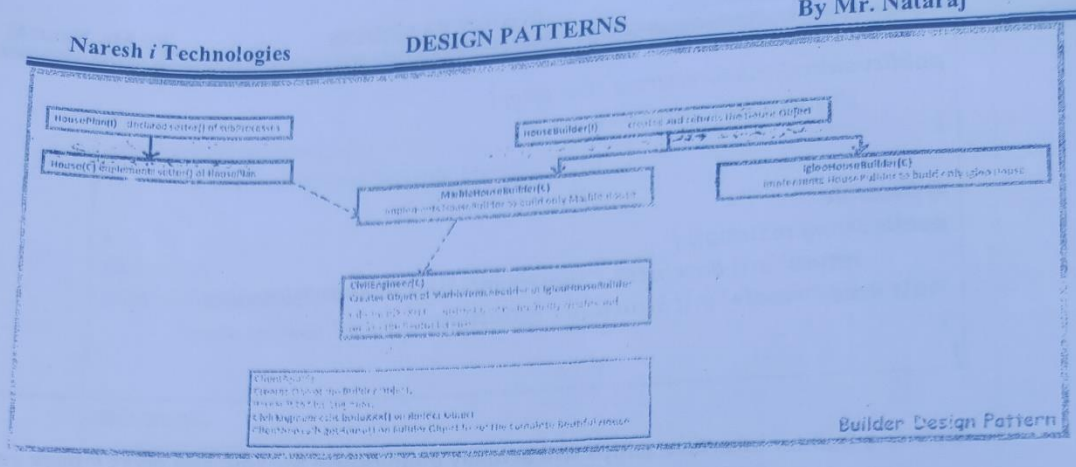
Application Areas: Use the Builder pattern when

- The creation algorithm of a complex object is independent from the parts that actually compose the object.
- The system needs to allow different representations for the objects that are being built.

Note

Each builder is independent of others. This improves modularity and makes the building of other builders easy. Because, each builder builds the final product step by step, we have more control on the final product.

When you want to have your own house, you will first go to the architect for getting the house plan. Once done You will go to the engineer submit your plan, The engineer will ask the suitable builder to take care of your house. Builder may be of various type like MarbleHouseBuilder, IglooHouseBuilder, BambooHouseBuilder.



Process:

1. **Define an Interface containing setter methods for your sub-process.**

HousePlan.java (optional)

```
publicinterfaceHousePlan {
publicvoidsetBasement(String basement);
publicvoidsetStructure(String structure);
publicvoidsetRoof(String roof);
publicvoidsetInterior(String interior);
}
```

2. **Define an Implementation of HousePlan, implementing those abstract methods**

House.java

```
publicclass House implementsHousePlan {
 private String basement;
 private String interior;
 private String roof;
 private String structure;

 publicvoidsetBasement(String basement) {
        this.basement=basement;
 }

 publicvoidsetInterior(String interior) {
        this.interior=interior;
 }

 publicvoidsetRoof(String roof) {
        this.roof=roof;
 }
```

```
publicvoidsetStructure(String structure) {
        this.structure=structure;
}

@Override
public String toString() {
        return"\n\t Basement :"+basement+"\n\t Interior:"+interior+
"\n\t Roof :"+roof+"\n\t Structure :"+structure;
}
}
```

3. Define a Builder Interface which will actually build House for You, and gives u back your dream house. This interface has a method to return your Dream Home.

HouseBuilder.java (Buidler Interface)

```
publicinterfaceHouseBuilder {
 publicvoidbuildBasement();
 publicvoidbuildStructure();
 publicvoidbuildRoof();
 publicvoidbuildInterior();
 public House getHouse();
}
```

4. Define Each Implementation of Builder interface. (ConcreteBuilder)

You may have varieties of Builder like MarbleHouseBuider, IglooHouseBuilder, BambooHouseBuilder etc. For each builder define one Implementation, which contains logic to build the house for You. Builder will create the Object of a House, sets the basement, interior, roof, Structure and will return that house using getHouse()

MarbleHouseBuilder.java

```
publicclassMarbleHouseBuilderimplementsHouseBuilder{
 private House house;
 publicMarbleHouseBuilder()
 {
        this.house=new House();
 }
 @Override
 publicvoidbuildBasement() {
        house.setBasement("Marble Basement");
```

```
}
@Override
publicvoidbuildInterior() {
        house.setInterior("Marble Structure");

}
@Override
publicvoidbuildRoof() {
        house.setRoof("Marble roof");

}
@Override
publicvoidbuildStructure() {
        house.setStructure("Marble Structure");

}
@Override
public House getHouse() {
        // TODO Auto-generated method stub
        returnhouse;
}
}
```

IglooHouseBuilder.java

```
publicclassIglooHouseBuilderimplementsHouseBuilder{
 private House house;
 publicIglooHouseBuilder()
 {
        this.house=new House();
 }
 @Override
 publicvoidbuildBasement() {
        house.setBasement("Ice Basement");

 }
 @Override
 publicvoidbuildInterior() {
        house.setInterior("Ice Structure");

 }
 @Override
 publicvoidbuildRoof() {
```

```
        house.setRoof("Ice roof");
    }
    @Override
    publicvoidbuildStructure() {
        house.setStructure("Ice Structure");
    }
    @Override
    public House getHouse() {
        // TODO Auto-generated method stub
        returnhouse;
    }
}
```

5. **Develop A delegate/Director class, that will use one of the Builder to create Your home.**

CivilEngineer.java

```
publicclassCivilEngineer {
    privateHouseBuilderhouseBuilder;

    publicCivilEngineer(HouseBuilder builder) {
        houseBuilder = builder;
    }

    public House getHouse() {
        returnhouseBuilder.getHouse();
    }

    publicvoidconstructHouse() {
        houseBuilder.buildBasement();
        houseBuilder.buildStructure();
        houseBuilder.buildRoof();
        houseBuilder.buildInterior();
    }
}
```

6. **Develop You Client Application, which will Use Builder and Engineer together to develop You Dream house.**

ClientApp.java

```
publicclassClientApp {
 publicstaticvoid main(String[] args) {
CivilEngineer engineer1=null,engineer2=null;
        House marbleHouse=null,iglooHouse=null;

HouseBuilderiglooBuilder=newIglooHouseBuilder();
        HouseBuildermarbleBuilder=newMarbleHouseBuilder();

        engineer1=newCivilEngineer(marbleBuilder);
        engineer1.constructHouse();
        marbleHouse=engineer1.getHouse();

        System.out.println("Builder constructed \n"+marbleHouse);

        engineer2=newCivilEngineer(iglooBuilder);
        engineer2.constructHouse();
        iglooHouse=engineer2.getHouse();

        System.out.println("Builder constructed \n"+iglooHouse);
 }
}
```

Strategy Pattern

This pattern allows to developreusable,flexible,extensible,easily maintainable Software App as loosely coupled collection of interchangeable parts.

While developing multiple classes having dependency it is recommended to follow this design pattern. This pattern is not a spring pattern, it can be used anywhere, since spring manages dependency between classes, it is recommended to use this pattern while developing spring application.

Strategy pattern is all about implementing 3 principles.

a) Prefer composition over inheritance.

b) Always code to interfaces and never code with implementationclasses

c) Code should be open to extension and must be closed for modification

a)Prefer composition over inheritance:

→If one class creates the object of other class to use its logics then it is called composition .

(Has –A relation)

```
class A{
.....
.....
}
class B
{
A a=new A();
}
```

→If one class extends from another class then it is called inheritance**(IS-A relation)**.

```
class A
{
.....
.....
}
class B extends A
{
.....
.....
}
```

Inheritance is having following limitations in java.

i)It doesn't support multiple inheritance

```
class C extends A,B
{
.....
.....
}
class C {
A a=new A();
B b=new B();
}
```

ii)Code becomes easily breakable / Code becomes delicate.

eg:

```
class Test{
publicint x()
{
.....
.....
return 100;
}
}
class Demo extends Test {
publicint x(){
.....
```

```
.....
return 1000;
 }
}
```

If we change return type or param type of 'x()' in "Test" we can not compile "Demo" class because overriding rules (Let us assume "Test","Demo" class given by 2 different developers)

Solution:

```
class A{
publicint x(){
 .....
return 100;
}
}
class B
{
A a=new A();
publicint y(){
int res=a.x();
}
}
```

- If 'x()' method return type is changed in "A", then there is no need of doing major modifications in 'B'.
- If x() return type is changed to float then we need to write y() in 'B' as shown below

```
publicint y(){
```

```
float res=a.x();
.....
.....
}
```

*The other classes (like sub classes) using B class, need not to have any modifications

b)Always code to interfaces and never code to implementations:

```
class A{
.....
}
class B{
.....
.....
}
class Test{
A a=new new A();
.....
.....
}
```

- Assigning class "B" object instead of of class "A" object in Test class is not possible with minimum modifications.

Solution:

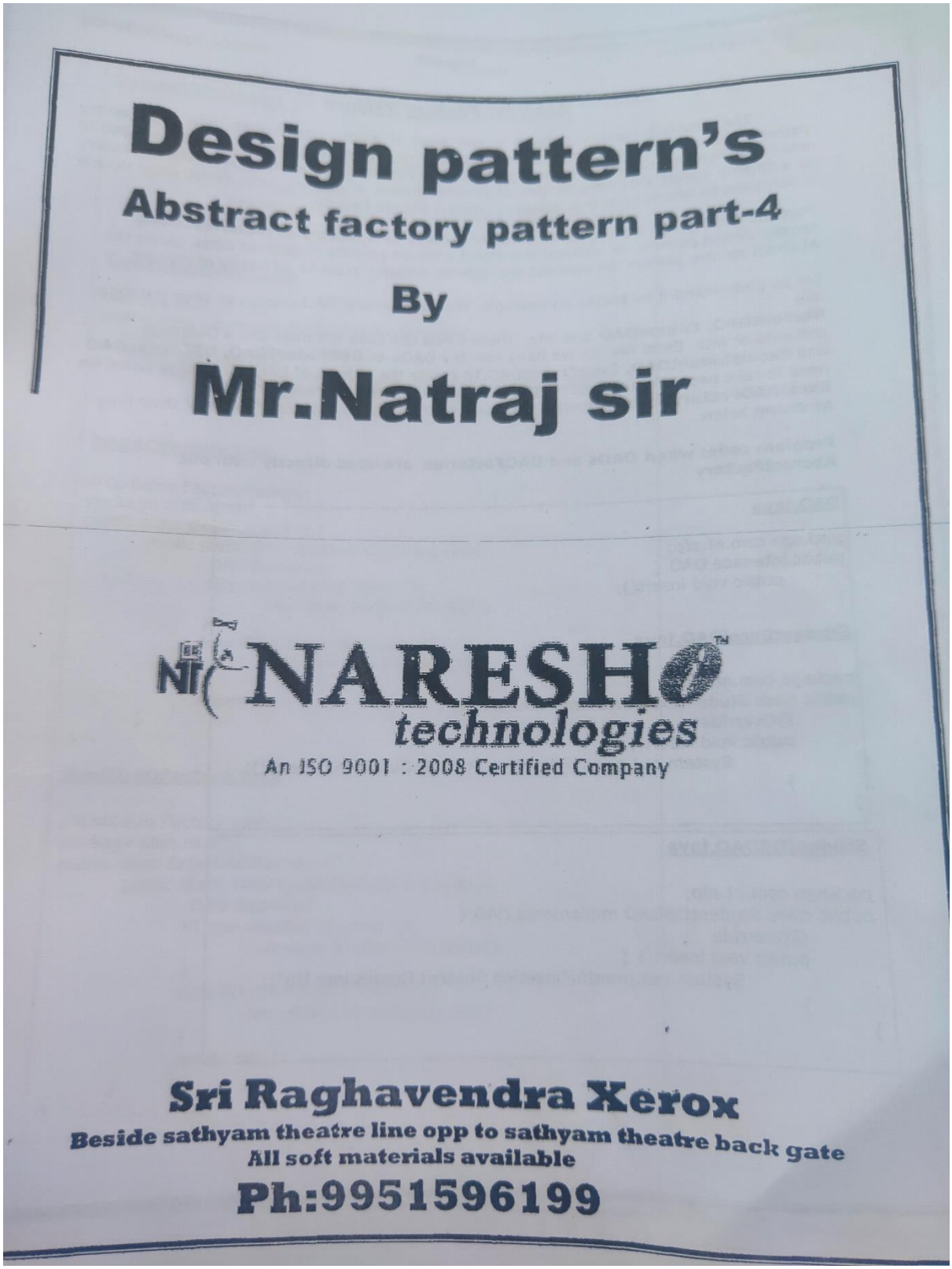
```
InterfaceX{
.....
.....
}
```

```
class A implements X{
.....
.....
}
class B implements X{
.....
.....
}
class Test{
X x=new A();
(Or)
X x=new B();
.....
.....
}
```

- When we code with interfaces we can achieve loosly coupling between target class and dependent class.

More improvised Solution

```
interface X{
.....
}
class A implements X{
.....
}
class B implements X{
```


Design pattern's
Abstract factory pattern part-4
By
Mr.Natraj sir
NARESH
technologies
An ISO 9001 : 2008 Certified Company
Sri Raghavendra Xerox
Beside sathyam theatre line opp to sathyam theatre back gate
All soft materials available
Ph:9951596199

Abstract Factory Pattern

The Abstract Factory pattern is one level of abstraction higher than the Factory pattern. You can use this pattern to return one of several related classes of objects, each of which can return several different objects on request. In other words, the Abstract Factory is a factory object that returns one of several groups of classes. You might even decide which class to return from that group by using a Simple Factory.

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

Let us understand it by taking an example. We have several DAO classes to save the data like
StudentDAO, CourseDAO and etc, these DAOs can save the data into a Database software or into Excel file. So we have now the DAOs as **DBStudentDAO, DBCourseDAO** and **ExcelStudentDAO,** ExcelCourseDAO.To create the objects of DAOs of related types we need to take two different factories(SimpleFactories). **DBDAOFactory** and **ExcelDAOFactory**, these factories take care of creating the objects of DAOs of their type as shown below.

Problem code: when DAOs and DAOFactories are used directly with out AbsractFactory

DAO.java

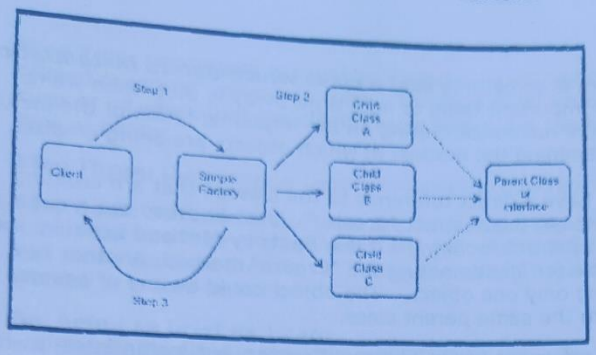
```
package com.nt.afp;
public interface DAO {
        public void insert();
}
```

StudentExcelDAO.java

```
package com.nt.afp;
public class StudentExcelDAO implements DAO {
        @Override
        public void insert() {
                System.out.println("inseting Student Details into Excel");
        }
}
```

StudentDBDAO.java

```
package com.nt.afp;
public class StudentDBDAO implements DAO {
        @Override
        public void insert() {
                System.out.println("inseting Student Details into DB");
        }
}
```



Simple Factory

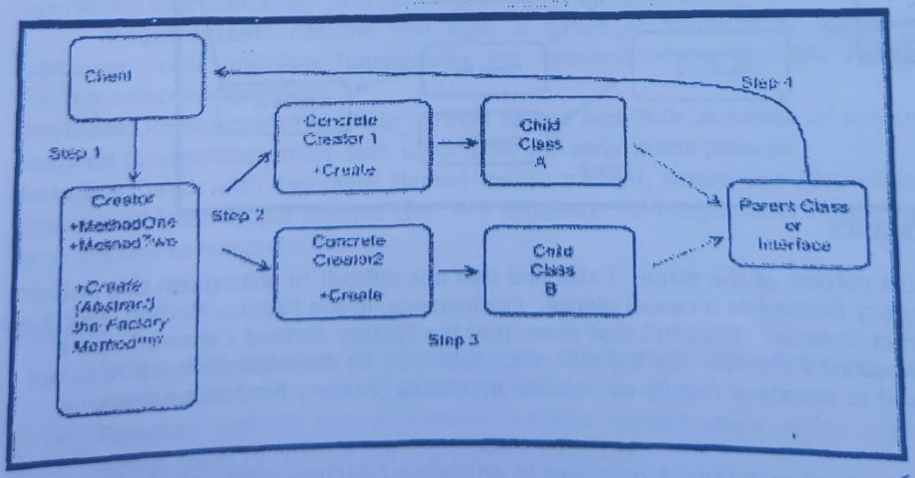
The easiest way for me to remember this is that a simple factory is called directly by the class which wants to create an object (the calling class is referred to as the "client"). The simple factory returns one of many different classes. All the classes that a simple factory can return either inherit from the same parent class, or implement the same interface.

Step One: you call a method in the factory. Here it makes sense to use a static method. The parameters for your call tell the factory which class to create.

Step Two: the factory creates your object. The only thing to note is that of all the objects it can create, the objects have the same parent class, or implement the same interface.

Step Three: factory returns the object, and this is why step two makes sense. Since the client didn't know what was going to be returned, the client is expecting a type that matches the parent class /interface.

Sticklers will note that this is not an "official" design pattern. While this is true, I find it useful and think that it merits discussion!



Step Two: the factory is capable of producing multiple types. This is where the "family of related products" comes into play. The objects which can be created still have a parent class or interface that the client knows about, but the key point is there is more than one type of parent.

Step Three: the concrete factory creates the concrete objects.

Step Four: the concrete objects are returned to the client. Again, the client doesn't really know what the type of the objects are, just that are a children of the parents.
See those concrete factories? Notice something vaguely familiar? There using the Factory Method to create objects.

So, being as brief as I can:
- **A Simple factory** is normally called by the client via a static method, and returns one of several objects that all inherit/implement the same parent.

- **The Factory Method design** is really all about a "create" method that is implemented by sub classes.It

provides set fo rules and guidelines to factories that are creating objects for same family classes.
- **Abstract Factory design** is about returning a family of related objects to the client suing same factory. It normally uses the Super Factory to create one Factory class object and this Factory class return other
Related classes objects..

TemplateMethod DesignPattern

If we take a look at the dictionary definition of a template we can see that a template is a preset format, used as a starting point for a particular application so that the format does not have to be recreated each time it is used. On the same idea ,the template method is based. A **template method** defines an algorithm in a base class using abstract operations that subclasses override to provide concrete behavior.

Requirement: Assume you have a series of functionality to be invoked in a particular order and few functionality are common where are few functionalities are Implementation dependent.
Problem: Remembering multiple method names and their sequence of invocation that are required to complete certain task is complex and error prone process.
Solution: Work with Template Method design pattern, where one java method definition contains all the multiple method calls in a sequence. So programmer needs to call only one java method to complete the task.

Intent:

- ✓ Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- ✓ Template Method lets subclasses redefine certain steps of an algorithm without letting them to change the algorithm's structure.

Application Areas: The Template Method pattern should be used:

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- When refactoring is performed and common behavior is identified among classes. A abstract base class containing all the common code (in the template method) should be created to avoid code duplication.

An application framework allows you to inherit from a class or set of classes and create a new Application, reusing most of the code in the existing classes and overriding one or more methods in order to customize the application to your needs. A fundamental concept in the application framework is the Template Method which is typically hidden beneath the covers and drives the application by calling the various methods in the base class (some of which you have overridden in order to create the application).

For **Ex,** the **process()** method of predefined RequestProcessor class (in Struts 1.X) is Template Method because it internally calls 16+ processXxx() methods in a sequence to complete the task (processing the request i.e. trapped by the ActionServlet).

An important characteristic of the Template Method is that it is defined in the base class and cannot be changed. It's sometimes a **private** method but it's virtually always **final**. It calls other base-class methods (the ones you override) in order to do its job, but it is usually called only as part of an initialization process (and thus the client programmer isn't necessarily able to call it directly).

Steps for implementing Template Method Design Pattern

1. Define Your Abstract class containing your abstract method and concrete method. The concrete method are the methods with common logic whereas abstract methods are the methods whose logics depends on the implementation class and it will be defined in the child classes.

   Lets take the example of a IT company who usually hires freshers in there organisation.

   The hiring could be in 2 format.ie a fresher that will be deployed in any technology and a fresher who is already trained in some technology like Java, dotNet. If the company is small,they will prefer to get readily trained Java and dotNet freshers. For both of them aptitude test, Group Discussion, HR rounds are common. However they will have their own Technical round and System test round. But important point here is there is a proper sequence of all this rounds, and the sequence is Aptitude, GroupDiscussion, Technical, System Test and HR. From the client application,we will have to call this methods in sequence, and if you misplace the sequence, it will be impacted. So instead Lets defined a method, which contains only method call in a sequence, so that the client application now needs to call only that method, and since that method contains the format or pattern of call, it is called as **template method.**

```
.....
}
class Test{
X x;
public void setX(X x){
this.x=x;
}
}
```

```
Test t=new Test();
t.setX (new A());
t.setX (new B());
```

Note:

Without modifying the code of "Test" class we can assign either the object for class A or class B. This gives loosely coupling.

b) Code must be open for extension and must be closed for modifications.

If we follow second principle perfectly by taking the methods of **implementation classes A,B as final methods** our code becomes closed for modification. Similarly it allows to add more implementation classes for interface X having new logics, This makes our code open for extension.

➔FlipKart,DTDC/BlueDart classes are dependent classes

Example

Courier.java

```
package com.nt.comps;
public interface Courier {
    public void deliver(intorderId);
```

```
}
```

DTDC.java

```
package com.nt.comps;
public class DTDC implements Courier {
    @Override
    public void deliver(intorderId) {
    System.out.println("DTDC: delivering orderId:"+orderId+" order items");
    }
}
```

BlueDart.java

```
package com.nt.comps;
public class BlueDart implements Courier {
    @Override
    public final void deliver(intorderId) {
    System.out.println("BlueDart: delivering orderId:"+orderId+" order items");
    }
}
```

FlipKart.java

```
package com.nt.comps;
import java.util.Random;
public class FlipKart {
    private Courier courier;
    public void setCourier(Courier courier) {
        this.courier = courier;
    }
    public String shopping(String []items){
        intorderId=0;
        // generate OrderId
        orderId=new Random().nextInt(1000);
        courier.deliver(orderId);
        return "OrderId"+orderId+" billAmt::"+items.length*1000;
    }
}
```

FlipKartFactory.java

```
package com.nt.factory;
import com.nt.comps.BlueDart;
import com.nt.comps.Courier;
import com.nt.comps.DTDC;
import com.nt.comps.FlipKart;

public class FlipKartFactory {

    public static FlipKartcreateFlipKartWithCourier(String courierName){
        FlipKartflipKart=null;
      Courier courier=null;
        if(courierName.equals("dtdc")){
                flipKart=new FlipKart();
                courier=new DTDC();
                flipKart.setCourier(courier);
        }
        else if(courierName.equals("blueDart")){
                flipKart=new FlipKart();
                courier=new BlueDart();
                flipKart.setCourier(courier);
        }
        else{
                throw new IllegalArgumentException("UnAvailable courier");
        }
        return flipKart;
    }
}//class
```

ClientApp.java

```
package com.nt.test;
import com.nt.comps.FlipKart;
import com.nt.factory.FlipKartFactory;
public class ClientApp {
  public static void main(String[] args) {
 FlipKart kart=null,kart1=null;
 // Get FlipKartobj using Factory
        kart=FlipKartFactory.createFlipKartWithCourier("dtdc");
        System.out.println(kart.shopping(new String[]{"shirt","trouser"}));
```

```
        System.out.println("....................");
        kart1=FlipKartFactory.createFlipKartWithCourier("blueDart");
        System.out.println(kart1.shopping(new String[]{"CRJ","TIJ"}));
    }//main
}//class
```

Decorator/Wrapper Design Pattern

Decorator is one of the popularly used structural patterns. It adds dynamic functionalities/responsibilities 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 design pattern.

Designing classes through inheritance to get additional functionalities is a bad practice because it Just increases number of classes to get different verities' of functionalities.

Problem Code (Using Inheritance):

==========

IceCream.java

```
package com.nt.dcp;

public interface IceCream {
 public void prepare();

}
```

VanilaIceCream.java

```
package com.nt.dcp;

public class VanillaIceCream implements IceCream {

 @Override
 public void prepare() {
        System.out.println("preparing Vanila Ice cream");
 }

}
```

ButterScotch.java

```
package com.nt.dcp;
```

```
public class ButterScotchIceCream implements IceCream {

 @Override
 public void prepare() {
        System.out.println("Preparing ButterScotchIcecream");

 }
}
```

DryFruitButterScotchIceCream.java

```
package com.nt.dcp;

public class DryFruitButterScotchIceCream extends ButterScotchIceCream {

 @Override
 public void prepare() {
        super.prepare();
        addDryFuits();
 }

 private void addDryFuits(){
        System.out.println("adding DryFruits Vanilla Icream");
 }
}
```

DryFruitVanillaIceCream.java

```
package com.nt.dcp;

public class DryFruitVanillaIceCream extends VanillaIceCream {

 @Override
 public void prepare() {
        super.prepare();
        addDryFuits();
 }

 private void addDryFuits(){
        System.out.println("adding DryFruits Vanilla Icream");
 }
}
```

HoneyVanillaIceCream.java

```
package com.nt.dcp;

public class HoneyVanillaIceCream extends VanillaIceCream {

 @Override
 public void prepare() {
        super.prepare();
        addDryFuits();
 }

 private void addDryFuits(){
        System.out.println("adding DryFruits Vanilla Icream");
 }
}
```

→Like this we need to develop many classes as the sub classes to get more functionalities on the top of existing functionalities which is not a good practice, To overcome this problem use
Decorator/Wrapper Design pattern

For example, IceCream is the object which is already prepared and consider as a root/base object. As requested by the customer/client we might need to add some additional toppings on it like DryFruits or Honey.Important point is only for the IceCream that that the customer requested without effecting otherIceCreams these additional functionalities should be added. You can consider these toppings/additions as additional responsibilities/functionalities added by the Decorator/Wrapper.

→There are important participants of the Decorator design pattern:
a) **Component:-** IceCream is the base interface.

b) **Concrete component**: - Normal IceCream is the concrete implementation of the IceCream interface.
c) **Decorator:-** It is the abstract class who holds the reference of the Concrete componentand also implements the component interface

d) **Concrete Decorator: -** Who extendsfrom the abstract decorator and adds additional responsibilities/functionalaties to the Concrete component.

Example

a) Develope Component Interface

IceCream.java

```
package com.nt.dcp;
```

```
public interface IceCream {
 public void prepare();

}
```

b) Develop Concrete Component classes implementing Component classes

VanillaIceCream.java

```
package com.nt.dcp;

public class VanillaIceCream implements IceCream {

 @Override
 public void prepare() {
        System.out.println("preparing Vanila Ice cream");
 }

}
```

ButterScotchIceCream.java

```
package com.nt.dcp;

public class ButterScotchIceCream implements IceCream {

 @Override
 public void prepare() {
        System.out.println("Preparing ButterScotchIcecream");

 }

}
```

c) Develop Abstract Decorator

→Now we want to add some toppings on the regular Vanilla/ButterscotchIceCreams but We don't want tomodify all the objects of IceCream and we do not want to use inheritance rather we want to decorate instances of the Vanilla/Butterscotch,For this create Abstract Decorator implementing from IceCream(I) and maintain reference of IceCreamto add toppings.

IceCreamDecorator.java

```
package com.nt.dcp;

public abstract class IceCreamDecoratorimplements IceCream{
private IceCreamiceCream;
```

```
 public IceCreamDecorator(IceCreamiceCream){
this.iceCream=iceCream;
 }

 public void prepare(){
iceCream.prepare();
 }
}
```

d) Develop Concrete Decorator class

→create an concrete decorator classes extending from Abstract Decorator to add DryFruit,Honey Toppings as shown below.

HoneyIceCreamDecorator.java

```
package com.nt.dcp;
public class HoneyIceCreamDecorator extends IceCreamDecorator{

  public HoneyIceCreamDecorator(IceCreamiceCream){
    super(iceCream);
  }

 public void prepare(){
super.prepare();
addHoney();
 }
 private void addHoney(){
System.out.println("adding honey...");
 }
}
```

DryFruitIceCreamDecorator.java

```
package com.nt.dcp;
public class DryFruitIceCreamDecorator extends IceCreamDecorator{

  public DryFruitIceCreamDecorator(IceCreamiceCream){
    super(iceCream);
  }

 public void prepare(){
super.prepare();
addDryFruits();
 }
```

```
 private void addDryFruits(){
System.out.println("adding dryfruits");
  }
}
```

e) Test the Application....

ClientApp.java

```
package test.problem;

import com.nt.dcp.ButterScotchIceCream;
import com.nt.dcp.DryFruitIceCreamDecorator;
import com.nt.dcp.HoneyIceCreamDecorator;
import com.nt.dcp.IceCream;
import com.nt.dcp.VanillaIceCream;


public static void main(String[] args) {
IceCreamvic=new VanillaIceCream();
IceCreamdfvic=new DryFruitIceCreamDecorator(vic);
dfvic.prepare();
System.out.println("----------");

IceCream vic1=new VanillaIceCream();
   vic1.prepare();

System.out.println("----------");

IceCreamhvic=new HoneyIceCreamDecorator(new ButterScotchIceCream());
hvic.prepare();
System.out.println("----------");

IceCream hvic1=new HoneyIceCreamDecorator(new ButterScotchIceCream());
IceCream dfhvic2=new DryFruitIceCreamDecorator(hvic1);
   dfhvic2.prepare();
System.out.println("----------");
}


}
```

JDK Example : All I/O stream classes

→DataInputStream dis=new DataInputStream(new FileInputStream("abc.txt"));

→BufferedReaderbr=new BufferedReader(new InputStreamReader("abc.txt"));

FlyWeight Pattern

According to GoF, **flyweight design pattern** intent is:

"Use sharing to support large numbers of fine-grained objects efficiently"

Flyweight design pattern is a **Structural design pattern** like Facade pattern, Adapter PatternandDecorator pattern.Flyweight design pattern is usedwhen we need to create a lot of Objects of a class. Since every object consumes memory space that can be crucial for low memory devices, such as mobile devices or embedded systems, flyweight design pattern can be applied to reduce the load on memory by sharing objects.

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 application, the entire mobile memory will be filled up with these objects and makes your mobile run quickly out of memory.

Before we apply flyweight design pattern, we need to consider following factors:

- The number of Objects to be created by application should be huge.
- The object creation is heavy on memory and it can be time consuming too.
- The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of an Object should be defined by the client program.

To apply flyweight pattern, we need to divide Object property into **intrinsic(sharable)** and **extrinsic(non-sharable)** properties. Intrinsic properties make the Object unique whereas extrinsic properties are set by client code method call arguments and used to perform different operations. For example, an Object Circle can have extrinsic properties such as color and width.

For applying flyweight pattern, we need to create a **Flyweight factory** that returns the shared objects. For our example, let's say we need to create a drawing with lines and Ovals. So we will have an interface Shape and its concrete implementations as *Line* and *Oval*. Oval class will have intrinsic property to determine whether to fill the Oval with

given color or not whereas Line will not have any intrinsic property.

Flyweight Interface and Concrete Classes

Shape.java

```
package com.nt.fwp;

public interface Shape {
  public void draw(int arg0,String fillColor,StringlineStyle);

}
```

Square.java

```
package com.nt.fwp;

public class Square implements Shape{
   private String label;

   public Square(){
            label="square";
          System.out.println("Square:0-param constructor");
   }

  public void draw(intside,StringfillColor,StringlineStyle){
System.out.println("Drawing "+label+"having side "+side +" with fillColor"+fillColor+
            "and line style"+lineStyle);
   }
}
```

Circle.java

```
package com.nt.fwp;
public class Circle implements Shape{
   private String label;

  public Circle(){
            label="circle";
         System.out.println("Circle:0-param constructor");
  }

public void draw(intradius,StringfillColor,StringlineStyle){
System.out.println("Drawing "+label+"having radius "+radius +" with fillColor"+fillColor+
          "and fill style"+lineStyle);

  }
}
```

➔Flyweight pattern can be used for Objects that takes a lot of time while instantiated.

Flyweight Factory

The flyweight factory will be used by client programs to instantiate the Object, so we need to keep a map(cache) of Objects in the factory that should not be accessible by client application. Whenever client program makes a call to get an instance of Object, it should be returned from the HashMap, if not found then create a new Object and put in the Map and then return it. We need to make sure that all the intrinsic properties are considered while creating the Object.

Our flyweight factory class looks like below code.

ShapeFactory.java

```
package com.nt.fwp;

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

public class ShapeFactory {
  private static Map<String,Shape>shapeCache=new HashMap<String,Shape>();

   public synchronized static  Shape  getShape(String shapeType){
      if(!shapeCache.containsKey(shapeType)){
         if(shapeType.equals("square")){
            Shape square=new Square();
shapeCache.put(shapeType,square);
         }
        else if(shapeType.equals("circle")){
          Shape circle=new Circle();
shapeCache.put(shapeType,circle);
        }
      }
     return  shapeCache.get(shapeType);
   }//method
 }//class
```

Flyweight Pattern Client Example

ClientApp.java

```
package test.solution;

import com.nt.fwp.Circle;
import com.nt.fwp.Shape;
import com.nt.fwp.ShapeFactory;
import com.nt.fwp.Square;

public class ClientApp {

 public static void main(String[] args) {
          final int SIZE=500;
         for(int i=0;i<SIZE;++i)
         {
           Shape shape=ShapeFactory.getShape("square");
        shape.draw(i+1,"white"+i,"dashed"+i);
                }
System.out.println("----------------------------------");
        for(int i=0;i<SIZE;++i)
         {
```

```
        Shape shape=ShapeFactory.getShape("circle");
      shape.draw(i+1,"red"+i,"dotted"+i);
       }

}//main
}//class
```

Adapter Design Pattern

Adapter design pattern is one of the **structural design patternand** it'sused so that **two unrelated interfaces can work together**. The object that joins these unrelated interface is called an **Adapter**. As a real life example, we can think of a mobile charger as an adapter because mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket.

Let's take an example to understand better. We have PayPal Component as external Service; it takes the Bank Code, Payment Gate Way card Code to perform Oline Payment. But Customer/Cient gives Bank name, and Payment gate way name like VISA,MASTER and etc .. to E-Commerce website like FlipKart.com . In this case we can write an Adapter class which takes the bankName, Payment Gateway name and maps them toBankCode and Payment Gateway /Card Code.

Let Us develop PayPal comp as Extenal Service using Singleton Design Pattern

PaypalComp.java

```
package com.nt.extenal;

public class PaypalComp {
 private static PaypalComp instance;
 private PaypalComp(){

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

```
}
public String approveAmount(intcardNo,intCardCode,intbankCode,floatamt){
   //DB interactions and communications are required here
return cardNo+" has been approved to pay "+amt +" from "+bankCode;
}

}
```

The above class contains the logic for Online Payment expecting CardCode/Payment Gate way code and BankCode .Now client instead of giving Bank code, Card Code he is giving Bank name and Card name /Payment gateway name for Online Payment .
In this case the interface the client expected is different from the actual implementation which has been designed. To fix this problem we need to write one **adapter class** **as shown below**

PayShoppingAmtAdapter.java

```
package com.nt.ap;
import com.nt.extenal.PaypalComp;
public class PayShoppingAmtAdapter {
  public String payAmount(intcardNo,StringcardName,StringbankName,floatamt){
        intcardCode=0,bankCode=0;
        PaypalComp comp=null;
              String paymentMsg=null;
        // get Card codes from  DB s/w
        if(cardName.equals("Visa"))
              cardCode=111;
        else if(cardName.equals("Master"))
              cardCode=222;
        // get bank codes from Db s/w
        if(bankName.equals("ICICI"))
              bankCode=1001;
        else if(bankName.equals("HDFC"))
              bankCode=1002;
        // use PaypalComp
```

```
        comp=PaypalComp.getInstance();
        paymentMsg=comp.approveAmount(cardNo, cardCode, bankCode, amt);

        return paymentMsg;
}

}
```

Now the Online Shopping websites can use the PayPal comp with bank Name and card name because the Adapter will take care of mapping bank name to Bank Code and Card name to CardCode/Payment Gateway code.

PayShoppingAmount.java

```
package com.nt.ap;

public interface PayShoppingAmount {
 public String payAmount(intcardNo,StringcardName,StringbankName,floatamt);
}
```

PayShoppingAmountimpl.java

```
package com.nt.ap;

public class PayShoppingAmountImpl implements PayShoppingAmount {

 @Override
 public String payAmount(intcardNo, String cardName, String bankName, float amt) {
        PayShoppingAmtAdapter adapter=null;
        String paymentMsg=null;
        //Use adapter class for payment service
        adapter=new PayShoppingAmtAdapter();
        paymentMsg=adapter.payAmount(cardNo, cardName, bankName, amt);

        return paymentMsg;
 }//method
}//class
```

ClientApp.java

```
package test;

import com.nt.ap.PayShoppingAmount;
import com.nt.ap.PayShoppingAmountImpl;
```

```
public class ClientApp {
 public static void main(String[] args) {
PayShoppingAmount shopping=null;
    String cfrmMsg=null;
shopping=new PayShoppingAmountImpl();
cfrmMsg=shopping.payAmount(554544, "VISA", "ICICI",3000);
System.out.println(cfrmMsg);
 }
}
```