

UNIT – IV

Interactive systems and the MVC architecture: Introduction The MVC architectural pattern, analyzing a simple drawing program designing the system, designing of the subsystems, getting into implementation, implementing undo operation drawing incomplete items, adding a new feature pattern based solutions.

4.1 Introduction The MVC architectural pattern:

The Model-View-Controller (MVC) framework is an architectural/design pattern that separates an application into three main logical components **Model**, **View**, and **Controller**. Each architectural component is built to handle specific development aspects of an application. It isolates the business logic and presentation layer from each other. It was traditionally used for desktop **graphical user interfaces (GUIs)**. Nowadays, MVC is one of the most frequently used industry-standard web development frameworks to create scalable and extensible projects. It is also used for designing mobile apps.

MVC was created by Trygve Reenskaug. The main goal of this design pattern was to solve the problem of users controlling a large and complex data set by splitting a large application into specific sections that all have their own purpose.

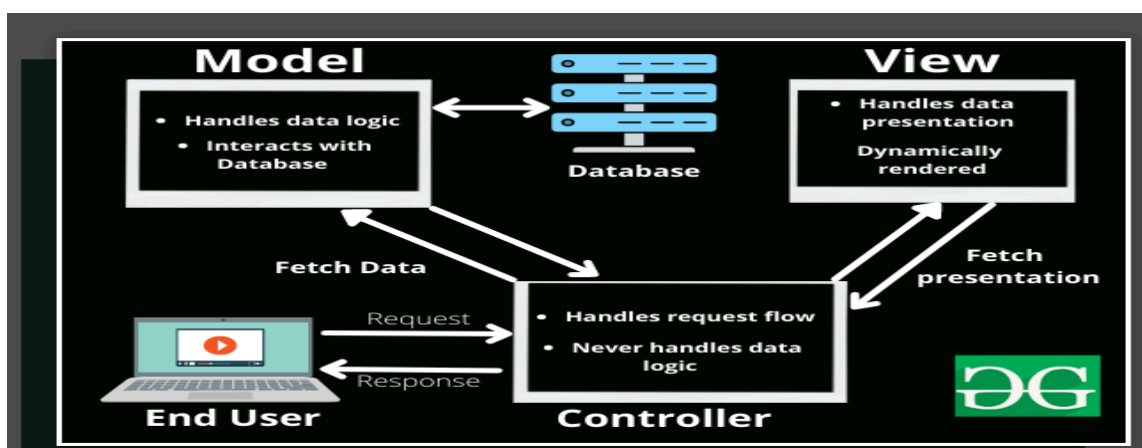
Features of MVC

- It provides a clear separation of **business logic, UI logic, and input logic**.
- It offers full control over your HTML and URLs which makes it easy to design web application architecture.
- It is a powerful URL-mapping component using which we can build applications that have comprehensible and searchable URLs.
- It supports Test Driven Development (TDD).

Components of MVC

The MVC framework includes the following 3 components:

- Controller
- Model
- View



Controller:

The controller is the component that enables the interconnection between the views and the model so it acts as an intermediary. The controller doesn't have to worry about handling data logic, it just tells

the model what to do. It processes all the business logic and incoming requests, manipulates data using the **Model** component, and interact with the **View** to render the final output.

Responsibilities:

- Receiving user input and interpreting it.
- Updating the Model based on user actions.
- Selecting and displaying the appropriate View.

Example: In a bookstore application, the Controller would handle actions such as searching for a book, adding a book to the cart, or checking out.

View:

The **View** component is used for all the UI logic of the application. It generates a user interface for the user. Views are created by the data which is collected by the model component but these data aren't taken directly but through the controller. It only interacts with the controller.

Responsibilities:

- Rendering data to the user in a specific format.
- Displaying the user interface elements.
- Updating the display when the Model changes.

Example: In a bookstore application, the View would display the list of books, book details, and provide input fields for searching or filtering books.

Model:

The **Model** component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. It can add or retrieve data from the database. It responds to the controller's request because the controller can't interact with the database by itself. The model interacts with the database and gives the required data back to the controller.

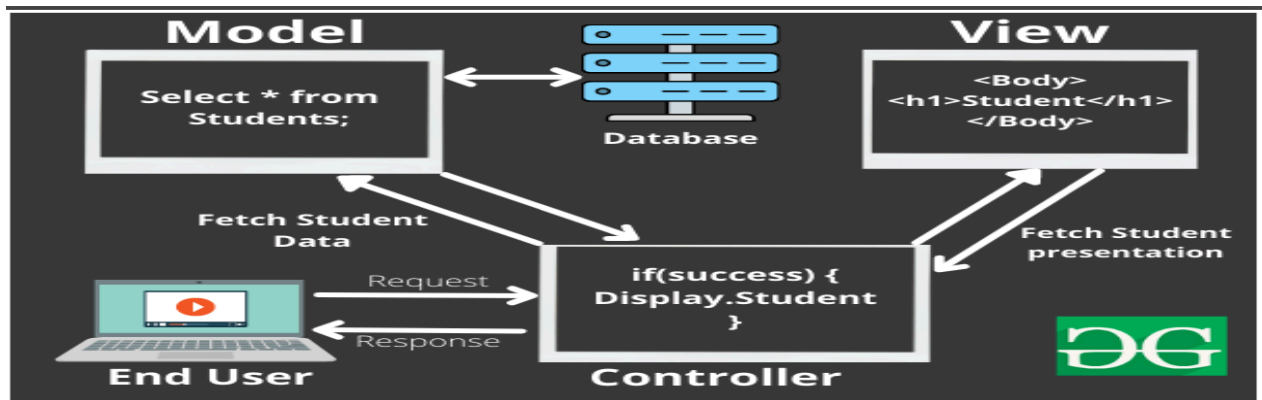
Responsibilities:

- Managing data: CRUD (Create, Read, Update, Delete) operations.
- Enforcing business rules.
- Notifying the View and Controller of state changes.

Example: In a bookstore application, the Model would handle data related to books, such as the book title, author, price, and stock level.

Working of the MVC framework with Example

Let's imagine an end-user sends a request to a server to get a list of students studying in a class. The server would then send that request to that particular controller that handles students. That controller would then request the model that handles students to return a list of all students studying in a class.



The model would query the database for the list of all students and then return that list back to the controller. If the response back from the model was successful, then the controller would ask the view associated with students to return a presentation of the list of students. This view would take the list of students from the controller and render the list into HTML that can be used by the browser. The controller would then take that presentation and returns it back to the user. Thus ending the request. If earlier the model returned an error, the controller would handle that error by asking the view that handles errors to render a presentation for that particular error. That error presentation would then be returned to the user instead of the student list presentation.

As we can see from the above example, the model handles all of the data. The view handles all of the presentations and the controller just tells the model and view of what to do. This is the basic architecture and working of the MVC framework.

The MVC architectural pattern allows us to adhere to the following design principles:

1. **Divide and conquer.** The three components can be somewhat independently designed.
2. **Increase cohesion.** The components have stronger layer cohesion than if the view and controller were together in a single UI layer.
3. **Reduce coupling.** The communication channels between the three components are minimal and easy to find.
4. **Increase reuse.** The view and controller normally make extensive use of reusable components for various kinds of UI controls. The UI, however, will become application-specific, therefore it will not be easily reusable.
5. **Design for flexibility.** It is usually quite easy to change the UI by changing the view, the controller, or both.

Advantages of MVC

- Codes are easy to maintain and they can be extended easily.
- The MVC **model** component can be tested separately.
- The components of MVC can be developed simultaneously.
- It reduces complexity by dividing an application into three units. **Model, view, and controller.**
- It supports **Test Driven Development (TDD)**.
- It works well for Web apps that are supported by large teams of web designers and developers.
- This architecture helps to test components independently as all classes and objects are independent of each other
- **Search Engine Optimization (SEO)** Friendly.

Disadvantages of MVC

- It is difficult to read, change, test, and reuse this model
- It is not suitable for building small applications.
- The inefficiency of data access in view.
- The framework navigation can be complex as it introduces new layers of abstraction which requires users to adapt to the decomposition criteria of MVC.
- Increased complexity and Inefficiency of data

4.2 Analyzing a simple drawing program designing the system

Designing a program is an important step in software development, as it determines how the application will function and interact with users. Understanding program requirements is fundamental, including identifying inputs, processing, and outputs (through the IPO model). Techniques like pseudocode help outline logic before coding begins. Flowcharts visually represent control flow while structure charts break down the components of a program. Additionally, Entity-Relationship Diagrams (ERD) model data relations, and UML diagrams show system design clearly. Use case diagrams capture functional interactions, state diagrams illustrate object states, and activity diagrams visualize workflows. All these techniques ensure effective communication throughout the project.

1. Understanding the Program Requirements
2. Using Pseudocode for Planning
3. Creating Flowcharts for Clarity
4. Developing Hierarchy or Structure Charts
5. Modeling with Entity-Relationship Diagrams
6. Utilizing Unified Modeling Language Diagrams
7. Constructing Use Case Diagrams
8. Designing State Diagrams for Objects
9. Representing Workflows with Activity Diagrams
10. Illustrating Components with Component Diagrams
11. Showing Deployment with Deployment Diagrams
12. Planning Testing with Evaluation Diagrams
13. Documenting Architecture with Diagrams
14. Iterative Design and Prototyping Process

A **simple drawing program** allows users to create and manipulate graphical objects such as lines, rectangles, circles, and freehand drawings. Before implementation, the system is analyzed by identifying its **requirements, objects, interactions, and responsibilities**.

. System Requirements

- Create graphical objects.
- Select and edit existing objects.
- Move, resize, and delete objects.
- Save and load drawings.
- Support **Undo/Redo** operations.
- Provide immediate visual feedback during drawing.

Drawing Maintains all graphical objects

Shape Common abstraction for drawing objects

Line Stores and displays a line

Rectangle Stores and displays a rectangle

Ellipse Stores and displays an ellipse

Point Represents coordinates

Tool

Defines the
current drawing
operation

3. MVC-Based System Design

Model

- Drawing
- Shapes
- Selection
- Drawing state
- Undo/Redo information

View

- Drawing canvas
- Tool bar
- Menus
- Shape visualization

Controller

- Handles mouse and keyboard events
- Determines the selected tool
- Creates or modifies shapes
- Invokes commands

4. Basic Interaction

User → Controller → Model → View

For example, when drawing a rectangle:

1. User selects **Rectangle Tool**.
2. User presses the mouse button on the canvas.
3. Controller records the starting point.
4. User drags the mouse.
5. View displays a temporary rectangle.
6. User releases the mouse.
7. Controller creates the completed rectangle.
8. Model stores the rectangle.
9. View redraws the updated drawing.

5. Important Design Issue: Incomplete Items

While the user is dragging the mouse, the rectangle is **incomplete**.

Mouse Down



Start Point



Mouse Move → Temporary Shape → View Feedback



Mouse Up



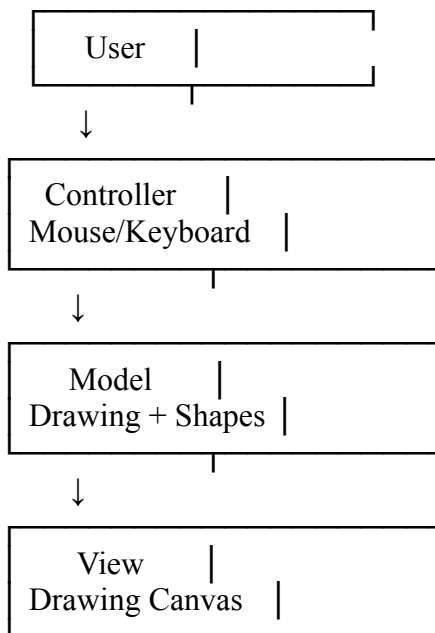
Completed Shape



Add to Drawing Model

The incomplete shape should generally **not be treated as a permanent drawing object or undoable operation** until the drawing action is completed.

6. Overall Architecture



Key design goal: keep the **drawing logic (Model)** independent from the **user interface (View)**, while the **Controller** coordinates user actions. This makes the drawing program easier to extend with new shapes, tools, and features such as Undo/Redo.

4.3 Designing of the Subsystems – Simple Drawing Program

After analyzing the drawing program, the next step is to divide the system into **smaller, manageable subsystems**. Each subsystem performs a specific responsibility and communicates with other subsystems through well-defined interfaces.

1. Drawing Model Subsystem

Responsible for maintaining the actual drawing data.

Responsibilities:

- Maintain the collection of shapes.
- Store properties such as position, size, and type.
- Add, remove, and modify shapes.

- Maintain selection information.
- Provide the current drawing state.

Examples: Drawing, Shape, Line, Rectangle, Ellipse

2. User Interface (View) Subsystem

Responsible for displaying the drawing and user-interface components.

Responsibilities:

- Display the drawing canvas.
- Display shapes on the screen.
- Show selected objects.
- Display menus, buttons, and toolbars.
- Provide visual feedback while drawing.

Examples: DrawingCanvas, Toolbar, Menu

3. Controller / Input Subsystem

Responsible for processing user actions.

Responsibilities:

- Handle mouse events.
- Handle keyboard events.
- Identify the selected drawing tool.
- Convert user actions into system operations.
- Coordinate changes between Model and View.

Example:

Mouse Down → Start Drawing

Mouse Move → Update Temporary Shape

Mouse Up → Complete Shape

4. Command Subsystem

Responsible for representing operations performed by the user.

Responsibilities:

- Create commands for drawing operations.
- Execute commands.
- Reverse commands for Undo.
- Maintain command history.

Examples:

CreateShapeCommand

MoveShapeCommand

DeleteShapeCommand

This subsystem is particularly useful for implementing **Undo/Redo**.

5. Shape / Tool Subsystem

Responsible for different types of drawing operations.

Shapes:

- Line

- Rectangle
- Ellipse
- Polygon
- Freehand drawing

Tools:

- Selection Tool
- Line Tool
- Rectangle Tool
- Ellipse Tool
- Eraser Tool

A common Shape interface can allow new shapes to be added without changing the entire system.

6. Persistence Subsystem

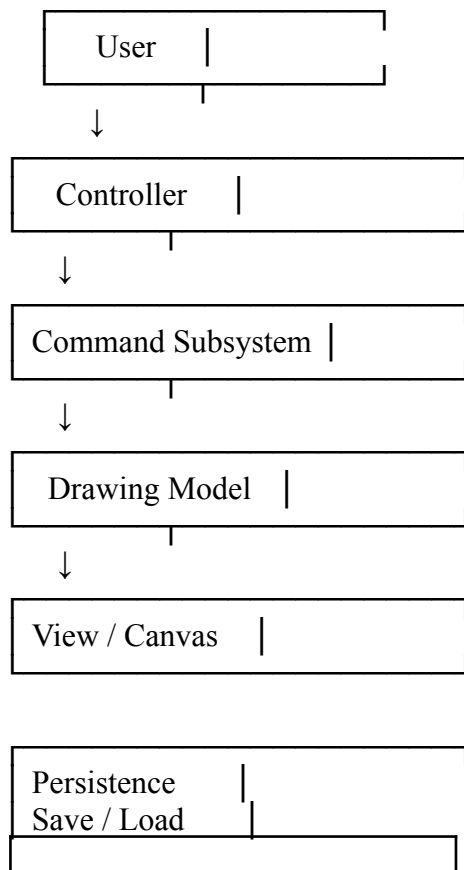
Responsible for saving and loading drawings.

Responsibilities:

- Save drawing data to a file.
- Load previously saved drawings.
- Convert shapes into a storable format.
- Restore the drawing when a file is opened.

Drawing → Save → File

File → Load → Drawing

7. Subsystem Interaction

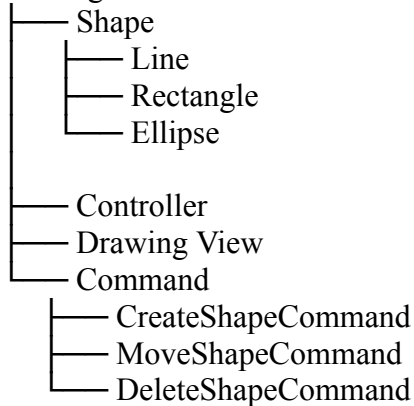
4.4 Getting into Implementation – Simple Drawing Program

After designing the system and its subsystems, the next step is **implementation**. At this stage, the identified classes, interfaces, and interactions are converted into actual program components.

1. Identify the Main Classes

The drawing application can be implemented using classes such as:

Drawing



2. Implement the Model

The **Model** maintains the drawing information.

Drawing

- shapes: List<Shape>

+ add Shape ()
 + remove Shape ()
 + get Shapes ()
 + clear ()

The Shape abstraction defines common operations:

Shape

+ draw ()
 + move ()
 + resize ()
 + contains ()

Specific shapes such as Line, Rectangle, and Ellipse implement these operations.

3. Implement the View

The **View** displays the model on the screen.

Drawing View

+ display Drawing ()
 + draw Shape ()
 + refresh ()
 + show Selection ()

Whenever the model changes, the view is refreshed so that the user sees the latest drawing.

4. Implement the Controller

The Controller handles user interactions.

Controller

+ mouse Pressed ()

+ mouse Dragged ()
 + mouse Released ()
 + key Pressed ()

For example:

Mouse Pressed



Record starting point



Mouse Dragged



Update temporary shape



Mouse Released



Create completed shape

5. Implement Incomplete Drawing Items

During mouse dragging, the shape is not yet complete.

Temporary Shape



Mouse Movement



Update Position/Size



Display Preview



Mouse Release



Commit Shape to Model

The temporary shape can be maintained separately until the user completes the operation.

6. Implement Commands

User operations can be represented as **Command objects**.

Command

+ execute ()

+ undo ()

For example:

CreateShapeCommand

+ execute () → Add shape

+ undo () → Remove shape

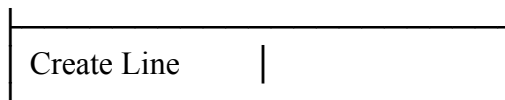
This structure makes it easier to implement **Undo and Redo**.

4.5 Implement Undo

A stack can maintain completed commands:

Command History

Delete Rectangle	← Latest
Move Circle	



When the user selects **Undo**:

Undo



Remove latest command



Call undo ()



Update Model



Refresh View

Implementation Sequence

A practical implementation order is:

Model → View → Controller → Commands → Undo/Redo → Save/Load → New Features

This incremental approach allows each subsystem to be developed and tested before integrating the complete drawing application.

4.6 Adding a New Feature – Pattern-Based Solutions:

Once the basic drawing program is implemented, new features may need to be added without disturbing the existing system. **Design patterns** help us extend the system in a structured and reusable way.

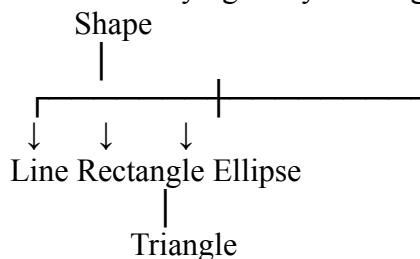
1. Example: Adding a New Shape

Suppose the existing drawing program supports:

- Line
- Rectangle
- Ellipse

Now we want to add a **Triangle**.

Instead of modifying many existing classes, we can use suitable design patterns.



The new Triangle class implements the existing Shape abstraction.

2. Factory Pattern – Creating New Shapes

A **Factory** can centralize the creation of shapes.

User selects Triangle



Shape Factory



Triangle Object



Drawing

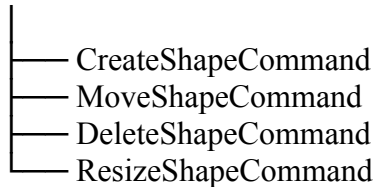
Advantages:

- Object creation is centralized.
- Controller does not need to know the detailed creation process.
- New shape types can be added more easily.

3. Command Pattern – Adding Operations

The **Command Pattern** represents user actions as objects.

Command



Each command can provide:

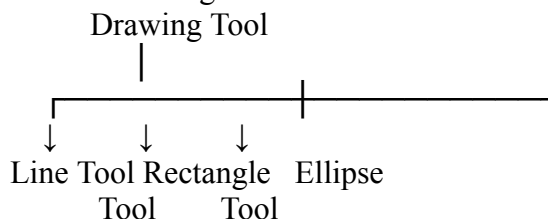
execute ()

undo ()

This makes new operations easier to integrate with **Undo/Redo**.

4. Strategy Pattern – Different Drawing Tools

Different drawing tools can be treated as interchangeable strategies.

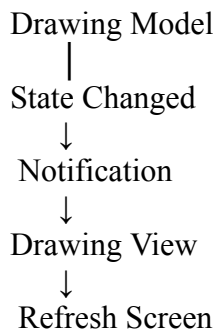


The Controller can select the appropriate strategy based on the user's choice.

Benefit: New drawing behaviors can be added without heavily modifying the Controller.

5. Observer Pattern – Updating the View

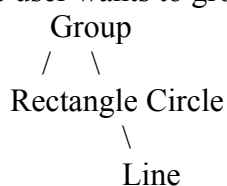
When the Model changes, the View needs to be updated.



The **Observer Pattern** can establish this relationship.

6. Composite Pattern – Grouping Shapes

If the user wants to group several shapes:

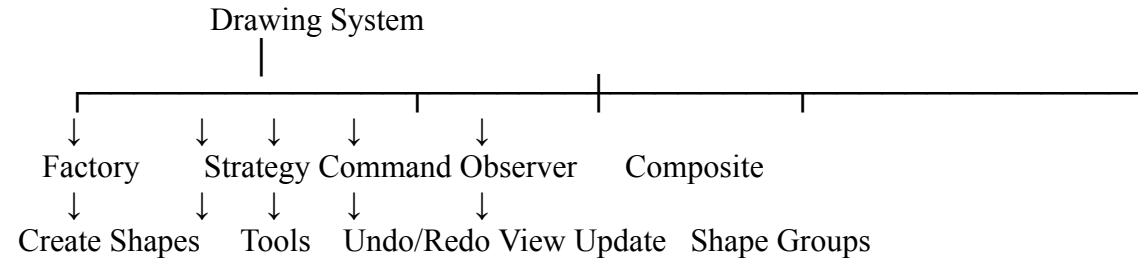


A **Composite Pattern** can treat a group of shapes as a single drawing object.

For example:

- Move the group
- Delete the group
- Resize the group
-

7. Pattern-Based Architecture



4.6 Benefits of Pattern-Based Solutions:

- **Extensibility:** New features can be added with fewer changes.
- **Reusability:** Existing design components can be reused.
- **Maintainability:** Responsibilities remain clearly separated.
- **Flexibility:** Different tools and behaviors can be exchanged.
- **Undo/Redo Support:** Commands provide a natural structure for reversible operations.
- **Reduced Coupling:** Classes depend more on abstractions than concrete implementations.