</

Design Pattern

Presented by Parsa Attaran

/>

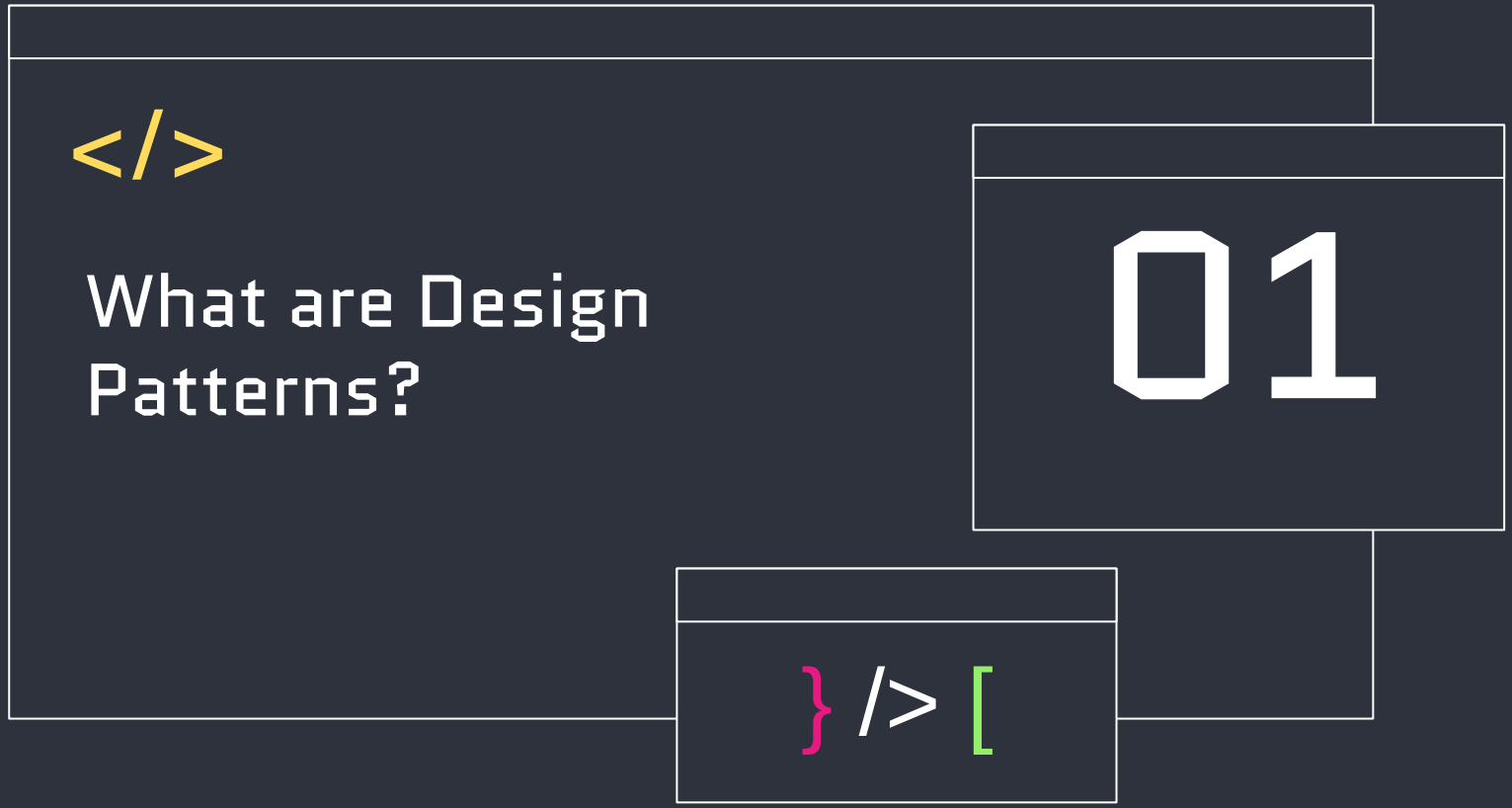
} /> [

AP Spring 1404
Dr.Mojtaba Vahidi Asl

1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1

</ Table of contents

1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

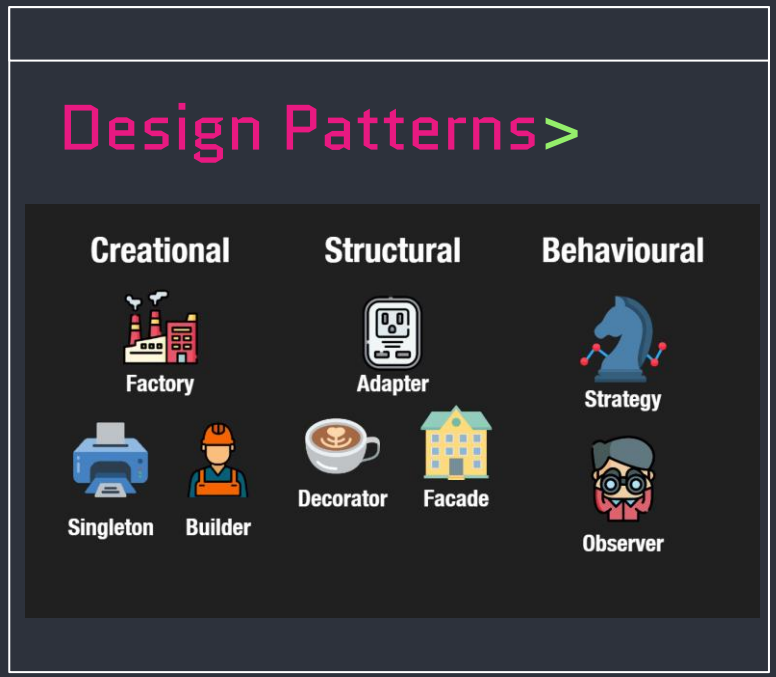


What are Design Patterns?

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 0 1 1 0 1

</ What are Design Patterns?

- Design patterns are structured approaches using classes and objects that solve recurring design problems within specific contexts.
- They provide reusable, generalized solutions to common software development challenges, representing best practices established by experience.
- Applying design patterns helps developers communicate ideas and solutions more clearly, improving collaboration and coding consistency.



1 0 1 1 0 1 1 0 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1

</ Advantages of Design Patterns

Reusability

Avoid reinventing the wheel each time.

Maintainability

Easier to modify and debug code.

Scalability

Design flexible and adaptable software.

Standardization

Common vocabulary and structure across different projects.

Collaboration

Easier for multiple developers to work on the same codebase.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</>

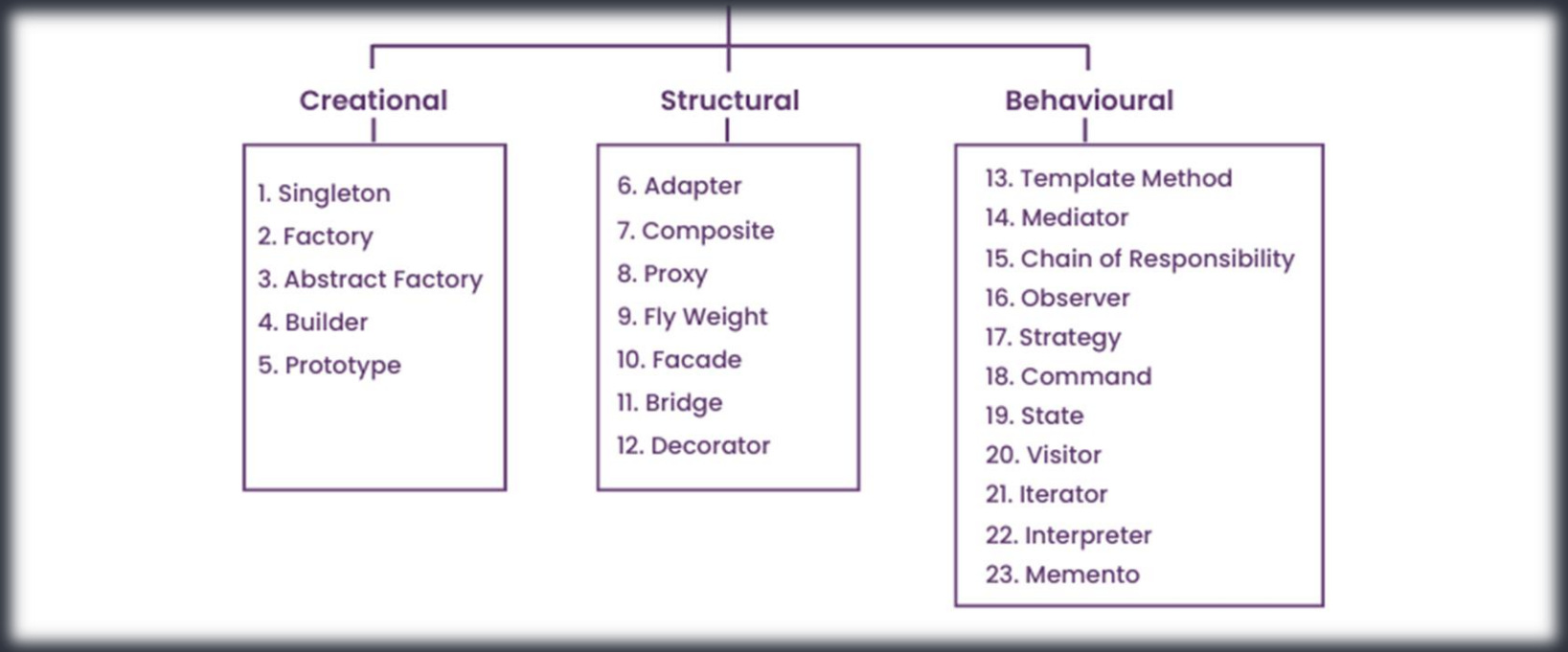
Types of Design patterns

02

} /> [

1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1

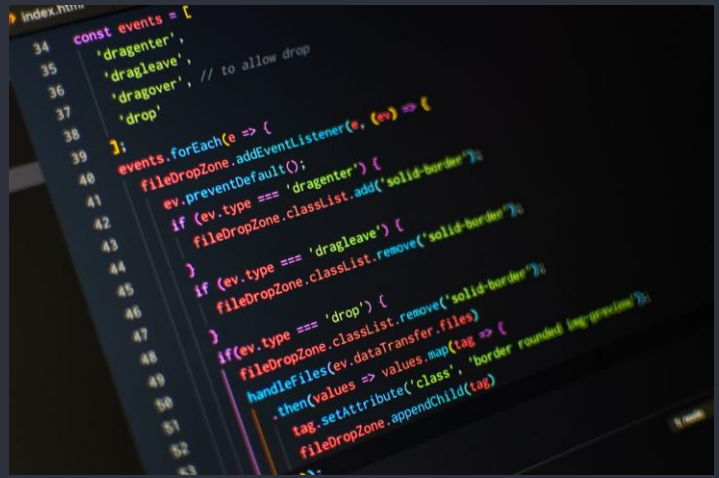
</ Different Types of Design Patterns



</ Creational

Creational design patterns focus on efficient and flexible object creation. They decouple the system from the process of instantiating objects, making the code more maintainable, adaptable and independent of how objects are constructed or represented.

Creational />



1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

</ introduction to creational design patterns methods

Factory

Defines an interface for creating objects but lets subclasses decide which class to instantiate.

Abstract Factory

Provides an interface to create families of related objects without specifying their concrete classes.

Builder

Separates the construction of a complex object from its representation, allowing step-by-step creation.

Singleton

Ensures that only one instance of a class exists and provides a global point of access.

Prototype

Creates new objects by cloning existing instances instead of instantiating new ones.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Structural

- Focus on organizing and connecting classes/objects so they operate as larger, more flexible systems.
- Emphasize relationships and dependencies over object creation.
- Clear links improve code reuse, extensibility, and comprehension of the program’s architecture.
- Because components are well-structured and loosely coupled, the system is easier to maintain, modify, and scale.

Structural />

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Introduction to structual design patterns methods

Composite

Treats individual objects and groups of objects uniformly in a tree structure.

Decorator

Dynamically adds new responsibilities to objects without modifying their code.

Facade

Provides a simplified interface to a complex subsystem.

Adapter

Converts one interface into another to make incompatible classes work together.

Bridge

Decouples abstraction from implementation so they can vary independently

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Behavioral

- Focus on how objects/classes communicate and collaborate.
- Clarify responsibility: which object handles which task.
- Structure collaboration among multiple objects to complete workflows.
- Improve changeability and extensibility without large rewrites.
- Analogy: like a team where each member has a role and clear coordination to work smoothly.

Behavioral />

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Introduction to behavioral design patterns methods

Observer

Defines a one-to-many dependency so when one object changes, dependents are notified.

Strategy

Defines a family of algorithms and makes them interchangeable at runtime.

Command

Encapsulates a request as an object to parameterize clients and support undo/redo.

Template

Defines the skeleton of an algorithm, letting subclasses override specific steps.

Chain Of Responsibility

Passes a request along a chain of handlers until one processes it.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</>

Singleton Design Pattern

03

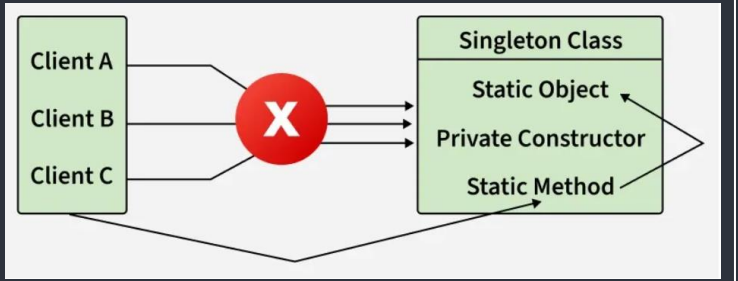
} /> [

1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1

</ What is singleton design pattern?

- Ensures a class has only one instance and offers a global access point.
- Used for centralized resource control e.g., database, configuration, logging.
- Core principles:• Maintain a single instance only. Provide easy global access to it.• Restrict instantiation (e.g., hide constructors).
- Concept name comes from mathematical “singleton”, meaning a single unique element.

Singleton>



1 0 1 1 0 1 1 0 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Features of the Singleton design pattern

Single Instance

Ensures only one object of the class exists in the classloader.

Thread Safety

Can be designed to work correctly in multithreaded environments.

Global Access Point

Provides a centralized way to access the instance.

Resource Management

Useful for managing shared resources like configurations, logging or database connections.

Lazy or Eager Initialization

An Instance can be created at class load time (eager) or when first needed (lazy).

Flexibility in Implementation

Can be implemented using eager initialization, lazy initialization, double-checked locking or an inner static class.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ When would you like to use singleton?

</ **

Ensuring a Single Instance

Choose Singleton when you need to guarantee only one instance and allow future subclassing.
This approach lets you replace or enhance the implementation later without changing the interface, ensuring that client code continues to work without modification.

} /> [

Supporting Future Extension

Use it to guarantee one unique instance across the program when shared state or resource management is required.

/> **

Using It with Caution

Apply Singleton sparingly; excessive use leads to hidden dependencies, tight coupling, and poor testability.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ 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.

} /> [

```
// Static member to hold the single instance
private static Singleton instance;
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Key Components

2- Private Constructor

- Uses a private constructor to block external instantiation of the class.
- Guarantees the class retains full control over object creation.
- Ensures only the intended single instance can ever be created.

```
// Private constructor to prevent external instantiation
class Singleton {

    // Making the constructor as Private
    private Singleton()
    {
        // Initialization code here
    }
}
```

} /> [

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Key Components

3-Static Factory Method

- The static factory method gives a global access point to the Singleton object.
- When requested, it creates a new instance if none exists, or returns the existing one.
- This method controls instance creation and maintains the “single object” rule of the pattern.

```
// Static factory method for global access
public static Singleton getInstance()
{
    // Check if an instance exists
    if (instance == null) {
        // If no instance exists, create one
        instance = new Singleton();
    }
    // Return the existing instance
    return instance;
}
```

</ **

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ How to implement Singleton

Classic (Lazy Initialization)

In this method, the class instance is not created at the time of class loading.

Instead, it is initialized only when it is first needed – typically on the first call to the getInstance() method.

The main advantage is that resources are only allocated if the instance is actually used.

Its drawback is that the classic implementation is not thread-safe, so multiple threads could create multiple instances unless additional synchronization is added.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ How to implement Singleton

Classic (Lazy Initialization)

```
class Singleton {
    private static Singleton obj;

    // private constructor to force use of getInstance() to create
Singleton object
    private Singleton() {}

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

Note: Singleton obj is not created until we need it and call the getInstance() method. This is called lazy instantiation. The main problem with the above method is that it is not thread-safe. Consider the following execution sequence.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ How to implement Singleton

Eager Initialization (Static Block)

In this method, class 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.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ How to implement Singleton

Eager Initialization (Static Block)

We create 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.

```
class Singleton {
    private static Singleton obj = new Singleton();
    private Singleton() {}

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

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ How to implement Singleton

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.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ How to implement Singleton

Static Inner Class (Best Java-Specific Way)

```
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;
    }
}
```

A private static inner class holds the Singleton instance.
The getInstance() method accesses that inner class's field.
Since the inner class loads only upon first access, the INSTANCE is created lazily and exactly once.
Being static, the instance remains unique and thread-safe throughout the program.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</>

MVC Design Pattern

04

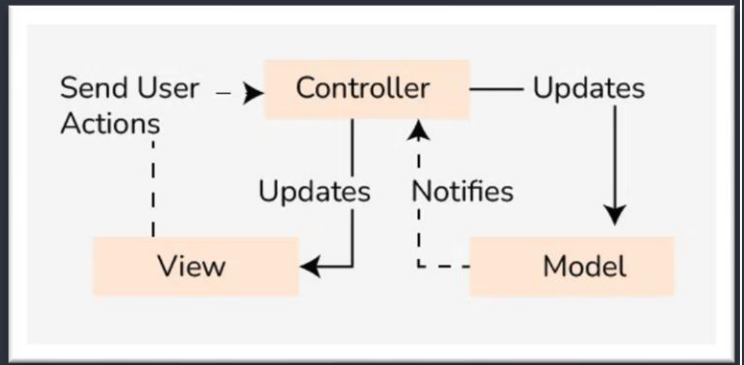
} /> [

1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1

</ What is MVC design pattern?

- The MVC pattern divides an application into Model, View, and Controller — each handling data, presentation, and control logic separately.
- This separation of concerns makes the system simpler to maintain and extend, since updating one part doesn't affect the others.
- In essence, MVC provides a structured architecture that enhances clarity, scalability, and flexibility.

MVC>: software architecture pattern

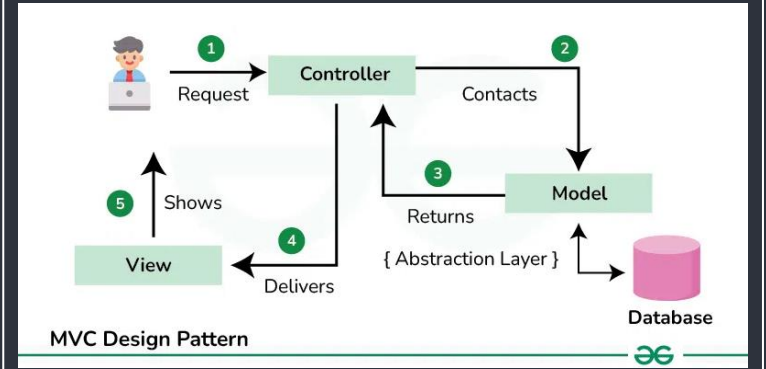


1 0 1 1 0 1 1 0 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1

</ Why use MVC design pattern?

- MVC divides an application into Model (data), View (UI), and Controller (logic/connection).
- This separation lets developers work on components independently, fix or extend features without breaking others.
- It simplifies testing, supports smooth feature addition, and produces clean, maintainable code.
- Overall, MVC improves organization, flexibility, and software quality.

MVC>



1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1

</ Components of the MVC Design Pattern

</ **

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.

} /> [

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.

/> **

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.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Communication between the Components

- **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.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Communication between the Components

- **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.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Several design patterns for implementing the MVC model

MVC Component	Related GoF Design Patterns	Purpose
Model	Observer, Strategy	Data changes notify views automatically; flexible business logic.
View	Composite, Decorator	Build complex UIs from smaller reusable parts.
Controller	Command, Strategy, Facade (conceptually)	Handle user actions and coordinate between model and view.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</>

MVC Design Pattern Simple Example

05

} /> [

1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1

</ Online Bookstore with MVC Pattern

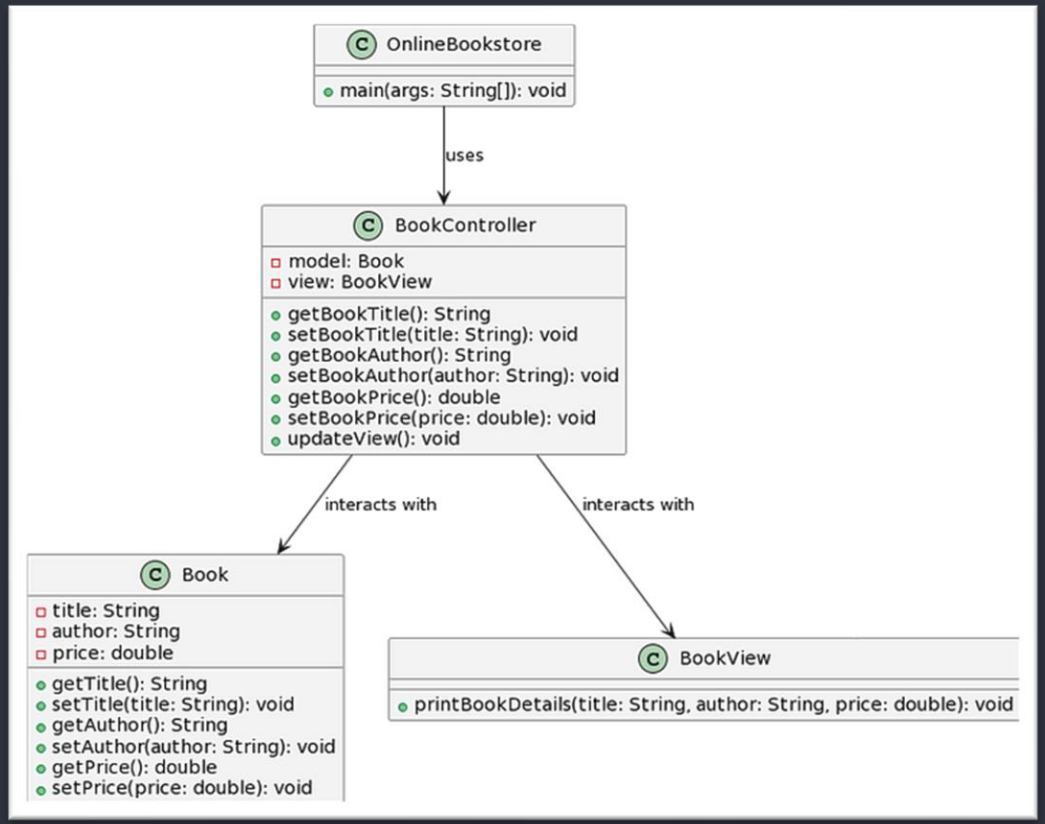
In this real-time use case, we will create a simplified online bookstore application using the MVC pattern in Java. The application will display a list of books and their details. We will have the following components:

1. Model: `Book.java`
2. View: `BookView.java`
3. Controller: `BookController.java`
4. Main class to run the application: `OnlineBookstore.java`

Online Bookstore >

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Problem solution UML



1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1

</ Model
Book.java

```java
public class Book {
    private String title;
    private String author;
    private double price;

    public Book(String title, String author, double price) {
        this.title = title;
        this.author = author;
        this.price = price;
    }

    // Getters and Setters
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author = author; }
    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price; }
}
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ View

BookView.java

```java
public class BookView {
    public void printBookDetails(String title, String author, double price){
        System.out.println("Book: ");
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        System.out.println("Price: $" + price);
    }
}
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Controller

BookController.java

```java
public class BookController {
    private Book model;
    private BookView view;

    public BookController(Book model, BookView view) {
        this.model = model;
        this.view = view;
    }

    // Methods to interact with the model
    public String getBookTitle() { return model.getTitle(); }
    public void setBookTitle(String title) { model.setTitle(title); }
    public String getBookAuthor() { return model.getAuthor(); }
    public void setBookAuthor(String author) { model.setAuthor(author); }
    public double getBookPrice() { return model.getPrice(); }
    public void setBookPrice(double price) { model.setPrice(price); }

    // Method to update the view
    public void updateView() {
        view.printBookDetails(model.getTitle(), model.getAuthor(), model.getPrice());
    }
}
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Main class

OnlineBookstore.java

```java
public class OnlineBookstore {
    public static void main(String[] args) {
        // Create a book instance
        Book book = new Book("Effective Java", "Joshua Bloch", 39.99);

        // Create a book view instance
        BookView view = new BookView();

        // Create a book controller instance
        BookController controller = new BookController(book, view);

        // Display the initial book details
        controller.updateView();

        // Update the book details and display the updated details
        controller.setBookTitle("Clean Code");
        controller.setBookAuthor("Robert C. Martin");
        controller.setBookPrice(34.99);
        controller.updateView();
    }
}
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</>

When to use or not to use the MVC Design Pattern

06

} /> [

1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1

</ When to Use the MVC Design Pattern

Complex Applications

Use MVC for apps with many features and user interactions, like e-commerce sites. It helps organize code and manage complexity.

Frequent UI Changes

If the UI needs regular updates, MVC allows changes to the View without affecting the underlying logic.

Reusability of Component

If you want to reuse parts of your app in other projects, MVC's modular structure makes this easier.

Testing Requirements

MVC supports thorough testing, allowing you to test each component separately for better quality control.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ When not to Use the MVC Design Pattern

Simple Applications

For small apps with limited functionality, MVC can add unnecessary complexity. A simpler approach may be better

Real-time Applications

MVC may not work well for apps that require immediate updates, like online games or chat apps.

Tightly Coupled UI and Logic

If the UI and business logic are closely linked, MVC might complicate things further.

Limited Resources

For small teams or those unfamiliar with MVC, simpler designs can lead to faster development and fewer issues.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Thanks!

Do you have any questions?

parsa.attaran84@gmail.com
+98 9350620344
@In_depthSci



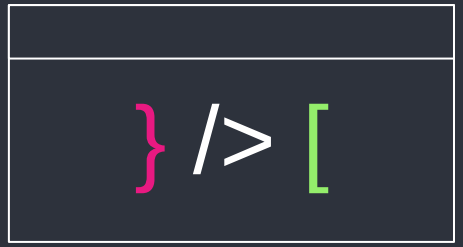
} /> [

/>

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon**, and infographics & images by **Freepik**

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 0 1

</ Resources



Geeksforgeeks.org

- Java Design Patterns tutorial
- Singleton Method Design Pattern
- MVC Design Pattern

Dev.to

- Design Patterns and their benefits

medium.com

- 6.1. Model-View-Controller (MVC)

Wikipedia.org

- Design Patterns
- Singleton

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1