UNIT II – SOFTWARE DESIGN AND UML DIAGRAMS [9 hours]

Design Principles (Modularity, Reusability, Abstraction), UML Diagrams: Use Case, Class, Activity, Sequence, Introduction to Design Patterns (Singleton, Factory, MVC),Building Simple System Architecture (Layered & Client-Server).

INTRODUCTION TO DESIGN PATTERNS

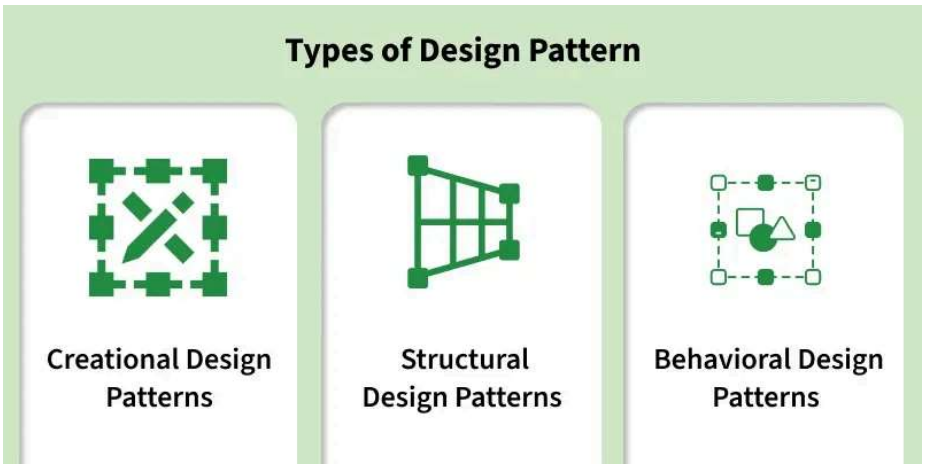
Design patterns are reusable solutions to common software design problems. They help create clean, scalable, and maintainable code by providing proven design templates.

- Reusable solutions to recurring problems.
- Improve code structure and maintainability.
- Enhance scalability, flexibility, and reusability.
- Standardize design practices across projects.
- Categorized into Creational, Structural, Behavioral.

Types of Design Patterns

Creational, Structural, and Behavioral Design Patterns -three categories that define how objects are created, organized, and how they interact.

Basically, there are several types of design patterns that are commonly used in software development.



Creational Design Patterns

These Creational Design Patterns deal with object creation in a flexible and efficient manner. They help you control how and when objects are instantiated.

- Singleton Pattern
- Factory Method Pattern
- Abstract Factory Pattern
- Builder Pattern
- Prototype Pattern
- Object Pool Pattern
- Lazy Initialization

Structural Design Patterns

Structural patterns explain how classes and objects are combined to form larger structures. They improve code flexibility by simplifying relationships between components.

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

Behavioral Design Patterns

Behavioral patterns define how objects communicate and distribute responsibilities. They help manage workflows, interactions, and decision-making within a system

- Observer Pattern
- Strategy Pattern
- Command Pattern
- Chain of Responsibility Pattern
- Template Method Pattern

- Iterator Pattern
- State Pattern
- Mediator Pattern
- Memento Pattern
- Visitor Pattern

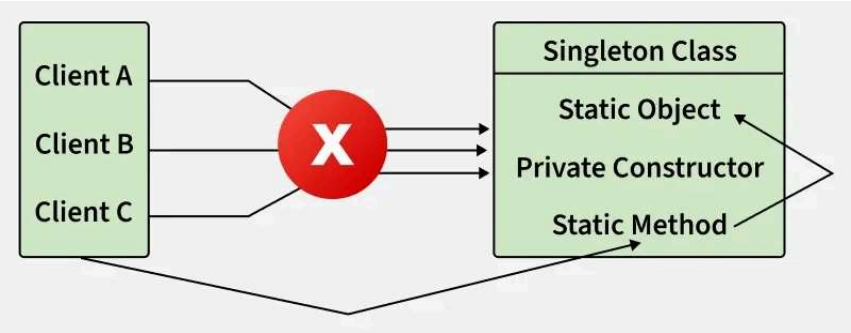
MVC Design Pattern

MVC separates an application into Model, View, and Controller for clean architecture. It improves scalability, maintainability, and parallel development in large systems

---

Singleton pattern

The Singleton Design Pattern ensures that a class has only one instance and provides a global access point to it. It is used when we want centralized control of resources, such as managing database connections, configuration settings or logging.



Real-World Applications of the Singleton Pattern

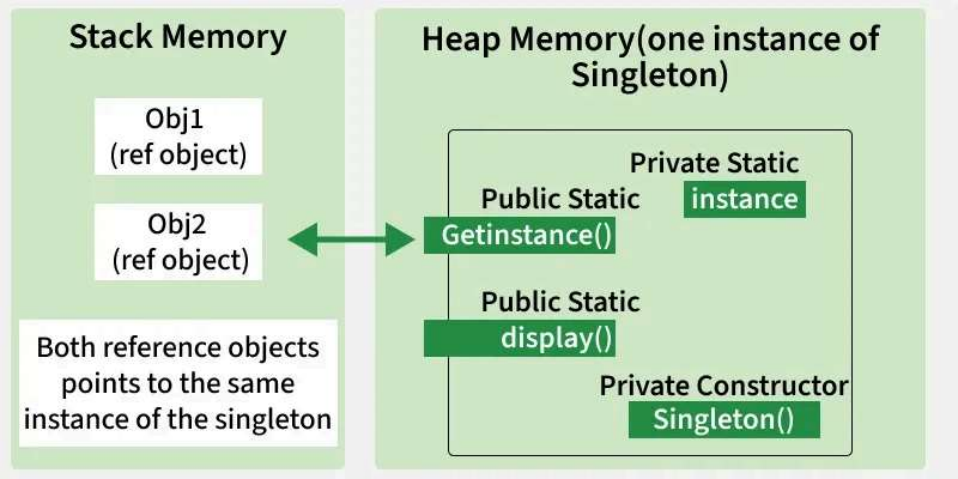
1. **Logging Systems :** Maintain a consistent logging mechanism across an application.
2. **Configuration Managers :** Centralize access to configuration settings.
3. **Database Connections :** Manage a single point of database access.
4. **Thread Pools :** Efficiently manage a pool of threads for concurrent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

Features of the Singleton Design Pattern

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.

Key Components

Below are the main key components of Singleton Method Design Pattern:



Key Components

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()
   {
   }
}
```

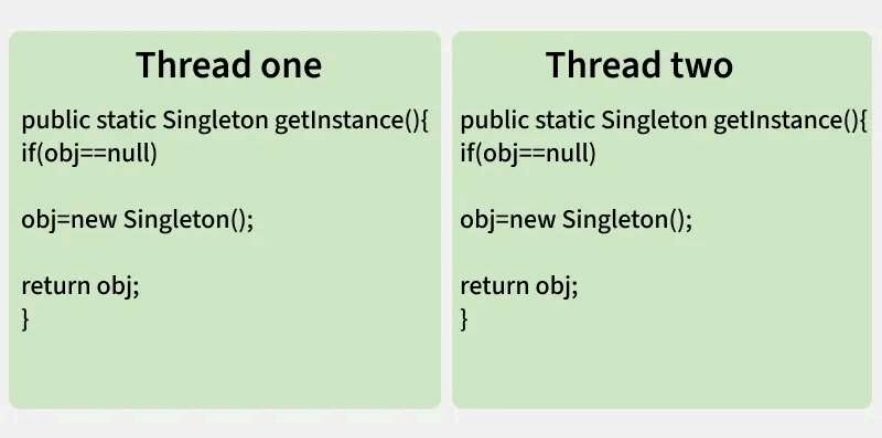
3. Static Factory Method

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 SingletonInner{
    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 SingletonInner 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.

Example: 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.

Factory Method Pattern

The Factory Method is a creational design pattern that defines an interface for creating objects but lets subclasses decide which object to instantiate. It promotes loose coupling by delegating object creation to a method, making the system more flexible and extensible.

- 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.

Features of Factory Method Design Pattern

- 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 behaviors.

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 = pClient.getVehicle();
    if (pVehicle != null) {
      pVehicle.printVehicle();
    }
    pClient.cleanup();
```

```
    }
}
```

Output

I am two wheeler

Issues with the Current Design

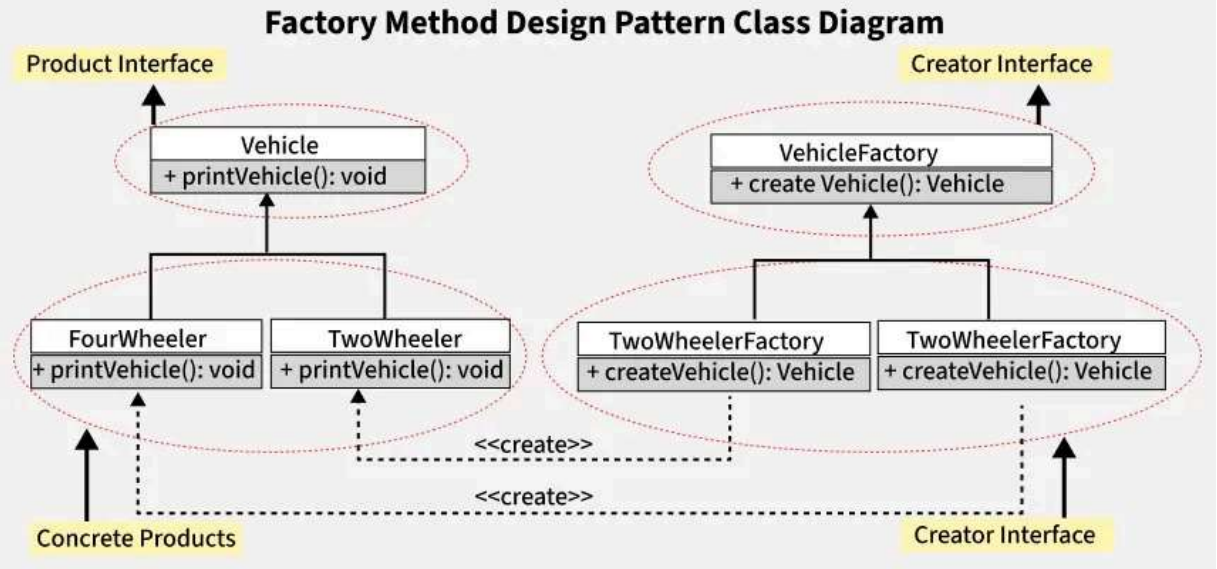
- Tight coupling: Client depends directly on product classes.
- Violation of SRP: Client handles both product creation and usage.
- Hard to extend: Adding a new vehicle requires modifying the client.

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(twoWheelerFactory);
      Vehicle twoWheeler = twoWheelerClient.getVehicle();
      twoWheeler.printVehicle();
      VehicleFactory fourWheelerFactory = new FourWheelerFactory();
```

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

Output

I am two wheeler

I am four wheeler

MODEL VIEW CONTROLLER

The MVC design pattern is a software architecture pattern that separates an application into three main components: Model, View, and Controller, making it easier to manage and maintain the codebase. It also allows for the reusability of components and promotes a more modular approach to software development.

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.

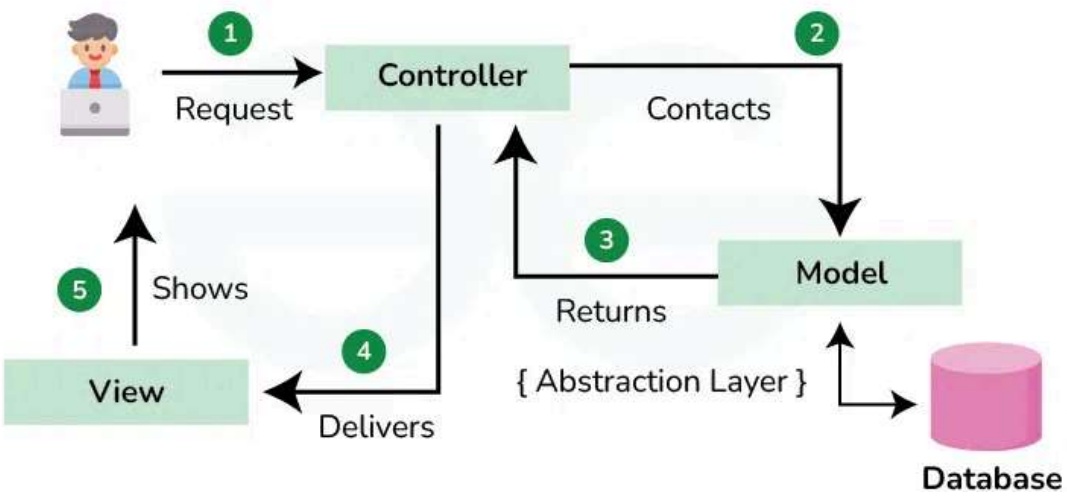
- The MVC pattern separates the concerns of an application into three distinct components, each responsible for a specific aspect of the application's functionality.
- This separation of concerns makes the application easier to maintain and extend, as changes to one component do not require changes to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

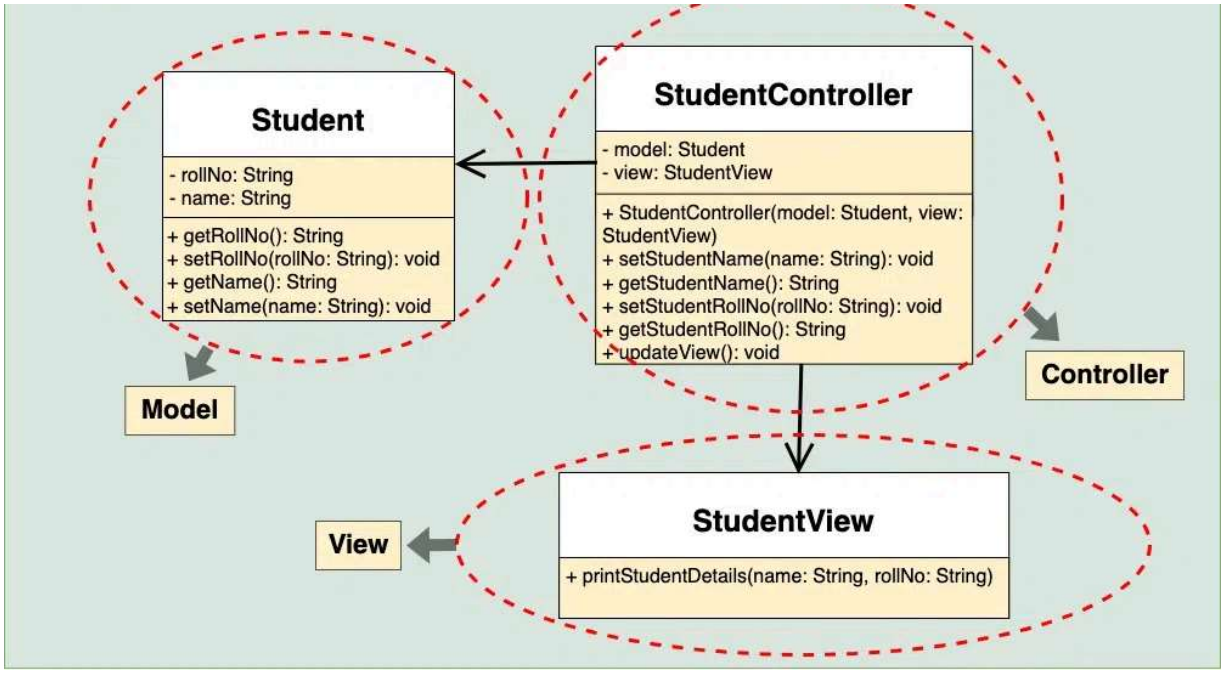
Communication between the Components

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.

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 setRollNo(String rollNo) {
     this.rollNo = 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 (**printStudentDetails**) 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**, **setStudentRollNo**) and to update the View (**updateView**).

```
class StudentController {
  private Student model;
  private StudentView view;
  public StudentController(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 setStudentRollNo(String rollNo) {
     model.setRollNo(rollNo);
  }
  public String getStudentRollNo() {
     return model.getRollNo();
  }
  public void updateView() {
     view.printStudentDetails(model.getName(), model.getRollNo());
  }
}
```

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.rollNo = 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 StudentController(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 setStudentRollNo(String rollNo) {
      model.setRollNo(rollNo);
  }
  public String getStudentRollNo() {
      return model.getRollNo();
  }
  public void updateView() {
      view.printStudentDetails(model.getName(), model.getRollNo());
  }
}
```

```
public class MVCPattern {
   public static void main(String[] args) {
      Student model = retriveStudentFromDatabase();
      StudentView view = new StudentView();
      StudentController controller = new StudentController(model, view);
      controller.updateView();
      controller.setStudentName("Vikram");
      controller.updateView();
   }
   private static Student retriveStudentFromDatabase() {
      Student student = new Student();
      student.setName("Lokesh");
      student.setRollNo("15UCS157");
      return student;
   }
}
```

```
Student:
Name: Lokesh
Roll No: 15UCS157
Student:
Name: Vikram
Roll No: 15UCS157
```
