Date _____ Day

"DESIGN PATTERN"

"A design pattern is a general solution to a comm problem in a ~~context~~ software design."

↳ These are repeatable / reusable solutions

↳ These are already tested & proven solutions use by many experienced designers.

▷ TYPES OF DESIGN PATTERN :

- Structural
- Creational
- Behaviour

① Structural Pattern :

↳ These design patterns are concerned with how classes and objects can be composed, to form large 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.

Example : Adapter Pattern, Facade Pattern

Date _____ Day _____

Behavioral Patterns :

↳ These are concerned with the interaction and responsibility of objects

↳ In these design patterns, the interaction b/w the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.

↳ Focuses on communication b/w objects

Example : Observer Pattern

3. Creational Patterns :

↳ Deals with how objects are created

↳ Increase the system's flexibility in terms of what, who, how and when the object is created.

Further classified into

↳ Class - creational Pattern

↳ Object Creational Pattern

↳ Example : Factory Pattern, Singleton

→ Patterns may consist of smaller pattern /sub-patterns

→ Class diag are used to express design design patt-

Date _____ Day _____

① Singleton Pattern :

→ Is design pattern se ek class ka ek hi object ban sakta hai.

→ Following are the steps to make singleton desi

① Define a private static attribute in the class

② Define a public static accessor function in the class.

↳ this will be used to create the object. and returns the object to outer world

③ Do "lazy initialization" (initialized the object only when it is 1st used) in acessor function

④ Define the constructor as protected

↳ So it cannot be instantiate outside the class

⑤ Clients may use accessor function to manipulate the class object.

```
public class Singleton {          // this can be any name

    private static Singleton obj;  // a unique variable that holds one and only one instance of class.
    // any other attributes
    private Singleton() {}
```

as it is private so no one can instantiate outside class

Date Day.

this is static function therefore can be accessed anywhere outside class without the object creation.

```
public static Singleton getInstance()
{
    if (obj == null) → agr pehle se object create nhi hua tha to hi new object create hoga or else
        obj = new Singleton();
    }
    return instan obj ;  → pura hi return hgayega.
}

// any other useful methods
// ye ek tsha ki normal class hi hai bus object
// instantiation par restriction lgdi hai

}
```

! But there is a problem.

↳ If two threads runs in parallel and simultaneously calls.

Singleton myObj = Singleton.getInstance();

Then two instance will be created.

↳ For this we need to add synchronize key in getInstance so that if one thread executes getInstance other should wait.

Public static synchronize Singleton getInstance(){ }

Date _____ Day _____

```
Public SingletonDemo {
 public static void main (String[] args) {
    // Instantiation using 2 threads
    Thread t1 = new Thread (new Runnable () {

        public void run ()
        {
            Singleton myObj = Singleton.getInstance();
        }

    });

    Thread t2 = new ~~as it is above~~ ();

    t1.start();
    t2.start();
  }
}
```

Date _____ Day _____

② Adapter Design Pattern :

→ This design pattern works as an interface blw two incompatible interfaces.

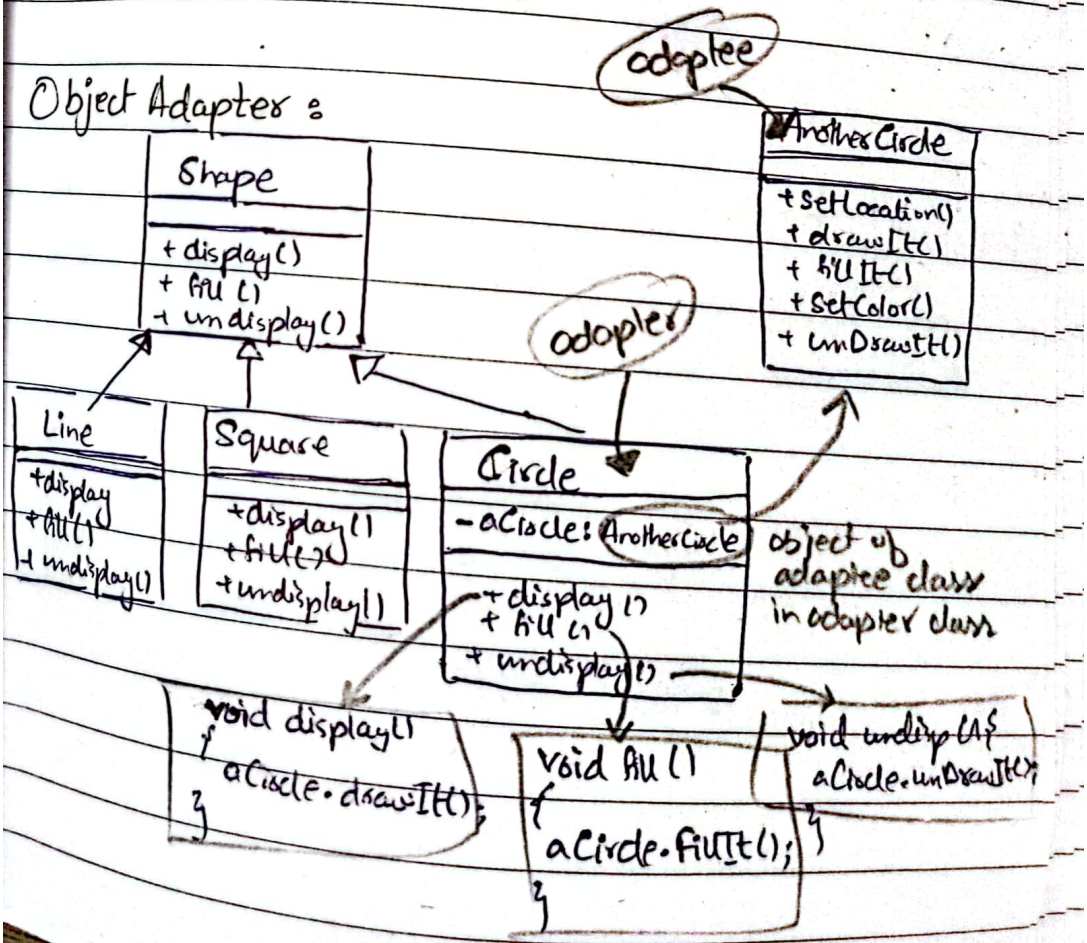
→ Use adapter when you need a way to create a new interface for an object that does the right stuff but has wrong interface.

→ Two Kinds of Adapter Pattern

↳ Object Adapter Pattern
- The adapter class contains adaptee object (the one having wrong interface) (the class which behaviour we want to use)

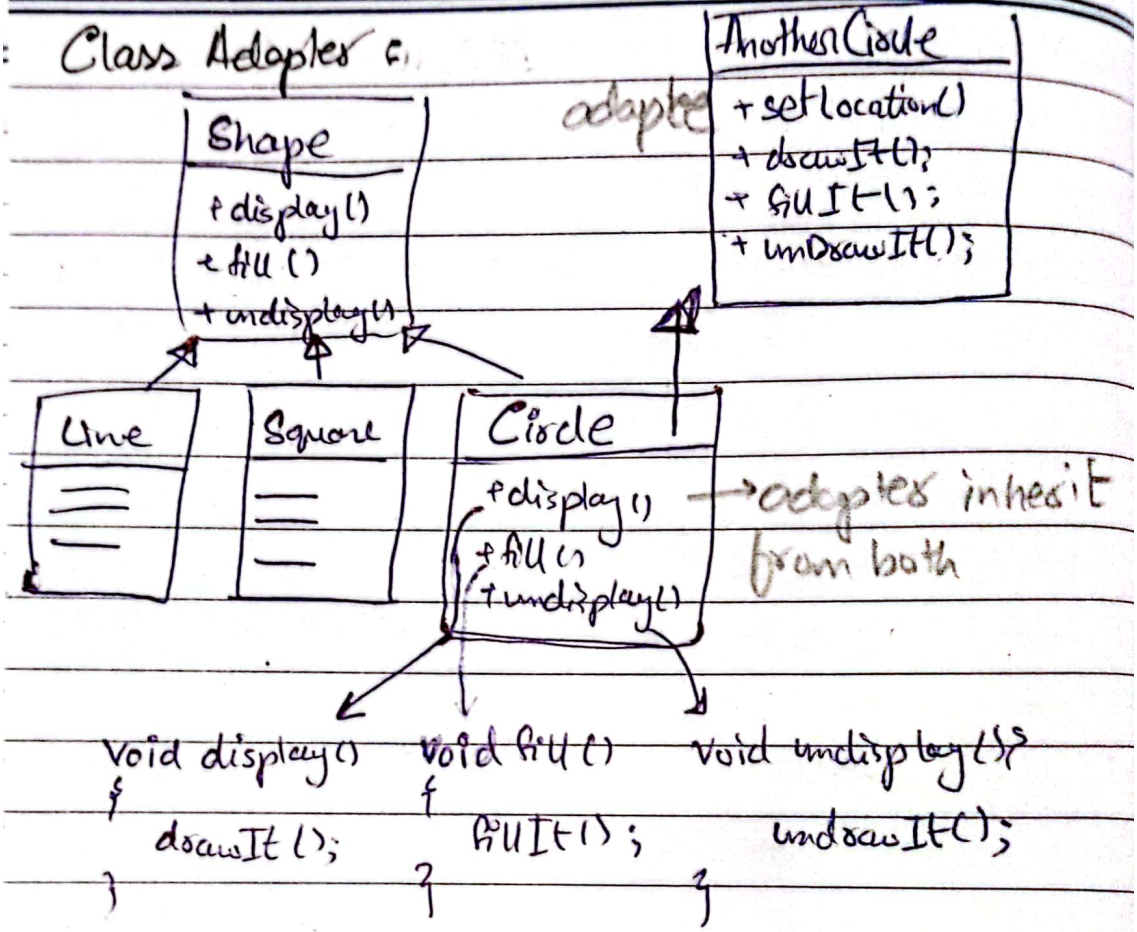
↳ Class Adapter Pattern
- Adapter class is inherited from both adaptee and abstract class

Object Adapter :

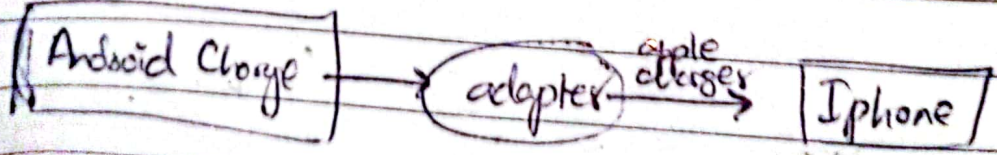


Date _____ Day _____

Class Adapter :



" Convert the services offered by class to the client in such a way that functionality will not be changed and the class will become reusable "



Page No. ☐

Date Day

```
public class Iphone13 {   // Client Class

    private AppleCharger appleCharger;

    public Iphone13(AppleCharger ac){
        this.appleCharger = ac;
    }

    public void chargeIphone(){
        appleCharger.chargePhone();
    }

}
```

the class which client uses

```
~~public class A~~
interface AppleCharger {   // Target Class

    void chagePhone();
}
```

Apdater will implement this class so that it provides the functionality required by client.

Date _____ Day _____

```
public class AndroidCharger { // Adaptee

    public void chargeAndroidPhone(){

        System.out.println("Your android phone is charging");
    }
}
```

```
public class AdapterCharger implements AppleCharger {
                                    // An object of adptee
    private AndroidCharger charger;

    public AdapterCharger(AndroidCharger c){
        this.charger = c;
    }
                                    // adapter implements the function of Target class
    public void chargePhone(){
        // overriding the implementation
        charger.chargeAndroidPhone();
        System.out.println("Your phone is charging thru adapter");
    }
                                    // calling the function of adaptee
}
```

Date _____ Day _____

```
public class Demo {

    public static void main (String[] args)
    {
                                    adapter uses the
                                    adaptee
                        charger
        AppleChargesh = new AdapterCharger (new Android);
                                                charger

        Iphone13 iphone = new Iphone13(charges);

        iphone.chargeIphone();
                ↓
            the client using the adaptee to
            charge the phone
    }
```

▷ To Use an Adapter:

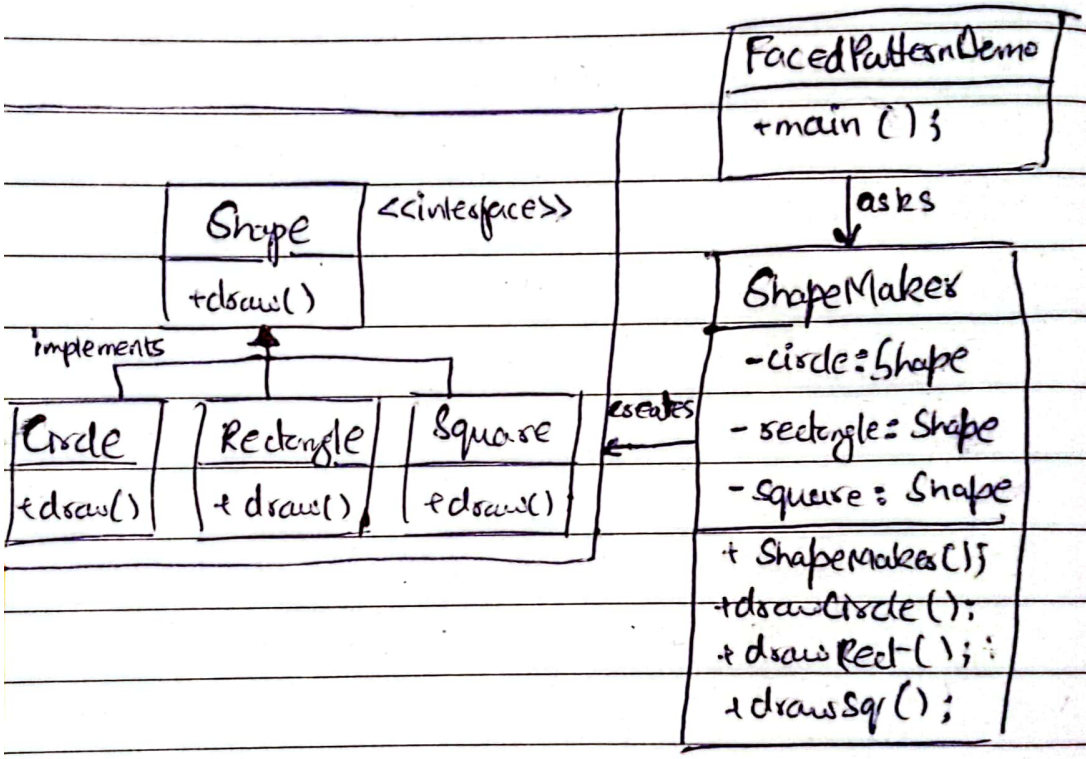
- The client makes a req/ to the adapter by calling a method on it using the target interface
- The adapter translates that req/ on the adaptee using the adaptee interface
- client recieve the result of the call and is unaware of adapter's presence.

Date _____ Day _____

③ Facade Design Pattern:

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

→ this pattern involves a single class which provide simplified methods req by client.




```
public interface shape {
    void draw();
}
//concrete classes implementing shape
public class Rectangle implements Shape {
   @Override
   public void draw() { System.out.println("Rectangle draw");
}
```

Date _____ Day _____

```
public class Square implements Shape {
    @Override
    public void draw() {
        Sys — ("Square draw");
    }
}

public class Circle implements Shape {
    @Override
    public void circle() {
        Sys — ("Circle draw");
    }
}
```

↗ Facade class

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

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

Date _____ Day _____

```
    public void drawCircle(){
        circle.draw();
    }

    public void drawRectangle(){
        rectangle.draw();
    }

    public void drawSquare(){
        square.draw();
    }
}
```

↗ the client code instead of making different objects for circle, sq, rectangle, it uses facade to achieve its results.

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

        ShapeMaker shapeMaker = new ShapeMaker();
```

↗ using Facade to draw various shape

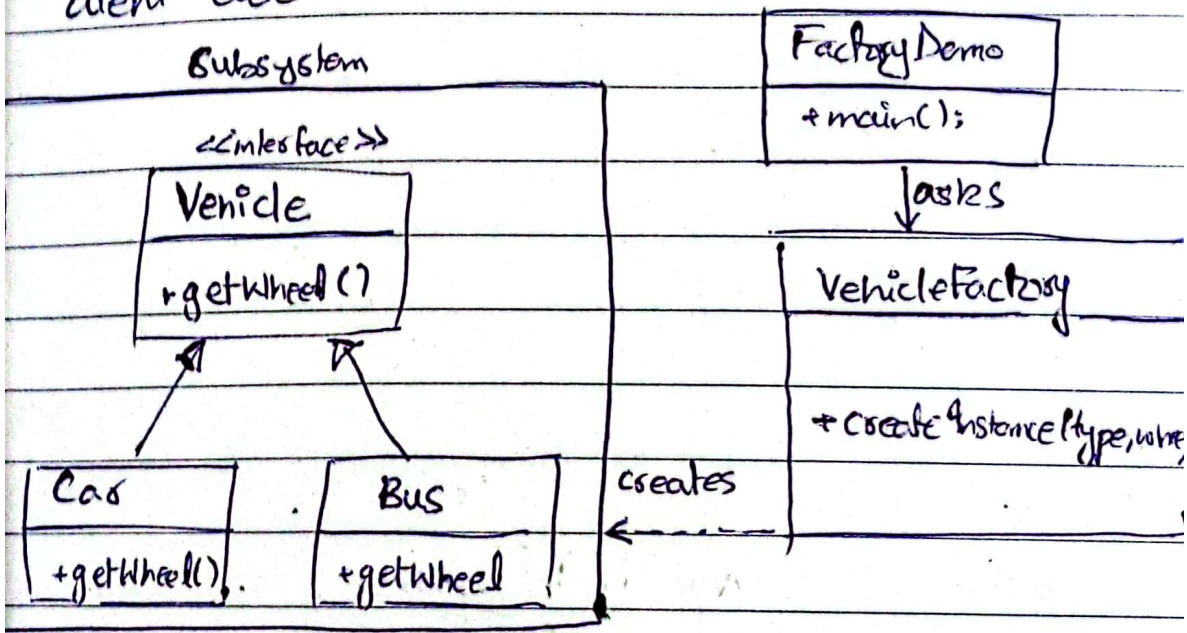
```
        shapeMaker.drawCircle();
        shapeMaker.drawSquare();
        shapeMaker.drawRectangle();
    }
}
```

Date _____ Day _____

4) Factory Design Pattern:

→ Factory pattern helps us ~~to do~~ in creation of objects.

→ All the business logic for creating an object is implemented in factory without exposing it to the client code.




```
interface Vehicle {
    public int getWheel();
}


class Car implements Vehicle {
    int wheel;

    Car(int wheel) {
        this.wheel = wheel;
    }
```

Date Day

```
@Override
public int getNWheel(){
    return this.wheel*6;
}
}

class Bus implements Vehicle{
    int wheel;

    Bus(int wheel){
        this.wheel = wheel;
    }

    @Override
    public int getNWheel(){
        return this.wheel;
    }
}
```

depending on type we create the vehicle

```
class VehicleFactory{

    public static VehicleFactory createInstance(int wheel){
        if(type.equalsIgnoreCase("car")){
            return new Car(wheel);
        }
```

Date _____ Day _____

```
else if (type.equalsIgnoreCase("Bus")){
        return new Bus(wheel);
    }
    return null;
  }
}

// Client Code
public class FactoryDemo {

    public static void main(String[] args){

        Vehicle car = VehicleFactory.createInstance
                                    ("car", 4);

        Vehicle bus = VehicleFactory.createInstance
                                    ("bus", 8);
    }
}
```