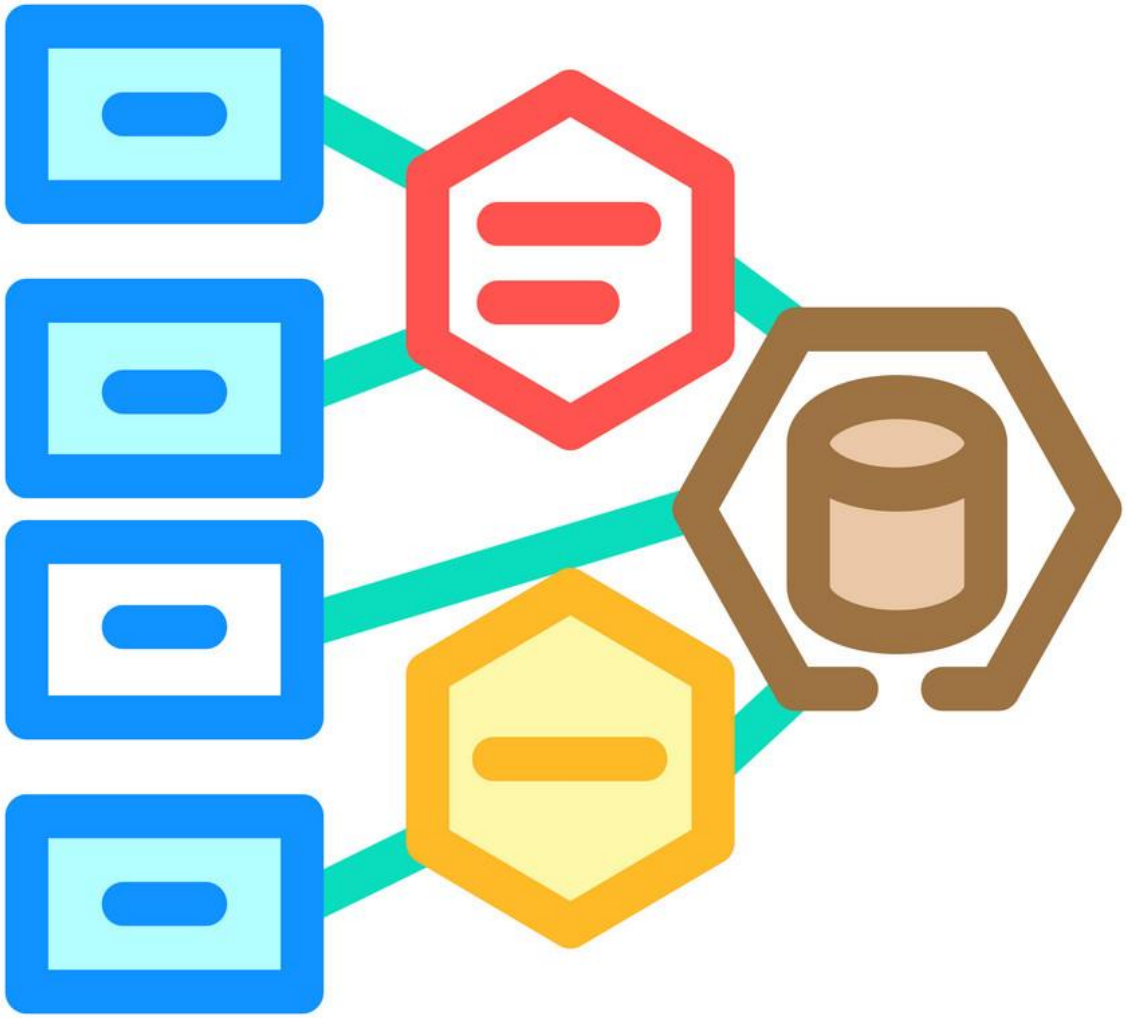
Software Design and architecture

Fall 2024

Dr. Natalia Chaudhry



Outline

- Architectural styles
  - ✓ Data flow architectures
  - ✓ Layered
  - ✓ Event-based
  - ✓ Data-centered
  - ✓ MVC
  - ✓ Multi-tier distributed
  - ✓ Service Oriented

- The **data-centric architectural style** focuses on organizing a software system around a central data repository that is crucial for the operations of the system.
- This style is often used when multiple systems or components need to access and update a common set of data.
- The central repository ensures that data integrity and consistency are maintained across all subsystems.

Key Characteristics:

Centralized Data Storage: All components interact with a shared data source (database, file system, or other storage mechanisms), which is the core of the architecture.

Loose Coupling: The components are loosely coupled since their primary interaction happens through the shared data rather than direct communication.

Concurrency and Synchronization: Since multiple components may access or update the data simultaneously, concurrency control mechanisms are important.

Scalability: It can scale well by optimizing the data store or adding more clients that use the data.

Benefits:

- **Data Consistency**: Having a single source of truth for all components.
- **Reusability**: The data repository can be used by various components or systems.
- **Maintainability**: Changes to data management can be centralized.

Challenges:

- **Performance Bottlenecks**: Since all components depend on a central repository, it may become a performance bottleneck under high loads.
- **Data Contention**: Concurrency issues can arise when multiple components try to read and write simultaneously.
- **Complexity in Data Management**: Handling large volumes of data and ensuring it remains consistent can be complex.

Enterprise Resource Planning (ERP) Systems

All departments use the same data repository for storing and retrieving information, ensuring data consistency across the organization.

Real-life Example: A manufacturing company using an ERP system where inventory data updates in real-time, reflecting accurate stock levels for multiple departments.

Banking Systems

Core banking systems, used by financial institutions, rely on centralized databases to manage transactions, customer data, and account information.

Multiple banking applications (e.g., ATMs, online banking, mobile apps) access a central data repository to ensure consistency across all services.

Real-life Example: A customer withdrawing money at an ATM and immediately seeing the updated balance in their mobile banking app

Health Information Systems (HIS)

Hospitals and clinics use health information systems to manage patient data, including medical records, test results, prescriptions, and billing.

Different healthcare professionals (doctors, nurses, administrators) access a central database to get real-time updates on patient health records.

Real-life Example: A doctor reviewing a patient's test results in real-time while a nurse is updating the same patient's prescription details.

Version Control Systems (VCS)

Distributed version control systems like Git, where code from multiple developers is managed in a central repository.

Multiple developers work on different parts of a project and commit their code changes to a central repository, which maintains the project's history and state.

Real-life Example: Teams of software engineers collaborating on a software project using GitHub, where all code changes are stored centrally and can be tracked and merged.

For **banking systems**, both **layered architecture** and **data-centric architecture** are suitable, depending on the specific needs:

- **Layered Architecture** is more suitable for managing complexity, as it divides the system into layers like presentation, business logic, and data access. This modularity improves **maintainability** and **scalability**.

- **Data-Centric Architecture** is ideal if the focus is on **shared data consistency**, such as in transaction management where multiple subsystems (e.g., ATMs, mobile apps) access a central database.

In practice, banking systems often use a **hybrid approach**: layered for managing business logic and data-centric for ensuring shared, consistent access to core data.

Layered architecture generally **scales better** than data-centric architecture, especially in large systems like banking, due to its separation of concerns:

- **Layered Architecture** can scale horizontally and vertically by distributing different layers (e.g., presentation, business, and data layers) across multiple servers, allowing the system to handle increased load in each area independently.
- **Data-Centric Architecture** while scalable, may face **bottlenecks** at the centralized data repository. Scaling often requires more complex solutions, such as database sharding or replication, which adds complexity.

Outline

- Architectural styles
  - ✓ Data flow architectures
  - ✓ Layered
  - ✓ Event-based
  - ✓ Data-centered
  - ✓ MVC
  - ✓ Multi-tier distributed
  - ✓ Service Oriented

Model-View-Controller (MVC) is a software architectural pattern commonly used for developing user interfaces that separates the application into three interconnected components:

Model:

- Manages the data and business logic of the application.
- It responds to requests for data and updates the data when commanded by the controller. It is independent of the user interface.

Example: In an e-commerce application, the "Product" object that handles product details (like name, price, and inventory) is part of the Model.

View:

- Represents the user interface (UI) and displays the data.
- It generates the UI based on the data provided by the Model.
- It also sends user input to the Controller.

Example: The HTML/CSS layout or the UI screen where users see product details on an e-commerce site is the View.

Controller:

- Acts as an intermediary between the Model and the View.
- It handles user input (like clicks, form submissions), updates the Model based on that input, and determines which View should be shown.

Example: If a user clicks "Add to Cart," the Controller updates the cart Model and refreshes the View to show the updated cart..

Key Benefits:

- **Separation of Concerns**: Each component has a clear, distinct responsibility, making the system easier to maintain and scale.
- **Reusability**: The Model can be reused across multiple Views, and different Views can present the same Model data in different ways.
- **Testability**: Components can be tested independently.

The Model can be reused across multiple Views, and different Views can present the same Model data in different ways

Model: Let's consider a Product model that contains the data and business logic related to products in the store. The model has attributes like:

Product Name

PriceDescription

Stock Quantity

Category

Images

This Product Model can be reused across multiple views in different contexts:

Product Listing Page (Grid View)
Displays products in a grid format. Here, only a subset of the Model's data is displayed, such as the product name, price, and thumbnail image.

The **View** pulls data from the **Product Model** (e.g., name, price, image) and presents them in a grid layout to allow users to browse through multiple products quickly.

Product Detail Page

Shows detailed information about a single product, such as the full description, large image, price, reviews, and availability.

This **View** pulls the same **Product Model** but displays more detailed information (e.g., full description, images, stock, reviews) in a rich format for customers interested in a specific product.

Shopping Cart Summary

Displays products added to the cart, typically showing the product name, quantity, and price.

The **View** again uses the same **Product Model** but only presents limited information (product name, price) relevant to the cart functionality.

Admin Dashboard (Product Management)

For administrators, this view might show the product details in an editable form, allowing admins to modify the product name, price, category, stock quantity, and description.

The **View** uses the **Product Model** for form-based editing, allowing administrators to update product details directly in the system.

Model (Product): Same data (e.g., product name, price, image, description) is being reused.
Views: Different representations of the data are shown (grid view, detailed view, cart summary, admin form).

By using the same **Model** across these **Views**, the system avoids duplication of data logic and ensures that all views are synchronized with the correct data from the Model.

Usecases:

- **Web applications**: Frameworks like Ruby on Rails, Django, and ASP.NET MVC are based on the MVC pattern.
- **Desktop applications**: MVC is often used in GUI-based systems like Java Swing applications.

Challenges

Increased Complexity for Simple Applications

For smaller or simple applications, using MVC can be overkill. Breaking the application into three layers (Model, View, Controller) may add unnecessary complexity and boilerplate code.

It can make simple apps harder to implement and maintain, as the code is more scattered across components.

Tight Coupling between Controller and View

Although the Model is independent, the **Controller and View can become tightly coupled** if not designed carefully. The controller often has logic to update the view, leading to dependency on the presentation logic.

Changes in the View may require updates in the Controller, reducing flexibility and increasing maintenance effort.

Difficulty in Managing Complex UIs

For complex UIs (e.g., dashboards with numerous components), managing multiple Views and their corresponding Controllers can become challenging. Controllers often handle a lot of logic and input validation, which can lead to bloated controller code.

The pattern may result in **large and hard-to-manage controllers** (also known as "Fat Controllers").

Maintenance and Scalability Issues for Large Applications

As applications grow, the MVC pattern can lead to **scalability issues**. For large projects, the separation of concerns can get blurred, leading to difficulties in maintaining the codebase. Maintaining consistency in the structure across a large team can also be hard.

It can be challenging to keep models and controllers well-organized, resulting in spaghetti code.

Testing Challenges

Unit testing can become more challenging in an MVC architecture since there is often tight coupling between components, especially when the Controller directly interacts with both the Model and View.

Writing isolated unit tests for Controllers and Views may require mocking dependencies, increasing the testing complexity.

Home task:

Explore Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM) Patterns

MVC:

- **Model:** Represents the data and business logic of the application.
- **View:** Displays the data (UI).
- **Controller:** Handles user input, updates the Model, and refreshes the View. It acts as the mediator between the Model and View.

MVP:

- **Model:** Same as MVC, representing the data and business logic.
- **View:** Displays the UI and delegates user actions to the Presenter.
- **Presenter:** Acts as a middleman between the View and Model, processing user input, updating the Model, and manipulating the View.

MVVM:

- **Model:** Represents data and business logic.
- **View:** Represents the UI elements.
- **ViewModel:** Exposes data and commands to the View and manipulates the Model. It enables data binding, allowing automatic updates between the View and ViewModel.

Aspect	MVC	MVP	MVVM
Structure	Model, View, Controller	Model, View, Presenter	Model, View, ViewModel
Communication Flow	View → Controller → Model → View	View → Presenter → Model → View	View ↔ ViewModel ↔ Model
Data Binding	Manual updates	Manual updates	Automatic data binding
Testability	Moderate	High	High
Complexity	Simple	Moderate	High
Best For	Less complex web apps	Desktop applications	Rich UI applications

That’s it