

OVERVIEW **Interactive and Hierarchical Systems: Model-View-Controller and Layered**

The last two system types. Interactive systems care most about usability and modifiability, and their mainstream pattern is Model-View-Controller: the model holds the data and core functionality, views display it, and controllers turn user input into requests on the model, which then notifies every view. Hierarchical systems are built in levels; in the layered pattern each layer provides services to the layer above through a defined interface and uses only the layer below, which gives modifiability, portability, security and reuse.

MVC roles

Model = data and functionality; View = output representation; Controller = paired with a view, handles input.

The MVC cycle

User → controller → model (updates) → notifies views → views refresh → user. Views observe the model.

MVC qualities

Modifiability (swap or add UIs), usability (different UIs for different users), reusability.

Layered

Each layer offers services upward through an interface; qualities are modifiability, portability, security, reusability.

HOW THE LECTURE IS ORGANIZED

Slides 2–5 Interactive systems and the three MVC components

Slides 6–12 MVC: structure, the real-estate example (component and sequence diagrams), qualities

Slides 13–16 Hierarchical systems and the layered pattern

Slide 17 What is next: detailed design

EXAM FOCUS

Final 252 Q4-1 (2.5 marks) asked you to recognise MVC in a children's app and draw its component diagram; Midterm 261 Q6 matched a museum-ticketing system to MVC and a payroll system to layered.

HOW TO READ THIS PAGE

Every original slide is shown unchanged. **Yellow marks** on a slide point to a numbered note: hover or tap a mark to jump to its note, and hover a note to see its marks. Questions tagged **Past exam** appeared on a real paper; **Prediction** questions are likely angles, not confirmed.



Software Design and Architecture

[Architecture Patterns] – Chapter 04, L03

NOTES

- 1 Final architecture-patterns lecture: the patterns for interactive systems (MVC) and hierarchical systems (layered). After this, the course moves from architecture to detailed design.

SLIDE 2 / 17 Lecture Outlines

Lecture Outlines

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

NOTES

- 1 Interactive systems → **Model-View-Controller (MVC)**.
- 2 Hierarchical systems → **layered**.

SLIDE 3 / 17 Interactive Systems

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).

NOTES

- 1 **Interactive** systems support user interaction, typically through user interfaces.
- 2 Two quality attributes matter most: **usability** (the UI must be easy to use) and **modifiability** (UIs change often, so changing them must be cheap).
- 3 The **mainstream pattern** for interactive systems is **MVC**, because it isolates the part that changes most (the UI) from the data and rules.

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.

NOTES

- 1 MVC is for interactive applications that need **flexible** human-computer interfaces: the UI can be replaced or multiplied without touching the core.
- 2 The three component types (the table):
 - Model**
Represents the system's core: its major processing capabilities and data.
 - View**
The output representation of the system (graphical or console-based).
 - Controller**
Associated with a view; handles user input.

EXAM ANGLE

Prediction

Q1

In MVC, which component handles user input?

- A. Controller
- B. Model
- C. View
- D. Broker

Answer: A. Controller

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).

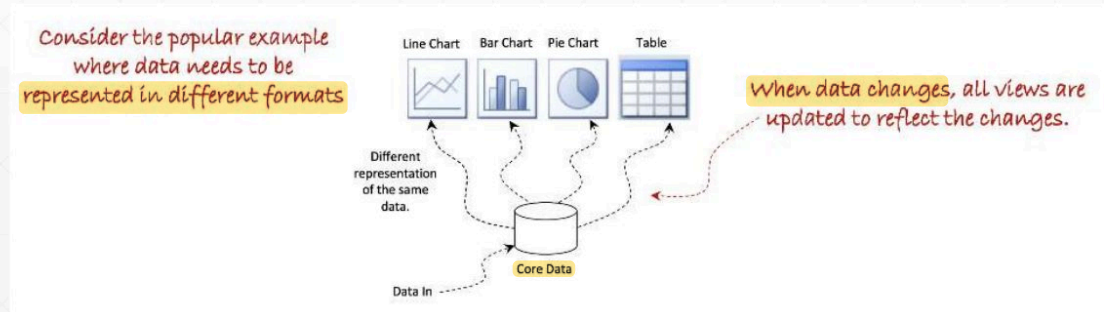
Summary:

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

NOTES

- 1 Inside a **mobile app**, MVC organises the code: the **model** manages the app's data (local storage, cache, data already fetched), the **view** is the UI (screens, buttons, forms), the **controller** handles input, calls services, and updates model and view.
- 2 Note: the **controller** (or a service layer) usually calls the server, not the model directly.
- 3 The same app is also part of a **distributed** system: the mobile client sends requests (login, messages, balance) through APIs to a server that applies business rules and uses a database. Summary: MVC inside the app, client-server outside it. Patterns combine at different levels.

MODEL-VIEW-CONTROLLER ARCHITECTURAL PATTERN



Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

6

NOTES

- 1 The motivating example: the same **core data** is shown in different formats (line chart, bar chart, pie chart, table). Each is a different representation of one data set.
- 2 When the data changes, **all views must update** to reflect it. MVC achieves that by keeping one model and letting every view observe it, instead of each view keeping its own copy that could drift.

EXAM ANGLE

Prediction

Q1

Which pattern best fits an app that shows the same data in several synchronised views that the user can edit?

- A. MVC
- B. Pipes and filters
- C. Blackboard
- D. Broker

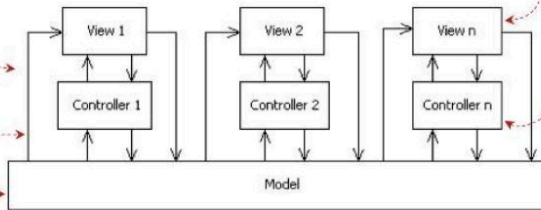
Answer: A. MVC

MODEL-VIEW-CONTROLLER ARCHITECTURAL PATTERN

Box-and-line diagram of the MVC architectural pattern

When Model changes, it updates its Views

Model: Encapsulates core data and functionality



View: Display information to the user. Every View is associated with a Controller.

Controller: Receives input and translates it to service requests.

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).

7

NOTES

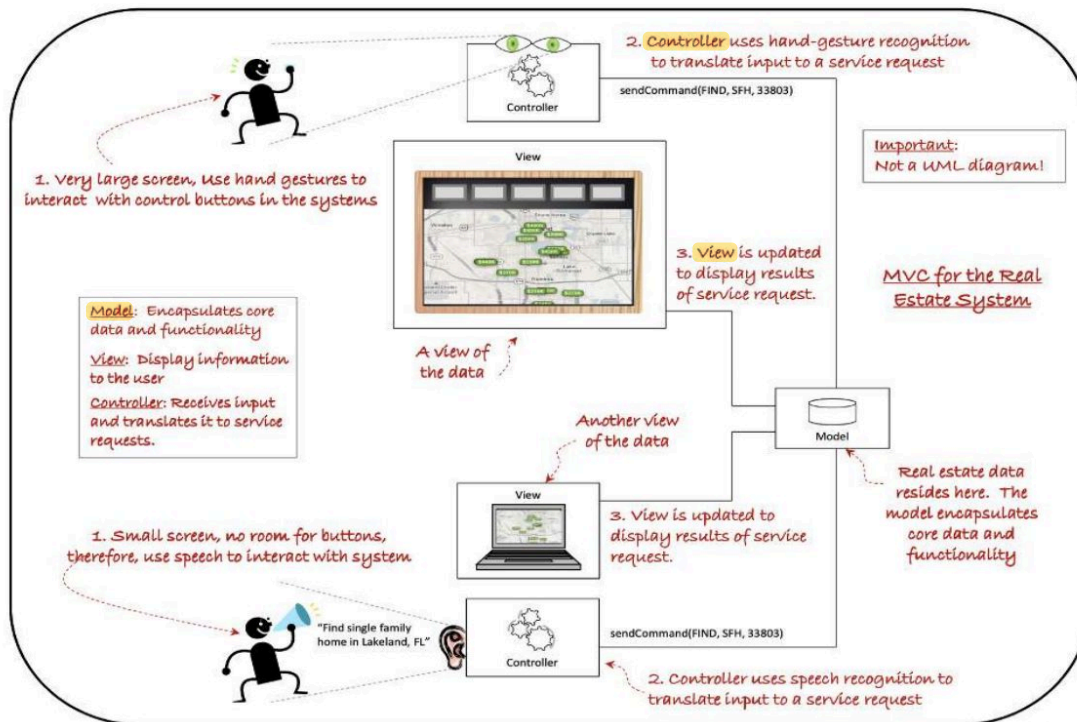
- 1 **Model:** holds the core data and system functionality; when the data changes, it updates (notifies) the views. In the box-and-line diagram one model sits under all the views and controllers.
- 2 **View:** displays information to the user. Every view is linked with a controller (View 1/Controller 1 ... View n/Controller n).
- 3 **Controller:** receives user input and translates it into service requests for the model.
- 4 **Flow:** the user interacts with a controller → the controller updates the model → the model notifies the views → the views present the updated information. Then the cycle repeats.
- 5 **Example** (changing a profile name): the user types a new name and clicks Save; the controller receives it and asks the model to update; the model changes the name and notifies the view; the view fetches the new data and refreshes the screen.

Past exam · Final 252 · Q4-1 · 2.5 marks**Q1**

Safari Explorer: a mobile app helps preschoolers (ages 3–5) learn about animals and their sounds. It must be highly responsive to touch and visually engaging. It has a Database with animal names, image file paths and audio clips, an Interactive Map (the graphical world map of animals), and a Selection Handler that detects a tap on a continent or animal and decides what to display next. Identify the architecture pattern that fits and draw its component diagram.

Answer: Model-View-Controller (MVC). The UI must stay responsive and visual while the data and input handling change independently. **Model:** the Animal Database (names, image paths, audio clips).

View: the Interactive Map (shows continents, animals, images and plays sounds). **Controller:** the Selection Handler (receives the tap, asks the model for the selected animal and tells the view what to show next). Diagram: Child taps the View → View passes the user event to the Controller → Controller selects the animal in the Model → Model updates/notifies the View.



NOTES

- 1 MVC for a **real-estate** system with two very different interfaces (not a UML diagram). The **model** in the middle holds the real-estate data and the core functionality.
- 2 Two controllers translate input into the same kind of service request, `sendCommand(FIND, 5FH, 33803)` : on the very large screen, a controller uses **hand-gesture recognition**; on the small screen (no room for buttons), a controller uses **speech recognition** ("Find single family home in Lakeland, FL").
- 3 Each **view** is updated to display the results of the service request: two views of the same data. Adding a third interface would mean adding a view-controller pair, without touching the model.

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).

NOTES

- 1 **User:** on the large screen uses hand gestures; on the small screen uses voice commands.
- 2 **Controller:** connects view and model, converting gestures or voice into actions such as fetching or filtering property data.
- 3 **Model:** stores the real-estate data (properties, prices, locations), provides it to the views and updates when controllers ask.
- 4 **View:** shows the information and updates dynamically after each user action. In short: user → controller → model → view → user.

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

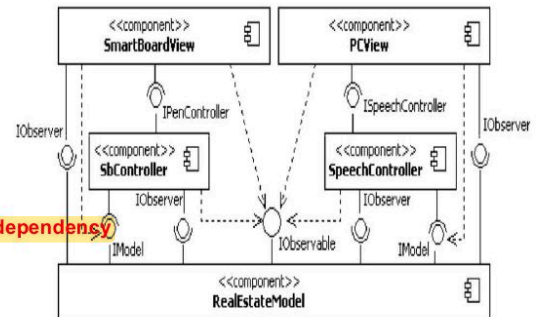
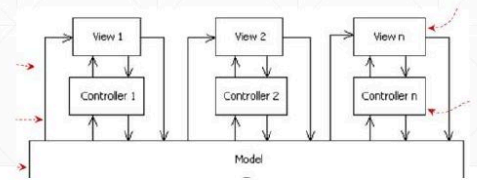
SmartBoardView: Displays info for large-screen (gesture/pen-based).
PCView: Displays info for smaller screen (voice-based).
Both register as **IObservers** to the **RealEstateModel**, so they update automatically when the Model changes.

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 **IObservable**.
Views refresh to display updated real estate data.

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



NOTES

- 1 **RealEstateModel**: the central data component. It provides **IObservable** (views register for updates) and **IModel** (data access and commands).
- 2 **Controllers**: **SbController** (SmartBoard) handles pen/gesture input through **IPenController**; **SpeechController** handles voice through **ISpeechController**. Both require **IModel** to act on the model.
- 3 **Views**: **SmartBoardView** (large screen) and **PCView** (small screen). Both provide **IObservable** and register with the model, so they update automatically when it changes.
- 4 Why both provide and require: a view needs the model (to get data) and the model needs the views (to notify them). That two-way dependency is created by the **Observer** pattern (Chapter 7).
- 5 **Flow**: user → controller (pen or speech) → RealEstateModel updated → model notifies views through **IObservable** → views refresh.

EXAM ANGLE

Prediction

Q1

In the real-estate MVC component diagram, why does the model provide **IObservable** while each view provides **IObserver**?

Answer: Views register with the model through **IObservable**, and the model calls each view back through **IObserver** when its data changes (the Observer pattern). So a view requires the model for data, and the model requires the views to notify them.

SLIDE 11 / 17 MVC: real-estate sequence diagram

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)*

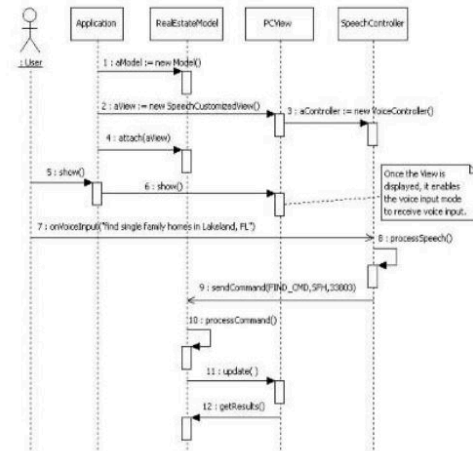
Application Class Methods:

- `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



11

NOTES

- 1 SpeechController:** `processSpeech()` is called when voice input arrives; `sendCommand(FIND_CMD, 5FH, 33803)` sends the parsed command to the model (messages 8–9).
- 2 RealEstateModel:** `processCommand()` handles the FIND command, `update()` changes its internal state (and notifies views), `getResults()` is how views fetch the search results (messages 10–12).
- 3 PCView:** `show()` (called twice) displays the view; enabling voice input mode is an internal state change, not a message.
- 4 Application** (set-up, messages 1–5): creates the model, the speech-customised view and the voice controller, and `attach(aView)` attaches the view to the model.
- 5 User:** `onVoiceInput("find single family homes in Lakeland, FL")` starts the interaction (message 7). Reading order: set-up, then input → controller → model → view.

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

Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

12

NOTES

- 1 The quality properties table: **modifiability** (easy to exchange, enhance or add user interfaces), **usability** (because UIs are exchangeable, systems can be configured with different interfaces for different groups of users), **reusability** (with concerns separated, model, view and controller components can be reused in other systems).
- 2 MVC has many **variations**: Microsoft's 1990s **Document-View** architecture fused view and controller into one component; **presentation-abstraction-control** is a more extensive variant (the slide calls it "process-abstraction-controller").
- 3 Remember the three qualities as the payoff of separating the UI: change it, tailor it, reuse the parts. The cost is more objects and update paths, which isn't worth it for a trivial static screen.

EXAM ANGLE

Prediction

Q1

True or false: MVC lets the user interface change without changing the model.

Answer: True — Views and controllers depend on the model, not the other way round (apart from notifications), so UIs can be swapped or added.

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**

NOTES

- 1 **Hierarchical** systems can be decomposed and structured in levels.
- 2 Two common patterns: **main program and subroutine** (a main routine calls subroutines, which call lower ones) and **layered**. This lecture covers layered.

SLIDE 14 / 17 Extra: Hierarchal Systems

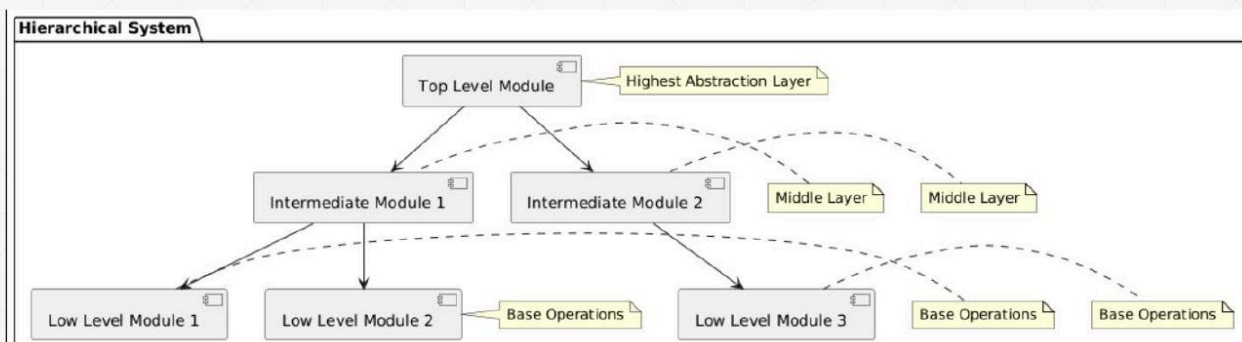
Extra: Hierarchal Systems

•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.



NOTES

- 1 Hierarchical systems are structured in **layers**, each with a specific role, separating concerns vertically (layers) or horizontally (modules).
- 2 Example: an **operating system**, from hardware interaction at the bottom to user-interface management at the top.
- 3 The diagram: a top-level module (highest abstraction) uses intermediate modules (middle layer), which use low-level modules (base operations). Calls go downward; higher levels never need to know how lower ones work.

SLIDE 15 / 17 Layered Pattern

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

NOTES

- 1 **Modifiability:** dependencies stay local within a layer; components only reach others through a well-defined, unified interface, so a layer component can be swapped for an enhanced one.
- 2 **Portability:** everything that touches the platform's APIs is encapsulated in one layer; port that layer and the higher layers move to the new platform unchanged.
- 3 **Security:** the controlled hierarchy makes it easy to insert security components, e.g. a layer that encrypts and decrypts all incoming and outgoing data.
- 4 **Reusability:** compartmentalised layer services are easier to reuse.

Past exam · Midterm 261 · Q6 (one of seven)

Q1

A payroll application must allow the user interface, salary-calculation rules, and data-storage technology to evolve with minimal impact on one another. For example, replacing the database should not require rewriting payroll policies, and interface code should not contain persistence logic. Which architectural style?

Answer: Layered: UI, business rules and data storage are separate layers that interact only through defined interfaces, so each can change independently.

SLIDE 16 / 17 Layered Pattern

Layered Pattern

SatcomLayer Component

- **Purpose:** Handles satellite communication functions
- **Interface:** ISatcom (provides satellite communication services to other layers)

SecurityLayer Component

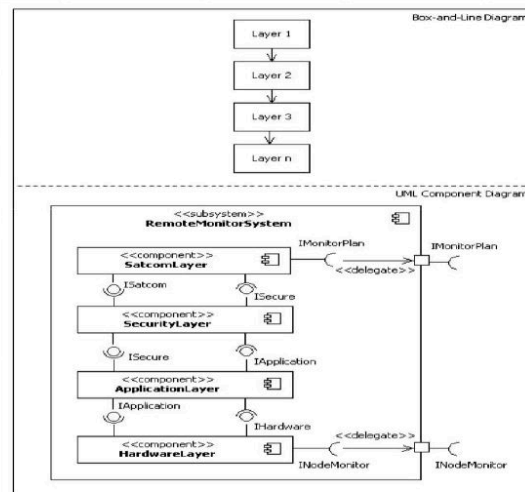
- **Purpose:** Manages security and encryption functions
- **Interface:** ISecure (provides security services, requires application services)

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)



Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

16

NOTES

- 1 A layered remote-monitoring system, top to bottom. **SatcomLayer** handles satellite communication; it provides **ISatcom** to the outside.
- 2 **SecurityLayer** handles security and encryption; it provides **ISecure** to the layer above and requires the application layer's services.
- 3 **ApplicationLayer** holds the main logic and business rules; it provides **IApplication** and requires hardware access.
- 4 **HardwareLayer** controls the physical devices and sensors; it provides **IHardware**. In the UML diagram each layer requires exactly the interface of the layer below; the subsystem's external ports (IMonitorPlan, INodeMonitor) are delegated to the top and bottom layers. Each extra layer adds a call, which is the pattern's performance cost.

Prediction

Q1

In a strict layered architecture, which layers may a layer use, and what is the main drawback?

Answer: Only the layer directly below it, through that layer's interface. Drawback: requests pass through every layer, adding performance overhead (and sometimes pass-through code).

SLIDE 17 / 17 What's next

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.

Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

17

NOTES

- 1 This completes the architectural patterns: blackboard, pipes and filters, client-server, broker, **MVC** and **layered**.
- 2 Next: **detailed design** (Chapter 5), which starts once the architecture is sufficiently complete.

Original slides from the SE322 lecture deck, with study notes. Check exact wording against the [original deck](#). Notes and questions are study aids. Made by Shoug Alomran.