COMP 3550

7.4 — ARCHITECTURAL DESIGN PATTERNS

Week 7: Design Patterns & Architecture

WHAT IS SOFTWARE ARCHITECTURE?

We’ve seen it a little but now let’s really dig in
The big picture: how your whole system fits together.

Definition:
Architectural design patterns define the system-wide structure and how major components interact.
They go beyond individual classes or objects and shape the entire application's flow, scalability, and maintainability.

WHAT IS SOFTWARE ARCHITECTURE?

We’ve seen it a little but now let’s really dig in
The big picture: how your whole system fits together.

Key Characteristics:

- High-level: affects modules, services, and layers, not just classes
- Guides how data flows, how responsibilities are divided, and how change is managed
- Influences:
  - Performance
  - Testability
  - Modularity
  - Team development boundaries

WHAT IS SOFTWARE ARCHITECTURE?

We’ve seen it a little but now let’s really dig in
The big picture: how your whole system fits together.

Examples:

- Layered Architecture (As seen in our projects and will see in a moment)
- MVC (Model–View–Controller) (Coming Soon in This Deck)
- Microservices
- Event-Driven Architecture
- Hexagonal / Ports & Adapters

WHAT IS SOFTWARE ARCHITECTURE?

Examples:

- **Layered Architecture (As seen in our projects and will see in a moment)**
  - Spring Framework (Java) – classic 3-tier architecture
  - .NET Applications – UI Layer → Business Layer → DAL
  - Enterprise JavaBeans (EJB) systems
  - Many traditional banking, insurance, and healthcare systems
- MVC (Model–View–Controller) (Coming Soon in This Deck)
- Microservices
- Event-Driven Architecture
- Hexagonal / Ports & Adapters

WHAT IS SOFTWARE ARCHITECTURE?

Examples:

- Layered Architecture (As seen in our projects and will see in a moment)
- **MVC (Model–View–Controller) (Coming Soon in This Deck)**
  - Ruby on Rails — textbook MVC pattern
  - Django (Python) — follows MTV (Model–Template–View), a flavor of MVC
  - ASP.NET MVC
  - Spring MVC
  - Angular (loosely MVC-inspired client-side framework)
  - Web applications with user-facing UIs
- Microservices
- Event-Driven Architecture
- Hexagonal / Ports & Adapters

WHAT IS SOFTWARE ARCHITECTURE?

Examples:

- Layered Architecture (As seen in our projects and will see in a moment)
- MVC (Model–View–Controller) (Coming Soon in This Deck)
- **Microservices**
  - Netflix – pioneered cloud-native microservices at scale
  - Amazon – each team owns a "two-pizza" service
  - Spotify – autonomous feature teams deploy their own services
  - Uber – hundreds of small services for rides, payments, logistics
  - Docker, Kubernetes, REST/gRPC APIs, CI/CD pipelines
- Event-Driven Architecture
- Hexagonal / Ports & Adapters

WHAT IS SOFTWARE ARCHITECTURE?

Examples:

- Layered Architecture (As seen in our projects and will see in a moment)
- MVC (Model–View–Controller) (Coming Soon in This Deck)
- Microservices
- **Event-Driven Architecture**
  - Airbnb – uses event streams for analytics and user behavior
  - Uber – event bus architecture for real-time coordination
  - eBay – CQRS and event sourcing for user and order data
  - Slack – message-passing architecture internally
  - Often combined with microservices
- Hexagonal / Ports & Adapters

WHAT IS SOFTWARE ARCHITECTURE?

Examples:

- Layered Architecture (As seen in our projects and will see in a moment)
- MVC (Model–View–Controller) (Coming Soon in This Deck)
- Microservices
- Event-Driven Architecture
- **Hexagonal / Ports & Adapters**
  - Clojure applications often favor hexagonal design
  - Domain-Driven Design (DDD) enthusiasts and companies
  - Alistair Cockburn’s own implementations (inventor of the pattern)
  - Many modern Spring Boot apps using @Component-driven interfaces

MODEL-VIEW-CONTROLLER (MVC)

Component	Role
Model	Holds the data and business logic. Knows what to do, but not how to display it.
View	Displays the data (UI). Passive — waits for updates from the Model or actions from the Controller.
Controller	Responds to user input, updates the Model, and may trigger View updates.

MODEL-VIEW-CONTROLLER (MVC)

```
// Controller (ActionListener)
button.addActionListener(e -> {
    model.updateName(nameField.getText());   // Controller calls Model
    view.refresh();                          // Controller updates View
});
```

Imagine a restaurant:

- Model = Kitchen (makes food)
- View = Waiter (shows you the menu & serves)
- Controller = You (place order & give commands)

MODEL-VIEW-CONTROLLER (MVC)

```
// Controller (ActionListener)
button.addActionListener(e -> {
    model.updateName(nameField.getText());   // Controller calls Model
    view.refresh();                          // Controller updates View
});
```

Imagine a restaurant:

- Model = Kitchen (makes food)
- View = Waiter (shows you the menu & serves)
- Controller = You (place order & give commands)

Notice the separation of concerns: changes in UI don’t break the logic, and vice versa.

LAYERED ARCHITECTURE

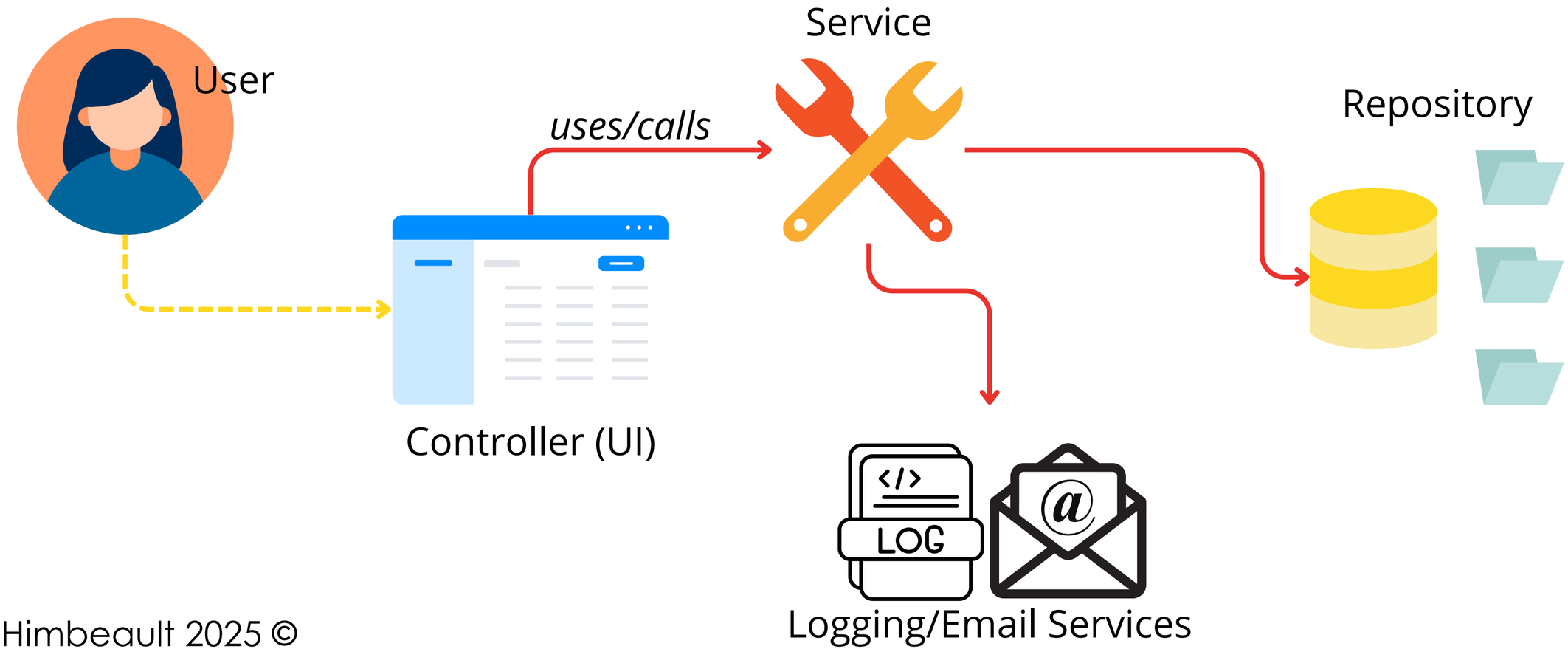
Separate your concerns, one layer at a time.

Let’s look at a classic 4-layer system which is very similar to ours without the infrastructure, since we do not need it

Layer	Responsibility	Example (Web App)
Presentation	Handles input/output, UI logic	Controller (OrderController)
Logic (Service)	Business rules, domain logic	Service (OrderService)
Persistence	Communicates with databases, CRUD operations	Repository (OrderRepository)
Infrastructure	External systems: email, file storage, payment gateways, etc.	EmailSender, Logger, StripeClient

LAYERED ARCHITECTURE

Just like in an office, you don’t call the CEO to get a paperclip. The receptionist asks the admin, the admin asks the supply team. Each layer has a role.



BENEFITS OF LAYERED/MVC

- **Clear Separation of Concerns**
  - Logic, UI, and data access live in different places — less chaos
  - Developers can work on one part without breaking another
  - You really see it’s benefit:
    - Adding new features
- Easier Testing
- Faster Onboarding
- Better Maintainability
- Reusability

BENEFITS OF LAYERED/MVC

- Clear Separation of Concerns
- **Easier Testing**
  - You can test business logic without spinning up the UI
  - Each layer/component can be mocked or stubbed in isolation
  - You really see it’s benefit:
    - Writing unit & integration tests
- Faster Onboarding
- Better Maintainability
- Reusability

BENEFITS OF LAYERED/MVC

- Clear Separation of Concerns
- Easier Testing
- **Faster Onboarding**
  - New devs can understand the system one layer at a time
  - Predictable structure = less ramp-up time
  - You really see it’s benefit:
    - New team members
- Better Maintainability
- Reusability

BENEFITS OF LAYERED/MVC

- Clear Separation of Concerns
- Easier Testing
- Faster Onboarding
- **Better Maintainability**
  - Easier to change tech in one part (e.g., replace database)
  - Less risk of bugs spreading across unrelated parts
  - You really see it’s benefit:
    - Long-term projects
- Reusability

BENEFITS OF LAYERED/MVC

- Clear Separation of Concerns
- Easier Testing
- Faster Onboarding
- Better Maintainability
- **Reusability**
  - Service or logic layers can be reused across different UIs
  - You really see it’s benefit:
    - Web + Mobile apps using same logic

CHOOSING THE RIGHT ARCHITECTURE

Design is a conversation, not a prescription.

Start by Asking These Questions:

- **What are we building?**
  - Simple CRUD app? High-throughput service? Real-time chat?
- **How many people will work on it — now and later?**
  - Solo project vs. large, distributed teams
- **What are our priorities?**
  - Speed to market? Maintainability? Scalability? Testability?
- **How volatile are the requirements?**
  - Do we need flexibility for change? Plug-and-play components?
- **How complex is the domain logic?**
  - Can we get by with MVC or do we need DDD-style separation?
- **Will parts of the system evolve or be reused independently?**
  - If yes? consider modular, hexagonal, or microservice architecture

CHOOSING THE RIGHT ARCHITECTURE

Design is a conversation, not a prescription.

Team Discussion Prompts:

Question	Why It Matters
What could change in the next 6–12 months?	Helps identify points of flexibility
What do we want to test independently?	Drives separation and interfaces
How much tech debt can we afford?	Influences how much structure to add
Who else will need to understand this code?	Shapes simplicity vs. complexity

PROJECT PAUSE & REFLECT

Consider your own project for a moment. Label parts of your own project using either MVC or layered terms.

Where do responsibilities blur?

Here is a list of terms to consider:

PROJECT PAUSE & REFLECT

Consider your own project for a moment. Label parts of your own project using either MVC or layered terms.

Where do responsibilities blur?

Here is a list of terms to consider:

MVC Terms:

- Model
  - Domain logic
  - Data structures / business rules
  - Validation logic
- View
  - UI components (Swing panels, web templates, etc.)
  - Output formatting
- Controller
  - Handles user input (button clicks, form submissions)
  - Orchestrates Model + View

Layered Architecture Terms:

- Presentation Layer
  - Input/output logic
  - Controllers, UI code
- Service Layer
  - Application logic (what happens after a button is clicked)
  - Rules and coordination
- Persistence Layer
  - CRUD operations
  - DAOs, repositories
- Infrastructure Layer
  - Email senders, loggers, API clients, file access, etc