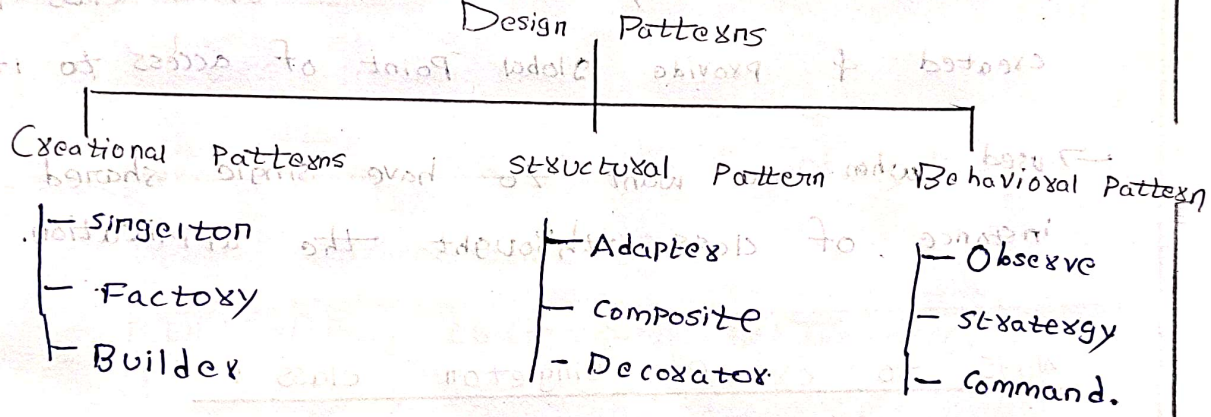
17-6-25. 10:00 AM

DESIGN PATTERNS :-

→ A design pattern in programming is a reusable solution to a common problem that occur during software development or design

→ These patterns provide structured approach to solve design & development issues.

→ used to create more maintainable, flexible & scalable code.



[1] Creational pattern :-

→ These pattern focus on object create mechanism.

→ Provide ways to create object in a manner that is flexible and maintanable.

→ Based on application specific requirements we choose a object creation mechanism.

[2] Structural pattern :-

→ These patterns focus on class & objects to create large structures while keeping them flexible and efficient.

→ Helps us to ensure that classes and objects can work together efficiently to acheive a goal.

[3] Behavioral Pattern :-

→ used for interaction & communication between object and classes.

→ Provide solution for efficiently manage flow of control, behaviour between object.

* Creational Pattern :-

[1] Singelton Pattern :-

→ Ensures that only one instance of a class is created & provide global point of access to it.

→ used when we want to have single shared instance of class thought the application.

Ways to create Singleton class :-

[1] Eager intialisation

[2] Lazy intialisation

[3] Synchronized block.

[4] Double check

[5] Bill Pugh solution.

[6] Enum

Main class :-

```
Public class main
{
    Public static void main (String args[])
    {
        DBConnection obj = DB Connection.getInstance();
    }
}
```

Eager Intialisation :-

```
Public class DBConnection
{
    private static DB Connection canobject = new DB Connection();

    private DBConnection () { }

    private static DBConnection getInstance()
    {
        return canObject;
    }
}
```

Lazy intialization :-

```
Public class DB Connection
{
    private static DBConnection conObject;

    private DB Connection () { }.

    Public static DB Connection getInstance()
    {
        if (conObject == null)
            conObject = new DB Connection();

        return conObject;
    }
}
```

Synchronization BLOCK :-

```
Public class DB Connection
{
    private static DB Connection conObject

    private DBConnection () { };

    Synchronized public static DB connection getInstance()
    {
        if (conObject == null)
            ConObject = new DB Connection()

        return conObject;
    }
}
```

Double check locking system:-

```
Public class DB Connection
{
    Private static volatile DB Connection con=new DB Connection
    Private DBConnection c){}.
    Public static DB Connection getInstance()
    {
        Synchronized (DB Connection.class)
        {
            if (con==null)
            {
                con = new DBConnection();
            }
        }
        return con.
    }
```

Big Pugh singleton:- (uses eager intialisation)

```
Public class DBC
{
    Private DBC c){}
    Private static class DBHelper
    {
        private static final DB = INSTANCE_OBJECT=new DB
    }
    Public static DB getInstance()
    {
        return DBHelper.INSTANCE_OBJECT.
    }
}
```

ENUM:-

```
Public enum DBC
{
    INSTANCE
        Public static doSomething c){
        }
}
```

Builder Pattern :-

→ Builder Pattern is a creational design pattern that allows you to construct complex objects step-by-step Seperating the construction logic from the representation.

→ useful when objects have many optional fields

→ Avoid telescoping Constructors.

Without Builder pattern

Person.Java :-

```
Public class Person
{
    private String name;
    private int age;

    Public Person (String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    Public void display ()
    {
        S.O.P ("Name" + name + ..);
    }
}
```

Test.Java:

```
Public class Test
{
    Public static void main (String args[])
    {
        Person P = new person ("virat", 35);
        P.display ();
    }
}
```

With builder Patton

person.Java.

```
Public class Person
{
    private String name;
    private int age;
    private Person (Builder builder)
    {
        this.name = buider.name;
        this.age = builder.age;
    }
    Public void display () {
        S.O.P (name + ...);
    }
    Public static class Builder{
        private String name;
        private int age;
        Public Builder setName (String
        {
            this.name = name;
            return this;
        }
        Public Buider setAge (int age)
        {
            this.age = age;
            return this;
        }
        Public Person build () {
            return new Person (this
    } }
```

Inside main

```
Person P = new person.Builder()
            .setName("Kohli")
            .setAge(35)
            .build();
P.display();
```

Factory Pattern :-

→ A creational design pattern that provides an interface to create objects in a superclass but allows subclass to alter the type of objects that will be created.

→ Avoid using new directly in the client.

→ We will use factory pattern when we need to creat object based on input or condition.

Code :-

```
Public interface shape {
    void draw();
}

Public class circle implements shape {
    Public void draw () {
        system.out.println ("Drawing a circle");
    }
}

Public class Square implements shape {
    Public void draw () {
        system.out.println ("Drawing a square");
    }
}

Public class shape Factory
{
    Public Shape getShape (String type)
    {
        if (type.equalsIgnoreCase ("circle")) {
            return new Circle();
        } else if (type.equalsIgnoreCase ("square")) {
            return new Square();
        }
        return null;
    }
}
```

Main.java

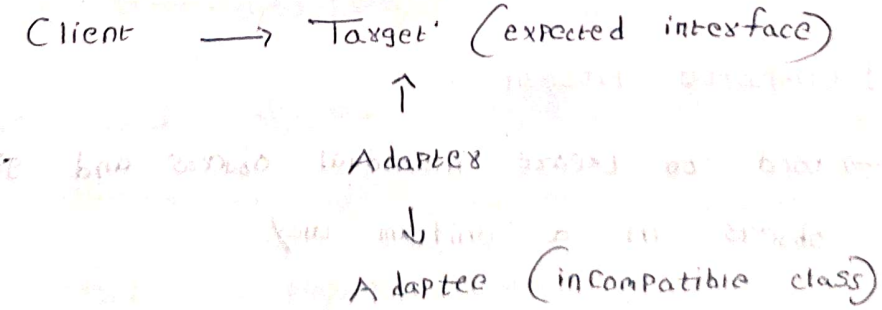
```
Public class Main
{
    Public static void main (String[] args)
    {
        ShapeFactory factory = new ShapeFactory();
        Shape s1 = factory.getShape ("circle");
        s1.draw();
        Shape s2 = factory.getShape ("square");
        s2.draw();
    }
}
```

Structural Pattern :-

Adapter:

→ Adapter Pattern is a structural Pattern ~~that~~ allows object with incompatible interfaces to work ~~together~~ by converting one interface into another.

→ Bridge ~~the~~ gap between two incompatible interfaces.



Code:

```
class OldPrinter {
    public void printOld() {
        System.out.println("old printer");
    }
}
```

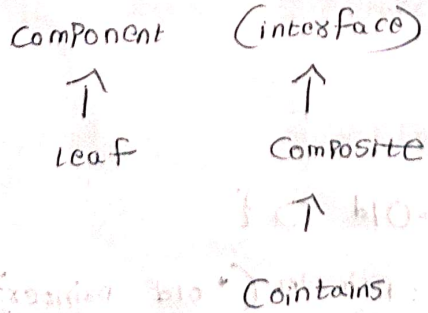
```
interface PrinterInterface {
    void print();
}

class PrinterAdapter implements PrinterInterface {
    private OldPrinter oldPrinter;
    public PrinterAdapter(OldPrinter oldPrinter) {
        this.oldPrinter = oldPrinter;
    }
    public void print() {
        oldPrinter.printOld();
    }
}

public class Test {
    public static void main(String[] args) {
        OldPrinter old = new OldPrinter();
        PrinterInterface adapter = new PrinterAdapter(old);
        adapter.print();
    }
}
```

[2] Composite Pattern :-

→ used to create individual objects and groups of objects in a uniform way.



```
public interface Employee {
    void showDetails();
}

public class Developer implements Employee {
    private String name;
    public Developer(String name) {
        this.name = name;
    }
    public void showDetails() {
        System.out.println("Developer :" + name);
    }
}

public class Manager implements Employee {
    private String name;
    private List<Employee> team = new ArrayList<>();
    public Manager(String name) {
        this.name = name;
    }
    public void add(Employee emp) {
        team.add(emp);
    }
    public void showDetails() {
        System.out.println("Manager :" + name);
        for (Employee e : team) {
            e.showDetails();
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Developer dev1 = new Developer("Rahul");
        Developer dev2 = new Developer("Kiran");
```

```
manager manager = new Manager ("Virat");
manager.add (dev1);
manager.add (dev2);
manager.showDetails();
}
```

Behav

Behavioral Pattern :-

Observe Pattern :-

→ where an object maintains a list of dependents and notifies them automatically when its state changes.

```
Public interface observer {
        void update (String message);
}

Public class Follower ; implements observer {
        Private String name;
        Public Follower (String name) {
                this.name = name;
        }
        Public void update (String message) {
                System.out.println (name);
        }
}

Public interface Subject {
        Void addObserver (Observer o);
        Void removeObserver (Observer o);
        Void notifyObserver (String msg);
}

import java.util.*;

Public class Channel implements Subject {
        Private List <observer> observers = new ArrayList<>();
        Public void addObserver (observer o) {
                observs.add (o);
        }
}
```

```
Public class Test {
    Public static void main (String [] args) {
        channel chanel =new channelC);
        Followes f1 =new Follower ("vivat);
        channel.addObserves (f1);
        channel.Upload ("Highlights");
    }
}
```