24/12/2025

Design Pattern

Creational ~~Event-driven Architecture~~

Singleton Design Pattern

- globally accessible single object
- Only one instance/object of a class is created

Example — Database connection, Logger, Configuration setting, cache Manager.

Key rules —

✓ Private constructor — no one can create obj using 'new'
✓ Private static instance — stores the single obj
✓ Public static method — returns the same instance every time.

code:

```
class Singleton{
private static Singleton instance; //private static instance
private Singleton() { //private constructor
    S.out ("Singleton created);
}
public static Singleton getInstance(){
    if(instance==null){
        instance = new Singleton();
    }
    return instance;
}}
```

Main:

```
psvm { Singleton obj1 = Singleton.getInstance();
       Singleton obj2 = "" . "" ();  }}
```

*** subclasses or factory class decide which obj to instantiate

Factory Design Pattern:

- creates object without exposing creational logic, using 'new'
- Uses a factory class to decide which obj to create

Use when –
- obj creation logic is complex
- loose coupling is needed
- multiple subclasses exist

□ code:

Interface:
```
Inteface Shape{
    void draw();
}
```

class①:
```
class circle implements Shape{
    public void draw(){
        Sout("Drawing Circle");
}}
```

class②:
```
Class Rectangle implements Shape{
    public void draw(){
        Sout("Drawing Rectangle");
}}
```

Factory Class:
```
Class ShapeFactory {
public Shape getShape(String shapeType){
    if(shapeType ==null){
        return null;
    }
    if(shapeType.equalsIgnoreCase("Circle")){
        return new Circle();
    else if
(shapeType.equalsIgnoreCase(
                "RECTANGLE"))
{ return new Reactangle();
}
return null;
}
}
```

```
psvm { //main
ShapeFactory sf = new
        ShapeFactory();
Shape s1=sf.getShape("circle");
s1.draw();
Shape s2 = " "
s2.draw();
}
}
```

→ Factory of factories

Abstract Factory :-

- provide interface for creating families of related or depend obj without specifying the concrete classes.
- Multiple factories
- More structured

Product Interface:

```
① interface Keyboard {
      void type();
  }
② interface Mouse {
      void click();
  }
```

Concrete Products:

```
① class PCkeyboard implements Keyboard {
      public void type() {
          sout("Typing with PC Keyboard");
      }
  }
  class PCMouse implements Mouse {
      public void click() {
          sout("Clicking with PC Mouse");
      }
  }
② class MacKeyboard implements ...
      publick void type() {
          - - - -
      }
  }
  class MacMouse { - - - - }
```

Abstract Factory:

```
interface ComputerFactory {
    Keyboard createKeyboard();
    Mouse createMouse();
}
```

Concrete Factories:

```
class PCfactory implements
                    ComputerFactory {
  public keyboard createKeyboard();
  return new PCKeyboard();
```

```
public Mouse createMouse() {
  return new PCMouse();
}
Class MacFactory implements
        ComputerFactory {
- - - - .
- - - - .
}
```

```
psvm { //Main
ComputerFactory f = new
        PCFactory();
Keyboard k = f.createkeyboard();
Mouse m = f.creatMouse();
Keyboard.type();
```

→ construct complex obj and allows different representations using same construction process

Builder Design Pattern:

- create complex step-by-step
- separates object construction from its representation
- Avoids many constructor parameter

Without builder:

- ✗ too many parameter
- ✗ confusing
- ✗ hard to remember order

- ✓ set values step-by-step
- ✓ No. large construction
- ✓ build() → final obj
- ✓ obj become immutable

code:

product class:

```
class Student {
    private int id;
    private String name;
    private String department;

    private Student(StudentBuilder builder) {
        this.id = builder.id;
        this.name = builder.name;
        this.dept = builder.dept;
    }
```

Builder class:

```
    static class StudentBuilder { //static inner builder class
        private int id;
        - - - - - -
        - - - - - -

        public StudentBuilder setId(int id) {
            this.id = id;
            return this;
        }
        - - - - - - - - setName() {...}
        - - - - - - - - setDept() {...}
        public Student build() {
            return new Student(this);
        }
    } //std class ends
```

```
psvm
Student s = new Student
        .StudentBuider()
    .setId(192)
    .setName("Raya")
    .setdept("CSE")
    .build();
}
```

Structural

Adapter Pattern :- Allows objects with incompatible interfaces to collaborate.

Use When →

- if existing class interface doesn't match what we need
- you want to reuse old code / existing subclasses

□ Code :

Target
```
interface MusicPlayer { //Target interface
    void play(String fileName);
}
```

Adaptee
```
class AdvancedPlayer { // existing / 3rd party class
    public void startPlayback(String audiofile) {
        sout("Playing:" + audiofile);
    } }
```

Adapter
```
class Adapter implements MusicPlayer {
    private AdvancedPlayer ap = new AdvancedPlayer();
@override
    public void play(String fileName) {
        ap.startPlayback(fileName);
    } }
```

Client Code:
```
public class Client {
    psvm {
        MusicPlayer mp = new Adapter();
        mp.play("song.mp3");
    }
```

Decorator Pattern:

- ✓ attach/add new behaviour/features dynamically
- ✓ uses composition instead of inheritance
- ✓ reduce class explosion
- ✓ doesn't change/modify original class

Code:

```
interface Coffee {
    String getDescription();
    int cost();
}
class SimpleCoffee implements Coffee{
    public String getDescription(){
        return "SimpleCoffee";
    }
    public int cost(){
        return 50;
    }
}
```

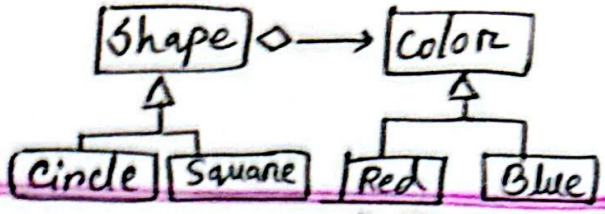
Decorator:

```
abstract class CoffeeDecorator
        implements Coffee {
    protected Coffee coffee;
    public CoffeeDecorator(Coffee c){
        this.coffee = c;
    }
}
```

Concrete Decorator

```
class MilkDecorator extends
        CoffeeDecorator{
    public MilkDecorator(Coffee c){
        super(c);
    }
    public String Description{---}
    int cost{---}
}
--- SuganDecorator{--}
--- Whippedcream{---}
```

```
psvm{
    Coffee c = new SimpleCoffee();
    coffee = new MilkDecorator(c);
    Coffee = new SuganDec(c);
    c.getDescription();
}
```



Bridge Pattern:

- ✓ Separates abstraction from implementation
- ✓ Allows both to change independently.
- ✓ Avoid large inheritance hierarchies
- ✓ Connects abstraction and implementation using composition, not inheritence. (Decouples abstraction)

- Abstraction ⟶ high level operation
- Implementation ⟶ low level "

Code:

Interface:

```
interface Color {
    void applyColor();
}
```

Concrete:

```
class RedColor implements Color{
    public void applyColor(){
        sout("Red Color");
    }
}
" BlueColor " {...}
```

Abstraction:

```
abstract class Shape {
    protected Color color;
    protected Shape(Color c){
        this.color = c;
    }
    abstract void draw();
}
```

```
class Circle extends Shape {
    Circle(Color c){
        super(c);
    }
    public void draw(){
        ...
    }
class Square extends
                Shape {
    ...
    color.applyColor();
}
```

```
psvm{
Shape redCircle = new
Circle(new RedColor());
redCircle.draw();

Shape blueSquare = " "
blueSquare.draw();
}
}
```

✓ Switch implementation at runtime
✓ Use when you want to divide and organize a monolithic class with several functionality

Proxy Pattern:

✓ provides a placeholder / representation for another obj
✓ Control access to the real object
✓ Add extra behaviour without changing real obj

Client → Proxy → Real object

- Use when —
  - obj creation is expensive
  - access control is needed
  - Lazy loading is required
  - logging / security is required

Types:
- virtual proxy - Lazy loading
- Protection " - Access control
- Remote " - Remote obj access

Code:

Subject Interface:

```
interface Internet{
    void connectTo(String server);
}
```

Real Subject:

```
class RealInternet implements Internet{
public void connectTo(String server){
    Sout("Connecting to "+server);
}}
```

Proxy:

```
Class ProxyInternet implements Internet{
  private Internet r = new RealInternet();
    public void connectTo(String server){
if(server.equalsIgnoreCase
        ("blocked.com")){
  Sout("Access denied");}
else{
  r.connectTo(server);
}}}
```

```
psvm{
Internet i = new ProxyInter(
i.connectTo("google.com");
i.connectTo("blocked.com")
}}
```

Facade Pattern: ✓ Hides internal complexity

✓ provide a simple interface of a complex system

- Use when —
  - system is complex
  - you want loose coupling
  - make code readable
  - c