Unit - 2

Design Pattern

Definition of Design pattern

- ❖ Design patterns are reusable solutions to common software design problems that help developers build cleaner and more maintainable systems.
- ❖ Each pattern Describes a problem which occurs over and over again in our environment, and then describes the core of the problem
- ❖ Problems are addressed without rediscovering solutions from scratch.
- ❖ "My wheel is rounder. Design Patterns are the best solutions for the re-occurring problems in the application programming environment.
- ❖ Nearly a universal standard.
- ❖ Responsible for design pattern analysis in other areas, including GUIs.
- ❖ Mainly used in Object Oriented programming.

Types of Design Pattern

- ❖ Creational Design Patterns
- ❖ Structural Design Patterns
- ❖ Behavioural Design Patterns

Creational Design Patterns

Creational Design Patterns focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed, and represented.

Two main themes:

- They keep information about the specific classes used in the system hidden.
- They hide the details of how instances of these classes are created and assembled.

Types:

1. Factory Method Design Pattern
2. Abstract Factory Method Design Pattern
3. Singleton Method Design Pattern
4. Prototype Method Design Pattern
5. Builder Method Design Pattern

1. Factory Method Design Pattern

The Factory pattern abstracts the process of object creation and allows subclasses or derived classes to determine which class to instantiate. The Factory pattern also provides a central factory class that encapsulates object creation logic.

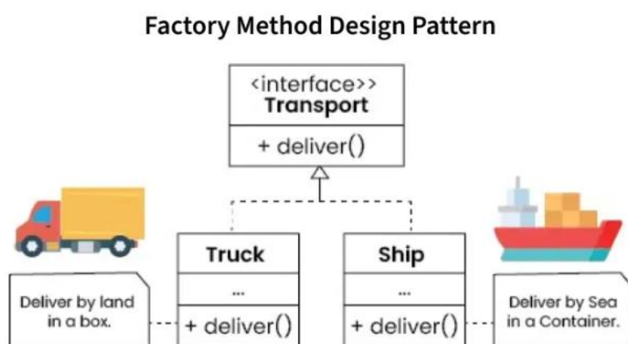
when to use Factory Method Design Pattern?

- A class can't anticipate the class of objects it must create.

- A class wants its subclass 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.

2.Abstract Factory Method Design Pattern

- It is almost similar to Factory Pattern
- Abstract Factory patterns work around a super-factory which creates other factories.
  - Subclasses override the factory method to produce specific object types.
  - Supports easy addition of new product types without modifying existing code.
  - Enhances maintainability and adaptability at runtime.



when to use Abstract Factory Method Design Pattern:

- 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.
- We want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Real World Use Cases

- Web Browsers (e.g., Chrome, Firefox)
  Browsers use factory methods to create different types of plugins or page renderers based on content type (e.g., PDF, HTML, Flash). This allows flexible and extensible handling of various media.
- Android OS (Activity Instantiation)
  In Android, activities are often created using factory methods internally to manage lifecycle and resource setup. Developers override methods like onCreate() while the system handles object creation.
- Payment Gateways (e.g., Stripe, PayPal)
  E-commerce platforms use factory methods to create different payment processors. Based on user selection, the factory returns an instance of the correct payment service without changing client logic.

- Game Development (e.g., Unity, Unreal Engine)
  Games use factory methods to spawn enemies, items, or NPCs dynamically based on game level or environment. This decouples object creation from the game logic and enables easy scalability.

Features

- Encapsulation of Object Creation: Clients don't know how objects are created.
- Loose Coupling: Reduces dependency between client and concrete classes.
- Scalability: New product types can be introduced without altering client code.
- Reusability: Common creation logic can be reused across factories.
- Flexibility: Supports multiple product families with minimal changes.
- Testability: Easy to use mock factories for unit testing.

Components of Factory Method Design Pattern

Below are the main components of Factory Design Pattern:

- Product: Abstract interface or class for objects created by the factory.
- Concrete Product: The actual object that implements the product interface.
- Creator (Factory Interface/Abstract Class): Declares the factory method.
- Concrete Creator (Concrete Factory): Implements the factory method to create specific products.

Factory Method Design Pattern Example

Below is the problem statement to understand Factory Method Design Pattern:

Consider a software application that needs to handle the creation of various types of vehicles, such as Two Wheelers, Three Wheelers and Four Wheelers. Each type of vehicle has its own specific properties and behaviours.

1.Without Factory Method Design Pattern

```
import java.io.*;

// Library classes abstract class Vehicle {

public abstract void printVehicle();

}

class TwoWheeler extends Vehicle

{ public void printVehicle(){

System.out.println("I am two wheeler");
```

```
}}
class FourWheeler extends Vehicle
{ public void printVehicle(){
System.out.println("I am four wheeler");
}}
// Client (or user) class
 class Client {
private Vehicle pVehicle;
 public Client(int type) {
if(type 1) {
pVehicle = new TwoWheeler();
} else if(type   2){
pVehicle = new FourWheeler();
} else{
pVehicle = null;
}}
public void cleanup()
{
 if(pVehicle != null)
{
pVehicle = null;
}}
public Vehicle getVehicle()
{
 return pVehicle;
}}
// Driver program public class GFG {
public static void main(String[] args)
{
 Client pClient = new Client(1);
```

```
Vehicle pVehicle = pC1ient.getVehicle();
 if(pVehicle != null){
pVehicle.printVehic1e();
}
pClient.c1eanup();
 }
}
```

Output

I am two wheeler.

Issues with the Current Design

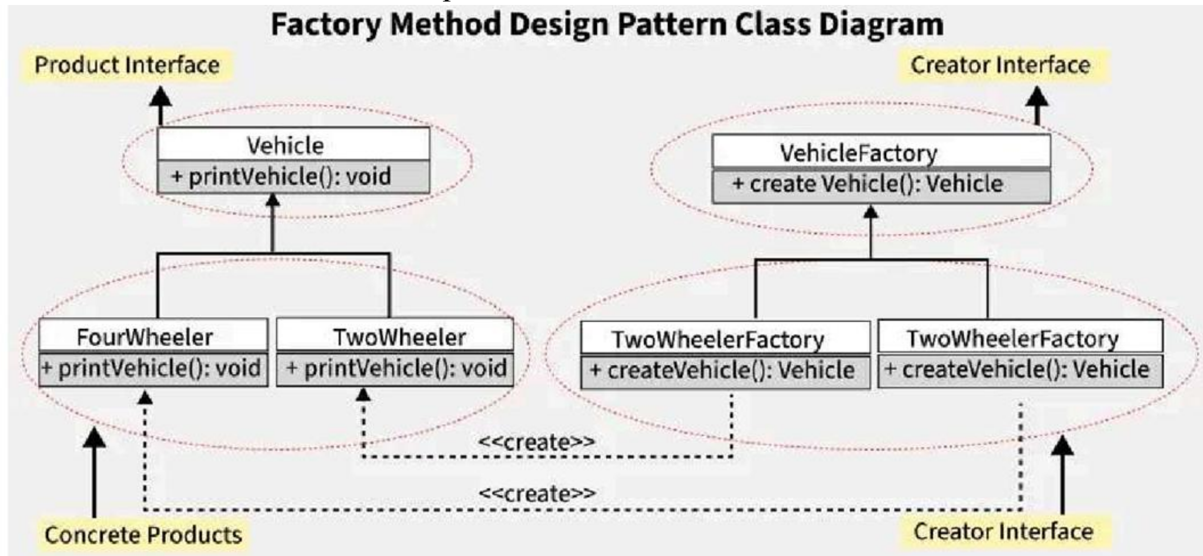
- Tight coupling: Client depends directly on product classes.
- Violation of SRP: Client handles both product creation and usage.
- Hard to extend: Adding a new vehicle requires modifying the client.

Solutions to the Problems

- ❖ Define a Factory Interface: Create an interface, VehicleFactory, with a method to produce vehicles.
- ❖ Create Specific Factories: Implement classes like TwoWheelerFactory and FourWheelerFactory that follow the VehicleFactory interface, providing methods for each vehicle type.
- ❖ Revise the Client Class: Change the Client class to use a VehicleFactory instance instead of creating vehicles directly. This way, it can request vehicles without using conditional logic.
- ❖ Enhance Flexibility: This structure allows for easy addition of new vehicle types by simply creating new factory classes, without needing to alter existing Client code.

2. With Factory Method Design Pattern

Let's breakdown the code into component wise code:



Factory Method Design Pattern

1. Product Interface

Product interface representing a vehicle

```
public abstract class Vehicle {
// Constructor to prevent direct instantiation
private Vehicle() {
throw new UnsupportedOperationException("Cannot construct Vehicle instances
directly");
}
// Abstract method to be implemented by subclasses
public abstract void printVehicle();
```

2.Concrete Products

Concrete product classes representing different types of vehicles

```
public class Vehicle
{
public void printVehicle()
{
// This method should be overridden
}}
public class TwoWheeler extends Vehicle
```

```
{ @Override
public void printVehicle()
{
System.out.println("I am two wheeler");
}
}
public class FourWheeler extends Vehicle
{ @Override
public void printVehicle()
{ System.out.println("I am four wheeler");
}}
```

3. Creator Interface (Factory Interface)

Factory interface defining the factory method

```
public interface VehicleFactory{
Vehicle createVehicle();
}
```

4. Concrete Creators (Concrete Factories)

Concrete factory class for TwoWheeler

```
public interface Vehicle {}
public class TwoWheeler implements Vehicle {}
public class FourWheeler implements Vehicle {}
public interface VehicleFactory{
Vehicle createVehicle();
}
public class TwoWheelerFactory implements VehicleFactory
{ public Vehicle createVehicle() {
 return new TwoWheeler();
}
}
```

```
public class FourWheelerFactory implements VehicleFactory
{ public Vehicle createVehicle() {
return new FourWheeler();
}
}
```

Complete Code of this example:

```
// Library classes
abstract class Vehicle {
public abstract void printVehicle();
}}
class TwoWheeler extends Vehicle
{ public void printVehicle() {
System.out.println("I am two wheeler");
}}
class FourWheeler extends Vehicle
{
public void printVehicle()
{
System.out.println("I am four wheeler");
}}
// Factory Interface
interface VehicleFactory{
Vehicle createVehicle();
}
// Concrete Factory for TwoWheeler
class TwoWheelerFactory implements VehicleFactory {
 public Vehicle createVehicle()
{
 return new TwoWheeler();
```

```
}}
// Concrete Factory for FourWheeler
class FourWheelerFactory implements VehicleFactory
{
 public Vehicle createVehicle()
{
return new FourWheeler();
}}
// Client class
class Client {
private Vehicle pVehicle;
public Client(VehicleFactory factory)
{ pVehicle = factory.createVehicle();
}
public Vehicle getVehicle()
{ return pVehicle;
}
}
// Driver program
 public class GFG {
public static void main(String[] args)
{
VehicleFactory twoWheelerFactory = new TwoWheelerFactory();
Client twoWheelerClient = new Client(twoWhee1erFactory);
Vehicle twoWheeler = twoWheelerClient.getVehic1e();
 twoWheeler.printVehic1e();
VehicleFactory fourWheelerFactory = new FourWheelerFactory();

Client fourWheelerClient = new Client(fourWheelerFactory);
 Vehicle fourWheeler = fourWheelerC1ient.getVehicle();
```

```
fourWheeler.printVehicle();
}
}
```

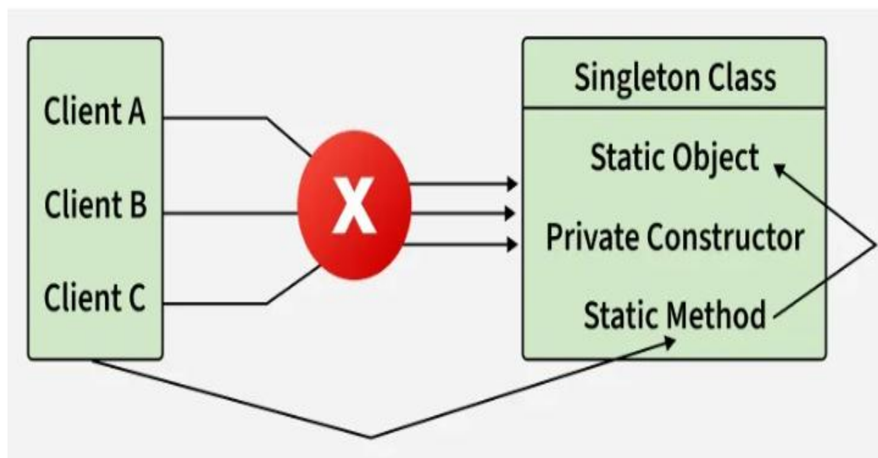
Output

I am two wheeler

I am four wheeler

3.Singleton Method Design Pattern

It ensures a class only has one instance, and provides a global point of access to it. It ensures a global access point as well. This pattern is proper when there should be a single class instance throughout the application.



Real-World Applications of the Singleton Pattern

1. Logging Systems : Maintain a consistent logging mechanism across an application.
2. Configuration Managers : Centralize access to configuration settings.
3. Database Connections : Manage a single point of database access.
4. Thread Pools : Efficiently manage a pool of threads for concurrent tasks.
5. Cache Managers, Print Spoolers (Single Printer Queue) and Runtime Environments (java.lang.Runtime is a singleton)

Features :

1.Single Instance: Ensures only one object of the class exists in the JVM.

2. Global Access Point: Provides a centralized way to access the instance.
3. Lazy or Eager Initialization: An Instance can be created at class load time (eager) or when first needed (lazy).
4. Thread Safety: Can be designed to work correctly in multithreaded environments.

5. Resource Management: Useful for managing shared resources like configurations, logging or database connections.
6. Flexibility in Implementation: Can be implemented using eager initialization, lazy initialization, double-checked locking or an inner static class.

when to use Singleton Method:

- 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.

Key Components

Below are the main key components of Singleton Method Design Pattern

1.Static Member

- The Singleton pattern or pattern Singleton employs a static member within the class. This static member ensures that memory is allocated only once, preserving the single instance of the Singleton class.

```
private static Singleton instance;
```

2.Private Constructor

The Singleton pattern or pattern singleton incorporates a private constructor, which serves as a barricade against external attempts to create instances of the Singleton class. This ensures that the class has control over its instantiation process.

```
class Singleton
{
private Singleton
{
}
}
```

3. Static Factory Method

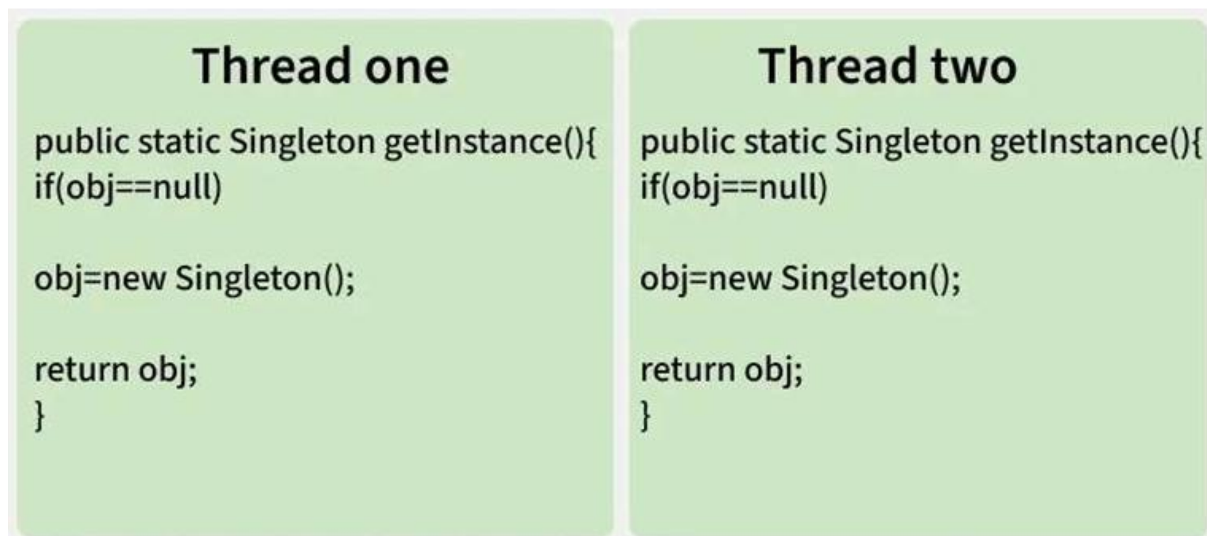
A crucial aspect of the Singleton pattern is the presence of a static factory method. This method acts as a gateway, providing a global point of access to the Singleton object. When someone requests an instance, this method either creates a new instance (if none exists) or returns the existing instance to the caller.

```
public static Singleton getInstance()
```

```
{
if(instance == null)
{
 instance = new Singleton();
}
return instance;
}
```

Different Ways to Implement Singleton Method Design Pattern

Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.



Various design options for implementation:

1. Classic (Lazy Initialization): In this method, class is initialized whether it is to be used or not. The main advantage of this method is its simplicity. You initiate the class at the time of class loading. Its drawback is that class is always initialized whether it is being used or not.

Example: Classical Java implementation of singleton design pattern

```
class Singleton {
private static Singleton obj;
private Singleton() {}
public static Singleton getInstance()
{
```

```
if(obj == null)
obj = new Singleton();
return obj;
}
}
```

Here we have declared **getInstance()** static so that we can call it without instantiating the class. The first time **getInstance()** is called it creates a new singleton object and after that, it just returns the same object.

This execution sequence creates two objects for the singleton. Therefore this classic implementation is not thread-safe.

2. Thread-Safe (Synchronized): Make getInstance() synchronized to implement Singleton Method Design Pattern

Example: Thread Synchronized Java implementation of singleton design pattern

```
class Singleton {
private static Singleton obj;
private Singleton() {}
public static synchronized Singleton getInstance()
{
if(obj == null)
obj = new Singleton();
return obj;
}
}
```

Here using synchronized makes sure that only one thread at a time can execute getInstance(). The main disadvantage of this method is that using synchronized every time while creating the singleton object is expensive and may decrease the performance of your program. However, if the performance of getInstance() is not critical for your application this method provides a clean and simple solution.

3. Eager Initialization (Static Block): In this method, class is initialized only when it is required. It can save you from instantiating the class when you don't need it. Generally, lazy initialization is used when we create a singleton class.

Example: Static initializer based Java implementation of singleton design pattern

```
class Singleton {
```

```
private static Singleton obj = new Singleton();
private Singleton() {}
public static Singleton getInstance()
{ return obj; }
```

Here we have created an instance of a singleton in a static initializer. JVM executes a static initializer when the class is loaded and hence this is guaranteed to be thread-safe. Use this method only when your singleton class is light and is used throughout the execution of your program.

4. Double-Checked Locking (Most Efficient): Use "Double Checked Locking" to implement singleton design pattern

Example: Double Checked Locking based Java implementation of singleton design pattern

```
class Singleton
{
private static volatile Singleton obj = null; private Singleton() {}
public static Singleton getInstance()
{
if(obj == null) {
synchronized (Singleton.class)
{
if(obj == null)
obj = new Singleton();
}
}
return obj;
}
}
```

We have declared the obj volatile which ensures that multiple threads offer the obj variable correctly when it is being initialized to the Singleton instance. This method drastically reduces the overhead of calling the synchronized method every time.

5. Static Inner Class (Best Java-Specific Way)

In Java, a Singleton can be implemented using a static inner class.

- **A class is loaded into memory only once by the JVM.**

- **An inner class is loaded only when it is referenced.**
- **Therefore, the Singleton instance is created lazily, only when the getInstance() method accesses the inner class.**

Example: using class loading concept singleton design pattern

```
public class Singleton {
private Singleton() {
 System.out.println(”Instance created");
}
private static class Singletonlnner
{
private static final Singleton INSTANCE=new Singleton();
}
public static Singleton getInstance()
{
return SingletonInner.INSTANCE;
}
}
```

In the above code, we are having a private static inner class Singletonlnner and having a private field. Through, getInstance() method of the singleton class, we will access the field of the inner class and due to being inner class, it will be loaded only one time at the time of accessing the INSTANCE field for the first time. And the INSTANCE is a static member due to which it will be initialized only once.

6. Enum Singleton: In Java, a Singleton can also be implemented using an enum, which is the simplest and safest approach. Enums are loaded by the JVM only once, and each enum constant is created exactly one time. Therefore, the Singleton instance is created safely when the enum is first accessed.

Esample: Enum-based Singleton

```
public enum Singleton
{
 INSTANCE;
public void dosomething()
{
 System.out.println("Doing something...");
```

```
 }
}
```

Explanation:

In this implementation

- **INSTANCE is the single allowed object of the enum.**
- **The JVM ensures that the enum is thread-safe, created only once, and cannot be instantiated again.**
- **It also automatically protects against serialization and reflection issues.**
- **When Singleton.INSTANCE is accessed for the first time, the enum is loaded and the instance is initialized only once.**

Implementation of the singleton Design pattern

Example: The implementation of the singleton Design pattern is very simple and consists of a single class.

```
import java.io.*;
class Singleton {
// static class
private static Singleton instance;
private Singleton()
{
System.out.println("Singleton is Instantiated.");
}
public static Singleton getInstance()
{
if(instance == null)
instance = new Singleton();
return instance;
}
public static void dosomething()
 {
System.out.println("Something is Done.");
}
```

```
}
class GFG
{(
public static void main(String[] args)
{
Singleton.getInstance().dosomething();
}
}
```

Output

Singleton is Instantiated.

Something is Done.

.

Advantage of Creational Design Patterns:

- ❖ They allow for greater flexibility in object creation.
- ❖ These patterns encapsulate the logic of object creation, which simplifies code management and promotes cleaner, more organized code.
- ❖ By centralizing the creation logic, these patterns promote reusability.
- ❖ They help manage complex object creation processes, making it easier to handle configurations and dependencies, especially when dealing with many related classes.

Structural Design Patterns:

Structural Design Patterns focus on organizing classes and objects to build larger, efficient, and maintainable software structures. They simplify relationships, support code reuse, and help create scalable architectures.

Main themes:

- This pattern is particularly useful for making independently developed class libraries work together.
- Structural Design 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.

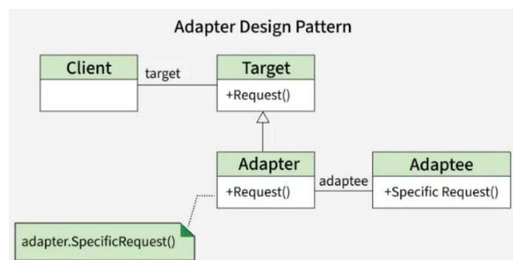
Types:

Structural Design Patterns

- Adapter Pattern
- Bridge Pattern
- Composite Pattern
- Decorator Pattern
- Facade Pattern
- Flyweight Pattern

ScholarHat

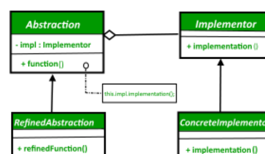
1. **Adapter:** The Adapter pattern converts the interface of a class into another interface that clients expect, enabling types with incompatible interfaces to work together as a bridge.



2. **The Bridge design** pattern allows you to separate the abstraction from the implementation. It is a structural design pattern.

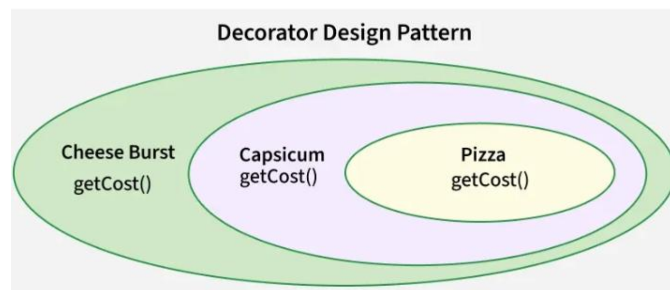
   2 parts in Bridge design pattern :

   - ❖ Abstraction

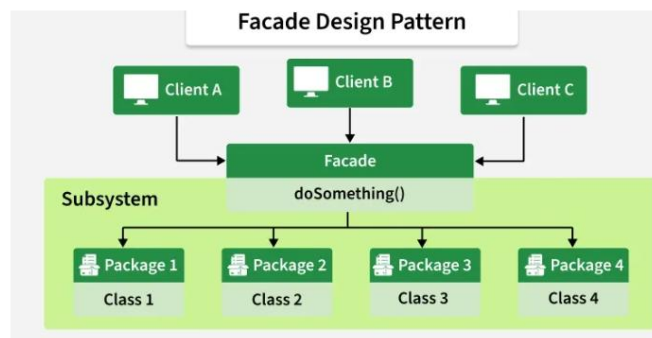


   - ❖ Implementation

2. **Decorator**: Decorator Design Pattern is a structural pattern that lets you dynamically add behavior to individual objects without changing other objects of the same class. It uses decorator classes to wrap concrete components, making functionality more flexible and reusable.



3. **Facade Design** Pattern is a Structural pattern from the Gang of Four that simplifies interactions with complex subsystems. It provides a unified, easy-to-use interface while hiding internal details, reducing complexity for clients and promoting cleaner, more maintainable code.



4.The Composite Pattern:

It is a structural design pattern that organizes objects into tree structures, enabling clients to treat individual and composite objects uniformly through a common interface.

5.Proxy Design Pattern

Proxy Design Pattern is a structural design pattern where a proxy object acts as a placeholder to control access to the real object. The client communicates with the proxy, which forwards requests to the real object. The proxy can also provide extra functionality such as access control, lazy initialization, logging, and caching.

6. Flyweight design pattern

The Flyweight design pattern is a structural pattern that optimizes memory usage by sharing a common state among multiple objects. It aims to reduce the number of objects created and to decrease memory footprint, which is particularly useful when dealing with a large number of similar objects.

Behavioral Design Patterns

1.Observer Design Pattern: Observer Design Pattern is a behavioral pattern that establishes a one-to-many dependency between objects. When the subject changes its state, all its

observers are automatically notified and updated. It focuses on enabling efficient communication and synchronization between objects in response to state changes.

2.Strategy Design Pattern: Strategy Design Pattern is a behavioral design pattern that allows you to define a family of algorithms or behaviors, put each of them in a separate class, and make them interchangeable at runtime. This pattern is useful when you want to dynamically change the behavior of a class without modifying its code.

3. Command Design Pattern: Command Design Pattern is a behavioral pattern that encapsulates a request as an object, decoupling the sender from the receiver. It allows requests to be queued, logged, parameterized, or undone/redone, providing flexibility and extensibility in executing operations.

4. Chain of Responsibility Design Pattern : The Chain of Responsibility design pattern is a behavioral design pattern that allows an object to pass a request along a chain of handlers. Each handler in the chain decides either to process the request or to pass it along the chain to the next handler.

5. Template Method Design Pattern : Template Method Design Pattern is a behavioral pattern that defines the skeleton of an algorithm in a base method while allowing subclasses to override specific steps without altering its overall structure. It's like a recipe: the main steps remain fixed, but details can be customized for variation.

6. Iterator Design Pattern : The Iterator design pattern is a behavioral design pattern that provides a way to access the elements of an aggregate object (like a list) sequentially without exposing its underlying representation. It defines a separate object, called an iterator, which encapsulates the details of traversing the elements of the aggregate, allowing the aggregate to change its internal structure without affecting the way its elements are accessed.

7. Mediator Design Pattern: The Mediator Design Pattern simplifies communication between multiple objects in a system by centralizing their interactions through a mediator. Instead of objects interacting directly, they communicate via a mediator, reducing dependencies and making the system easier to manage.

8. State Design Pattern: State Design Pattern is a behavioral design pattern that allows an object to change its behavior when its internal state changes. This pattern is particularly useful when an object's behavior depends on its state, and the state can change during the object's lifecycle.

9. Mediator Design Pattern : The Mediator Design Pattern simplifies communication between multiple objects in a system by centralizing their interactions through a mediator. Instead of objects interacting directly, they communicate via a mediator, reducing dependencies and making the system easier to manage.

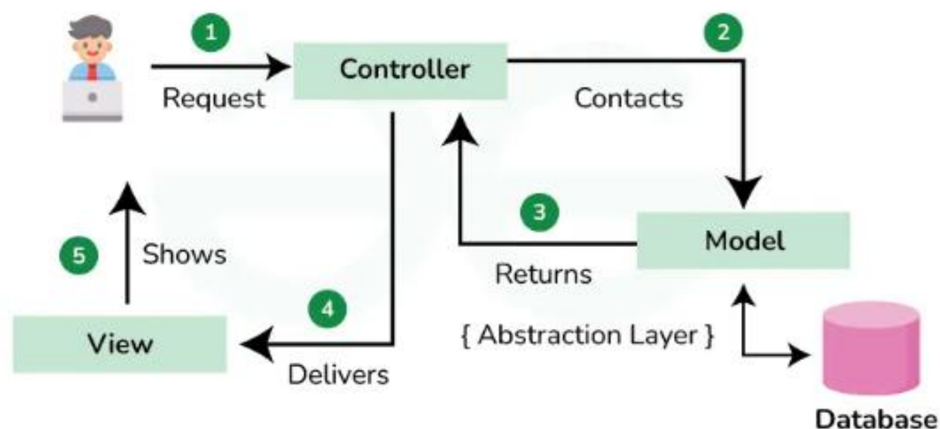
10.Memento Design Pattern: The Memento Design Pattern is a behavioral pattern that helps save and restore an object's state without exposing its internal details. It is like a "snapshot" that allows you to roll back changes if something goes wrong. It is widely used in undo-redo functionality in applications like text editors or games.

11.Visitor design pattern: An object-oriented programming method called the Visitor design pattern makes it possible to add new operations to preexisting classes without

changing them. It improves the modularity and maintainability of code, which makes it perfect for operations on a variety of object structures. Its advantages, and real-world applications are examined in this article.

MVC Design Pattern

The Model View Controller (MVC) design pattern specifies that an application consists of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects. The MVC pattern separates the concerns of an application into three distinct components, each responsible for a specific aspect of the application's functionality.This separation of concerns makes the application easier to maintain and extend, as changes to one component do not require changes to the other components.



What is the MVC Design Pattern?

The Model View Controller (MVC) design pattern specifies that an application consists of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.

- The MVC pattern separates the concerns of an application into three distinct components, each responsible for a specific aspect of the application's functionality.
- This separation of concerns makes the application easier to maintain and extend, as changes to one component do not require changes to the other components.

Why use MVC Design Pattern?

The MVC (Model-View-Controller) design pattern breaks an application into three parts: the Model (which handles data), the View (which is what users see), and the Controller (which connects the two). This makes it easier to work on each part separately, so you can update or fix things without messing up the whole app. It helps developers add new features smoothly, makes testing simpler, and allows for better user interfaces. Overall, MVC helps keep everything organized and improves the quality of the software.

Components of the MVC Design Pattern

1. Model : The Model component in the MVC (Model-View-Controller) design pattern demonstrates the data and business logic of an application. It is responsible for managing the application's data, processing business rules, and responding to requests for information from other components, such as the View and the Controller.

2. View: Displays the data from the Model to the user and sends user inputs to the Controller. It is passive and does not directly interact with the Model. Instead, it receives data from the Model and sends user inputs to the Controller for processing.

3. Controller: Controller acts as an intermediary between the Model and the View. It handles user input and updates the Model accordingly and updates the View to reflect changes in the Model. It contains application logic, such as input validation and data transformation.

Communication between the Components

This below communication flow ensures that each component is responsible for a specific aspect of the application's functionality, leading to a more maintainable and scalable architecture

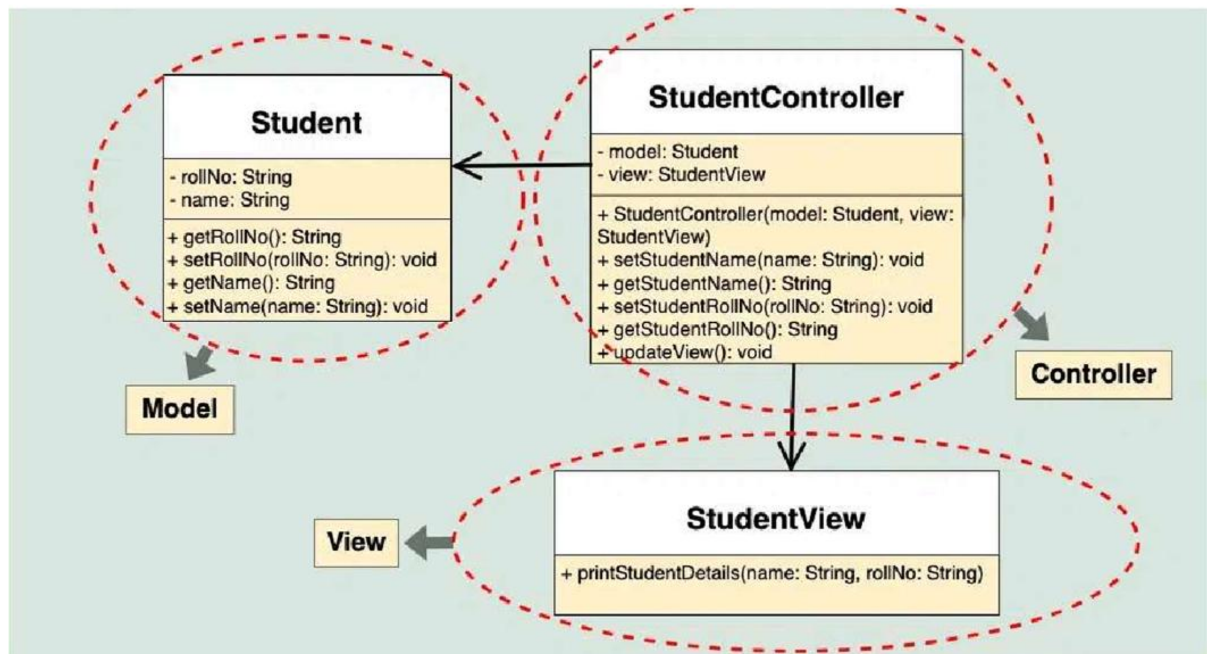
- User Interaction with View: The user interacts with the View, such as clicking a button or entering text into a form.
- View Receives User Input: The View receives the user input and forwards it to the Controller.
- Controller Processes User Input: The Controller receives the user input from the View. It interprets the input, performs any necessary operations (such as updating the Model), and decides how to respond.
- Controller Updates Model: The Controller updates the Model based on the user input or application logic.
- Model Notifies View of Changes: If the Model changes, it notifies the View.
- View Requests Data from Model: The View requests data from the Model to update its display.
- Controller Updates View: The Controller updates the View based on the changes in the Model or in response to user input.
- View Renders Updated UI: The View renders the updated UI based on the changes made by the Controller.

When to Use the MVC Design Pattern:

- Complex Applications
- Frequent UI Changes
- Reusability of Components
- Testing Requirements

Example of the MVC Design Pattern

- Below is the code of above problem statement using MVC Design Pattern:



1. Model (Student class)

Represents the data (student's name and roll number) and provides methods to access and modify this data.

```
class Student {
private String rollNo;
 private String name;
public String getRollNo()
{ return rollNo;
}
public void setRol1No(String rollNo)
{ this.ro1lNo = rollNo;
}
public String getName()
{ return name;
}
public void setName(String name)
{ this.name = name;
 }}
```

2. View (StudentView class)

Represents how the data (student details) should be displayed to the user. Contains a method (printStudentDetaiIs) to print the student's name and roll number.

```
class StudentView {

public void printStudentDetails(String studentName, String studentRollNo){
System.out.println("Student:");

System.out.println("Name: " + studentName); System.out.println("Roll No: " +
studentRollNo);

}}
```

3. Controller (StudentController class)

Acts as an intermediary between the Model and the View. Contains references to the Model and View objects. Provides methods to update the Model (e.g., setStudentName, setStudentRoIlNo) and to update the View (updateView).

```
class StudentController

{ private Student model;

private StudentView view;

public StudentContro1ler(Student model, StudentView view)

{ this.model = model;

This.view = view;

}

public void setStudentName(String name)

{ model.setName(name);

}

public String getStudentName()

{ return model.getName();

 }

public void setStudentRo1lNo(String rollNo)

{ model.setRol1No(rollNo);

}

public String getStudentRollNo()

{ return model.getRo11No();

}
```

```
public void updateView()
{
 view.printStudentDetai1s(model.getName(), model.getRo11No());
}}
```

Complete code for the above example

Below is the complete code for the above example:

```
 class Student {
private String rollNo; private String name;
public String getRollNo()
{
 return rollNo;
}
public void setRollNo(String rollNo)
{
this.ro11No = rollNo;
}
public String getName()
{
 return name;
}
public void setName(String name)
{
 this.name = name;
}
class StudentView
{
public void printStudentDetails(String studentName, String studentRollNo)
{ System.out.println("Student:");
System.out.println("Name: " + studentName);
System.out.println("Roll No: " + studentRollNo);
```

```
class StudentController
{ private Student model;
 private StudentView view;
public StudentControl1er(Student model, StudentView view)
{ this.model = model;
this.view = view;
}
public void setStudentName(String name)
{ model.setName(name);
}
public String getStudentName()
{ return model.getName();
}
public void setStudentRol1No(String rollNo)
{model.setRo11No(rollNo);
}
public String getStudentRollNo()
{ return model.getRollNo();
}
public void updateView()
{ view.printStudentDetails
{model.getName(), model.getRo11No());
}}
public class MVCPattern
{
public static void main(String[] args){
Student model = retriveStudentFromDatabase();
StudentView view = new StudentView();
StudentController controller = new StudentControl1er(model, view);
controller.updateView();
controller.setStudentName("Vikram");
```

```
controller.updateView
}
private static Student retriveStudentFromDatabase(){
{ Student student = new Student();
student.setName("Lokesh");
student.setRo1lNo("15UCS157");
return student;
}}
```

Output

Student:

Name: Lokesh

Roll No: 15UCS157

Student:

Name: Vikram

Roll No: 15UCS157