Design Pattern

Unit: I

Introduction to Design Pattern

Course Details
(B. Tech. 5th Sem)

Ibrar Ahmed
(Asst. Professor)
CSE Department

Faculty Introduction

Name	**Ibrar Ahmed**
Qualification	M. Tech. (Computer Engineering)
Designation	Assistant Professor
Department	Computer Science & Engineering
Total Experience	4 years
NIET Experience	1 years
Subject Taught	Design & Analysis of Algorithm, Data Structures, Artificial Intelligence, Soft Computing, C Programming, Web Technology, Discrete Mathematics.

Evaluation Scheme

B. TECH (CSE)
Evaluation Scheme

Session 2020-21	Third Year	SEMESTER V	ESC (3)	PCC (14)	ELC (6)	PW (1)

Sl. No.	Subject code	Subject	Periods			Evaluation Schemes				End Semester		Total	Credit	Course Type
			L	T	P	CT	TA	TOTAL	PS	TE	PE			
1		Design Thinking -II	2	1	0	30	20	50		100		150	3	ESC
2	20CS501	Database Management System	3	1	0	30	20	50		100		150	4	PCC
3	20CS502	Web Technology	3	0	0	30	20	50		100		150	3	PCC
4	20CS503	Compiler Design	3	1	0	30	20	50		100		150	4	PCC
5		Python Web development with Django	3	0	0	30	20	50		100		150	3	ELC
6		Design Pattern	3	0	0	30	20	50		100		150	3	ELC
7	P20CS501	Database Management System Lab	0	0	2				25		25	50	1	PCC
8	P20CS502	Web Technology Lab	0	0	2				25		25	50	1	PCC
9	P20CS503	Compiler Design Lab	0	0	2				25		25	50	1	PCC
10		Internship Assessment	0	0	2				50			50	1	PW
11		Constitution of India / Essence of Indian Traditional Knowledge	2	0	0	30	20	50		50		100	0	NC
12		MOOCs for Honors degree												

Syllabus

UNIT-I: Introduction of Design Pattern

Describing Design Patterns, Design Patterns in Smalltalk MVC, The Catalogue of Design Patterns, Organizing The Cato log, How Design Patterns solve, Design Problems, How to Select a Design pattern, How to Use a Design Pattern. Principle of least knowledge.

Syllabus

UNIT-II: Creational Design Pattern

A Case Study: Designing a Document Editor

Creational Patterns: Abstract Factory, Builder , Factory Method, Prototype , Singleton Pattern,

Syllabus

UNIT-III: Structural Design Pattern

Structural Pattern Part-I, Adapter, Bridge, Composite.

Structural Pattern Part-II, Decorator, Facade, Flyweight, Proxy.

Syllabus

UNIT-IV: Behavioral Design Patterns Part: I

Behavioral Patterns Part: I, Chain of Responsibility, Command, Interpreter, Iterator Pattern.

Behavioral Patterns Part: II, Mediator, Memento, Observer, Patterns.

Syllabus

UNIT-V: Behavioral Design Patterns Part: II

Behavioral Patterns Part: III, State, Strategy, Template Method, Visitor, What to Expect from Design Patterns.

Branch Wise Application

1. Real time web analytics
2. Digital Advertising
3. E-Commerce
4. Publishing
5. Massively Multiplayer Online Games
6. Backend Services and Messaging
7. Project Management & Collaboration
8. Real time Monitoring Services
9. Live Charting and Graphing
10. Group and Private Chat

Course Objective

In this semester, the students will

Study how to shows relationships and interactions between classes or objects..

Study to speed up the development process by providing well-tested, proven development/design paradigms.

Select a specific design pattern for the solution of a given design problem.

Create a catalogue entry for a simple design pattern whose purpose and application is understood.

Course Outcomes (COs)

At the end of course, the student will be able to:

CO1 : Construct a design consisting of collection of modules.

CO2 : Exploit well known design pattern such as Factory, visitor etc.

CO3 : Distinguish between different categories of design patterns.

CO4 : Ability to common design pattern for incremental development.

CO5 : Identify appropriate design pattern for a given problem and design the software using pattern oriented architecture.

Program Outcomes (POs)

Engineering Graduates will be able to:

PO1 : Engineering Knowledge

PO2 : Problem Analysis

PO3 : Design/Development of solutions

PO4 : Conduct Investigations of complex problems

PO5 : Modern tool usage

PO6 : The engineer and society

Program Outcomes (POs)

Engineering Graduates will be able to:

PO7 : Environment and sustainability

PO8 : Ethics

PO9 : Individual and teamwork

PO10 : Communication

PO11 : Project management and finance

PO12 : Life-long learning

COs - POs Mapping

CO.K	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	2	2	2	3	3	-	-	-	-	-	-	-
CO2	3	2	3	2	3	-	-	-	-	-	-	-
CO3	3	2	3	2	3	-	-	-	-	-	-	-
CO4	3	2	3	2	3	-	-	-	-	-	-	-
CO5	3	2	3	3	3	-	-	-	-	-	-	-
AVG	2.8	2.0	2.8	2.4	3.0	-	-	-	-	-	-	-

Program Specific Outcomes(PSOs)

S. No.	Program Specific Outcomes (PSO)	PSO Description
1	PSO1	Understand to shows relationships and interactions between classes or objects of a pattern.
2	PSO2	Study to speed up the development process by providing well-tested, proven development
3	PSO3	Select a specific design pattern for the solution of a given design problem
4	PSO4	Create a catalogue entry for a simple design pattern whose purpose and application is understood.

COs - PSOs Mapping

CO.K	PSO1	PSO2	PSO3	PSO4
CO1	3	-	-	-
CO2	3	3	-	-
CO3	3	3	-	-
CO4	3	3	-	-
CO5	3	3	-	-

Program Educational Objectives (PEOs)

Program Educational Objectives (PEOs)	PEOs Description
PEOs	To have an excellent scientific and engineering breadth so as to comprehend, analyze, design and provide sustainable solutions for real-life problems using state-of-the-art technologies.
PEOs	To have a successful career in industries, to pursue higher studies or to support entrepreneurial endeavors and to face the global challenges.
PEOs	To have an effective communication skills, professional attitude, ethical values and a desire to learn specific knowledge in emerging trends, technologies for research, innovation and product development and contribution to society.
PEOs	To have life-long learning for up-skilling and re-skilling for successful professional career as engineer, scientist, entrepreneur and bureaucrat for betterment of society.

Result Analysis(Department Result & Subject Result & Individual result

Name of the faculty	Subject code	Result % of clear passed
Mr. Sanjay Nayak		

Pattern of Online External Exam Question Paper (100 marks)

Printed page: ....

Subject Code: .....

Roll No: | | | | | | | | | | | | |

NOIDA INSTITUTE OF ENGINEERING AND TECHNOLOGY, GREATER NOIDA

(An Autonomous Institute Affiliated to AKTU, Lucknow)

B.Tech./MBA/MCA/M.Tech (Integrated)

(SEM:....THEORY EXAMINATION(2020-2021)

Subject .....

Time: 2 Hours

Max. Marks: 100

Pattern of Online External Exam Question Paper (100 marks)

		SECTION – A	**[30]**	**CO**
1.	**Attempt all parts- (MCQ, True False)Three Question From Each Unit**		**[15×2=30]**	
		UNIT-1		
	1-a.	**Question-**	**(2)**	
	1-b.	**Question-**	**(2)**	
	1-c.	**Question-**	**(2)**	
		UNIT-2		
	1-d.	**Question-**	**(2)**	
	1-e.	**Question-**	**(2)**	
	1-f.	**Question-**	**(2)**	
		UNIT-3		
	1-g.	**Question-**	**(2)**	
	1-h.	**Question-**	**(2)**	
	1-i.	**Question-**	**(2)**	
		UNIT-4		
	1-j.	**Question-**	**(2)**	
	1-k.	**Question-**	**(2)**	
	1-l.	**Question-**	**(2)**	
		UNIT-5		
	1-m.	**Question-**	**(2)**	
	1-n.	**Question-**	**(2)**	
	1-o.	**Question-**	**(2)**	

Pattern of Online External Exam Question Paper (100 marks)

		SECTION – B	**[20×2=40]**	**CO**
2.	**Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in Glossary)) Question from Unseen passage - Four Question From Unit-I**		**[4×2=08]**	**CO**
	Glossary- (Required words to be written)			
	2-a.	**Question-**	**(2)**	
	2-b.	**Question-**	**(2)**	
	2-c.	**Question-**	**(2)**	
	2-d.	**Question-**	**(2)**	
3.	**Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in Glossary) Question from Unseen passage - Four Question From Unit-II**		**[4×2=08]**	**CO**
		Glossary- (Required words to be written)		
	3-a.	**Question-**	**(2)**	
	3-b.	**Question-**	**(2)**	
	3-c.	**Question-**	**(2)**	
	3-d.	**Question-**	**(2)**	

Pattern of Online External Exam Question Paper (100 marks)

4.	Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in Glossary) Question from Unseen passage - Four Question From Unit-III		[4×2=08]	CO
		Glossary- (Required words to be written)		
	4-a.	Question-	(2)	
	4-b.	Question-	(2)	
	4-c.	Question-	(2)	
	4-d.	Question-	(2)	
5.	Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in Glossary) Question from Unseen passage - Four Question From Unit-IV		[4×2=08]	CO
		Glossary- (Required words to be written)		
	5-a.	Question-	(2)	
	5-b.	Question-	(2)	
	5-c.	Question-	(2)	
	5-d.	Question-	(2)	
6.	Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in Glossary) Question from Unseen passage - Four Question From Unit-V		[4×2=08]	CO
		Glossary- (Required words to be written)		
	6-a.	Question-	(2)	
	6-b.	Question-	(2)	
	6-c.	Question-	(2)	
	6-d.	Question-	(2)	

Pattern of Online External Exam Question Paper (100 marks)

SECTION – C				
7	Answer any10 out 15 of the following, Subjective Type Question, Three Question from Each Unit		[10×3=30]	CO
		UNIT-1		
	7-a.	-Question-	(3)	
	7-b.	-Question-	(3)	
	7-c.	-Question-	(3)	
		UNIT-2		
	7-d.	-Question-	(3)	
	7-e.	-Question-	(3)	
	7-f.	-Question-	(3)	
		UNIT-3		
	7-g.	-Question-	(3)	
	7-h.	-Question-	(3)	
	7-i.	-Question-	(3)	
		UNIT-4		
	7-j.	-Question-	(3)	
	7-k.	-Question-	(3)	
	7-l.	-Question-	(3)	
		UNIT-5		
	7-m	-Question-	(3)	
	7-n.	-Question-	(3)	
	7-o.	-Question-	(3)	

Prerequisite / Recap

- Student should have knowledge of object oriented analysis and design.
- Knowledge of Data structure and algorithm.
- knowledge of Programing language such as C/C++ etc.
- Good problem solving Skill .

Brief Introduction about the Subject with videos

YouTube /other Video Links

- https://youtu.be/rI4kdGLaUiQ?list=PL6n9fhu94yhUbctIoxoVTrklN3LMwTCmd
- https://youtu.be/v9ejT8FO-7I?list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc
- https://youtu.be/VGLjQuEQgkI?list=PLt4nG7RVVk1h9lxOYSOGI9pcP3I5oblbx

Unit III Content

- Structural Pattern Part-I :
- Adapter Pattern
- Bridge Pattern
- Composite Pattern
- Structural Pattern Part-II :
- Decorator Pattern
- Facade Pattern
- Flyweight Pattern
- Proxy Pattern

Unit III Objective

In Unit III, the students will be able to find

- Definitions of terms and concepts.
- The idea of a pattern.
- The origins of all design patterns.
- How Patterns Work in software design.
- Scope of development activity: applications, toolkits, frameworks.
- All Structural Pattern and their need.

Topic Objective

Topic : Structural Pattern (Adapter)

- In this topic, the students will gain , The idea of a Structural design pattern, It concerned with how classes and objects can be composed, to form larger structures. The structural design patterns simplifies the structure by identifying the relationships.

Structural design patterns

Structural design patterns:-

- Structural design patterns are concerned with how classes and objects can be composed, to form larger structures.
- The structural design patterns simplifies the structure by identifying the relationships.
- These patterns focus on, how the classes inherit from each other and how they are composed from other classes.
- structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships among entities. Examples of Structural Patterns include Adapter pattern, Bridge, Composite etc.

Structural patterns (Adapter Pattern)

Adapter Pattern:-

- An Adapter Pattern says that just "converts the interface of a class into another interface that a client wants".
- In other words, to provide the interface according to client requirement while using the services of a class with a different interface.
- The Adapter Pattern is also known as Wrapper.

Advantage of Adapter Pattern:-

- It allows two or more previously incompatible objects to interact.
- It allows reusability of existing functionality.

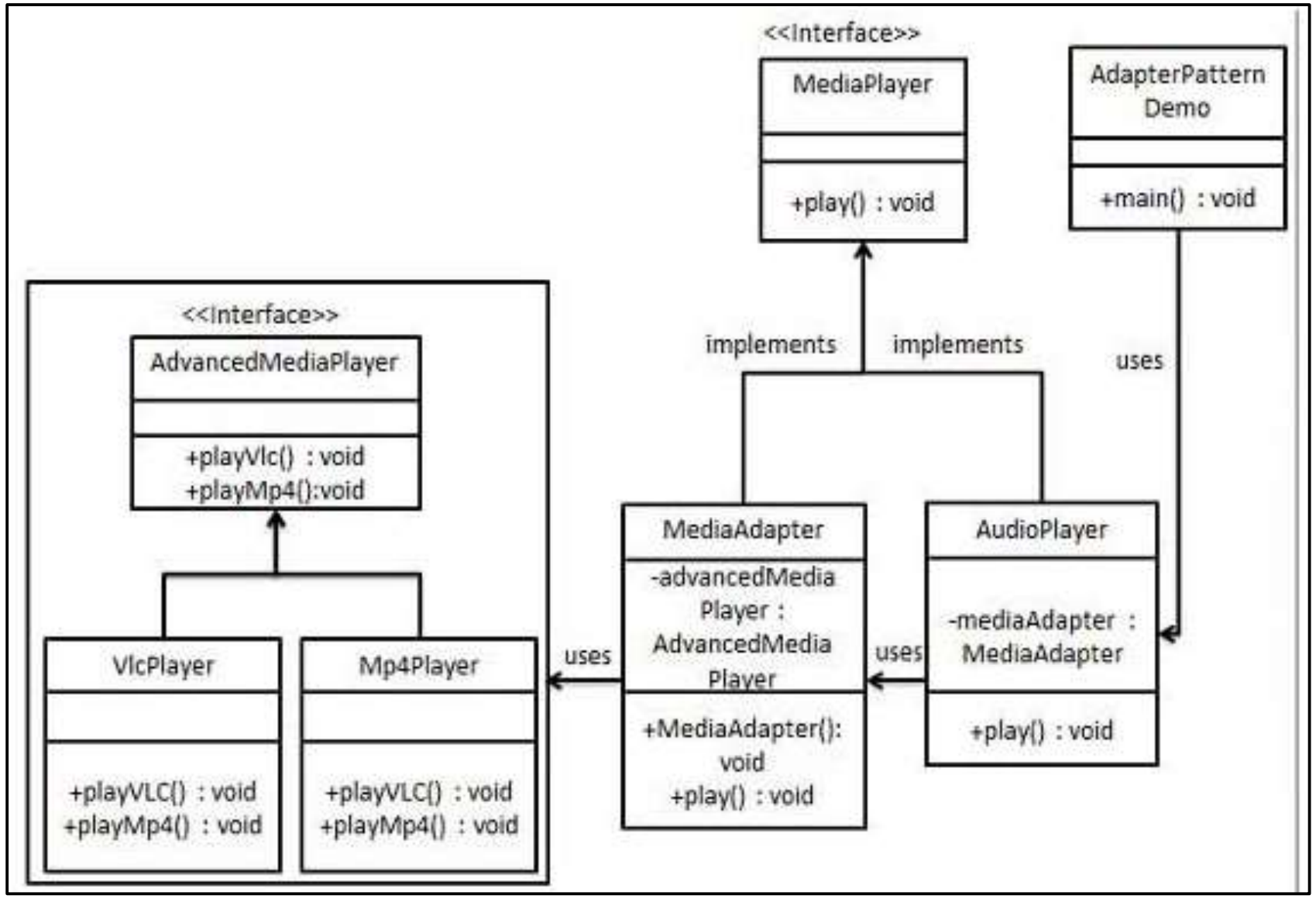
Structural patterns (Adapter Pattern)

Usage of Adapter pattern:-

It is used

- When an object needs to utilize an existing class with an incompatible interface.
- When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.
- When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.

UML\Structure for Adapter Design Pattern



Implementation of (Adapter Design Pattern)

We have a MediaPlayer interface and a concrete class AudioPlayer implementing the MediaPlayer interface. AudioPlayer can play mp3 format audio files by default.

We are having another interface AdvancedMediaPlayer and concrete classes implementing the AdvancedMediaPlayer interface. These classes can play vlc and mp4 format files.

We want to make AudioPlayer to play other formats as well. To attain this, we have created an adapter class MediaAdapter which implements the MediaPlayer interface and uses AdvancedMediaPlayer objects to play the required format.

AudioPlayer uses the adapter class MediaAdapter passing it the desired audio type without knowing the actual class which can play the desired format. AdapterPatternDemo, our demo class will use AudioPlayer class to play various formats.

Implementation of (Adapter Design Pattern)

Step 1

Create interfaces for Media Player and Advanced Media Player.

MediaPlayer.java

```
public interface MediaPlayer {
    public void play(String audioType, String fileName);
}
```

AdvancedMediaPlayer.java

```
public interface AdvancedMediaPlayer {
    public void playVlc(String fileName);
    public void playMp4(String fileName);
}
```

Implementation of (Adapter Design Pattern)

Step 2

Create concrete classes implementing the *AdvancedMediaPlayer* interface.

VlcPlayer.java

```
public class VlcPlayer implements AdvancedMediaPlayer{
    @Override
    public void playVlc(String fileName) {
        System.out.println("Playing vlc file. Name: "+ fileName);
    }

    @Override
    public void playMp4(String fileName) {
        //do nothing
    }
}
```

Implementation of (Adapter Design Pattern)

Step 2

Mp4Player.java

```java
public class Mp4Player implements AdvancedMediaPlayer{

    @Override
    public void playVlc(String fileName) {
        //do nothing
    }

    @Override
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file. Name: "+ fileName);
    }
}
```

Implementation of (Adapter Design Pattern)

Step 3

Create adapter class implementing the *MediaPlayer* interface.

MediaAdapter.java

```
public class MediaAdapter implements MediaPlayer {

   AdvancedMediaPlayer advancedMusicPlayer;

   public MediaAdapter(String audioType){

      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();

      }else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }
```

Implementation of (Adapter Design Pattern)

Step 3

```
    @Override
    public void play(String audioType, String fileName) {

        if(audioType.equalsIgnoreCase("vlc")){
            advancedMusicPlayer.playVlc(fileName);
        }
        else if(audioType.equalsIgnoreCase("mp4")){
            advancedMusicPlayer.playMp4(fileName);
        }
    }
}
```

Implementation of (Adapter Design Pattern)

Step 4

Create concrete class implementing the *MediaPlayer* interface.

AudioPlayer.java

```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;

   @Override
   public void play(String audioType, String fileName) {

      //inbuilt support to play mp3 music files
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: " + fileName);
      }
```

Implementation of (Adapter Design Pattern)

Step 4

```
        //mediaAdapter is providing support to play other file formats
        else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
            mediaAdapter = new MediaAdapter(audioType);
            mediaAdapter.play(audioType, fileName);
        }

        else{
            System.out.println("Invalid media. " + audioType + " format not supported");
        }
    }
}
```

Implementation of (Adapter Design Pattern)

Step 5

Use the AudioPlayer to play different types of audio formats.

AdapterPatternDemo.java

```
public class AdapterPatternDemo {
   public static void main(String[] args) {
      AudioPlayer audioPlayer = new AudioPlayer();

      audioPlayer.play("mp3", "beyond the horizon.mp3");
      audioPlayer.play("mp4", "alone.mp4");
      audioPlayer.play("vlc", "far far away.vlc");
      audioPlayer.play("avi", "mind me.avi");
   }
}
```

Implementation of (Adapter Design Pattern)

Step 6

Verify the output.

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

Topic Objective

Topic : Structural Pattern (Bridge Design Pattern)

- In this topic, the students will gain , The idea of a Structural design pattern, It concerned with how classes and objects can be composed, to form larger structures. "decouple the functional abstraction from the implementation so that the two can vary independently".

Structural patterns (Bridge Pattern)

Bridge Design Pattern:-

- A Bridge Pattern says that just "decouple the functional abstraction from the implementation so that the two can vary independently".

- The Bridge Pattern is also known as Handle or Body. This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes.

- Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them.

Structural patterns (Bridge Pattern)

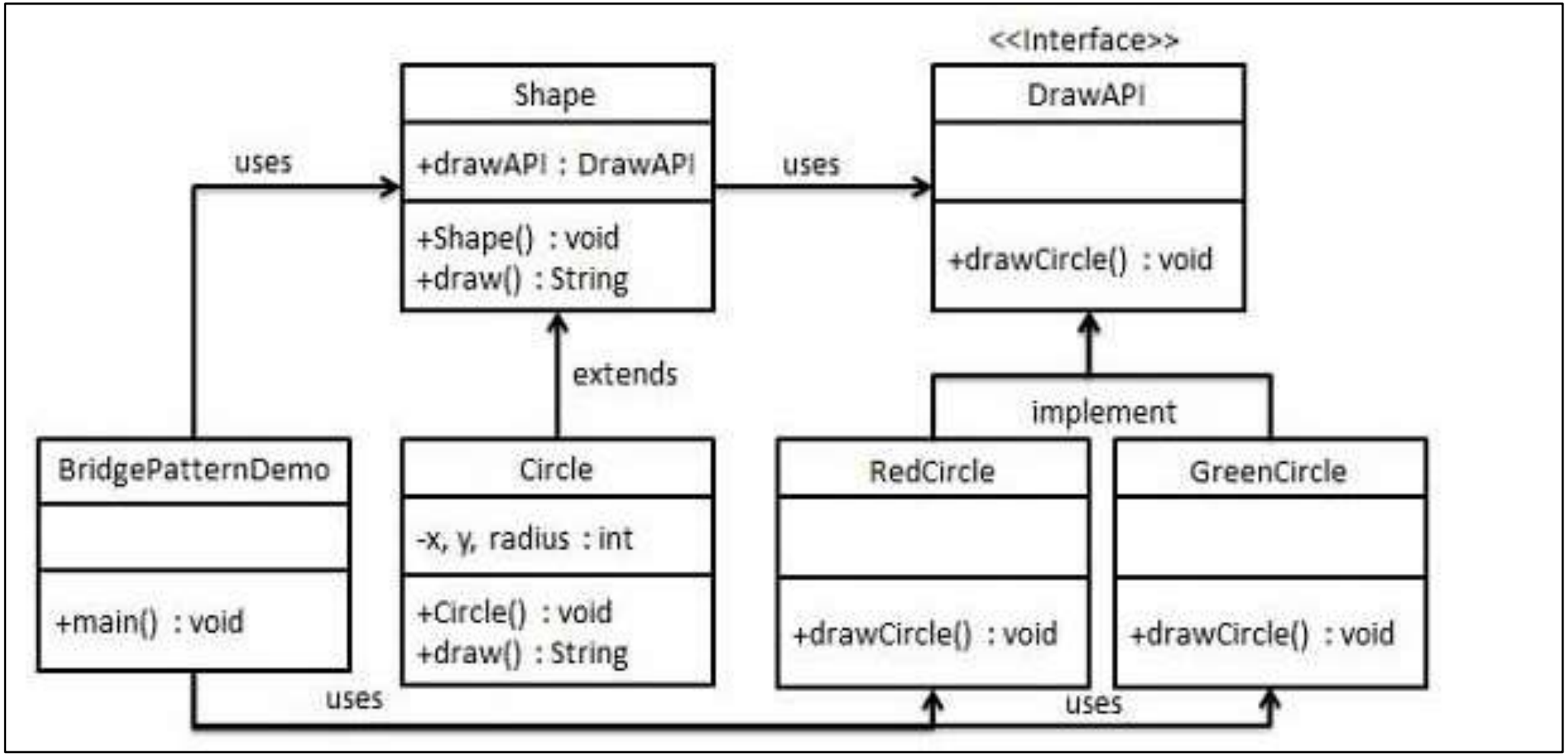
Advantage of Bridge Pattern:-

- It enables the separation of implementation from the interface.
- It improves the extensibility.
- It allows the hiding of implementation details from the client.

Usage of Bridge Pattern:-

- When you don't want a permanent binding between the functional abstraction and its implementation.
- When both the functional abstraction and its implementation need to extended using sub-classes.
- It is mostly used in those places where changes are made in the implementation does not affect the clients.

UML\Structure for Bridge Design Pattern



Implementation of (Bridge Design Pattern)

- We are demonstrating use of Bridge pattern via following example in which a circle can be drawn in different colors using same abstract class method but different bridge implementer classes.

- We have a DrawAPI interface which is acting as a bridge implementer and concrete classes RedCircle, GreenCircle implementing the DrawAPI interface. Shape is an abstract class and will use object of DrawAPI. BridgePatternDemo, our demo class will use Shape class to draw different colored circle.

Implementation of (Bridge Design Pattern)

Step 1

Create bridge implementer interface.

DrawAPI.java

```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}
```

Implementation of (Bridge Design Pattern)

Step 2

Create concrete bridge implementer classes implementing the *DrawAPI* interface.

RedCircle.java

```
public class RedCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: red, radius: '+ radius + ", x: " + x + ", " + y + "]");
    }
}
```

Implementation of (Bridge Design Pattern)

GreenCircle.java

```java
public class GreenCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: green, radius: " + radius + ", x: " + x + ", " + y + "]");
    }
}
```

Implementation of (Bridge Design Pattern)

Step 3

Create an abstract class *Shape* using the *DrawAPI* interface.

Shape.java

```
public abstract class Shape {
    protected DrawAPI drawAPI;

    protected Shape(DrawAPI drawAPI){
        this.drawAPI = drawAPI;
    }
    public abstract void draw();
}
```

Implementation of (Bridge Design Pattern)

Step 4

Create concrete class implementing the *Shape* interface.

Circle.java

```
public class Circle extends Shape {
   private int x, y, radius;

   public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }

   public void draw() {
      drawAPI.drawCircle(radius,x,y);
   }
}
```

Implementation of (Bridge Design Pattern)

Step 5

Use the *Shape* and *DrawAPI* classes to draw different colored circles.

BridgePatternDemo.java

```
public class BridgePatternDemo {
   public static void main(String[] args) {
      Shape redCircle = new Circle(100,100, 10, new RedCircle());
      Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

      redCircle.draw();
      greenCircle.draw();
   }
}
```

Implementation of (Bridge Design Pattern)

Step 6

Verify the output.

```
Drawing Circle[ color: red, radius: 10, x: 100, 100]
Drawing Circle[ color: green, radius: 10, x: 100, 100]
```

Topic Objective

Topic : Structural Pattern (Composite Design Pattern)

- In this topic, the students will gain , The idea of a Structural design pattern, A Composite Pattern says that just "allow clients to operate in generic manner on objects that may or may not represent a hierarchy of objects".

Structural patterns (Composite Pattern)

Composite Design Pattern:-

- A Composite Pattern says that just "allow clients to operate in generic manner on objects that may or may not represent a hierarchy of objects".

- 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 hierarchy. 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 that contains group of its own objects. This class provides ways to modify its group of same objects.

Structural patterns (Composite Design Pattern)

Advantage of Composite Design Pattern:-

- It defines class hierarchies that contain primitive and complex objects.
- It makes easier to you to add new kinds of components.
- It provides flexibility of structure with manageable class or interface.

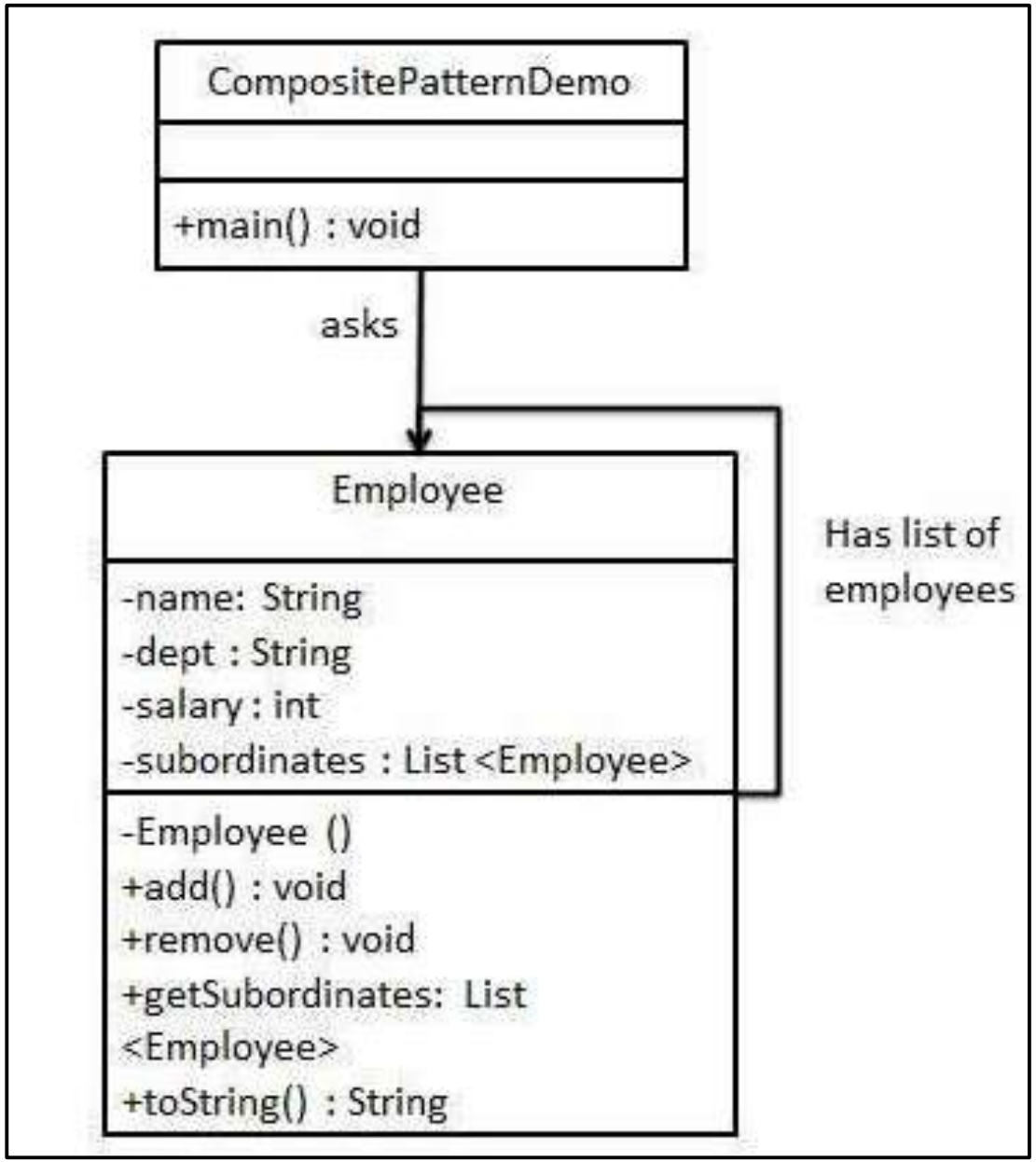
Usage of Composite Design Pattern:-

It is used:

- When you want to represent a full or partial hierarchy of objects.

- When the responsibilities are needed to be added dynamically to the individual objects without affecting other objects. Where the responsibility of object may vary from time to time.

UML\Structure for Composite Design Pattern



Implementation of (Composite Design Pattern)

- This pattern creates a class that 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 we will show employees hierarchy of an organization.
- We have a class Employee which acts as composite pattern actor class. CompositePatternDemo, our demo class will use Employee class to add department level hierarchy and print all employees.

Implementation of (Composite Design Pattern)

Step 1

Create *Employee* class having list of *Employee* objects.

Employee.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>();
   }
```

Implementation of (Composite Design Pattern)

Step 1

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

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

    public List<Employee> getSubordinates(){
      return subordinates;
    }

    public String toString(){
        return ("Employee :[ Name : " + name + ", dept : " + dept + ", salary :" + salary+" ]");
    }
}
```

Implementation of (Composite Design Pattern)

Step 2

Use the *Employee* class to create and print employee hierarchy.

CompositePatternDemo.java

```
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);
```

Implementation of (Composite Design Pattern)

```
        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);
            }
        }
    }
```

Implementation of (Composite Design Pattern)

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 ]
```

Topic Objective

Topic : Structural Pattern II (Decorator Pattern)

- In this topic, the students will gain , The idea of a Structural design pattern, A Decorator Pattern says that just "attach a flexible additional responsibilities to an object dynamically".

Structural patterns II (Decorator Pattern)

Decorator Design Pattern:-

- A Decorator Pattern says that just "attach a flexible additional responsibilities to an object dynamically".
- In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime. The Decorator Pattern is also known as Wrapper.
- Decorator pattern allows a user to add new functionality to an existing object without altering its structure.
- This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

Structural patterns (Decorator Design Pattern)

Advantage of Decorator Design Pattern:-

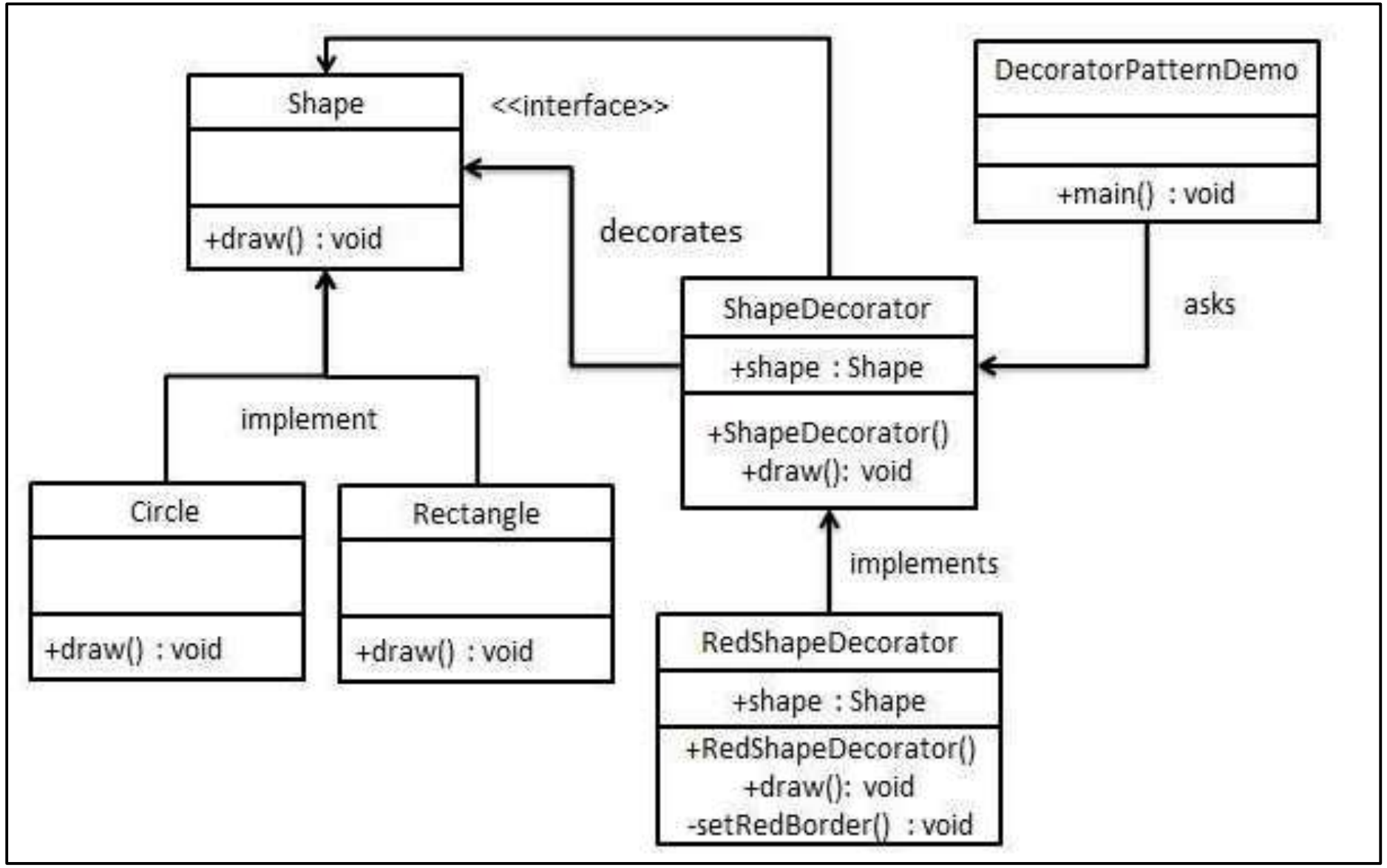
- It provides greater flexibility than static inheritance.
- It enhances the extensibility of the object, because changes are made by coding new classes. It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.

Usage of Decorator Design Pattern:-

It is used:

- When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
- When you want to add responsibilities to an object that you may want to change in future. Extending functionality by sub-classing is no longer practical.

UML\Structure for Decorator Design Pattern



Implementation of (Decorator Design Pattern)

- We are demonstrating the use of decorator pattern via following example in which we will decorate a shape with some color without alter shape class.

- We're going to create a Shape interface and concrete classes implementing the Shape interface. We will then create an abstract decorator class ShapeDecorator implementing the Shape interface and having Shape object as its instance variable.

- RedShapeDecorator is concrete class implementing ShapeDecorator.

- DecoratorPatternDemo, our demo class will use RedShapeDecorator to decorate Shape objects.

Implementation of (Decorator Design Pattern)

Step 1

Create an interface.

Shape.java

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

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

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

Circle.java

```
public class Circle implements Shape {

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

Implementation of (Decorator Design Pattern)

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();
   }
}
```

Implementation of (Decorator Design Pattern)

Step 4

Create concrete decorator class extending the *ShapeDecorator* class.

RedShapeDecorator.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");
    }
}
```

Implementation of (Decorator Design Pattern)

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();
   }
}
```

Implementation of (Decorator Design Pattern)

Step 6

Verify the output.

```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

Topic Objective

Topic : Structural Pattern II (Facade Pattern)

- In this topic, the students will gain , The idea of a Structural design pattern, A Facade Pattern says that just "just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client".

Structural patterns II (Facade Pattern)

Facade Design Pattern:-

- A Facade Pattern says that just "just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client".

- In other words, Facade Pattern describes a higher-level interface that makes the sub-system easier to use.

- Practically, every Abstract Factory is a type of Facade.

- Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system.

Structural patterns (Facade Design Pattern)

Advantage of Facade Design Pattern:-

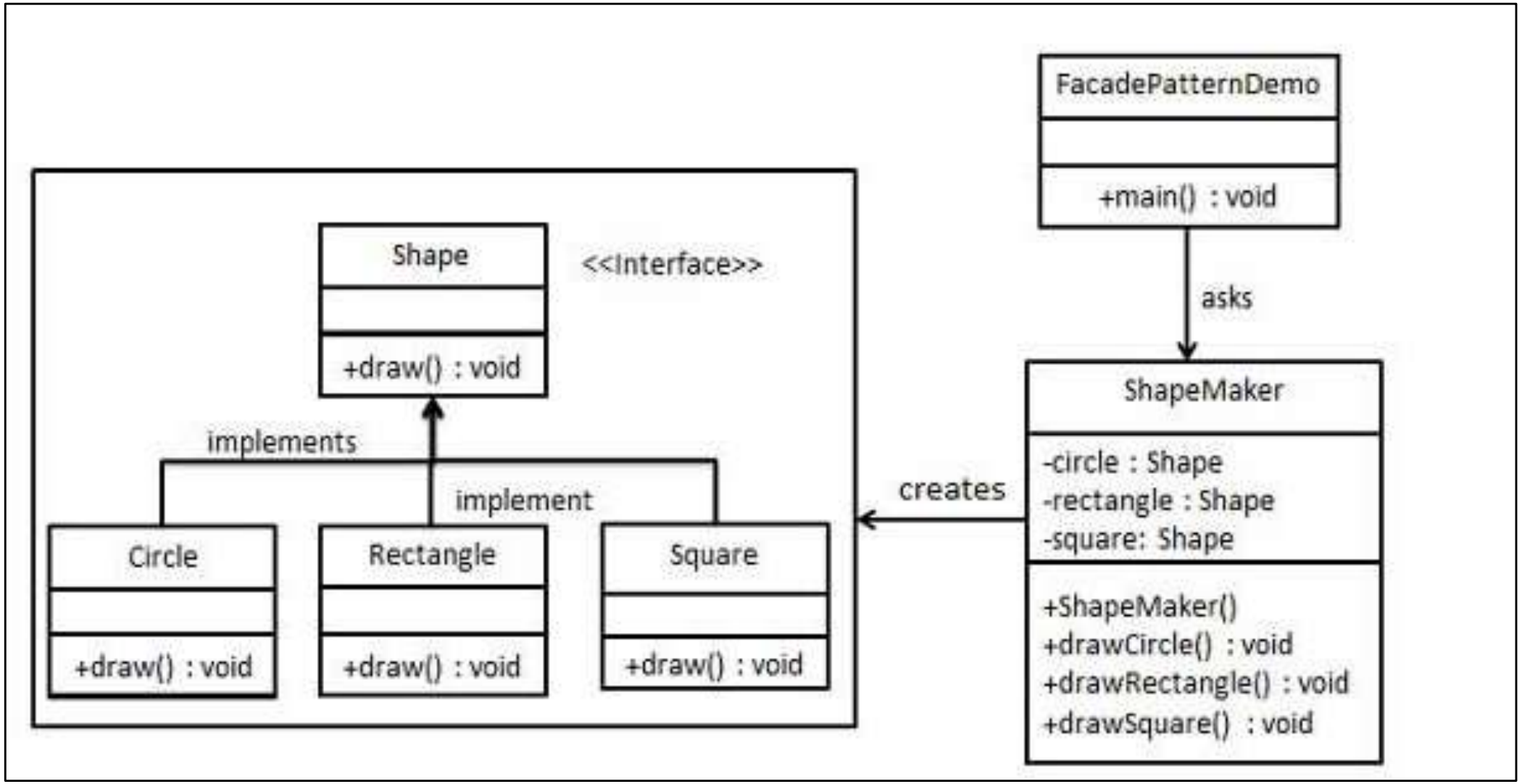
- It shields the clients from the complexities of the sub-system components.
- It promotes loose coupling between subsystems and its clients.

Usage of Facade Design Pattern:-

It is used:

- When you want to provide simple interface to a complex sub-system.
- When several dependencies exist between clients and the implementation classes of an abstraction.

UML\Structure for Facade Design Pattern



Implementation of (Facade Design Pattern)

- We are going to create a Shape interface and concrete classes implementing the Shape interface. A facade class ShapeMaker is defined as a next step.

- ShapeMaker class uses the concrete classes to delegate user calls to these classes. FacadePatternDemo, our demo class, will use ShapeMaker class to show the results.

- This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

Implementation of (Facade Design Pattern)

Step 1

Create an interface.

Shape.java

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

Implementation of (Facade Design Pattern)

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

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

Square.java

```
public class Square implements Shape {

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

Circle.java

```
public class Circle implements Shape {

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

Implementation of (Facade Design Pattern)

Step 3

Create a facade class.

ShapeMaker.java

```
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }

   public void drawCircle(){
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
}
```

Implementation of (Facade Design Pattern)

Step 4

Use the facade to draw various types of shapes.

FacadePatternDemo.java

```
public class FacadePatternDemo {
    public static void main(String[] args) {
        ShapeMaker shapeMaker = new ShapeMaker();

        shapeMaker.drawCircle();
        shapeMaker.drawRectangle();
        shapeMaker.drawSquare();
    }
}
```

Step 5

Verify the output.

```
Circle::draw()
Rectangle::draw()
Square::draw()
```

Topic Objective

Topic : Structural Pattern II (Flyweight Pattern)

- In this topic, the students will gain , The idea of a Structural design pattern, A Flyweight Pattern says that just "to reuse already existing similar kind of objects by storing them and create new object when no matching object is found".

Structural patterns II (Flyweight Pattern)

Flyweight Design Pattern:-

- A Flyweight Pattern says that just "to reuse already existing similar kind of objects by storing them and create new object when no matching object is found".

- Flyweight pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance. This type of design pattern comes under structural pattern as this pattern provides ways to decrease object count thus improving the object structure of application.

- Only 5 colors are available so color property is used to check already existing Circle objects.

Structural patterns (Flyweight Design Pattern)

Advantage of Flyweight Design Pattern:-

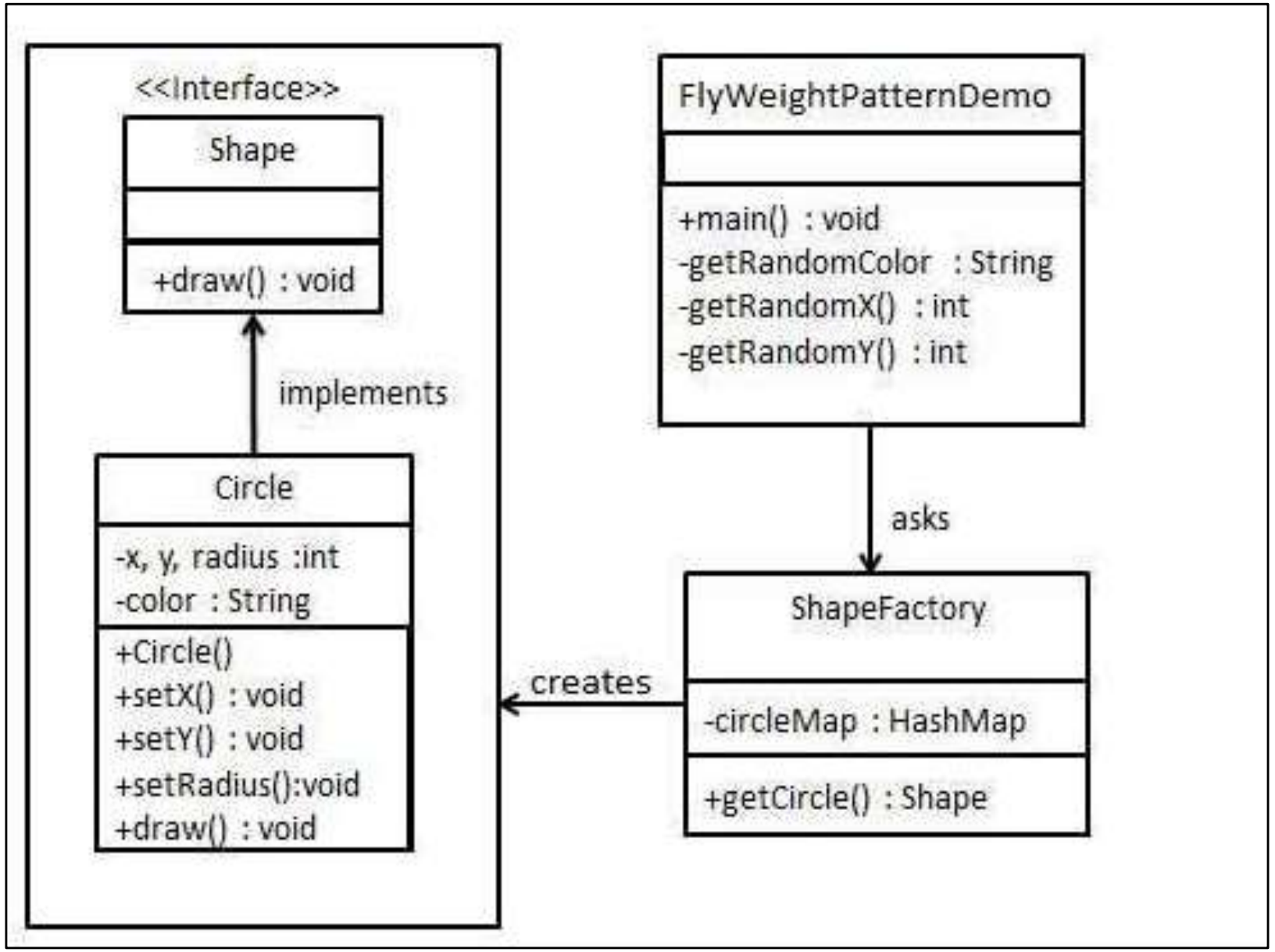
- It reduces the number of objects.
- It reduces the amount of memory and storage devices required if the objects are persisted

Usage of Flyweight Design Pattern:-

It is used:

- When an application uses number of objects
- When the storage cost is high because of the quantity of objects.
- When the application does not depend on object identity.

UML\Structure for Flyweight Design Pattern



Implementation of (Flyweight Design Pattern)

- We are going to create a Shape interface and concrete class Circle implementing the Shape interface. A factory class ShapeFactory is defined as a next step.

- ShapeFactory has a HashMap of Circle having key as color of the Circle object. Whenever a request comes to create a circle of particular color to ShapeFactory, it checks the circle object in its HashMap, if object of Circle found, that object is returned otherwise a new object is created, stored in hashmap for future use, and returned to client.

- FlyWeightPatternDemo, our demo class, will use ShapeFactory to get a Shape object. It will pass information (red / green / blue/ black / white) to ShapeFactory to get the circle of desired color it needs.

Implementation of (Flyweight Design Pattern)

Step 1

Create an interface.

Shape.java

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

Step 2

Create concrete class implementing the same interface.

Circle.java

```
public class Circle implements Shape {
    private String color;
    private int x;
    private int y;
    private int radius;

    public Circle(String color){
        this.color = color;
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }
```

Implementation of (Flyweight Design Pattern)

Step 2 cont…….

```
    public void setRadius(int radius) {
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Circle: Draw() [Color : " + color + ", x : " + x + ", y :" + y + ", radius :" + radius);
    }
}
```

Implementation of (Flyweight Design Pattern)

Step 3

Create a factory to generate object of concrete class based on given information.

ShapeFactory.java

```java
import java.util.HashMap;

public class ShapeFactory {

   // Uncomment the compiler directive line and
   // javac *.java will compile properly.
   // @SuppressWarnings("unchecked")
   private static final HashMap circleMap = new HashMap();

   public static Shape getCircle(String color) {
      Circle circle = (Circle)circleMap.get(color);

      if(circle == null) {
         circle = new Circle(color);
         circleMap.put(color, circle);
         System.out.println("Creating circle of color : " + color);
      }
      return circle;
   }
}
```

Implementation of (Flyweight Design Pattern)

Step 4

Use the factory to get object of concrete class by passing an information such as color.

FlyweightPatternDemo.java

```
public class FlyweightPatternDemo {
   private static final String colors[] = { "Red", "Green", "Blue", "White",
   public static void main(String[] args) {                        , "Black" };

      for(int i=0; i < 20; ++i) {
         Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());
         circle.setX(getRandomX());
         circle.setY(getRandomY());
         circle.setRadius(100);
         circle.draw();
      }
   }
   private static String getRandomColor() {
      return colors[(int)(Math.random()*colors.length)];
   }
   private static int getRandomX() {
      return (int)(Math.random()*100 );
   }
   private static int getRandomY() {
      return (int)(Math.random()*100);
   }
}
```

Implementation of (Flyweight Design Pattern)

Step 5

Verify the output.

```
Creating circle of color : Black
Circle: Draw() [Color : Black, x : 36, y :71, radius :100
Creating circle of color : Green
Circle: Draw() [Color : Green, x : 27, y :27, radius :100
Creating circle of color : White
Circle: Draw() [Color : White, x : 64, y :10, radius :100
Creating circle of color : Red
Circle: Draw() [Color : Red, x : 15, y :44, radius :100
Circle: Draw() [Color : Green, x : 19, y :10, radius :100
Circle: Draw() [Color : Green, x : 94, y :32, radius :100
Circle: Draw() [Color : White, x : 69, y :98, radius :100
Creating circle of color : Blue
Circle: Draw() [Color : Blue, x : 13, y :4, radius :100
Circle: Draw() [Color : Green, x : 21, y :21, radius :100
Circle: Draw() [Color : Blue, x : 55, y :86, radius :100
Circle: Draw() [Color : White, x : 90, y :70, radius :100
Circle: Draw() [Color : Green, x : 78, y :3, radius :100
Circle: Draw() [Color : Green, x : 64, y :89, radius :100
Circle: Draw() [Color : Blue, x : 3, y :91, radius :100
Circle: Draw() [Color : Blue, x : 62, y :82, radius :100
Circle: Draw() [Color : Green, x : 97, y :61, radius :100
Circle: Draw() [Color : Green, x : 86, y :12, radius :100
Circle: Draw() [Color : Green, x : 38, y :93, radius :100
Circle: Draw() [Color : Red, x : 76, y :82, radius :100
Circle: Draw() [Color : Blue, x : 95, y :82, radius :100
```

Topic Objective

Topic : Structural Pattern II (Proxy Pattern)

- In this topic, the students will gain , The idea of a Structural design pattern, According to GoF, a Proxy Pattern "provides the control for accessing the original object".

Structural patterns II (Proxy Pattern)

Proxy Design Pattern:-

- Simply, proxy means an object representing another object.According to GoF, a Proxy Pattern "provides the control for accessing the original object".
- So, we can perform many operations like hiding the information of original object, on demand loading etc.Proxy pattern is also known as Surrogate or Placeholder.
- 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.

Structural patterns (Proxy Design Pattern)

Advantage of Proxy Design Pattern:-

- It provides the protection to the original object from the outside world.

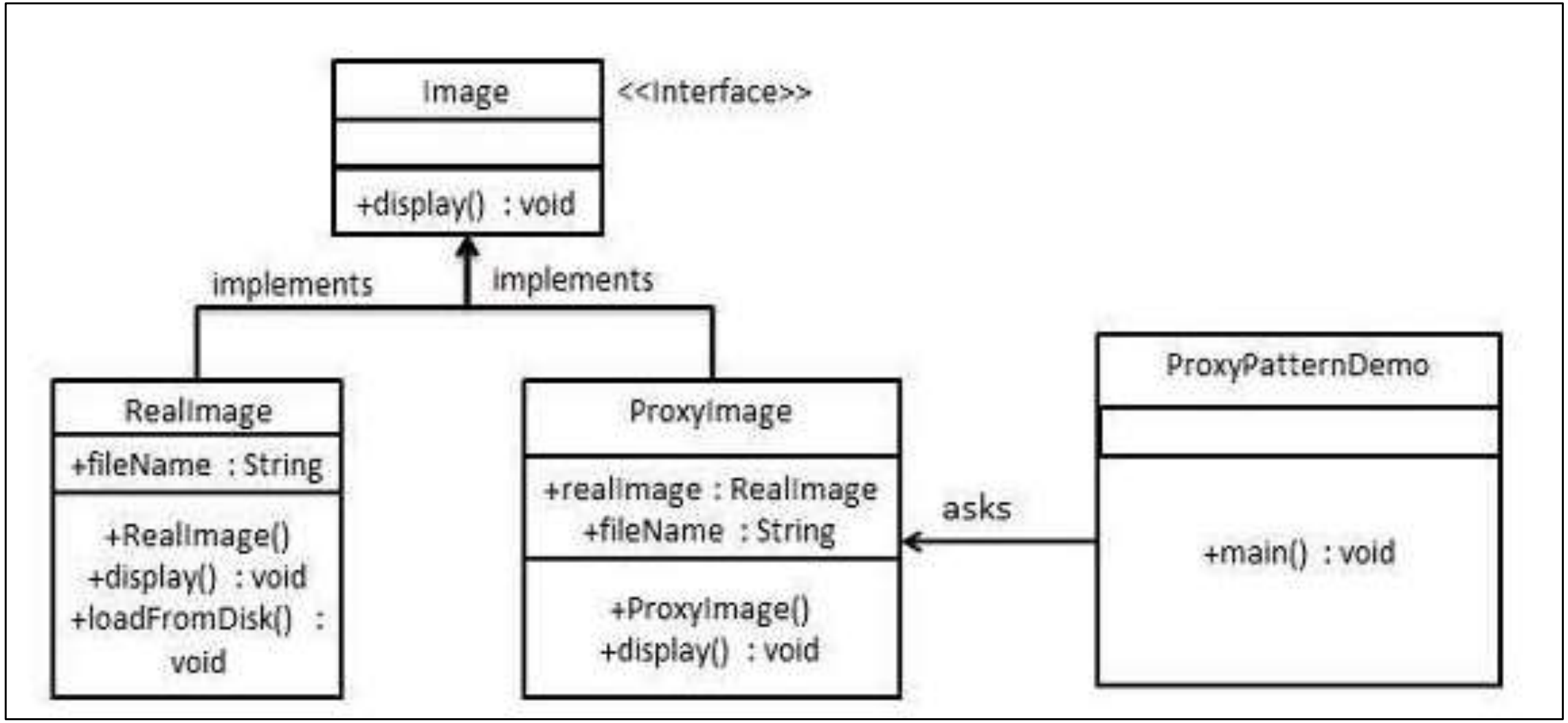
Usage of Proxy Design Pattern:-

It is used:

- It can be used in Virtual Proxy scenario---Consider a situation where there is multiple database call to extract huge size image.

- It can be used in Protective Proxy scenario---It acts as an authorization layer to verify that whether the actual user has access the appropriate content or not.

UML\Structure for Proxy Design Pattern



Implementation of (Proxy Design Pattern)

- We are going to create an Image interface and concrete classes implementing the Image interface. ProxyImage is a a proxy class to reduce memory footprint of RealImage object loading.

- ProxyPatternDemo, our demo class, will use ProxyImage to get an Image object to load and display as it needs.

- In proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern.

Implementation of (Proxy Design Pattern)

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);
    }
}
```

Implementation of (Proxy Design Pattern)

Step -2 Cont.....

ProxyImage.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();
   }
}
```

Implementation of (Proxy Design Pattern)

Step 3

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

ProxyPatternDemo.java

```
public class ProxyPatternDemo {

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

      //image will be loaded from disk
      image.display();
      System.out.println("");

      //image will not be loaded from disk
      image.display();
   }
}
```

Step 4

Verify the output.

```
Loading test_10mb.jpg
Displaying test_10mb.jpg

Displaying test_10mb.jpg
```

Daily Quiz

You want to minimize development cost by reusing methods? Which design pattern would you choose?

A. Adapter Pattern
B. Singleton Pattern
C. **Delegation pattern**
D. Immutable Pattern

Which design pattern defines one-to-many dependency among objects?

A. Singleton pattern
B. Facade Pattern
C. **Observer pattern**
D. Factory method pattern

Daily Quiz

Which design pattern suggest multiple classes through which request is passed and multiple but only relevant classes carry out operations on the request.

A. Singleton pattern
B. Chain of responsibility pattern
C. State pattern
D. Bridge pattern

Most user interface design patterns fall with in one of ____ categories of patterns.

A. 5
B. 10
C. 25
D. 100

Weekly Assignment

1. What are Design Patterns in Java? What are the types of design patterns in Java.
2. What are the Structural Patterns and What Is Façade Pattern.
3. What Is Flyweight Design Pattern.
4. Explain Facade Pattern in Java.
5. Explain the Proxy pattern.

Topic Link (YouTube & NPTEL Video Links)

YouTube /other Video Links

- https://youtu.be/rI4kdGLaUiQ?list=PL6n9fhu94yhUbctIoxoVTrklN3LMwTCmd
- https://youtu.be/v9ejT8FO-7I?list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc
- https://youtu.be/VGLjQuEQgkI?list=PLt4nG7RVVk1h9lxOYSOGI9pcP3I5oblbx

MCQ (End of Unit)

1. 1**Design patterns can be classified in _____ categories.**

- ❑ 2
- ❑ 3
- ❑ 4

2. Which design patterns are specifically concerned with communication between objects?

- ❑ Creational Patterns
- ❑ Structural Patterns
- ❑ Behavioral Patterns
- ❑ J2EE Patterns

MCQ (End of Unit)

3. Which design pattern provides a single class which provides simplified methods required by client and delegates call to those methods?

- ❑ Adapter pattern
- ❑ Builder pattern
- ❑ **Facade pattern**
- ❑ Prototype pattern

4. Which design pattern suggests multiple classes through which request is passed and multiple but only relevant classes carry out operations on the request?

- ❑ Singleton pattern
- ❑ **Chain of responsibility pattern**
- ❑ State pattern
- ❑ Bridge pattern

Glossary Questions

Top 10 design pattern interview questions

1. What are design patterns?
2. How are design patterns categorized?
3. Explain the benefits of design patterns in Java.
4. Describe the factory pattern.
5. Differentiate ordinary and abstract factory design patterns.
6. What do you think are the advantages of builder design patterns?
7. How is the bridge pattern different from the adapter pattern?
8. What is a command pattern?
9. Describe the singleton pattern along with its advantages and disadvantages.
10. What are anti patterns?

Expected Questions for University Exam

- What are design patterns?
- How are design patterns categorized?
- Explain the benefits of design patterns in Java.
- Describe the factory pattern.
- Differentiate ordinary and abstract factory design patterns.
- What do you think are the advantages of builder design patterns?
- How is the bridge pattern different from the adapter pattern?
- What is a command pattern?
- Describe the singleton pattern along with its advantages and disadvantages.
- What are anti patterns?

Recap of Unit

- **Till Now we understand,** The idea of a Structural design pattern, It concerned with how classes and objects can be composed, to form larger structures. The structural design patterns simplifies the structure by identifying the relationships.
- A Facade Pattern says that just "just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client".
- You also learn, a Proxy Pattern "provides the control for accessing the original object".
- A Flyweight Pattern says that just "to reuse already existing similar kind of objects by storing them and create new object when no matching object is found".