DEPARTMENT OF
COMPUTER SCIENCE

Software Design and Architecture

[Architecture Patterns] – Chapter 04, L03

Lecture Outlines

- Interactive Systems
  - Overview
  - Patterns:

    Model-View-Controller (MVC).
- Hierarchical Systems
  - Overview
  - Patterns:
    - Layered

Interactive Systems

- Interactive systems support user interactions, typically through user interfaces.
  - When designing these systems, two main quality attributes are of interest:
    - Usability
    - Modifiability
- The mainstream architectural pattern employed in most interactive systems is the Model-View-Controller (MVC).

Interactive Systems

- The MVC pattern is used in interactive applications that require flexible incorporation of human-computer interfaces. With the MVC, systems are decomposed into three main types of components:

Component	Description
Model	Component that represents the system's core, including its major processing capabilities and data.
View	Component that represents the output representation of the system (e.g., graphical output or console-based).
Controller	Component (associated with a view) that handles user inputs.

Extra:

Mobile App – Internal Architecture (MVC)

- **Model** → Manages the app's data (local storage, cached data, or data already fetched).
- **View** → User interface (screens, buttons, forms).
- **Controller** → Handles user input, communicates with services/APIs if needed, updates the Model and View.

Note: The **Controller (or service layer)** is usually responsible for calling the server, not the Model directly.

Mobile App – External Architecture (Distributed System)

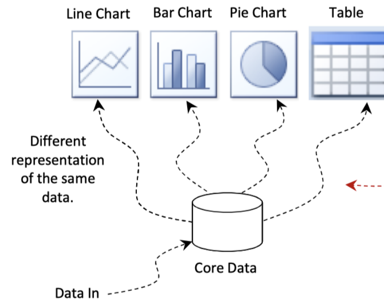
- **Client (Mobile App)** → Sends requests (login, get messages, check balance) through APIs.
- **Server** → Processes requests, applies business rules.
- **Database** → Stores persistent data (users, transactions, messages).

Summary:

- Inside the mobile app = **MVC** organizes the code (UI, logic, data).
- Outside the app = **Distributed System** (Client ↔ Server ↔ Database) enables communication and services.

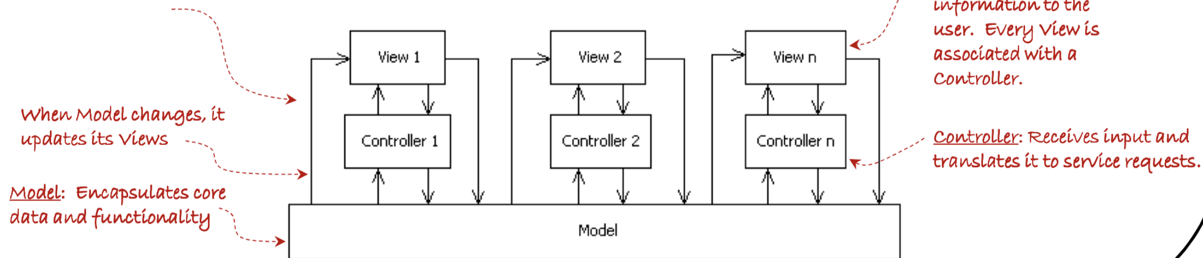
MODEL-VIEW-CONTROLLER ARCHITECTURAL PATTERN

Consider the popular example where data needs to be represented in different formats



When data changes, all views are updated to reflect the changes.

MODEL-VIEW-CONTROLLER ARCHITECTURAL PATTERN



Extra:

1.Model

1. Holds core data and system functionality.
2. When data changes, it updates the Views.

2.View

1. Displays information to the user.
2. Each View is linked with a Controller.

3.Controller

1. Receives user input.
2. Translates input into service requests for the Model.

4.Flow of interaction

1. User interacts with **Controller**.
2. Controller updates the **Model**.
3. Model notifies and updates the **View**.
4. View presents the updated information back to the user.

Example Scenario: User Updates Their Profile Name

1.User action (input)

1. The user types a new name and clicks **Save**.
2. This action goes to the **Controller**.

2.Controller

1. The Controller receives the input (new name).
2. It translates the input into a **request to update the Model**.

3.Model

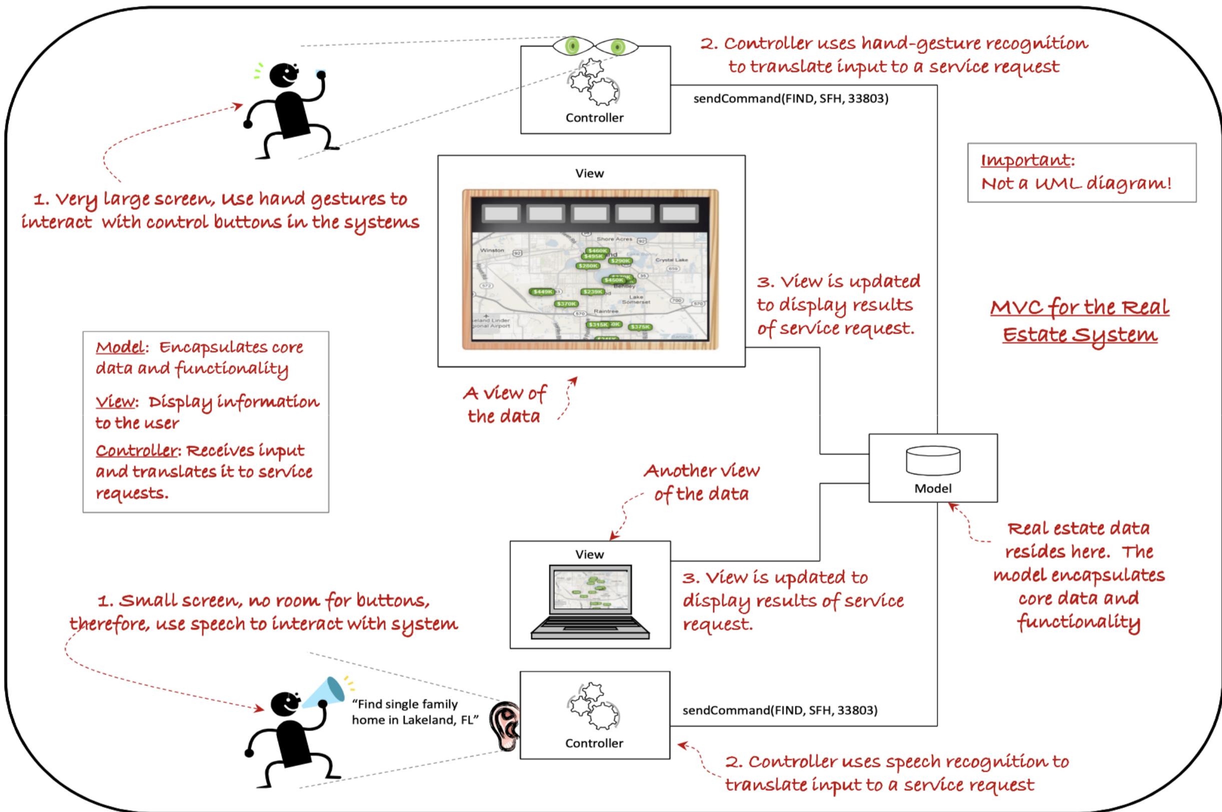
1. The Model updates the user's name in its data.
2. After updating, the Model **notifies the View** that data has changed.

4.View

1. The View retrieves the updated data from the Model.
2. It refreshes the screen to display the new name.

👉 In short: **Controller handles input** → **Model updates data** → **View shows changes**.

👉 **Flow summary**:
User → **Controller** → **Model** → **View** → **User** (cycle continues for next actions).



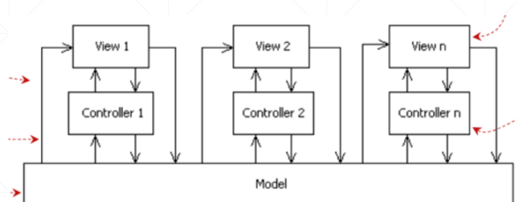
2. Controller uses hand-gesture recognition to translate input to a service request
Controller
sendCommand(FIND, SFH, 33803)
Important:
Not a UML diagram!
View
1. Very large screen, use hand gestures to interact with control buttons in the systems
3. View is updated to display results of service request.
MVC for the Real Estate System
Model: Encapsulates core data and functionality
View: Display information to the user
Controller: Receives input and translates it to service requests.
A view of the data
Another view of the data
Model
Real estate data resides here. The model encapsulates core data and functionality
View
3. View is updated to display results of service request.
1. Small screen, no room for buttons, therefore, use speech to interact with system
"Find single family home in Lakeland, FL"
Controller
sendCommand(FIND, SFH, 33803)
2. Controller uses speech recognition to translate input to a service request

Extra: Explanation of the System:

1. User
Large Screen: Uses hand gestures to navigate or control options.
Small Screen: Uses voice commands due to limited space.
2. Controller
Role: Connects View and Model.
Task: Converts user input (gestures/voice) into actions like fetching or filtering property data.
3. Model
Role: Stores real estate data (properties, prices, locations).
Task: Provides data to the View and updates based on Controller requests.
4. View
Role: Shows information to the user.
Task: Updates display dynamically based on user actions (gestures/voice).

👉 In short: **User** → **Controller** → **Model** → **View** → **User** (with gestures/voice as inputs).

MODEL-VIEW-CONTROLLER ARCHITECTURAL PATTERN



Extra:

1.RealEstateModel (Model)

Central data component storing real estate information.
Exposes **IObservable** for updates and **IModel** for data access.

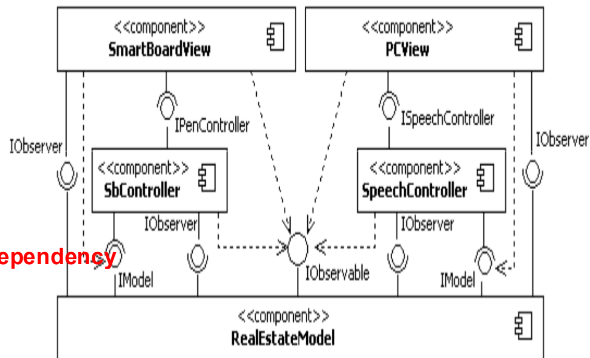
2.Controllers

SbController (SmartBoard Controller): Handles pen/gesture input via **IPenController**.
SpeechController: Handles voice commands via **ISpeechController**.
Both translate user input into actions on the **RealEstateModel** (through **IModel**).

3.Views

SmartBoardView: Displays info for large-screen (gesture/pen-based).
PCView: Displays info for smaller screen (voice-based).
Both register as **IObservers** to the **RealEstateModel**, so they update automatically when the Model changes.

Note:View requires the Model (to get the data).
But the Model also requires View (to send updates).
So, both **provide** and **require** exist at the same time because of the **two-way dependency created by the Observer pattern**.



4.Communication Flow

User interacts → Controller (pen or speech).
Controller updates the **RealEstateModel**.
Model notifies Views via **IObserver**.
Views refresh to display updated real estate data.

👉 In short: **User input** → **Controller** → **RealEstateModel** → **Views (auto-updated via Observer pattern).**

SpeechController Class Methods:

- processSpeech() (called when voice input is received)
- sendCommand(FIND_CMD, 5FH, 33803) (sends parsed command to model)

RealEstateModel Class Methods:

- processCommand() (processes the FIND command)
- update() (updates internal state)
- getResults() (retrieves search results)

PCView Class Methods:

•show() (called twice - displays the view)

•*Internally enables voice input mode* (not a method call, but internal state change)

Application Class Methods:

•new Model() (constructor)

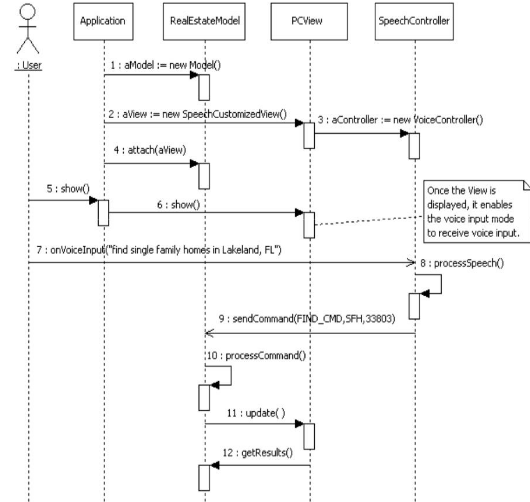
•new SpeechCustomizedViewModel() (constructor)

•new VoiceController() (constructor)

•attach(aView) (attaches view)

External/User:

•onVoiceInput("find single family homes in Lakeland, FL") (this is the user/system calling the controller's input handler)



MODEL-VIEW-CONTROLLER ARCHITECTURAL PATTERN

- Quality properties of the MVC architectural pattern include the ones specified below.

Quality	Description
Modifiability	Easy to exchange, enhance, or add additional user interfaces.
Usability	By allowing easy exchangeability of user interfaces, systems can be configured with different user interfaces to meet different usability needs of particular groups of customers.
Reusability	Be separating the concerns of the model, view, and controller components, the can all be reused in other systems.

- There are many variations of the MVC architectural pattern.
  - ✓ One popular variation includes the fusion of views and controller components, as made famous in the 1990s by Microsoft's Document-View architecture
  - ✓ Other more extensive variations include the process-abstraction-controller pattern.

Extra:
Modifiability - Easy to exchange or enhance user interfaces
Usability - Configure different UIs for different customer needs
Reusability - Model, View, and Controller components can be reused in other systems

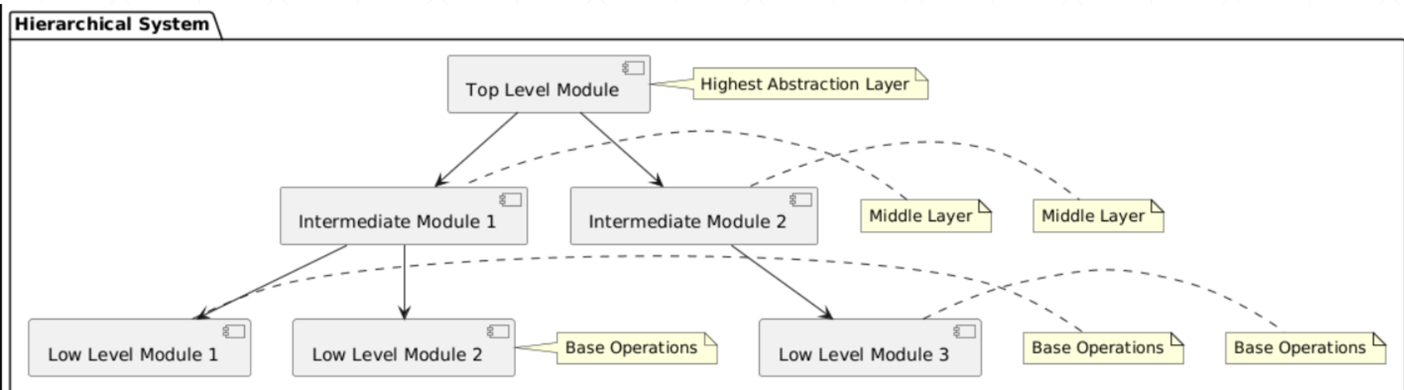
Hierarchical Systems

- Hierarchical systems can be decomposed and structured in hierarchical fashion. Two common architectural patterns for hierarchical systems are:
  - ✓ Main program and subroutine
  - ✓ Layered

Extra: Hierarchal Systems

•Description:
These systems are structured in layers, each layer having a specific role or responsibility, allowing for the separation of concerns across a vertical (layered) or horizontal (module) scale.

•Example:
•Operating systems where different layers handle tasks ranging from hardware interaction at the lowest level to user interface management at the highest level.



Layered Pattern

- Quality properties of the Layered architectural pattern include the ones specified below.

Quality	Description
Modifiability	Dependencies are kept local within layer components. Since components can only access other components through a well-defined and unified interface, the system can be modified easily by swapping layer components with other enhanced or new layer components.
Portability	Services that deal directly with platform's API's can be encapsulated using a system layer component. Higher level layers rely on this component for providing system services to the application, therefore, by porting the system's API layer to other platforms systems become more portable.
Security	The controlled hierarchical structure of layered systems allow for easy incorporation of security components to encrypt/decrypt incoming/outgoing data.
Reusability	By compartmentalizing each layer's services, they become easier to reuse.

Extra:
Modifiability - Easy to swap layer components through defined interfaces
Portability - Encapsulate platform APIs for easier cross-platform migration
Security - Easy to add security components in controlled hierarchical structure
Reusability - Compartmentalized layer services are easier to reuse

Layered Pattern

SatcomLayer Component

•**Purpose:** Handles satellite communication functions

•**Interface:** IJSatcom (provides satellite communication services to other layers)

SecurityLayer Component

•**Purpose:** Manages security and encryption functions

•**Interface:** IJSecure (provides security services, requires application services)

ApplicationLayer Component

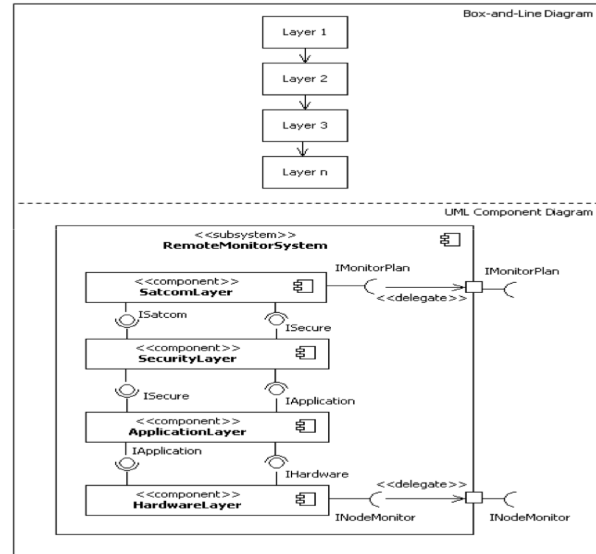
•**Purpose:** Contains main application logic and business rules

•**Interface:** IApplication (provides application services, requires hardware access)

HardwareLayer Component

•**Purpose:** Controls physical hardware devices and sensors

•**Interface:** IHardware (provides hardware access, requires monitoring services)



WHAT'S NEXT…

- • In this session, we continued the discussion on distributed systems and presented fundamentals concepts of interactive and hierarchical systems, together with essential architectural patterns for these systems, including:
  - ✓ MVC
  - ✓ Layered
- This finalizes our coverage of architectural patterns. In the next module, we start the discussion on detailed design, which is the next activity in the design process. Detailed design begins once the architecture of the software is sufficiently complete.