VIth Semester / B.E.
Design Pattern
LABORATORY (BECSE307T)

LAB MANUAL

Prepared By
Prof. Nusrat Anjum
Assistant Professor
Email: anjum.nusrat72@gmail.com

Prof. Nusrat Anjum
LABORATORY MANUAL CONTENTS

This manual is intended for the Third Year students of Computer Science and Engineering in the subject of Design Pattern. This manual typically contains practical/Lab Sessions related to the subject to enhance understanding.

The time spent on getting the design right before you start programming will almost always save you time in the end. It's much, much easier to make major changes on a design, which is after all just squiggly lines on paper, and then it is to make changes in hundreds or thousands of lines of code.

The design process is typically split into distinct phases: Object Oriented Design (OOD) and Design Pattern.

Students are advised to thoroughly go through this manual rather than only topics mentioned in the syllabus as practical aspects are the key to understanding and conceptual visualization of theoretical aspects covered in the books.

Good Luck for your Enjoyable Laboratory Sessions
DOs and DON’Ts in Laboratory:

1. Make entry in the Log Book as soon as you enter the Laboratory.

2. All the students should sit according to their roll numbers starting from their left to right.

3. All the students are supposed to enter the terminal number in the log book.

4. Do not change the terminal on which you are working.

5. All the students are expected to get at least the algorithm of the program/concept to be implemented.

6. Strictly observe the instructions given by the teacher/Lab Instructor.

Instruction for Laboratory Teachers:

1. Submission related to whatever lab work has been completed should be done during the next lab session. The immediate arrangements for printouts related to submission on the day of practical assignments.

2. Students should be taught for taking the printouts under the observation of lab teacher.

3. The promptness of submission should be encouraged by way of marking and evaluation patterns that will benefit the sincere students.

Prof. Nusrat Anjum       Prof. M. S. Khatib
Prof. Almas Ansari             HOD, CSE
Subject Incharge

Prof. Nusrat Anjum
Vision of CSE Department

To achieve excellent standards of quality education in the field of computer science and engineering, aiming towards development of ethically strong technical experts contributing to the profession in the global society.

Mission of CSE Department

1- To create outcome based education environment for learning and attaining career goals.
2- Provide latest tools in a learning ambience to enhance innovations, problem solving skills, leadership qualities team spirit and ethical responsibilities.
3- Inculcating awareness through innovative activities in the emerging areas of technology.

Program Educational Objectives (PEOs)

PEO1:- The graduates will have a strong foundation in mathematical, scientific, and engineering advanced and fundamental concepts, necessary to formulate, solve and analyze engineering problem in their career.

PEO2:- Graduates will be able to create and design computer support systems and have knowledge and skills to analyze, design, test and implement various software applications.

PEO3:- Graduates will work productively as computer science engineers towards betterment of society exhibiting ethical qualities.
Program Specific Outcomes (PSOs)

PSO1:- Foundation of mathematical concepts: To use mathematical methodologies and techniques for computing and solving problem using suitable mathematical analysis, data structures, database and algorithms as per the requirement.

PSO2:- Foundation of Computer System: The capability and ability to interpret and understand the fundamental concepts and methodology of computer systems and programming. Students can understand the functionality of hardware and software aspects of computer systems, networks and security.

PSO3:- Foundations of Software development: The ability to grasp the software development lifecycle and methodologies of software system and project development.
Programme Outcomes (POs):

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Prof. Nusrat Anjum
Course Outcome (CO’s)

CO1:
Identify the purpose and methods of common object-oriented design patterns.

CO2:
Develop programming for basic object-oriented programming concepts.

CO3:
Select and apply these patterns in their own designs for simple programs.

CO4:
Create documentation outlining the testable and complete design of a simple program.

CO5:
Produce and present documents for the purpose of capturing software requirements and specification.
**Index**

<table>
<thead>
<tr>
<th>Sr. No</th>
<th>Practical</th>
<th>Page no</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction to Design patterns. Describe the organization of catalog along with the following design patterns.</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>a. Creational Patterns.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. Structural Patterns.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. Behavioral Patterns.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Write a program to implement the following concepts in java.</td>
<td>08</td>
</tr>
<tr>
<td></td>
<td>b. Interface.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. Abstract class.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a. Write a Program to implement Factory pattern.</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>b. Write a program to implement abstract factory.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a. Write a Program to implement Singleton pattern.</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>b. Write a Program to implement Composite design pattern.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a. Write a program to implement decorator pattern.</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>b. Write a Program to implement proxy design pattern.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a. Write a Program to design chain of responsibility pattern.</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>b. Write a Program to design mediator pattern.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Write a program to implement iterator pattern.</td>
<td>53</td>
</tr>
<tr>
<td>8</td>
<td>Write a program to implement visitor pattern.</td>
<td>57</td>
</tr>
<tr>
<td>9</td>
<td>Case Study: Railway Reservation System</td>
<td>63</td>
</tr>
<tr>
<td>10</td>
<td>Describe the Methods to analyze the complexity of design patterns.</td>
<td>65</td>
</tr>
</tbody>
</table>
Practical No: 1
Aim: Introduction to Design patterns. Describe the organization of catalog along with the following design patterns.

a. Creational Patterns.
b. Structural Patterns.
c. Behavioral Patterns.

Theory:

Design patterns describe the relations and interactions of different class or objects or types. They do not specify the final class or types that will be used in any software code, but give an abstract view of the solution. Patterns show us how to build systems with good object oriented design qualities by reusing successful designs and architectures. Expressing proven techniques speed up the development process and make the design patterns, more accessible to developers of new system.

Good object oriented software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns and we can also avoid alternatives that compromise reusability.

Classification of Design Patterns

These patterns are also called as GOF patterns (Gang-Of-Four). Their names are listed in the table given below.

- Creational patterns concern the process of object creation
- Structural patterns deal with the composition of classes and objects
- Behavioural patterns characterize the ways in which classes and objects interact and distribute responsibility.
- Scope, which specifies whether the pattern applies primarily to classes or to objects:
  
  o Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static.

  o Object patterns deal with object relationships, which can be changed at run-time and are more dynamic.

The following table shows the classification of design patterns. Note, that those are the patterns introduced by the Gang of Four.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Design Patterns</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creational</td>
<td>Abstract Factory</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Builder</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Factory Method</td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td>Prototype</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td>Object</td>
</tr>
<tr>
<td>Structural</td>
<td>Adapter</td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td>Bridge</td>
<td>Object</td>
</tr>
<tr>
<td>Behavioral</td>
<td>Chain of Responsibility</td>
<td>Object</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Command</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>Interpreter</td>
<td>Class</td>
<td></td>
</tr>
<tr>
<td>Iterator</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>Mediator</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>Memento</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>Strategy</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>Template Method</td>
<td>Class</td>
<td></td>
</tr>
<tr>
<td>Visitor</td>
<td>Object</td>
<td></td>
</tr>
</tbody>
</table>

### Creational patterns

Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented.

A class creational pattern uses inheritance to vary the class that’s instantiated, whereas an object creational pattern will delegate instantiation to another object.

The creational design patterns allow configuring of a software system as a system with —product objects that vary widely in structure and functionality. Such configuration can be static, i.e., specified at compile-time, or dynamic, i.e., specified at run-time.

In software engineering, creational design patterns are 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 could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

- **Abstract Factory**
  Creates an instance of several families of classes
o **Builder**  
Separates object construction from its representation

o **Factory Method**  
Creates an instance of several derived classes

o **Object Pool**  
Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

o **Prototype**  
A fully initialized instance to be copied or cloned

o **Singleton**  
A class of which only a single instance can exist

---

**Structural design patterns**

Structural patterns are concerned with how classes and objects are composed to form larger structures.

Structural class patterns use inheritance to compose interfaces or implementations. For example, multiple inheritance can be seen as a kind of structural design patterns, since it uses inheritance to mix two or more classes into a new one.

Rather than composing interfaces or implementations, structural object patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

This design pattern is all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.
- **Adapter**
  Match interfaces of different classes

- **Bridge**
  Separates an object’s interface from its implementation

- **Composite**
  A tree structure of simple and composite objects

- **Decorator**
  Add responsibilities to objects dynamically

- **Facade**
  A single class that represents an entire subsystem

- **Flyweight**
  A fine-grained instance used for efficient sharing

- **Private Class Data**
  Restricts accessor/mutator access

- **Proxy**
  An object representing another object

**Behavioral design patterns**

Behavioral patterns are concerned with algorithms and the assignment of responsibility between objects. Behavioural patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that is difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Behavioral class patterns use inheritance to distribute behaviour between classes.

Behavioral object patterns use object composition rather than inheritance. For example, a behavioral object pattern can describe how a group of object might cooperate to perform a task that no single object can carry out by itself. A typical example is the Observer pattern from the Smalltalk (Model/View/Controller paradigm). Views are used to show the state of data
(contained in Model) and they are observers of this data. Whenever a model changes its state all views are notified and they can update the representation of the data in views.

This design patterns is all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

- **Chain of responsibility**
  A way of passing a request between a chain of objects

- **Command**
  Encapsulate a command request as an object

- **Interpreter**
  A way to include language elements in a program

- **Iterator**
  Sequentially access the elements of a collection

- **Mediator**
  Defines simplified communication between classes

- **Memento**
  Capture and restore an object's internal state

- **Null Object**
  Designed to act as a default value of an object

- **Observer**
  A way of notifying change to a number of classes

- **State**
  Alter an object's behavior when its state changes

- **Strategy**
  Encapsulates an algorithm inside a class

- **Template method**
  Defer the exact steps of an algorithm to a subclass

- **Visitor**
  Defines a new operation to a class without change

**Conclusion:**

**Thus we, have studied organization of catalog along with all design patterns successfully.**
Practical No: 2
Aim:

Write a program to implement the following concepts in java.

b. Interface.
c. Abstract class.

Theory:

- Method Overriding: Declaring a method in sub class which is already present in parent class is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method.

Method Overriding Example

Lets take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The Boy class extends Human class. Both the classes have a common method void eat(). Boy class is giving its own implementation to the eat() method or in other words it is overriding the eat() method.

The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating.

Sample Code:

class Human{
    //Overridden method
    public void eat()
    {
        System.out.println("Human is eating");
    }
} 
class Boy extends Human{
    //Overriding method
public void eat(){
    System.out.println("Boy is eating");
}

public static void main( String args[] ) {
    Boy obj = new Boy();
    //This will call the child class version of eat()
    obj.eat();
}

Output: Boy is eating.

Interface:

The interface in java is a mechanism to achieve abstraction. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java. It cannot be instantiated just like abstract class.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Sample Code:

```java
interface printable{
    void print();
}

class A6 implements printable{
    public void print(){System.out.println("Hello");}
}

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

Prof. Nusrat Anjum
A6 obj = new A6();

obj.print();

Output: Hello

- Abstract Class:

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

**abstract method**

A method that is declared as abstract and does not have implementation is known as abstract method.

**Sample Code:**

```java
abstract class Bike{
    abstract void run();
}

class Honda4 extends Bike{
    void run(){System.out.println("running safely..");}

    public static void main(String args[]){
        Bike obj = new Honda4();
```
obj.run();
}
}

Output: running safely

Conclusion: Thus we have executed basic java programs successfully.
Practical No:3
a. Aim: Write a Program to implement Factory pattern.

Theory:

- Defines an interface for creating objects but let sub-classes decide which of those instantiate.
- Enables the creator to defer Product creation to a sub-class.
- Factory pattern is one of the most used design pattern in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Intent:

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Also Known As:

- Virtual Constructor

Applicability:

Use the Factory Method pattern when

- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.
Sample code:

Step 1
Create an interface.

Shape.java

```java
public interface Shape {
    void draw();
}
```

Step 2
Create concrete classes implementing the same interface.

Rectangle.java

```java
public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}
```
Square.java

```java
public class Square implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}
```

Circle.java

```java
public class Circle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}
```

Step 3
Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```java
public class ShapeFactory {

    //use getShape method to get object of type shape
    public Shape getShape(String shapeType){
        if(shapeType == null){
```
return null;
}
if(shapeType.equalsIgnoreCase("CIRCLE")) {
    return new Circle();
}
else if(shapeType.equalsIgnoreCase("RECTANGLE")) {
    return new Rectangle();
}
else if(shapeType.equalsIgnoreCase("SQUARE")) {
    return new Square();
}
return null;

Step 4
Use the Factory to get object of concrete class by passing an information such as type.

FactoryPatternDemo.java

public class FactoryPatternDemo {

    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        //get an object of Circle and call its draw method.
        Shape shape1 = shapeFactory.getShape("CIRCLE");

        //call draw method of Circle

shape1.draw();

//get an object of Rectangle and call its draw method.
Shape shape2 = shapeFactory.getShape("RECTANGLE");

//call draw method of Rectangle
shape2.draw();

//get an object of Square and call its draw method.
Shape shape3 = shapeFactory.getShape("SQUARE");

//call draw method of circle
shape3.draw();

Step 5
Verify the output.

Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
b. Aim: Write a program to implement abstract factory.

Theory:

- Create instances of classes belonging to different families.
- Abstract Factory patterns works around a super-factory which creates other factories. This factory is also called as Factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- In Abstract Factory pattern an interface is responsible for creating a factory of related objects, without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

Intent:

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Also Known As: Kit

Applicability:

Use the Abstract Factory pattern when

- a system should be independent of how its products are created, composed, and represented.
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce this constraint.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.
Class Diagram:

Sample Code:

Step 1
Create an interface for Shapes.

*Shape.java*

```java
public interface Shape {
    void draw();
}
```

Step 2
Create concrete classes implementing the same interface.

*Rectangle.java*

```java
public class Rectangle implements Shape {
```
@Override
public void draw() {
    System.out.println("Inside Rectangle::draw() method.");
}

Square.java

public class Square implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}

Circle.java

public class Circle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}

Step 3
Create an interface for Colors.

Color.java

public interface Color {
    void fill();
}
Step 4
Create concrete classes implementing the same interface.

Red.java

```java
public class Red implements Color {

    @Override
    public void fill() {
        System.out.println("Inside Red::fill() method.");
    }
}
```

Green.java

```java
public class Green implements Color {

    @Override
    public void fill() {
        System.out.println("Inside Green::fill() method.");
    }
}
```

Blue.java

```java
public class Blue implements Color {

    @Override
    public void fill() {
        System.out.println("Inside Blue::fill() method.");
    }
}
```
Step 5
Create an Abstract class to get factories for Color and Shape Objects.

*AbstractFactory.java*

```java
public abstract class AbstractFactory {
    abstract Color getColor(String color);
    abstract Shape getShape(String shape);
}
```

Step 6
Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

*ShapeFactory.java*

```java
public class ShapeFactory extends AbstractFactory {

    @Override
    public Shape getShape(String shapeType) {

        if (shapeType == null) {
            return null;
        }

        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
```
return new Square();
}

return null;
}

@Override
Color getColor(String color) {
    return null;
}

ColorFactory.java

public class ColorFactory extends AbstractFactory {

    @Override
    public Shape getShape(String shapeType){
        return null;
    }

    @Override
    Color getColor(String color) {
        if(color == null){
            return null;
        }
    }
}
if (color.equalsIgnoreCase("RED")) {
    return new Red();
} else if (color.equalsIgnoreCase("GREEN")) {
    return new Green();
} else if (color.equalsIgnoreCase("BLUE")) {
    return new Blue();
}

return null;

Step 7
Create a Factory generator/producer class to get factories by passing an information such as Shape or Color

FactoryProducer.java

public class FactoryProducer {
    public static AbstractFactory getFactory(String choice) {
        if (choice.equalsIgnoreCase("SHAPE")) {
            return new ShapeFactory();
        }

        else if (choice.equalsIgnoreCase("COLOR")) {
            return new ColorFactory();
        }

        return null;
    }
}
Step 8
Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

AbstractFactoryPatternDemo.java

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

        //get shape factory
        AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");

        //get an object of Shape Circle
        Shape shape1 = shapeFactory.getShape("CIRCLE");

        //call draw method of Shape Circle
        shape1.draw();

        //get an object of Shape Rectangle
        Shape shape2 = shapeFactory.getShape("RECTANGLE");

        //call draw method of Shape Rectangle
        shape2.draw();

        //get an object of Shape Square
        Shape shape3 = shapeFactory.getShape("SQUARE");
    }
}
```
//call draw method of Shape Square
shape3.draw();

//get color factory
AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");

//get an object of Color Red
Color color1 = colorFactory.getColor("RED");

//call fill method of Red
color1.fill();

//get an object of Color Green
Color color2 = colorFactory.getColor("Green");

//call fill method of Green
color2.fill();

//get an object of Color Blue
Color color3 = colorFactory.getColor("BLUE");

//call fill method of Color Blue
color3.fill();

Step 9
Verify the output.
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside Red::fill() method.
Inside Green::fill() method.
Inside Blue::fill() method.

Conclusion:

Thus we have executed programs for factory pattern and abstract factory successfully.
Practical No: 4
a. Aim: Write a program to implement singleton pattern.

Theory:

- Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best way to create an object.

- This pattern involves a single class which is responsible to creates own object while making sure that only single object get created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class. Enables the creator to defer Product creation to a sub-class.

- A class with only one single possible instance.
  - Private constructor
  - Global access

- Intent

  Ensure a class has only one instance, and provide a global point of access to it.

  Encapsulated "just-in-time initialization" or "initialization on first use".

Applicability:

- Use the Singleton pattern when

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
Class Diagram:

Sample Code:

Step 1
Create a Singleton Class.

SingleObject.java

```java
public class SingleObject {
    // create an object of SingleObject
    private static SingleObject instance = new SingleObject();
    // make the constructor private so that this class cannot be
    // instantiated
    private SingleObject() {}
    // Get the only object available
    public static SingleObject getInstance() {
        return instance;
    }
}
```
public void showMessage() {
    System.out.println("Hello World!");
}

Step 2
Get the only object from the singleton class.

SingletonPatternDemo.java

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

        //illegal construct
        //Compile Time Error: The constructor SingleObject() is not visible
        //SingleObject object = new SingleObject();

        //Get the only object available
        SingleObject object = SingleObject.getInstance();

        //show the message
        object.showMessage();
    }
}

Step 3
Verify the output.

Hello World!
b. **Aim:** Write a Program to implement Composite design pattern.

**Theory:**

- Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchies.

- This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.

- This pattern creates a class contains group of its own objects. This class provides ways to modify its group of same objects.

- We are demonstrating use of Composite pattern via following example in which show employees hierarchy of an organization.

- **Intent**

  Compose objects into tree structures to represent part-whole hierarchies.

- **Applicability**

  Use the Composite pattern when

  - you want to represent part-whole hierarchies of objects.
  - you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

**Class Diagram:**

![Class Diagram](image_url)
Sample Code:

Step 1
Create Employee class having list of Employee objects.

Employee.java

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

public class Employee {
    private String name;
    private String dept;
    private int salary;
    private List<Employee> subordinates;

    // constructor
    public Employee(String name, String dept, int sal) {
        this.name = name;
        this.dept = dept;
        this.salary = sal;
        subordinates = new ArrayList<Employee>();
    }

    public void add(Employee e) {
        subordinates.add(e);
    }

    public void remove(Employee e) {
        subordinates.remove(e);
    }
}
```
Step 2

Use the Employee class to create and print employee hierarchy.

CompositePatternDemo.java

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

      Employee CEO = new Employee("John","CEO", 30000);

      Employee headSales = new Employee("Robert","Head Sales", 20000);

      Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

      Employee clerk1 = new Employee("Laura","Marketing", 10000);

      Employee clerk2 = new Employee("Bob","Marketing", 10000);

      Employee salesExecutive1 = new Employee("Richard","Sales", 10000);

    }
  }
Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

CEO.add(headSales);
CEO.add(headMarketing);

headSales.add(salesExecutive1);
headSales.add(salesExecutive2);

headMarketing.add(clerk1);
headMarketing.add(clerk2);

//print all employees of the organization
System.out.println(CEO);

for (Employee headEmployee : CEO.getSubordinates()) {
    System.out.println(headEmployee);

    for (Employee employee : headEmployee.getSubordinates()) {
        System.out.println(employee);
    }
}

Step 3
Verify the output.

Employee : [ Name : John, dept : CEO, salary : 30000 ]
Employee : [ Name : Robert, dept : Head Sales, salary : 20000 ]
Employee : [ Name : Richard, dept : Sales, salary : 10000 ]
Employee : [ Name : Rob, dept : Sales, salary : 10000 ]
Employee : [ Name : Michel, dept : Head Marketing, salary : 20000 ]
Employee : [ Name : Laura, dept : Marketing, salary : 10000 ]
Employee : [ Name : Bob, dept : Marketing, salary : 10000 ]

Conclusion:

Thus, we have executed programs for singleton and composite pattern.
Practical No: 5
a. Aim: Write a program to implement decorator pattern.

Theory:

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

Also Known As

Wrapper

Applicability

Use Decorator

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by sub classing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for sub classing.
Class Diagram:

Sample Code:

Step 1
Create an interface.

Shape.java

```java
public interface Shape {
    void draw();
}
```

Step 2
Create concrete classes implementing the same interface.

Rectangle.java

```java
public class Rectangle implements Shape {
    @Override
    public void draw() {
        super.draw();
    }
}
```
public void draw() {
    System.out.println("Shape: Rectangle");
}

Circle.java

public class Circle implements Shape {

    @Override
    public void draw() {
        System.out.println("Shape: Circle");
    }
}

Step 3
Create abstract decorator class implementing the Shape interface.

ShapeDecorator.java

public abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

    public ShapeDecorator(Shape decoratedShape) {
        this.decoratedShape = decoratedShape;
    }

    public void draw() {
        decoratedShape.draw();
    }
}
Step 4
Create concrete decorator class extending the *ShapeDecorator* class.

*RedShapeDecorator.java*

```java
public class RedShapeDecorator extends ShapeDecorator {

    public RedShapeDecorator(Shape decoratedShape) {
        super(decoratedShape);
    }

    @Override
    public void draw() {
        decoratedShape.draw();
        setRedBorder(decoratedShape);
    }

    private void setRedBorder(Shape decoratedShape) {
        System.out.println("Border Color: Red");
    }
}
```

Step 5
Use the *RedShapeDecorator* to decorate *Shape* objects.

*DecoratorPatternDemo.java*

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

        Shape circle = new Circle();
```
Shape redCircle = new RedShapeDecorator(new Circle());

Shape redRectangle = new RedShapeDecorator(new Rectangle());

System.out.println("Circle with normal border");
circle.draw();

System.out.println("\nCircle of red border");
redCircle.draw();

System.out.println("\nRectangle of red border");
redRectangle.draw();
b. Aim: Write a Program to implement proxy design pattern.

Theory:

- In Proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern.
- In Proxy pattern, we create object having original object to interface its functionality to outer world.

Intent

- Provide a surrogate or placeholder for another object to control access to it.

Also Known As

- Surrogate

Applicability:

- A **remote proxy provides a local representative for an object in a different** address space. NEXTSTEP [Add94] uses the class NXProxy for this purpose.
- **Virtual proxy creates expensive objects on demand.** The ImageProxy described in the Motivation is an example of such a proxy.

Class Diagram:
Sample Code:

Step 1
Create an interface.

*Image.java*

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

Step 2
Create concrete classes implementing the same interface.

*RealImage.java*

```java
public class RealImage implements Image {
    private String fileName;

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

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

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

```java
public class ProxyImage implements Image{

    private RealImage realImage;
    private String fileName;

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

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

Step 3

Use the ProxyImage to get object of RealImage class when required.

ProxyPatternDemo.java

```java
public class ProxyPatternDemo {

    public static void main(String[] args) {
        Image image = new ProxyImage("test_10mb.jpg");
    }
```
Step 4
Verify the output.

Loading test_10mb.jpg
Displaying test_10mb.jpg
Displaying test_10mb.jpg

Conclusion:
Thus, we have executed decorator and proxy pattern successfully.
Practical No: 6
a. Aim: Write a Program to design chain of responsibility pattern.

Theory:

As the name suggests, the chain of responsibility pattern creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request. This pattern comes under behavioral patterns.

In this pattern, normally each receiver contains reference to another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

Example: ATM (rupees of 1000, 500, 100 etc)

Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Applicability

Use Chain of Responsibility when

- more than one object may handle a request, and the handler isn't known \textit{a priori}. The handler should be ascertained automatically.
- you want to issue a request to one of several objects without specifying the receiver explicitly.
- the set of objects that can handle a request should be specified dynamically.

Class Diagram:
Sample Code:

Step 1
Create an abstract logger class.

AbstractLogger.java

```java
public abstract class AbstractLogger {
    public static int INFO = 1;
    public static int DEBUG = 2;
    public static int ERROR = 3;

    protected int level;

    //next element in chain or responsibility
    protected AbstractLogger nextLogger;
}
```
public void setNextLogger(AbstractLogger nextLogger){
    this.nextLogger = nextLogger;
}

public void logMessage(int level, String message){
    if(this.level <= level){
        write(message);
    }
    if(nextLogger !=null){
        nextLogger.logMessage(level, message);
    }
}

abstract protected void write(String message);

Step 2
Create concrete classes extending the logger.

ConsoleLogger.java

public class ConsoleLogger extends AbstractLogger {

    public ConsoleLogger(int level){
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("Standard Console::Logger: " + message);
    }
}
ErrorLogger.java

```java
public class ErrorLogger extends AbstractLogger {

    public ErrorLogger(int level) {
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("Error Console::Logger: " + message);
    }
}
```

FileLogger.java

```java
public class FileLogger extends AbstractLogger {

    public FileLogger(int level) {
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("File::Logger: " + message);
    }
}
```
Step 3
Create different types of loggers. Assign them error levels and set next logger in each logger. Next logger in each logger represents the part of the chain.

*ChainPatternDemo.java*

```java
public class ChainPatternDemo {

    private static AbstractLogger getChainOfLoggers()
    {
        AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
        AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
        AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);

        errorLogger.setNextLogger(fileLogger);
        fileLogger.setNextLogger(consoleLogger);

        return errorLogger;
    }

    public static void main(String[] args) {
        AbstractLogger loggerChain = getChainOfLoggers();

        loggerChain.logMessage(AbstractLogger.INFO, "This is an information.");

        loggerChain.logMessage(AbstractLogger.DEBUG, "This is an debug level information.");
    }
}
```
loggerChain.logMessage(AbstractLogger.ERROR,

    "This is an error information.");

{}
b. Aim: Write a Program to design mediator pattern.

Theory:

- Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintainability of the code by loose coupling. Mediator pattern falls under behavioral pattern category.

- **Intent**

  Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

- **Applicability**

  Use the Mediator pattern when:
  
  - a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
  
  - reusing an object is difficult because it refers to and communicates with many other objects.

Class Diagram:

```
MediatorPatternDemo

+main() : void

User

+name : String
+User() : void
+getName() : void
+setName() : void
+sendMessage() : void

ChatRoom

+showMessage() : void
```
Sample Code:

Step 1
Create mediator class.

*ChatRoom.java*

```java
import java.util.Date;

public class ChatRoom {
    public static void showMessage(User user, String message){
        System.out.println(new Date().toString() + " [" + user.getName() + "] : " + message);
    }
}
```

Step 2
Create user class

*User.java*

```java
public class User {
    private String name;

    public String getName() {
        return name;
    }

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

    public User(String name){
        this.name = name;
    }
```
public void sendMessage(String message){
    ChatRoom.showMessage(this.message);
}

Step 3
Use the User object to show communications between them.

MediatorPatternDemo.java

public class MediatorPatternDemo {
    public static void main(String[] args) {
        User robert = new User("Robert");
        User john = new User("John");
        robert.sendMessage("Hi! John!");
        john.sendMessage("Hello! Robert!");
    }
}

Step 4
Verify the output.

Thu Jan 31 16:05:46 IST 2013 [Robert] : Hi! John!

Conclusion:

Thus, we have executed programs for chain of responsibility and mediator pattern successfully.
Practical No: 7
Aim: Write a program to implement iterator pattern.

Theory:

Iterator pattern is very commonly used design pattern in Java and .Net programming environment. This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

Iterator pattern falls under behavioral pattern category.

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Also Known As

Cursor

Applicability

Use the Iterator pattern:

- to access an aggregate object’s contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Class Diagram:
Sample Code:

Step 1
Create interfaces.

*Iterator.java*

```java
public interface Iterator {
    public boolean hasNext();
    public Object next();
}
```

*Container.java*

```java
public interface Container {
    public Iterator getIterator();
}
```

Step 2
Create concrete class implementing the *Container* interface. This class has inner class *NameIterator* implementing the *Iterator* interface.

*NameRepository.java*

```java
public class NameRepository implements Container {
    public String names[] = {"Robert", "John", "Julie", "Lora"};

    @Override
    public Iterator getIterator() {
        return new NameIterator();
    }

    private class NameIterator implements Iterator {
```
int index;

@Override
public boolean hasNext() {
    if(index < names.length){
        return true;
    }
    return false;
}

@Override
public Object next() {
    if(this.hasNext()){
        return names[index++];
    }
    return null;
}

Step 3
Use the NameRepository to get iterator and print names.

IteratorPatternDemo.java
public class IteratorPatternDemo {

    public static void main(String[] args) {
        NameRepository namesRepository = new NameRepository();
        for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
            String name = (String)iter.next();
            System.out.println("Name : " + name);
        }
    }
}

Step 4
Verify the output.

Name : Robert
Name : John
Name : Julie
Name : Lora

Conclusion: Thus, we have executed iterator pattern successfully.
Practical No: 8
Aim: Write a program to implement visitor pattern.

Theory:

In Visitor pattern, we use a visitor class which changes the executing algorithm of an element class. By this way, execution algorithm of element can vary as and when visitor varies. This pattern comes under behavior pattern category. As per the pattern, element object has to accept the visitor object so that visitor object handles the operation on the element object.

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Applicability

Use the Visitor pattern when

- an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.

- many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.

- the classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes.
Class Diagram:

Sample Code:

Step 1
Define an interface to represent element.

*ComputerPart.java*

```java
public interface ComputerPart {
    public void accept(ComputerPartVisitor computerPartVisitor);
}
```

Step 2
Create concrete classes extending the above class.

*Keyboard.java*

```java
public class Keyboard implements ComputerPart {
}
```
@Override
public void accept(ComputerPartVisitor computerPartVisitor) {
    computerPartVisitor.visit(this);
}

Monitor.java

public class Monitor implements ComputerPart {

    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        computerPartVisitor.visit(this);
    }
}

Mouse.java

public class Mouse implements ComputerPart {

    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        computerPartVisitor.visit(this);
    }
}

Computer.java

public class Computer implements ComputerPart {

    ComputerPart[] parts;
}
public Computer() {
    parts = new ComputerPart[] { new Mouse(), new Keyboard(), new Monitor() };
}

@Override
public void accept(ComputerPartVisitor computerPartVisitor) {
    for (int i = 0; i < parts.length; i++) {
        parts[i].accept(computerPartVisitor);
    }
    computerPartVisitor.visit(this);
}

Step 3
Define an interface to represent visitor.

ComputerPartVisitor.java

public interface ComputerPartVisitor {
    public void visit(Computer computer);
    public void visit(Mouse mouse);
    public void visit(Keyboard keyboard);
    public void visit(Monitor monitor);
}

Step 4
Create concrete visitor implementing the above class.

ComputerPartDisplayVisitor.java

public class ComputerPartDisplayVisitor implements ComputerPartVisitor {


@Override
public void visit(Computer computer) {
    System.out.println("Displaying Computer.");
}

@Override
public void visit(Mouse mouse) {
    System.out.println("Displaying Mouse.");
}

@Override
public void visit(Keyboard keyboard) {
    System.out.println("Displaying Keyboard.");
}

@Override
public void visit(Monitor monitor) {
    System.out.println("Displaying Monitor.");
}

Step 5
Use the ComputerPartDisplayVisitor to display parts of Computer.

VisitorPatternDemo.java

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

        ComputerPart computer = new Computer();
        computer.accept(new ComputerPartDisplayVisitor());
Step 6
Verify the output.

Displaying Mouse.
Displaying Keyboard.
Displaying Monitor.
Displaying Computer.

Conclusion:
Thus, we have executed visitor pattern successfully.
Practical No: 9
Aim : Case Study: Banking System

Theory:

The IT Environment With over 50,000 physical servers, a growing private cloud environment (VMware vSphere, Microsoft HyperV, IBM PowerVM), and hundreds of storage arrays (EMC Symmetrix, HDS, netApp), managing IT operations at this large bank is no small feat. More than 1,600 IT engineers in over a dozen datacenters worldwide work closely with 3rd party partners and vendors to meet the bank’s service availability goals.

Challenges:

With millions of customers accessing the bank systems daily at ATMs, branches, online, and through multiple call centers, any downtime or service disruptions are practically unacceptable to the bank. With a growing portion of customers relying on online and mobile banking, 24/7 service reliability has become more critical than ever. To address these needs, major efforts and resources have been directed towards the creation of a robust high availability and disaster recovery infrastructure. In this complex infrastructure comprising multiple datacenters, configuration changes are undertaken daily by different groups in various parts of the environment. While each team was making an effort to apply best practices in its own domain, there was no visibility to the implications and risks introduced by such modifications on the overall stability, service availability, and DR readiness of critical systems. As the IT environment has grown in size and complexity, keeping production high availability and disaster recovery systems in complete sync across IT teams and domains (e.g., server, storage, databases and virtualization) has become an increasing challenge. Moreover, management was lacking visibility into how well the organization was keeping up with established Service Level Agreements (SLA’s) for availability (RTO), data protection (RPO), and retention objectives.

In Search of a Solution:

While overall service performance of the bank has been satisfactory, minor incidents over a several month period have alerted the bank’s IT stewards to hidden vulnerabilities and the lack of visibility to impending downtime and data loss risks. Although the bank managed to escape these incidents without a major service impact, other banks were not so lucky. Publicized outages at major financial institutions around the world caught the attention of senior management. They were determined to avoid seeing their bank suffer similar embarrassment and dire financial consequences. Following management’s directive, a committee was put in place to define the requirements for a solution: Proactively detect risks introduced by configuration
changes across the entire datacenter and DR environments. Analyze the potential impact of such risks on service availability levels and disaster recovery readiness. Help the relevant teams pinpoint the source of each risk identified. Provide management with a consolidated view of downtime and data loss risks across the entire environment. Measure adherence to availability and data recovery SLA’s (RPO, RTO, redundancy, DR capacity). Simplify internal and regulatory compliance reporting. Improve DR capacity management and planning. In addition, the IT team mandated the solution must operate in a nonintrusive manner (preferably agentless) and integrate with the service management system used by the organization.

**Getting a Solution in Place:**

Following an extensive search, the bank’s team identified Continuity Software’s AvailabilityGuard as the preferred solution for protecting the bank against service availability risks. The fact that AvailabilityGuard was already used by other large financial institutions gave the team an added measure of confidence in the decision. To ensure the proper process was put in place, the initial implementation was limited to 1,000 servers. A joint committee with members from the Business Continuity and IT Operations groups was established. Over the first six months of operating the system, the committee established and documented procedures for problem identification, management reporting, and continuous improvement. The flexibility of the AvailabilityGuard solution allowed the bank to seamlessly adapt the solution to support the availability assurance processes established by the organization, including:

- Daily scan of the IT infrastructure
- Automated routing of critical issues to the enterprise IT incident management system, ensuring that risks are addressed immediately as they are uncovered
- Weekly, monthly, and quarterly health and SLA compliance scores for each IT domain and business service delivered to management as well as internal and external auditors

One year following the initial implementation, AvailabilityGuard is now used to monitor all Tier One and Tier Two business services, encompassing over 6,000 servers.

**Results:**

Since the bank has implemented the AvailabilityGuard solution, downtime and data loss risks have decreased by over 70%. AvailabilityGuard has also been helping the bank break down many of the long-standing IT silos. AvailabilityGuard provides a unified platform that detects availability risks across all IT domains, enabling collaboration among the various teams. And since most issues are identified and resolved before they impact service availability, there is a 50% decrease in time spent on resolving emergency issues. With AvailabilityGuard well-integrated into the bank’s IT infrastructure and processes, planning for the next phase of implementation is underway, expanding the coverage of AvailabilityGuard to include all production and DR systems.
Aim: Describe the Methods to analyze the complexity of design patterns.

Theory:
- **Complexity analysis of the design pattern.**
- Design pattern are a method that enables reuse of code and good solution since it contain experiences from successful solution to other similar problem.
- The fact that pattern are flexible is not necessarily a benefit since extended functionality than patterns enables may not be used by the original application.
- One common experience from all the projects is the common vocabulary that pattern introduce which reduce the misunderstandings between team members and raise the abstraction level of discussion.
- The time developing a framework based on design pattern gain the productivity despite the high initial learning work of the framework.
- Patterns are not a silver bullet solution, developers will still have to deal with complex design, analysis and implementation issues.
• A design pattern affects different software quality attributes and the use of a pattern may result in an increase of flexibility but at the cost of an undesirable increase of complexity.

• Two categories were raised useful in analysis of the design pattern to avoid incorrect applied pattern.

1) The first category is patterns that were misused by software developer who had not understood rational behind the pattern.

2) The second category is patterns which do not fall into the first category but do not fulfill the requirements in the project. The second category included some estimated future needs of flexibility, the facts that required to change during the project cycle and thereby patterns which were perfect at first time.

* Following are the methods to analyze the complexity of design pattern.

4.3 Design problem and characteristics:
This study identified the system design problem to be addressed through the
Use of complexity reducing patterns specified series of complexity relevant parameters & desired resolution of selected candidic problem to be addressed in the object.

3.5B Design plan and methodology:
The section defines a design plan methodology for the rapid development of cyber-physical system based on the application of verified design pattern on verified system component models.

3.6B Design pattern models:
This section describe the design pattern developed in the project pattern were developed for sales, replication, leader selection and fusion.

4.7B System architecture models:
This section describe the architecture models developed in the project. These architecture models provide example that can be used to evaluate the design and verification tools created.
Conclusion:

Thus, we have studied Methods to analyze the complexity of design patterns.