Bachelor of Computer Science - SCS1303

Introduction to Software Engineering

Software Architecture

Prof. K. P. Hewagamage

UNIVERSITY OF COLOMBO SCHOOL OF COMPUTING

Why is Architecture Important?

Doghouse

You wouldn't build a skyscraper on a doghouse foundation.

Skyscraper

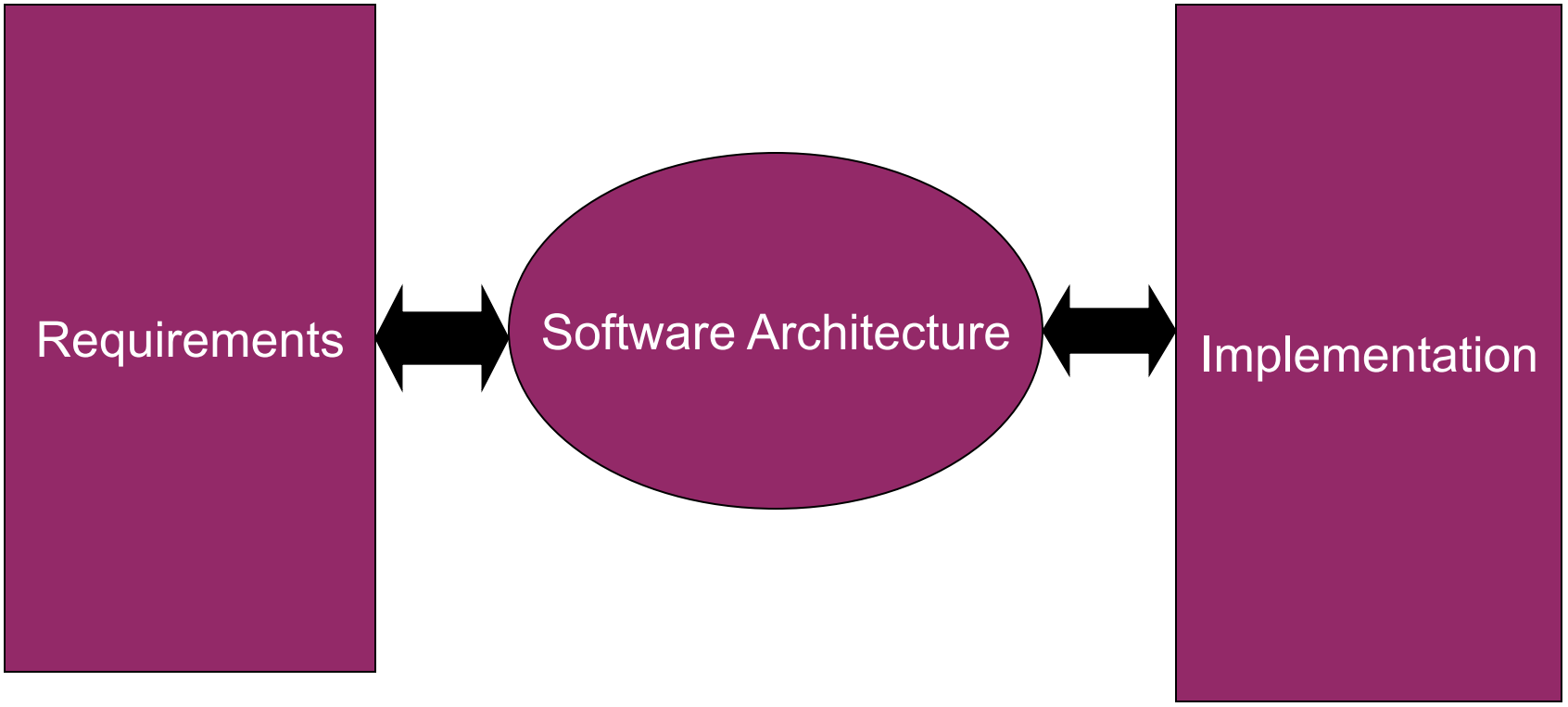
So why build complex software without a strong architectural foundation?

Sub Topics

- Introduction to Software Architecture
- Architectural Design
- Architectural Views and View Models
- Architectural Styles and Patterns

INTRODUCTION TO SOFTWARE ARCHITECTURE

Software Architecture as a Bridge



Architecture plays a key role as a bridge between requirements and implementation

Architecture in Plan-Driven Vs Agile

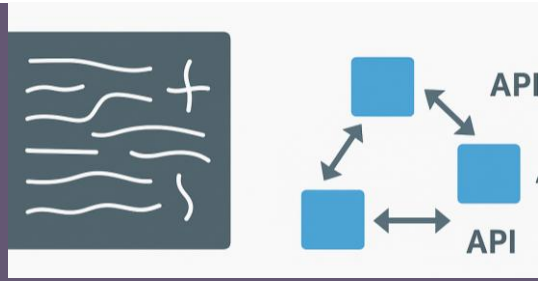
- It is generally accepted that an early stage of agile processes is to design an overall systems architecture.
- Refactoring the system architecture is usually expensive because it affects so many components in the system

What is Architecture?

- Defines the **structure** and **components** of a system.
- Describes how parts **interact** with each other and the environment.
- Guides how a system **evolves** over time.

"The fundamental organization of a system embodied in its components, their relationships, and the principles guiding its design."
— IEEE 1471

Architecture Is Everywhere



- Every system has an architecture—even if it's accidental.
- **Good architecture** is intentional, documented, and scalable.
- It influences performance, reliability, maintainability, and more.

Every system has an architecture, whether or not it is documented and understood.

Definition of Software Architecture

"Software architecture refers to the fundamental structure of a software system, encompassing its **key components**, their **interactions**, and the **guiding principles** for its **design and evolution**."

1. What are the **core elements** of the system?
2. How do they **interact** with each other and the environment?
3. What **principles** govern their design and long-term evolution?

1. Elements and Their Relationships

The **structure** of a software system is defined by how its parts are organized and how they communicate.

Two Types of Structures:

- **Static Structure** – design-time elements:
  - Classes, packages, modules, services, stored procedures, etc.
- **Dynamic Structure** – runtime behavior:
  - Object creation/deletion, message passing, task execution, system state

2. Properties for Interaction and Behavior

- **1. Externally Visible Behavior** (*What it does*):
  - Observable actions: data in/out, user responses, exposed API behavior
- **2. Quality Attributes** (*How it does it*):
  - Performance (throughput, latency)
  - Security (data access, threat protection)
  - Scalability (handles more users)
  - Maintainability, Usability, Reliability

Example: *"The system responds to a user request in under 200ms while ensuring data is encrypted and processed securely."*

Principles of Design and Evolution

Principles Guiding Design & Change:

- Guide system **implementation** and **extension**
- Ensure **consistency** across modules
- Promote **simplicity**, **scalability**, and **understandability**
- Support **future evolution** with minimal disruption

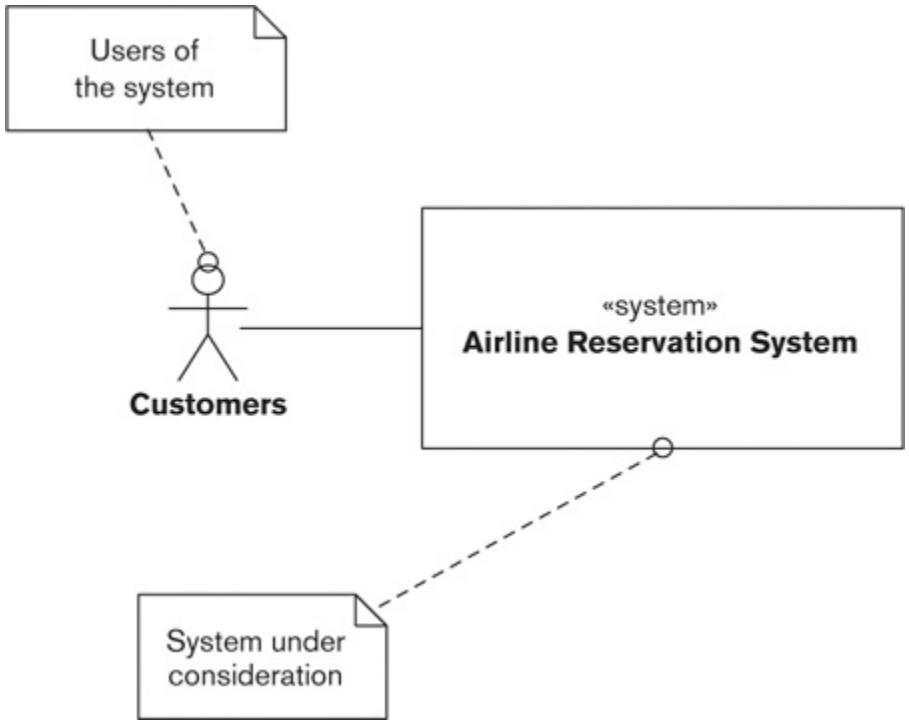
"A good architecture grows with the system, without becoming a bottleneck."

About the SW Architecture

- Architecture ≠ Code
- Architecture is about the **big picture**: structure, constraints, patterns.
- Design is about **detailed decisions** inside that structure.
- Code is the **realization** of those designs.

ARCHITECTURAL DESIGN

Functional & Quality View of a System: Airline Booking



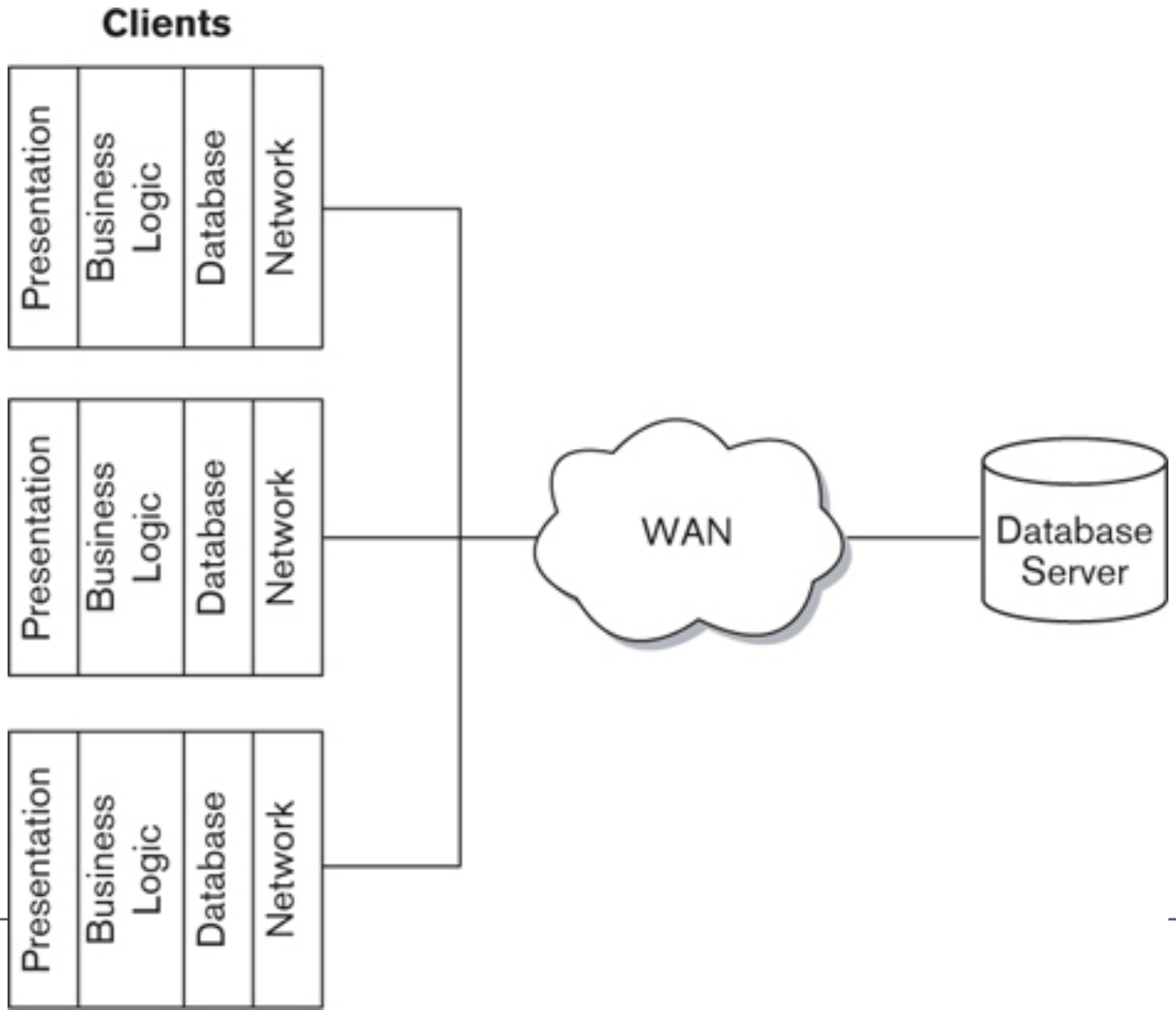
External Behavior:

- - Booking a flight
- - Canceling a reservation

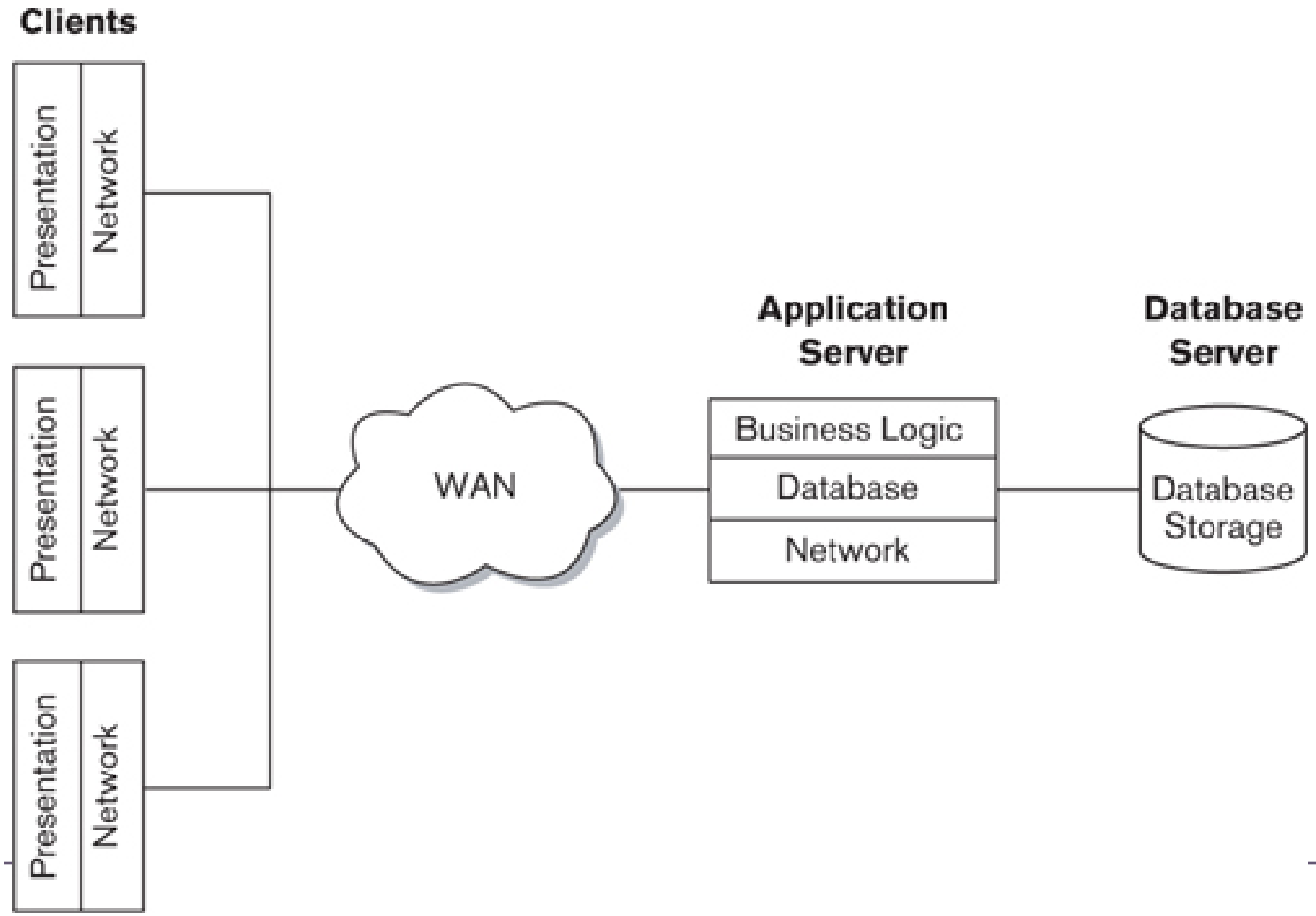
Internal Behavior (Quality Attributes):

- - Response time under 1s under peak load
- - 99.99% availability

Two-Tier Client/Server Architecture



Three Tier Client/Server Architecture



Choosing Two-Tier Vs Three-Tier Architectures

Architecture	Key Reason	Best Suited When
Two-Tier	Simplicity with low overhead	- The number of users is small - Business logic can reside on the client-System is deployed in a LAN or closed environment - Fast development with minimal infrastructure is preferred
Three-Tier	Scalability and separation of concerns	- Large number of concurrent users - System needs centralized, reusable business logic - Long-term maintenance, modifiability, and security are priorities - Deployment is in a distributed or cloud-based environment

What determines Architectural Design?

- Functional Requirements
- Non-Functional Requirements
- Design Constraints
- Quality Attributes requirements

Software architecture is the design blueprint that balances requirements with real-world constraints and long-term goals.

What Is a Candidate Architecture?

- A **candidate architecture** is a proposed structural solution—comprising static and dynamic elements—that is **designed to meet the system's functional and quality requirements**.
- Selecting a candidate architecture is like choosing the best-fitting design blueprint—it might not be perfect, but it's optimized for the **constraints, goals, and qualities** of your system context.

Architecture Selection

- Supports required system behaviors
  - e.g., handles transactions, delivers expected responses
- Satisfies quality attributes
  - Response time, throughput, availability, time to repair
- Fits the environment and constraints
  - Efficiently processes tasks given cost, technology, or team skills
- Trade-offs may vary between options
  - Example: One candidate may be easier to maintain but costlier to build.

What Is an Architectural Element?

An **architectural element** is a **building block** of a software system. It can be a component, module, or service that fulfills a specific role within the architecture.

The distinction of Elements

- **Component**: conceptual/design-level
- **Module**: implementation/packaging
- **Service**: runtime execution & exposure

Their responsibilities must be **well-defined**, with **clear boundaries** and **interfaces**.

Examples of Architectural Elements

Type	Definition	Examples
Component	A **logical unit** of functionality, often used during **design modeling**	- UserAuthComponent in a login system- PaymentProcessor in an e-commerce app
Module	A **packaged implementation unit** — deployable or reusable chunk of code	- UserModule in Angular or NestJS - A JAR file (product-core.jar) in Java- .NET DLL file
Service	A **runtime-accessible unit** with well-defined interfaces, often over network	REST API: GET /products/{id}- Microservice: InventoryService in a retail platform- AWS Lambda: ImageResizer function

Stakeholders in Software Architecture

Individuals or groups who **affect or are affected by** the architecture of a system.
Not limited to end users.

Why Stakeholders Matter in Architecture?

- Their **needs**, **concerns**, and **priorities** drive key architectural decisions.
- Every architecture is a response to **competing stakeholder demands**.
- Understanding them is **key to defining the architect's role**.

Types of Stakeholders

Type	Example
Users	Airline passengers using a booking system
Customers	Airline companies purchasing the system
Developers	Teams implementing the architecture
Managers	Project/product leads who manage scope, cost
Operators	System admins, DevOps
Testers	QA teams validating quality
Regulators	GDPR, aviation compliance bodies

Architectures exist to serve stakeholder needs — understanding those needs is the architect's most critical responsibility.

Stakeholders' Concerns

A **concern** is any interest or issue that a stakeholder has regarding the system's architecture.

It may include:

- A **requirement** (e.g., data must be encrypted)
- An **objective** (e.g., reduce time-to-market)
- A **constraint** (e.g., must use PostgreSQL)
- An **aspiration** (e.g., support AI integration in future)

Concerns….

Every stakeholder has unique concerns

Good architecture:

- **Identifies** and **understands** those concerns.
- **Balances trade-offs** between competing or conflicting concerns.
- Provides **transparent justifications** for decisions made.

A good architecture does **not please everyone** equally—it finds the **best compromise** that aligns with business goals and system context.

What is Architecture Description (AD)?

A **structured collection of documents and models** that communicates a system's architecture **clearly and consistently** to its stakeholders.

Why Is AD Important?

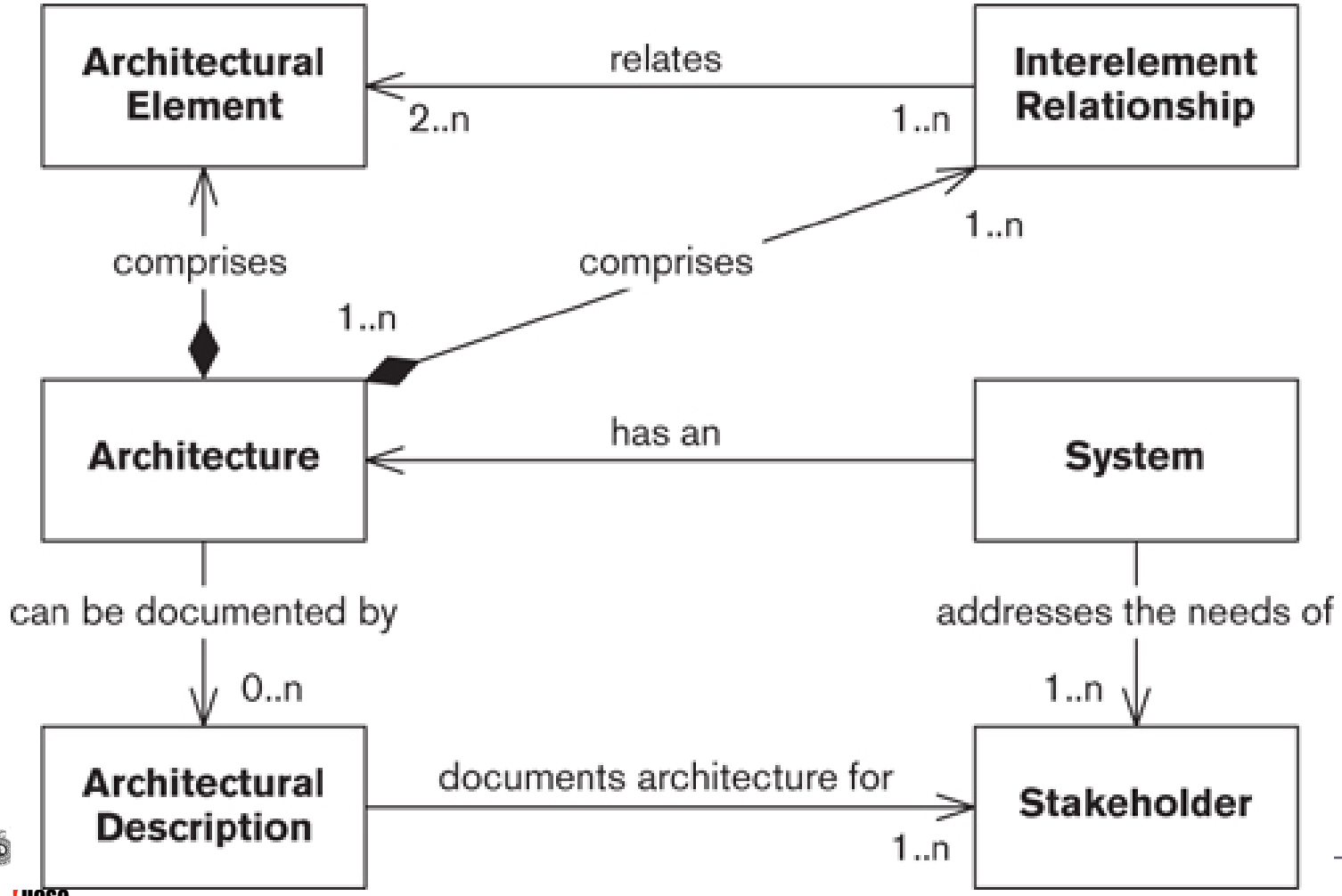
- Makes the **architecture understandable** to all stakeholders
- Shows how the architecture **addresses stakeholder concerns**
- Provides **evidence** that quality attributes and constraints are met

Contents of an AD

Element	Purpose
Architectural Models	Visual representations (e.g., views, diagrams)
Scope Definition	What the architecture includes and excludes
Design Principles	Key decisions, rationales, and constraints
Interface Definitions	Descriptions of component/service boundaries
Quality Attribute Justifications	Evidence for scalability, security, etc.

Every system has an architecture — but **only well-architected systems have effective descriptions** that allow stakeholders to make informed decisions.

Core Concepts and Relationships



What Influences Software Architecture?

Category	Influence Description
Requirements	Functional (what system should do) and Non-functional (how it should behave – e.g., performance, security)
Stakeholders	Users, developers, managers, operators, regulators – each with unique concerns
Development Organization	Team size, skills, budget, timeline, development process (plan-driven, agile, DevOps)
Technical Environment	Platforms, programming models (OO, REST, cloud-native), frameworks (.NET, Spring, React, etc.)
Business & Social Context	Market pressures, legal compliance (e.g., GDPR), social expectations (accessibility, ethical AI)
Past Experience	Lessons from previous projects, reuse of proven

The Roles of Software Architecture

Role	Description	Stakeholders Benefited
Understanding	Provides a high-level abstraction of the system for easier comprehension	Developers, Designers
Reuse	Enables reuse of components, modules, or frameworks across projects	Developers, Architects
Construction	Acts as a blueprint for development—showing structure and component interaction	Developers, Team Leads
Evolution	Guides how the system can evolve to accommodate future requirements	Architects, Product Owners
Analysis	Enables evaluation of quality attributes, consistency, and constraints	QA, Architects
Management	Acts as a milestone and planning tool for teams, schedules, risks, and resource	Managers, Project Leads

Architecture from the Stakeholders' View

Architecture impacts every role involved in the system—not just the developers:

Stakeholder	Primary Concern
User	Usability, reliability, availability
Customer	Cost, delivery timeline, alignment with business goals
Manager	Team coordination, modularity, project risk
Developer	Clarity of components, feasibility of implementation
Tester	Verifiability of system behavior and quality attributes

A **good architecture reflects and balances** these diverse concerns.

Architecture as a Communication Bridge

A well-described architecture:

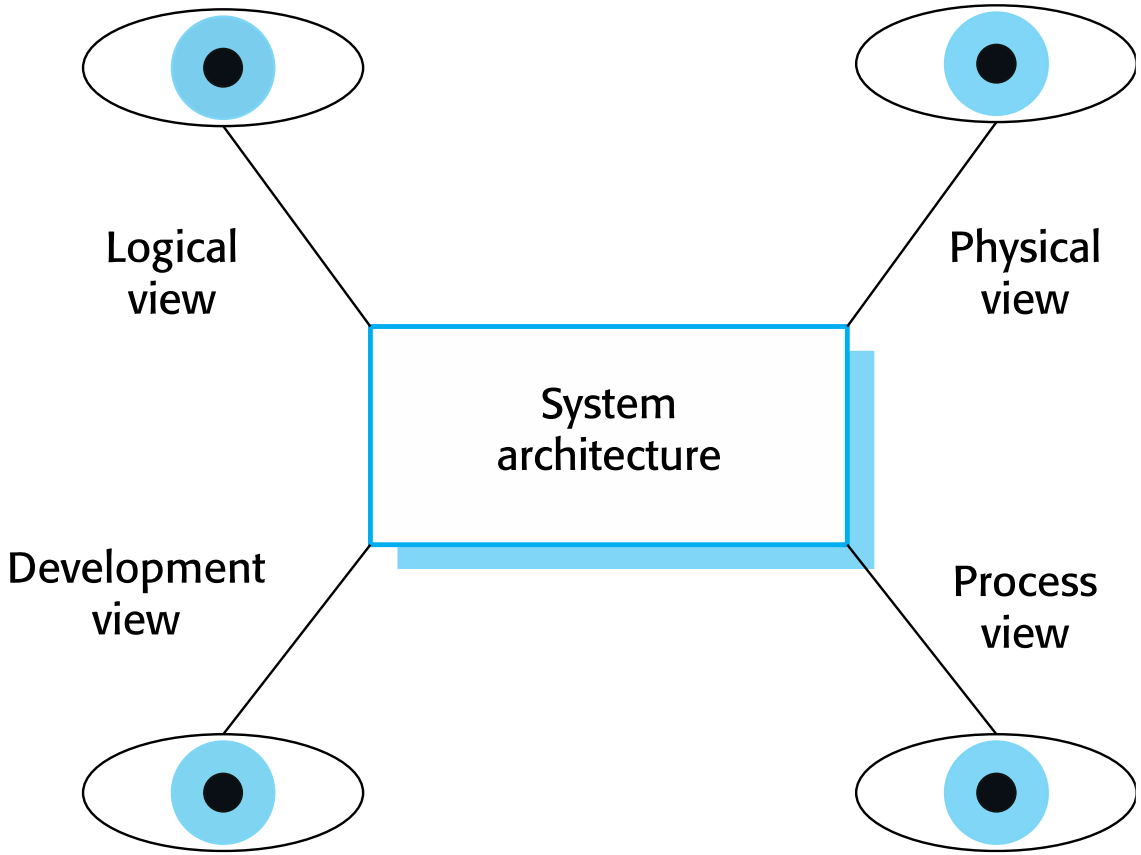
- Acts as a **common language** among stakeholders
- Presents a **shared abstraction** of the system
- Guides development, testing, maintenance, and decision-making

Why this matters:

- Stakeholders use architecture to **ask questions**, **raise concerns**, and **agree on direction**.
- Misunderstood or undocumented architecture leads to **misaligned expectations** and **technical failure**.

ARCHITECTURAL VIEWS

Architectural views



Why Do We Use Views?

The Challenge?

- A complex system cannot be understood or described effectively with a **single diagram or model**.

The Solution:

- Use a **set of interrelated views**, each representing specific aspects of the system for different stakeholder concerns.

This makes architecture:

- Understandable, Scalable, Stakeholder-focused

What Is an Architectural View?

- A **view** is a representation of part or all of a system's architecture, **from the perspective of specific stakeholder concerns**.

An Architecture View Includes:

- Scope (which part of the system?)
- Element types (components, services, nodes)
- Stakeholders/audience
- Concern(s) addressed (performance, modifiability, etc.)
- Level of detail (high-level vs. low-level)

Views help **communicate structure, decisions, and quality properties** clearly.

What Is a Viewpoint?

- A **viewpoint** is a **template or guideline** that tells us how to construct a specific type of view.

Each viewpoint defines:

- The **stakeholders** involved
- The **concerns** it addresses
- The **conventions**, **notations**, and **principles** to be used

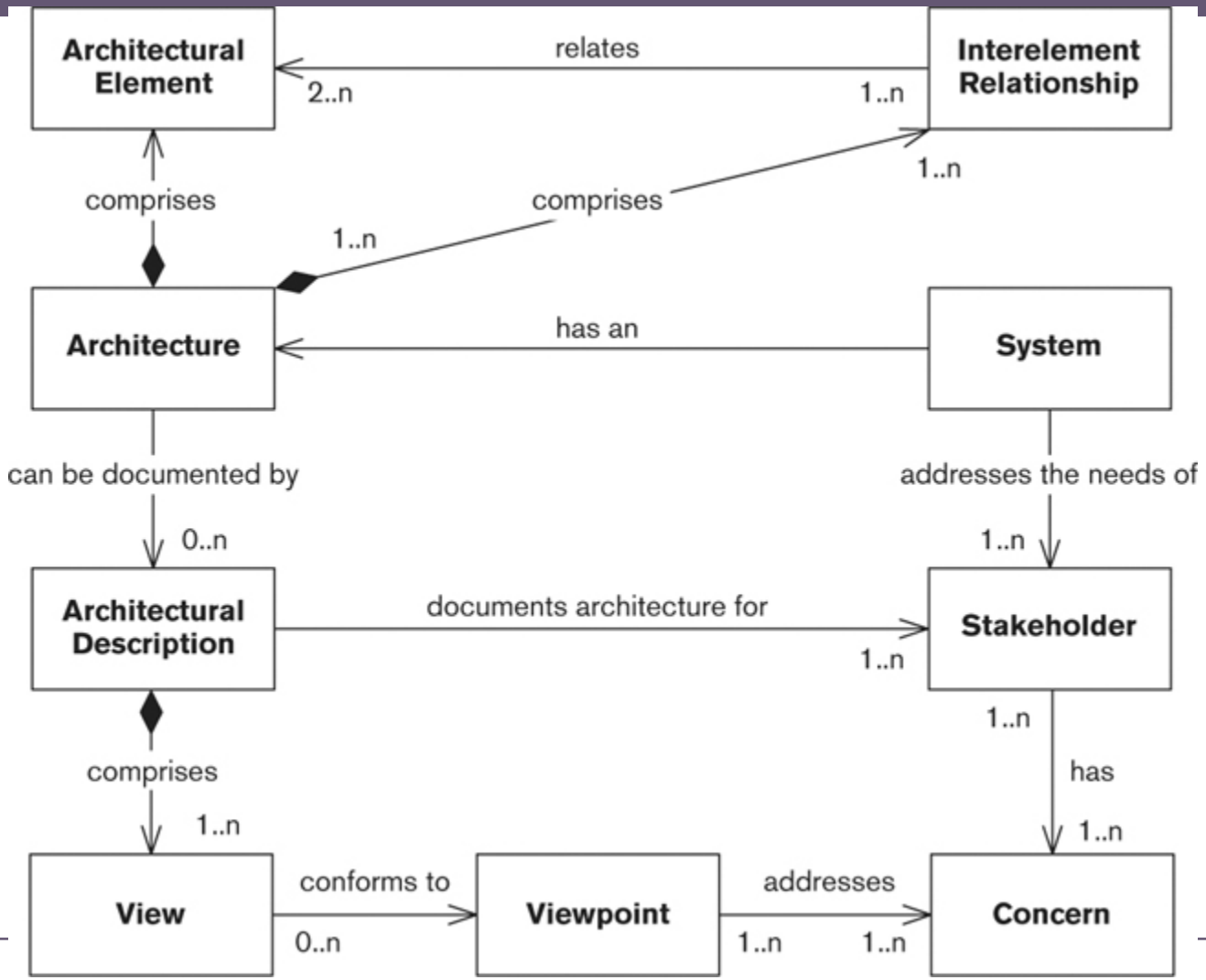
It's like a **lens** through which we look at the architecture.

How Viewpoints and Views Relate

Concept	Definition	Example
Viewpoint	A rulebook for creating a type of view	“Logical Viewpoint” defines how to model component structure
View	A specific model constructed using a viewpoint	“Login System Logical View” showing classes/modules and interfaces

Architectural views help **break down complexity**, and viewpoints ensure they are **consistent and purposeful**.

Relationships Views and Viewpoints



ARCHITECTURAL VIEW MODEL

Architectural View Models?

A **view model** is a structured framework that:

- Defines a **set of standard views**,
- Provides **guidance** on how to build them,
- Ensures different stakeholders' concerns are addressed consistently.

"View models act as the **architect's toolbox** — helping you decide *what views to create*, *why*, and *how*."

Classical & Standard-Based Architectural View Models

Model / Framework	Year	Key Views / Structure	Relevance
4+1 View Model *(Kruchten, RUP)*	1995	Logical, Development, Process, Physical + Scenarios	Widely taught in academia and UML-based projects
RM-ODP *(Reference Model for Open Distributed Processing)*	1995	Enterprise, Information, Computational, Engineering, Technology	Used in distributed, telecom, and defense systems
Siemens Four View Model	1999	Conceptual, Module, Execution, Code	Internal to Siemens; academic reference
IEEE 1471 *(now ISO/IEC 42010)*	2000+	Meta-model: stakeholders, concerns, views, viewpoints	Official global standard for architectural description
Garland & Anthony Model *(RUP context)*	2003	Aligns architectural models with Rational Unified Process	Historically used with RUP tools
Rozanski & Woods	2005	Functional, Information, Development, Deployment,	Popular in industry; practical guidance aligned

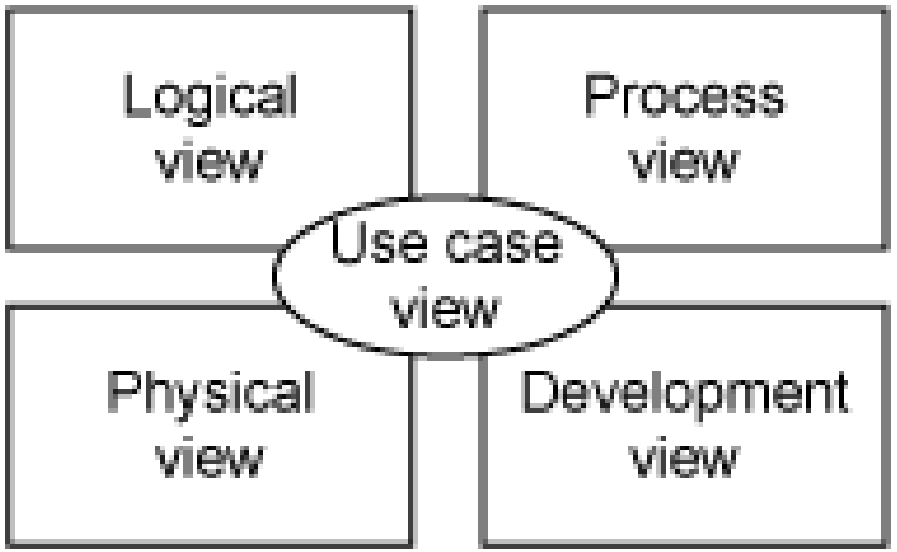
Modern / Agile-Aligned View Models

Model / Framework	Year	Key Views / Structure	Relevance
TOGAF ADM Views	2009+	Business, Application, Data, Technology	Widely used in enterprise architecture (EA)
Arc42	2017+	Template-based; blends functional, technical, deployment views	Practical, lightweight, ISO 42010-based; popular in EU
C4 Model *(Simon Brown)*	2016+	Context, Container, Component, Code	Popular in agile/dev teams; simple and scalable
SAFe Architecture Views	2018+	Enterprise, Solution, and System views	Used in large-scale agile orgs (SAFe Framework)

Architectural View Model: 4+1

- **Creator:** Philippe Kruchten, 1995.
- **Purpose:** Multiple, concurrent perspectives for complex system architecture.
- **Benefit:** Enhanced communication & collaboration among stakeholders.
- **The Views: 4 Primary:** Logical, Process, Physical, Development.

+1 Central: Use Case View (integrates & drives the others).



4+1: The Logical View

- **Focus:** Functional requirements & system decomposition.
- **Elements:** Classes, objects, and their relationships.

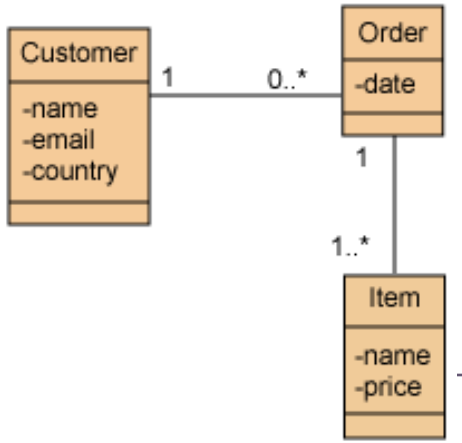
Representations:

- Class Diagrams (static structure)
- Sequence Diagrams (object interactions over time)
- Collaboration/Communication Diagrams (object interactions)
- **Goal:** Ensure functional coverage & conceptual clarity.

Logical Views: Key Diagrams

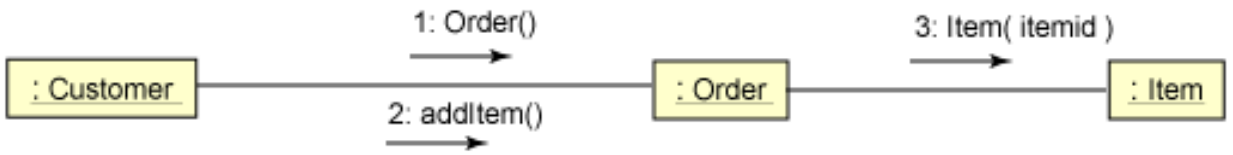
Class Diagram

- **Purpose:** Illustrates the **static structure** of the system, showing core entities (classes) and their relationships.



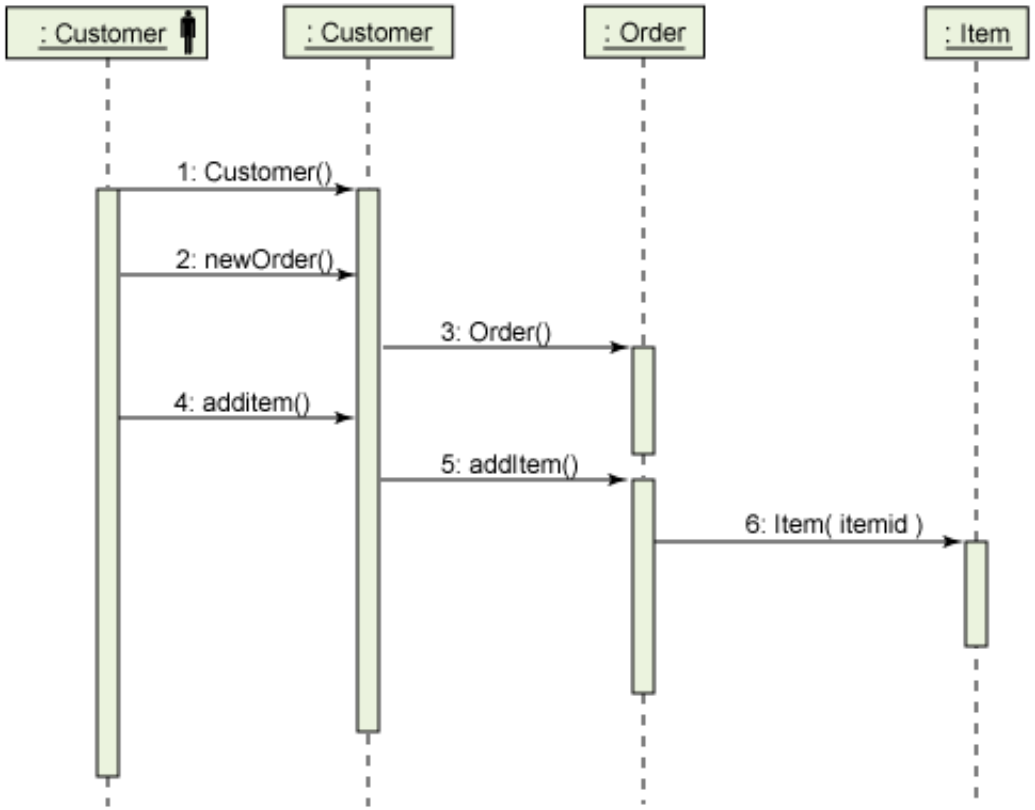
Collaboration (Communication) **Diagram**

Purpose: Depicts **object interactions** and message flow for a specific scenario, focusing on roles and relationships.



Logical Views: Sequence Diagram

- **Purpose:** Illustrates the **temporal order** of messages and interactions between objects over time.
- **Benefit:** Ideal for detailing and **fine-tuning** specific system scenarios and interaction logic.

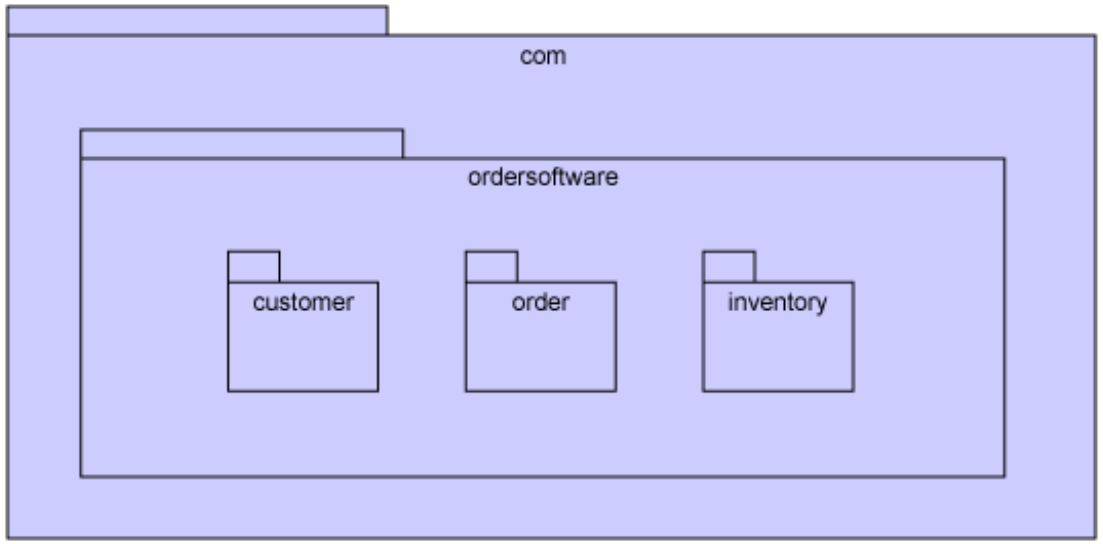


4+1: The Development View

- **Focus:** Describes the **static organization of software modules** for development and implementation.
- **Key Elements (Modules):** Larger building blocks than classes (e.g., **packages, subsystems, libraries**). These represent units of source code, compilation, and often deployment.
- **Purpose:** Helps manage complexity, define development structure, and organize the **actual files and directories** within the development environment (e.g., in a layered architecture).
- **Primary Tool:**

  llustrated using

  UML Package Diagrams.

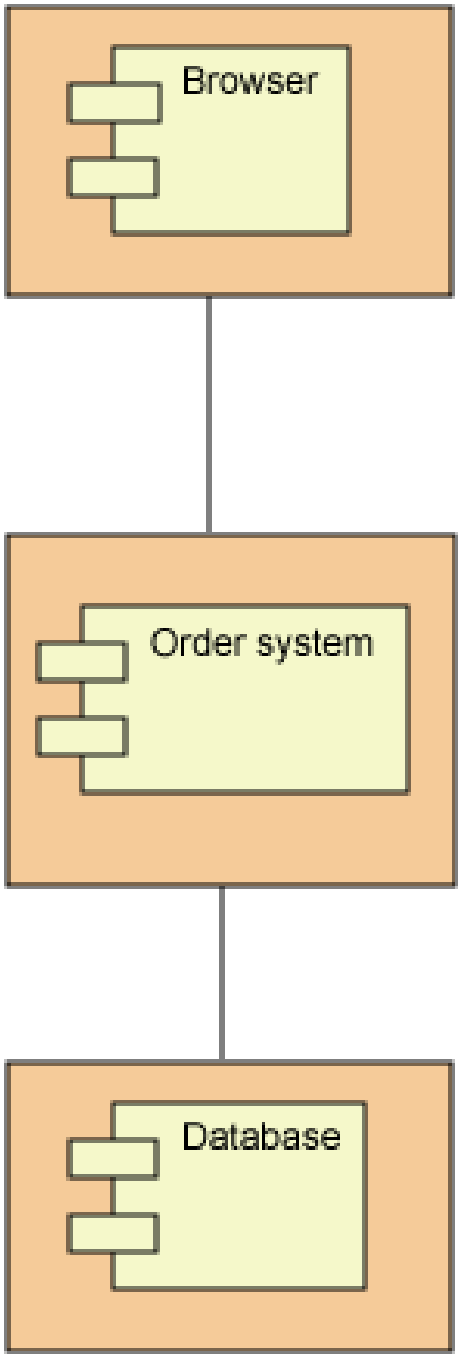


4+1: The Process View

- **Focus:** Describes the system's **runtime behavior, concurrency, and distribution**. It models the executable aspects.
- **Elements:** Consists of **independently executing components** (processes, tasks, threads) and their communication.
- **Purpose:** Crucial for addressing **non-functional requirements** like performance, scalability, reliability, and security.
- **Primary Tool:** Often visualized with **UML Activity Diagrams** or custom concurrency diagrams.

4+1: The Physical View

- **Focus:** Describes the system's **hardware topology** and how software components are **mapped and executed on physical nodes**.
- **Elements:** Consists of **nodes** (e.g., computers, servers, devices) and the **communication networks** connecting them.
- **Purpose:** Addresses critical **non-functional requirements** such as availability, reliability, performance, and scalability in a distributed environment.
- **Primary Tool:** Illustrated using **UML Deployment Diagrams**



4+1: The Use Case View (The "+1" View)

- **Focus:** Captures **functional requirements** from the perspective of external actors (users or other systems).
- **Role:** It **drives and validates** all other architectural views by describing how the system behaves under various scenarios.
- **Elements:** Consists of **use cases** and their corresponding **scenarios**.
- **Primary Tool:** Illustrated using **UML Use Case Diagrams**.

Beyond 4+1: The Rozanski & Woods Model

- **Evolution from 4+1:** The Rozanski & Woods (R&W) model is an **extended and refined framework** for documenting software architecture, addressing perceived gaps in earlier models like the 4+1.
- **Comprehensive Coverage:** It defines **seven distinct viewpoints** (instead of 4+1 views), ensuring a holistic understanding of a system from multiple stakeholder perspectives.
- **Each Viewpoint's Role:** Each viewpoint provides a focused perspective on a specific aspect of the system, tailored to different stakeholder concerns and questions.
- **Key Distinction:** R&W emphasize "viewpoints" as *abstractions* that define what aspects are important, leading to actual "views" (diagrams, models, documents) that describe the system for a specific viewpoint.

ARCHITECTURAL STYLES/PATTERNS

Architectural Styles/Pattern

- Stylized abstract description of good practice that has been tested in different systems and environments.
- Describe system organization that has been successful in previous systems
- Examples
  - Client-server (2, 3, …n tire patterns)
  - Layered Architectures
  - Model View Controllers (MVC)

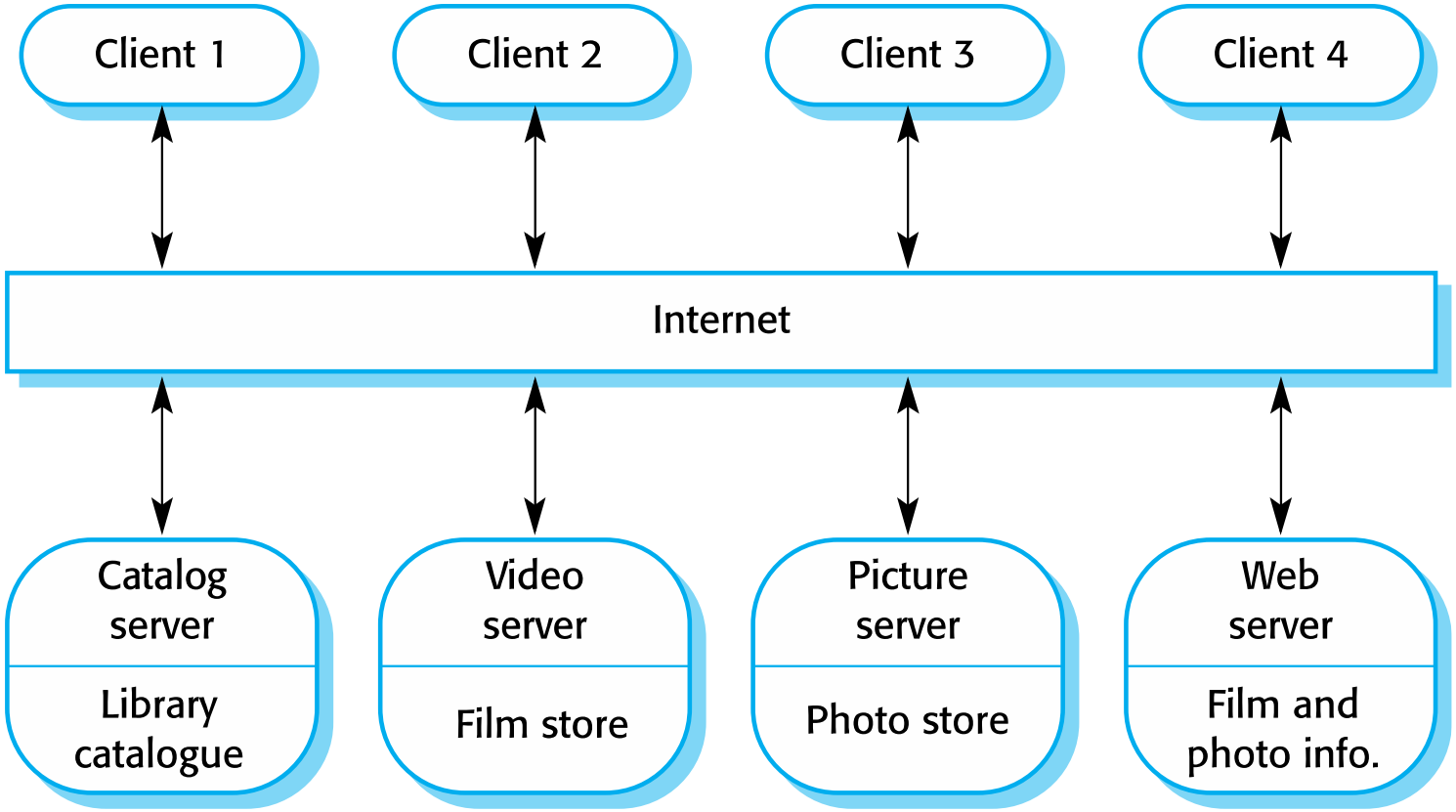
Architectural Patterns

- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when the are not useful.
- Patterns may be represented using tabular and graphical descriptions.

Client-server architecture

- Distributed system model which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

Example: A client–server architecture for a film library



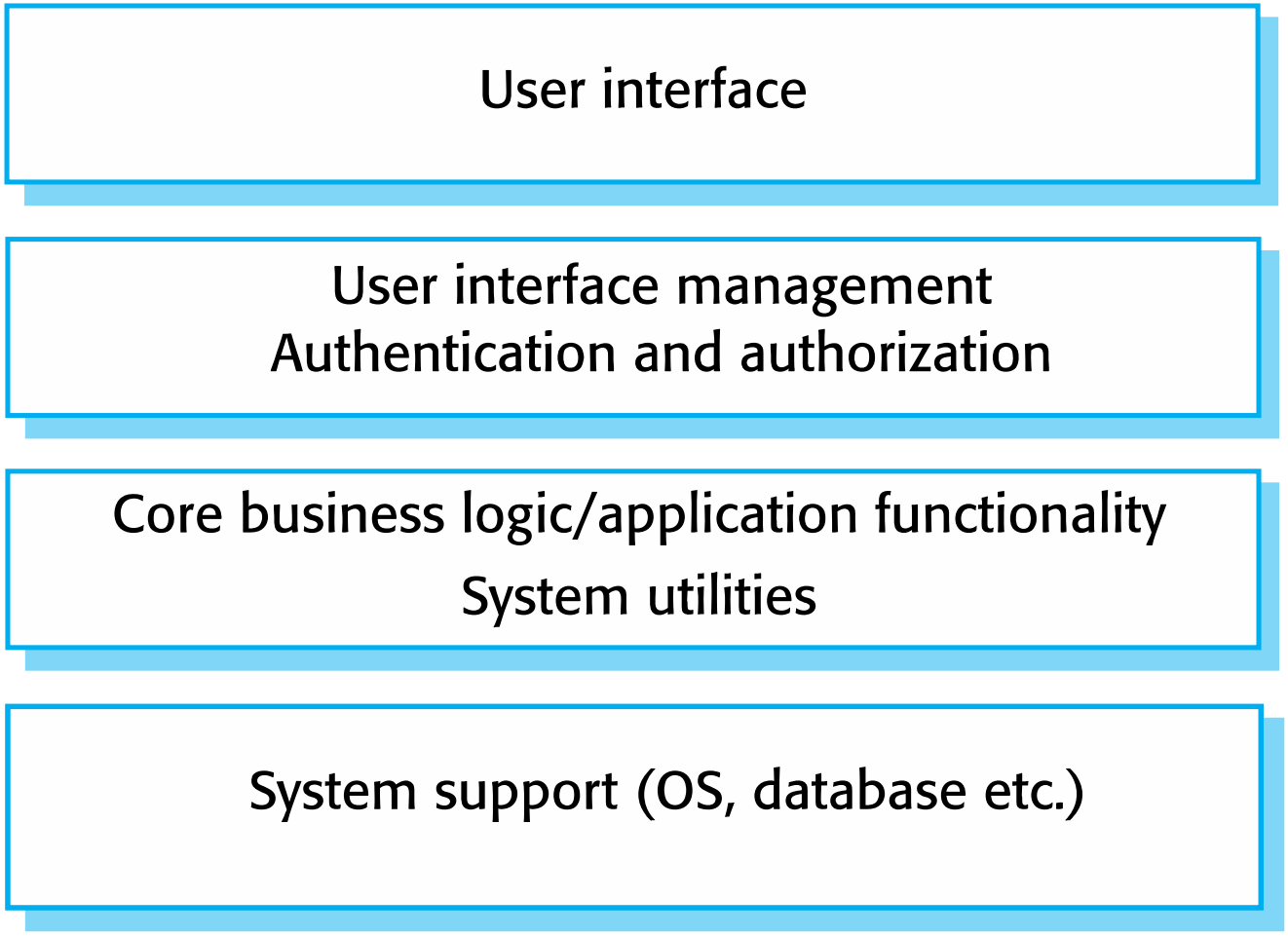
The Client–server pattern

Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Different users accessing a website on the server using their individual machines.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

Layered Architecture

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

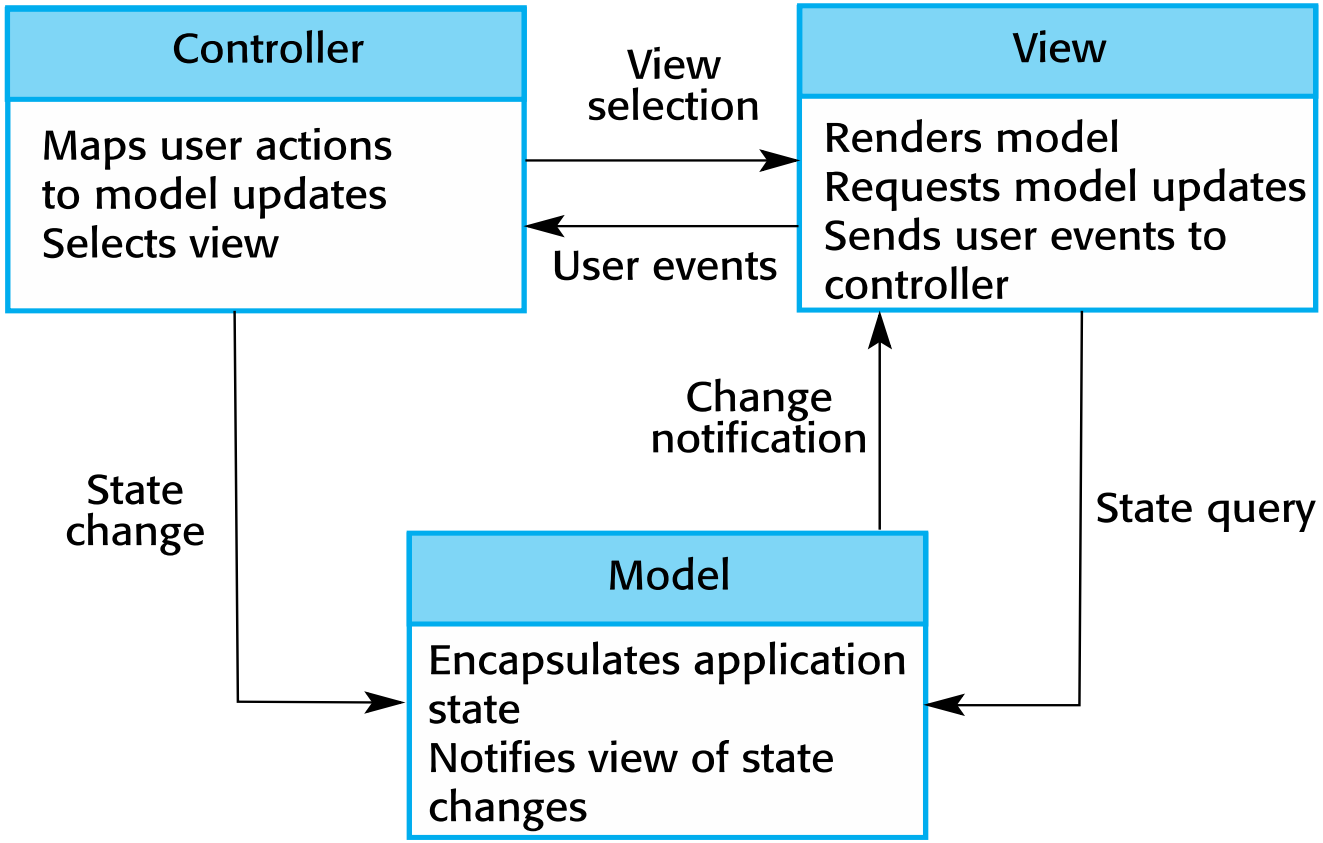
A Generic Layered Architecture



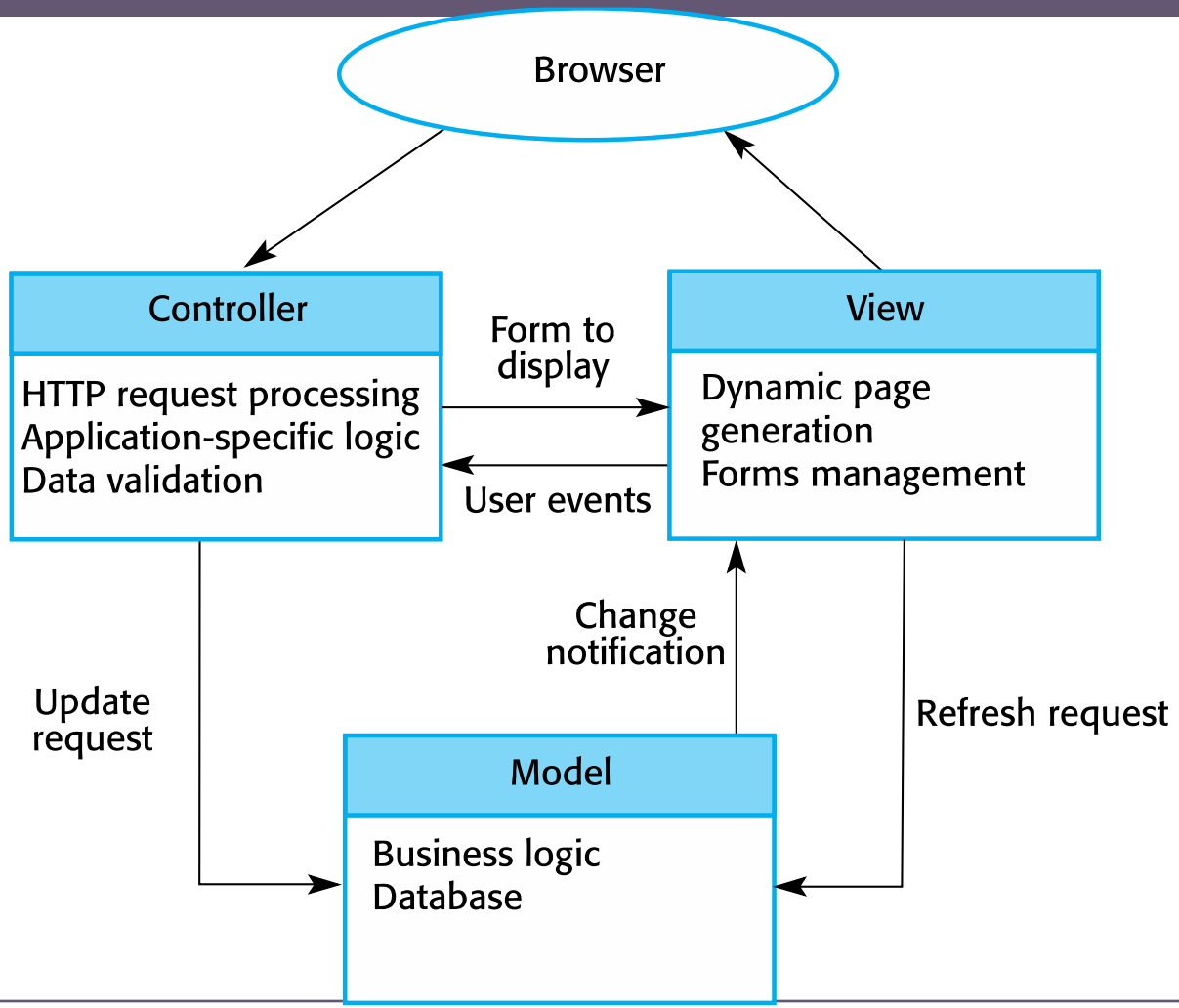
The Layered architecture pattern

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.
Example	A layered model of a system for sharing copyright documents held in different libraries,
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

The organization of the Model-View-Controller



Web application architecture using the MVC pattern



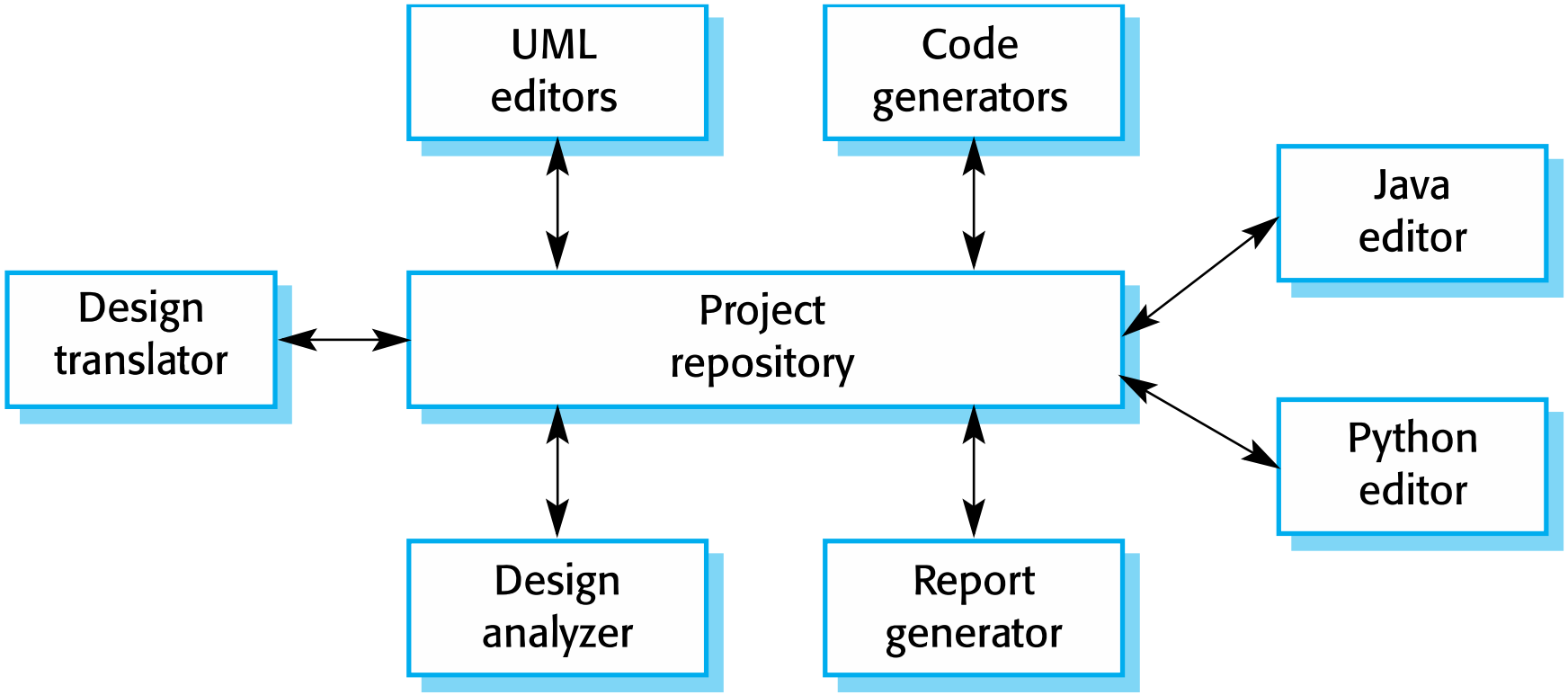
The Model-View-Controller (MVC) pattern

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.
Example	the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

Repository Architecture

- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used this is an efficient data sharing mechanism.

Example: A repository architecture for an IDE



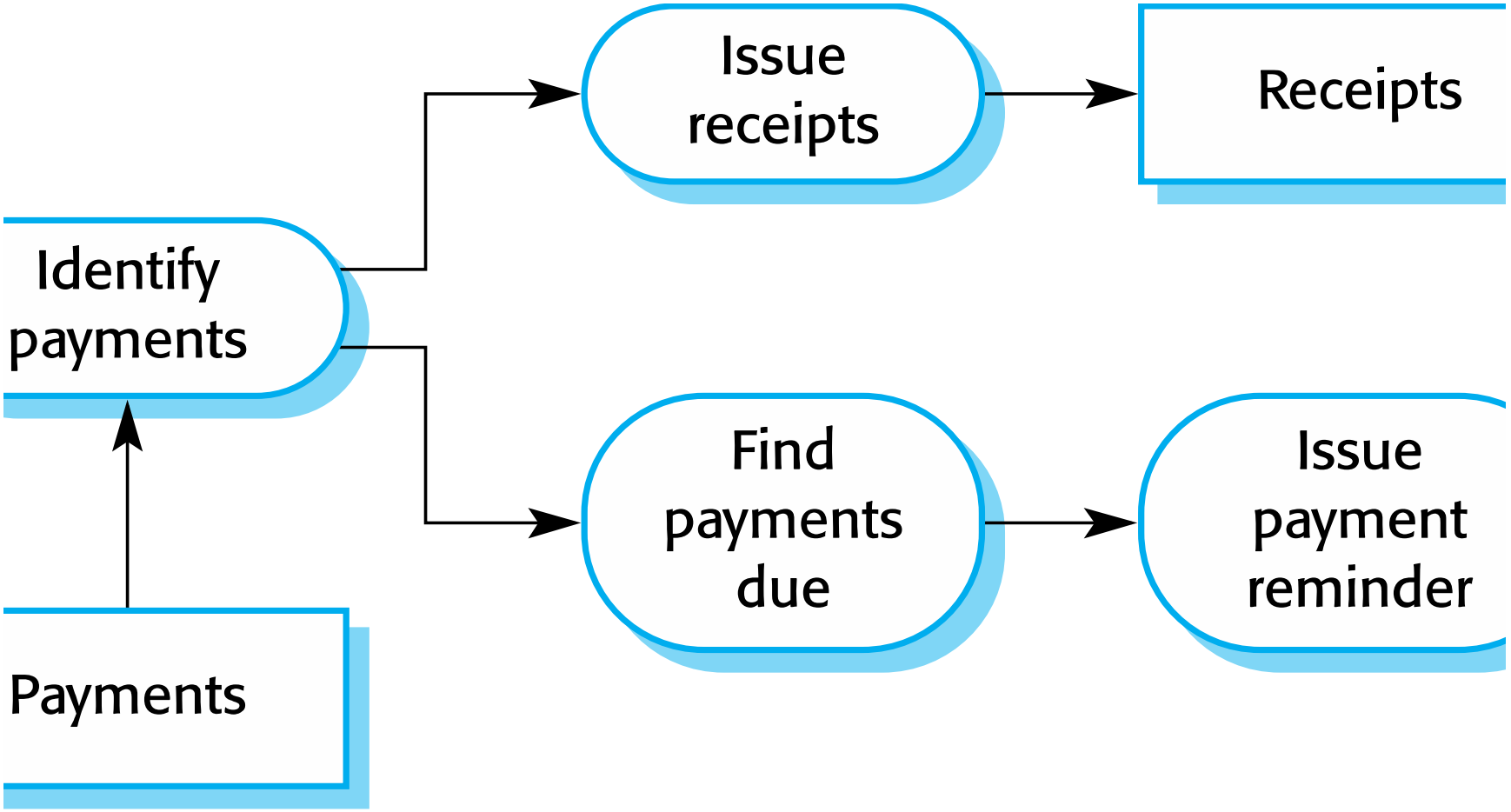
The Repository Pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

Pipe and Filter Architecture

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

An example of the pipe and filter architecture used in a payments system



The pipe and filter pattern

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.