Chapter 9: Architectural Design

Software Engineering

BTech Computer Science and Engineering

Duration: 1 Hour

Software Architecture: The Master Plan

Lecture Introduction

Topic: Designing the ‘Big Picture’ of Software Systems

The City Planning Analogy

Opening Hook

The Analogy: Think of building a city versus building a house.

City Requirements: A city needs a master plan—zoning (residential/commercial), transportation networks, and utilities.

Software Parallel: Software Architecture is that master plan for your system.

Why It Matters: It is the single most important design activity because it dictates:

- Performance
- Security
- Maintainability (Ability to change over the next decade).

Learning Objectives

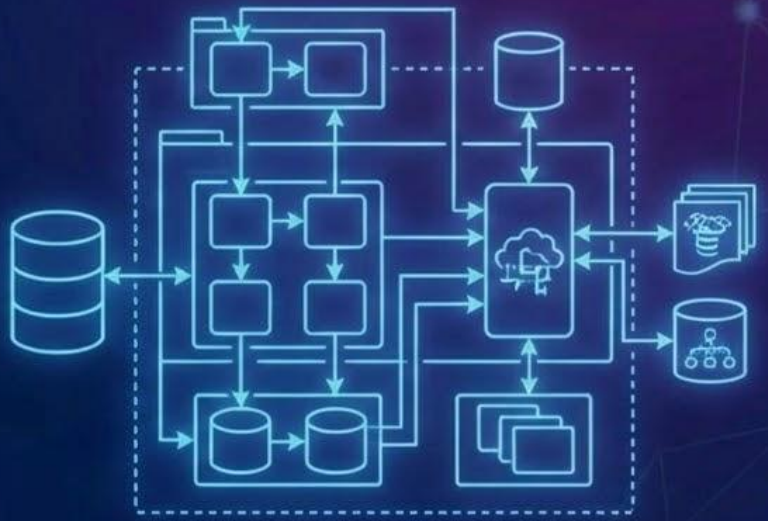
By the end of this lecture, you will be able to:

- **Define Architecture**: Clearly define software architecture and articulate its critical importance.
- **Distinguish Concepts**: Differentiate between architectural genres and architectural styles.
- **Describe Patterns**: Explain key architectural styles and patterns (e.g., Layered, Pipe & Filter).
- **Start Designing**: Explain the initial steps of architectural design:
  - Defining the System Context.
  - Identifying the Archetype.

The ‘Big Picture’ View

Visualizing the Scope

- **Architecture vs. Code**: Architecture is about the structure of the system, comprising software components, the externally visible properties of those components, and the relationships between them.
- **The High Stakes**: Architectural decisions are the hardest to change later in the project lifecycle.



Part 1: The Essence of Software Architecture

Section 9.1: Defining the Master Plan

- **Focus:** What it is, why it matters, and how we describe it.

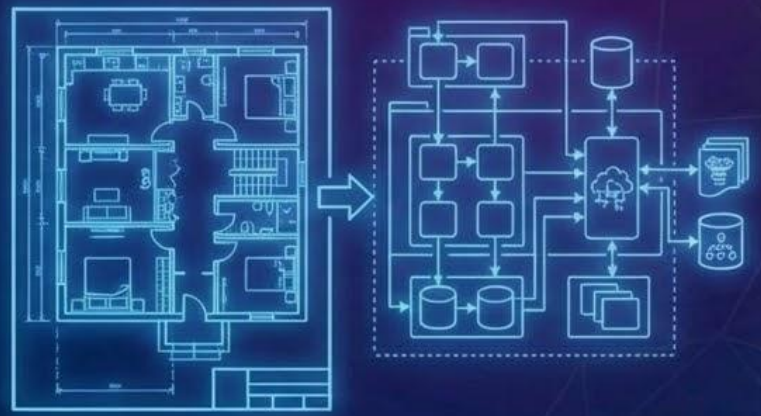
What Is Software Architecture?

Section 9.1.1: The Definition

- **IEEE 1471 Definition:** “The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.”
- **In Simple Terms:** It is the high-level structure—the “big boxes” of the system and how they communicate.
- **The Analogy:**

Building Blueprint:
Shows rooms, hallways, and follows building codes.

Software Architecture:
Shows components, connections, and follows design principles.



Why Is Architecture Important?

Section 9.1.2: The Foundational Impact

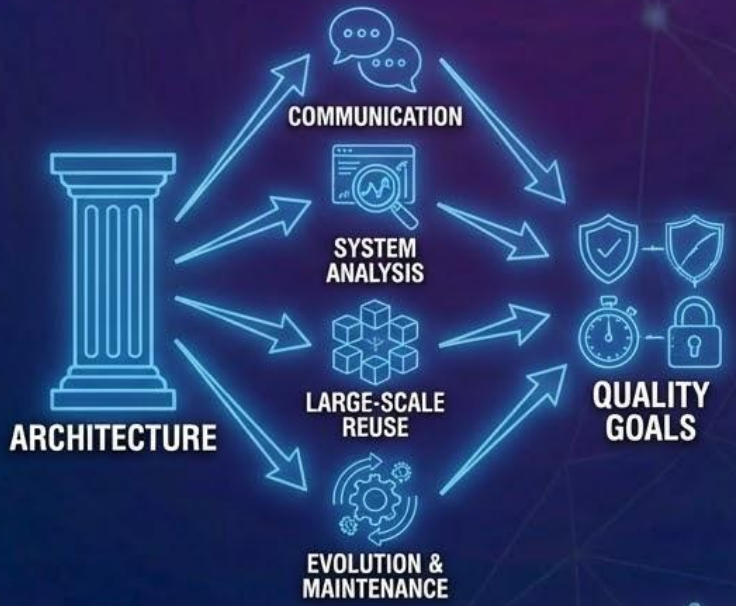
- **The High Stakes:** Architecture represents the earliest design decisions and the hardest to change later.
- **Key Areas of Influence:**

  Communication: A common technical blueprint for all stakeholders.

  System Analysis: Enables early reasoning about quality (e.g., “Can we handle 1 million users?”).

  Large-Scale Reuse: Architectures can be reused across similar product lines.

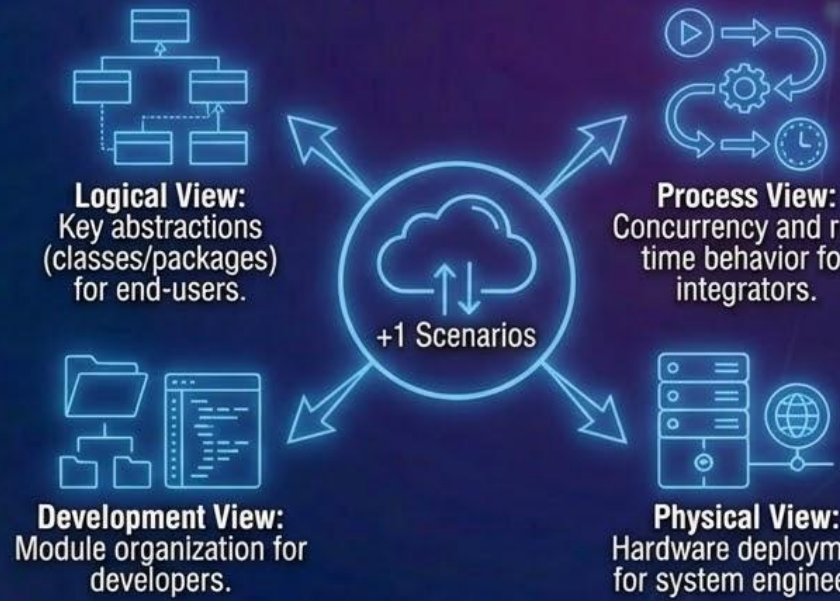
  Evolution & Maintenance: Good architecture localizes change, making the system resilient.
- **Bottom Line:** It enables or hinders every quality goal (performance, security, reliability).



Architectural Descriptions

Section 9.1.3: The 4+1 View Model

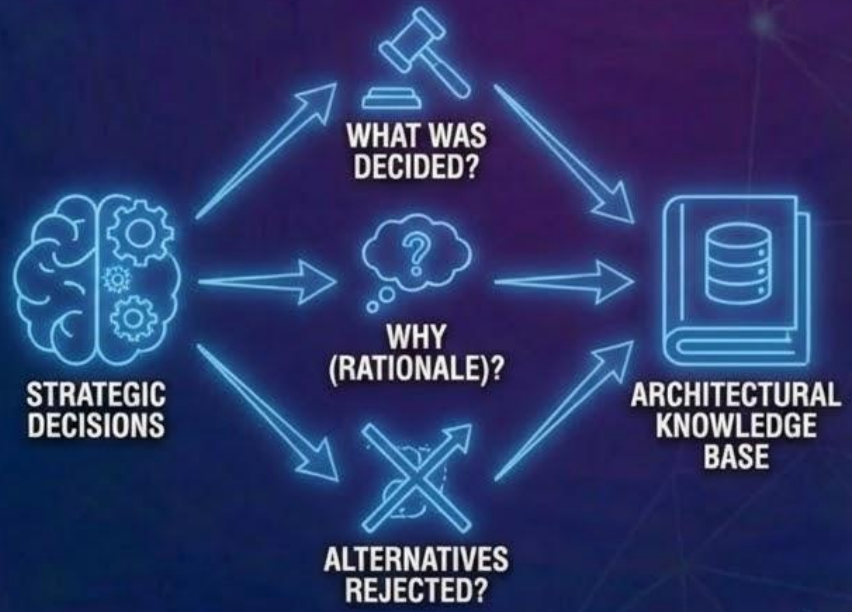
- **The Rule:** A single diagram is never enough to capture a complex system.
- **The Solution (Kruchten's 4+1 Views):**



Architectural Decisions

Section 9.1.4: Strategic Design

- **Nature of Architecture:** It is a collection of conscious, strategic decisions.
- **Documentation is Critical:** You must record:
  - What was decided?
  - Why (the rationale)?
  - What alternatives were rejected?
- **Value:** This creates an “Architectural Knowledge Base” that is essential for future maintainers to understand why the system works the way it does.



Part 2: Architectural Genres & Styles

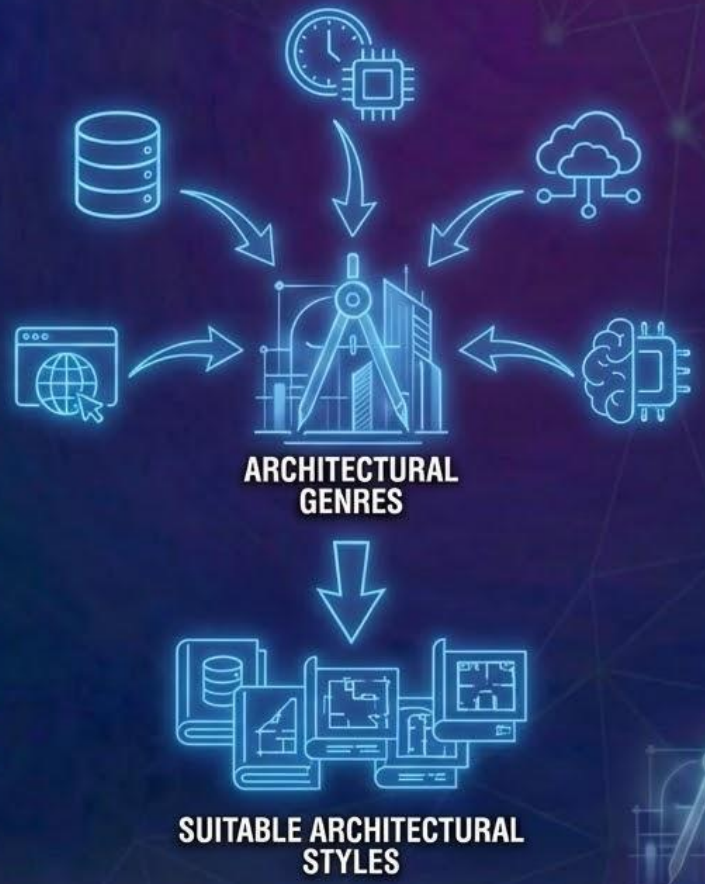
Section 9.2 & 9.3: Categories and Structural Patterns

Focus: Matching System Types to Structural Blueprints

Architectural Genres

Section 9.2: Broad Categories

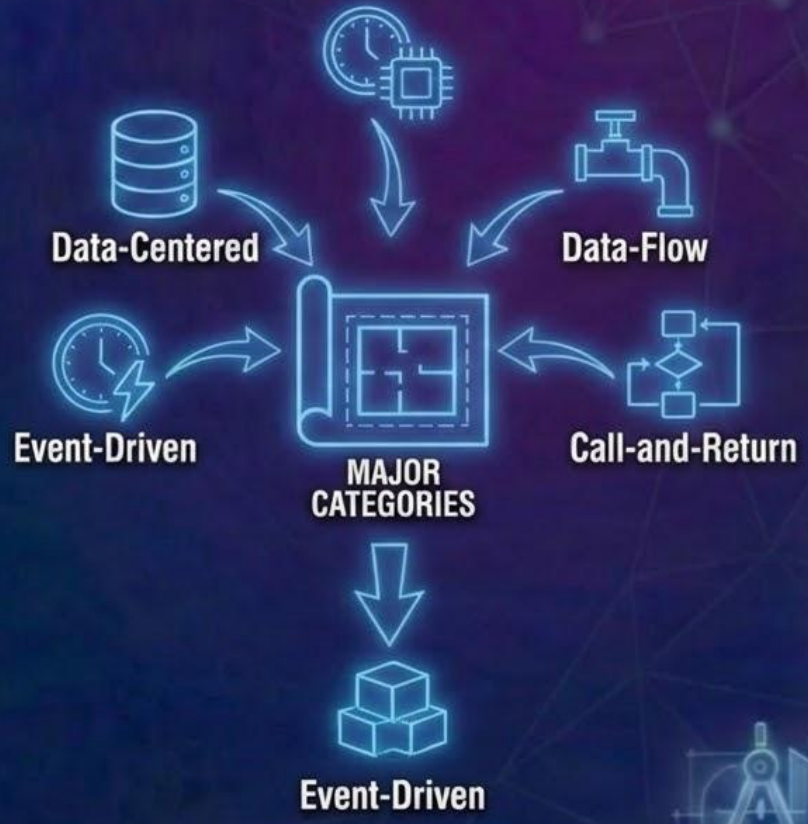
- **Definition:** Broad categories of software systems that suggest certain architectural forms.
- **Common Genres:**
  - **Information Systems:** Focus on databases and transaction processing.
  - **Real-Time & Embedded:** Systems constrained by timing and physical events.
  - **Distributed Systems:** Components communicating across networks.
  - **Web-Based Systems:** Network-centric applications.
  - **AI Systems:** Heuristic and learning-based processing.
- **Importance:** Identifying the genre points you toward suitable architectural styles.



Architectural Styles Overview

Section 9.3.1: A Brief Taxonomy

- **Definition:** An architectural style is a named, coordinated set of architectural constraints.
- **Purpose:** It serves as a pattern for the overall structure of the system.
- **Major Categories:**
  - Data-Centered
  - Data-Flow
  - Call-and-Return
  - Object-Oriented
  - Event-Driven



Data-Centered Architectures

The Repository Model

- **Core Idea:** A central data store (repository) acts as the core, accessed by independent components.
- **Examples:**
  - Repository Architecture: Shared database accessed by various apps.
  - Blackboard Architecture: Used in AI (e.g., speech recognition) where knowledge sources update a shared memory.
- **Pros:** Data consistency; easy integration of new components.
- **Cons:** The repository is a single point of failure and a potential performance bottleneck.

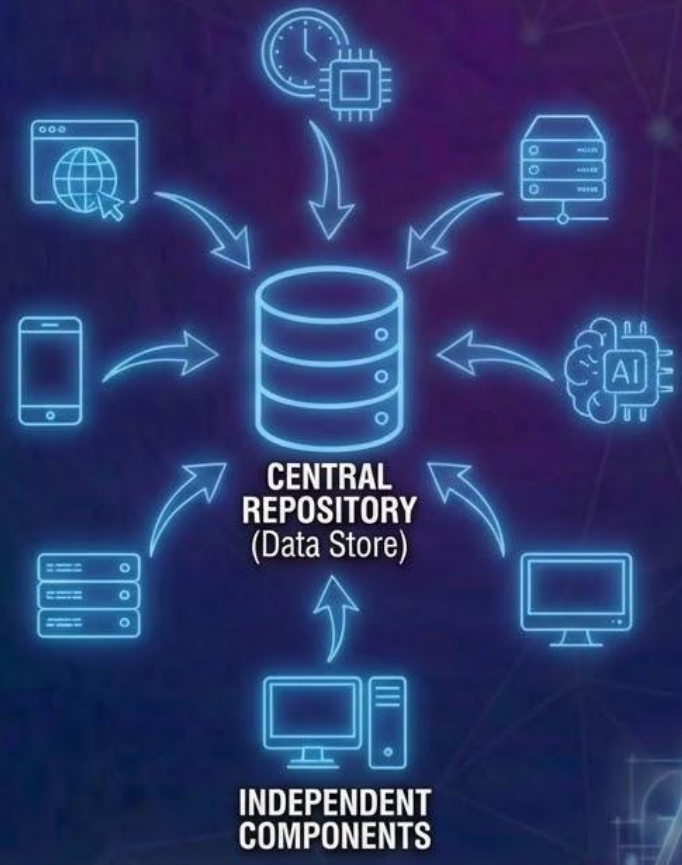
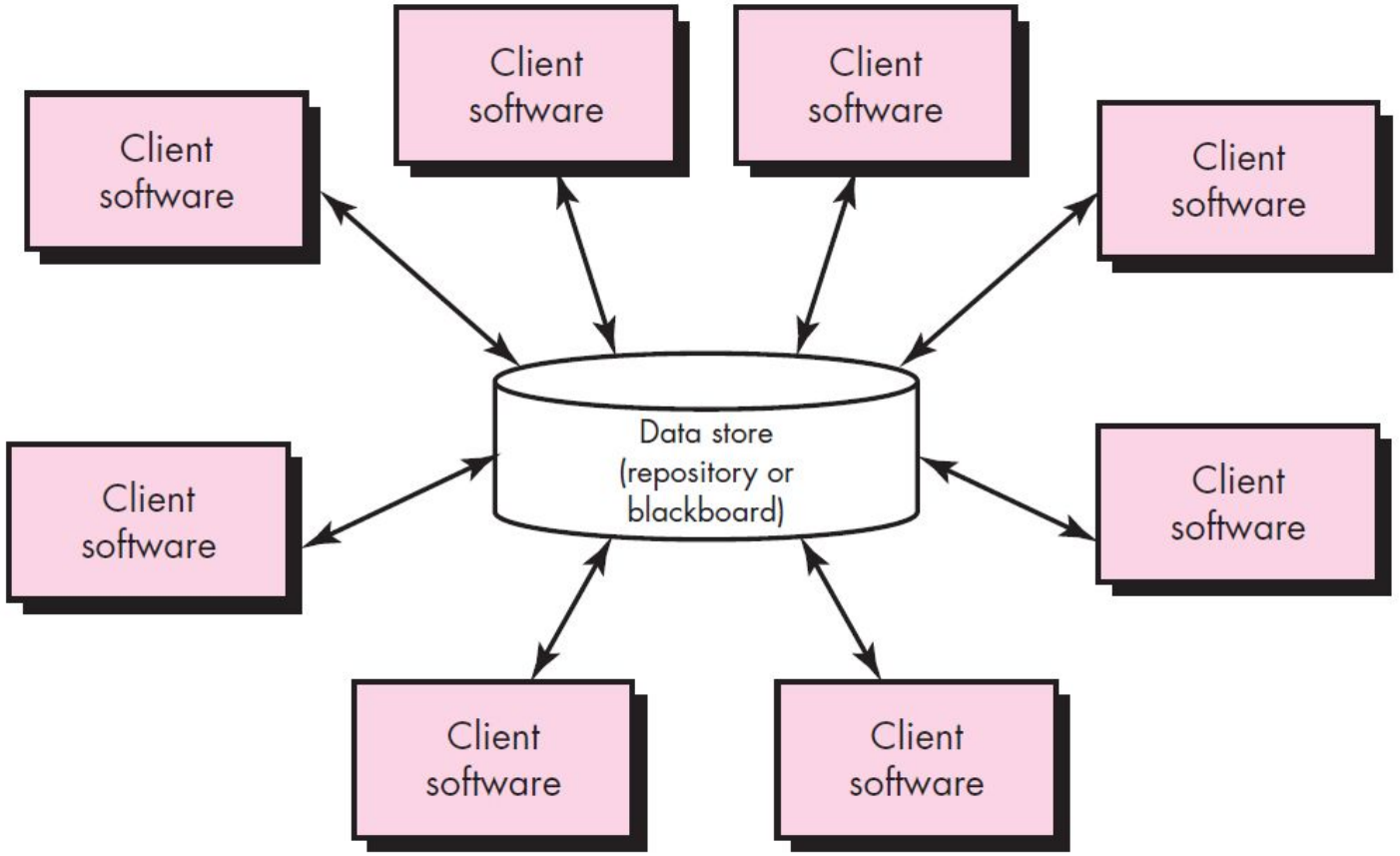


FIGURE 9.1

Data-centered architecture



Data-Flow Architectures

The Pipeline Model

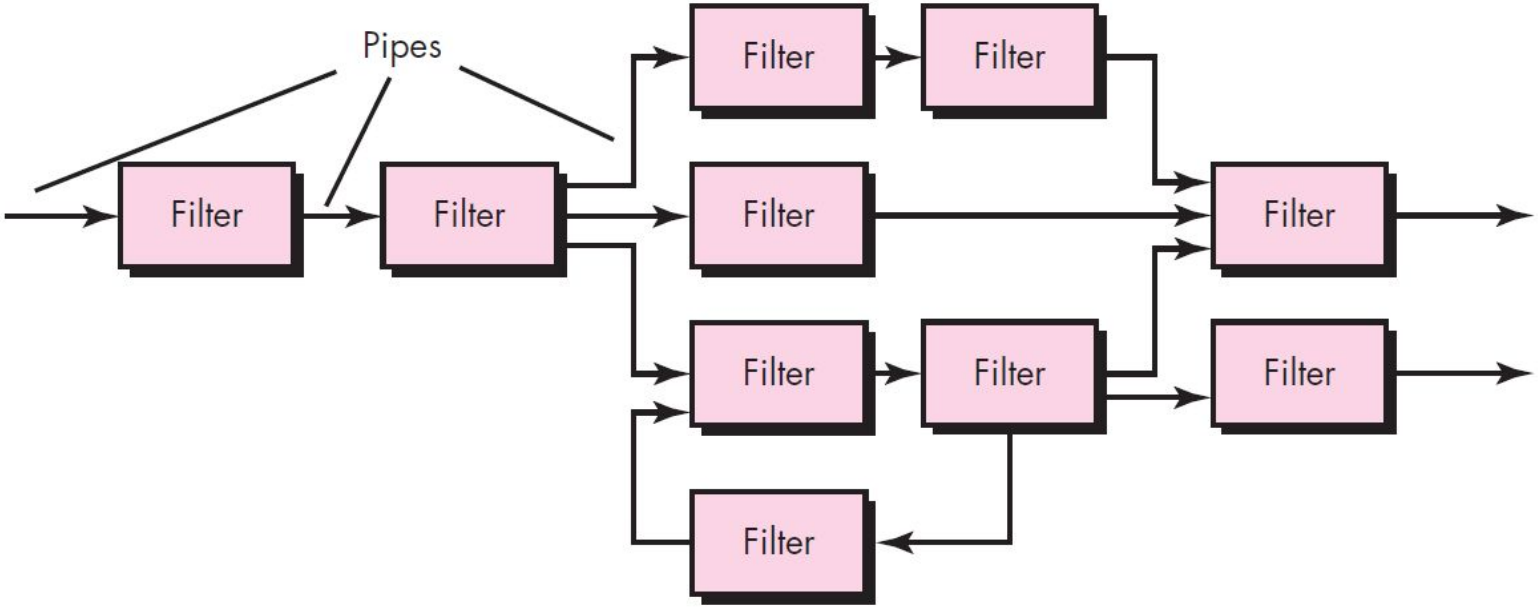
- **Core Idea:** The system is structured as a series of transformations on continuous data streams.
- **Example:** Pipe-and-Filter Style.

- **Unix Command Line:**

  ls | grep .txt | wc -l
- **Pros:** Excellent for batch processing; logical flow is easy to understand.
- **Cons:** Not interactive; often inefficient due to parsing overhead between filters.

FIGURE 9.2

Data-flow architecture



Pipes and filters

Call-and-Return Architectures

The Hierarchical Model

- **Core Idea:** A hierarchical control structure where a main program invokes sub-programs.
- **Examples:**
  - Main Program & Subroutine: Classic structured programming.
  - Layered Architecture: (e.g., OSI Model, 3-Tier Web App).
  - Client-Server: A classic 2-layer variation.
- **Pros:** Strong separation of concerns; layers can be reused or replaced.
- **Cons:** Performance can suffer as data must travel up and down through multiple layers.

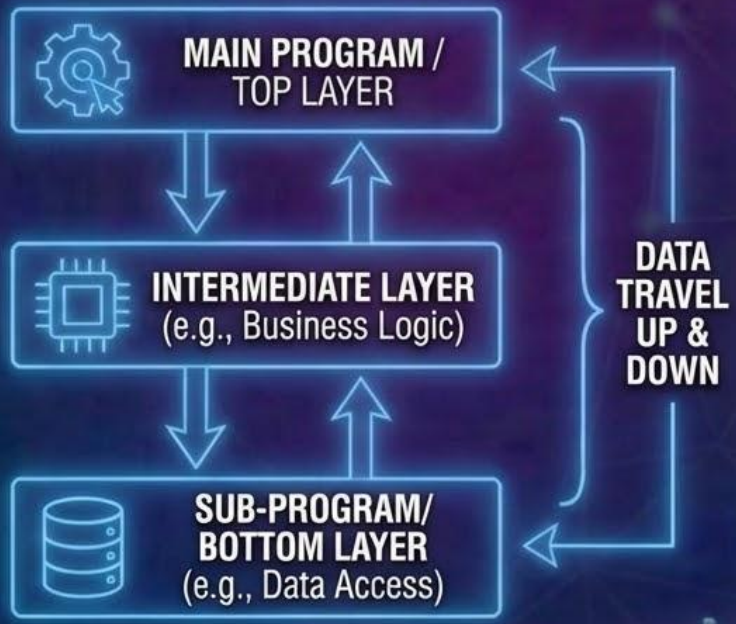
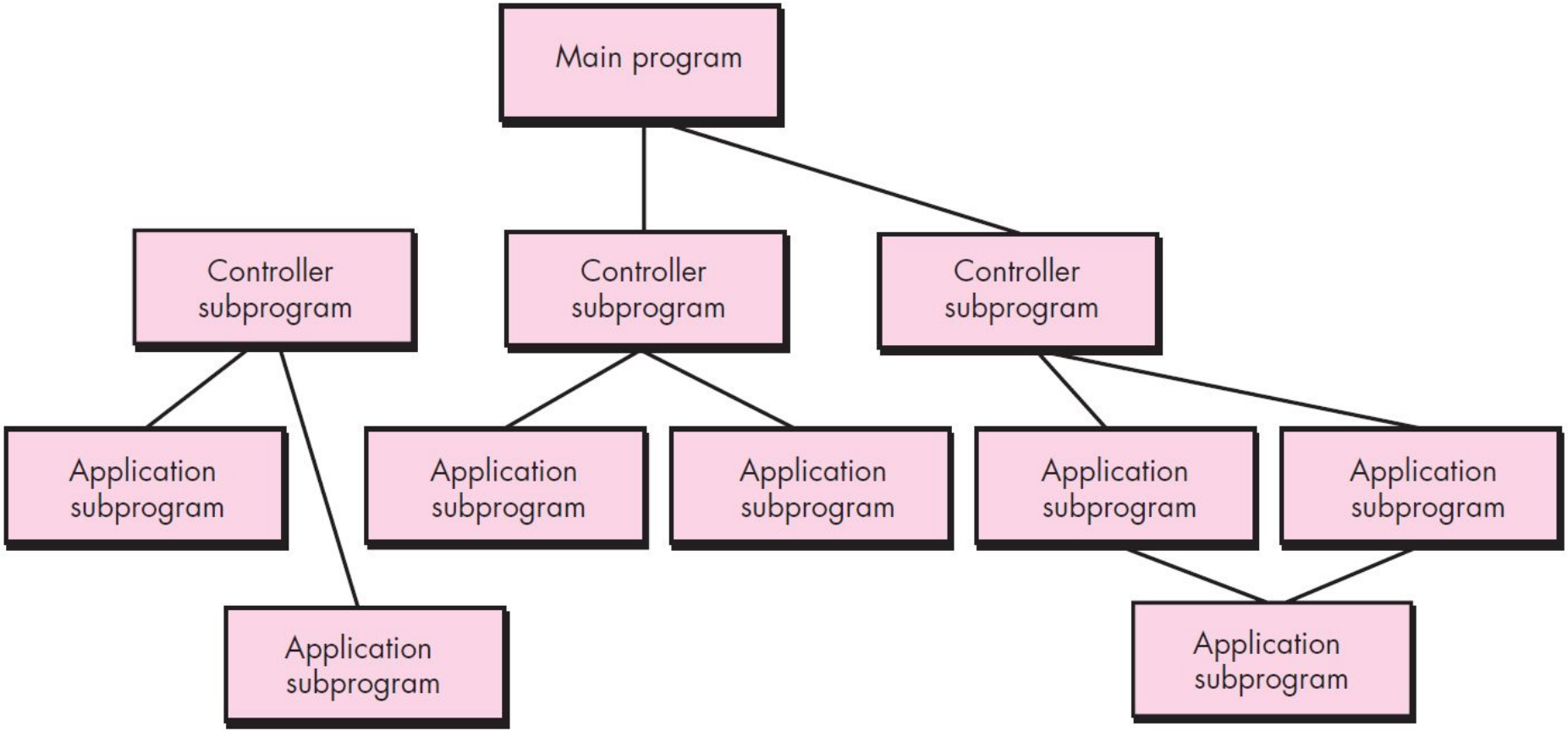


FIGURE 9.3 Main program/subprogram architecture

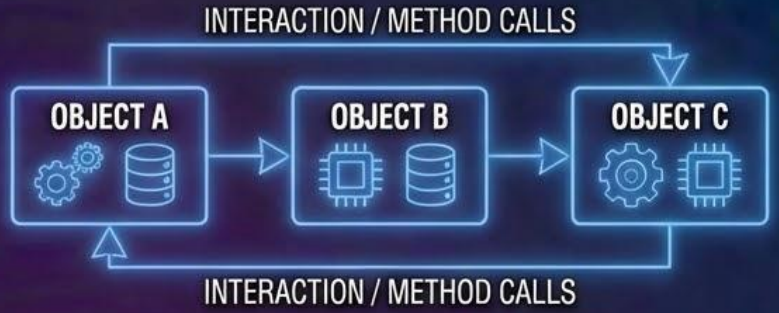


Object-Oriented & Event-Driven Styles

Modularity vs. Decoupling

Object-Oriented Architectures:

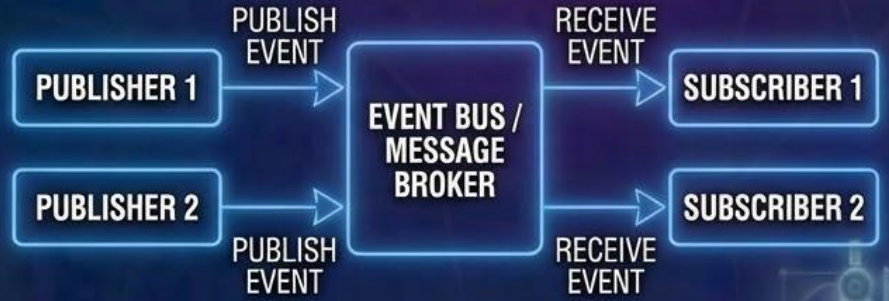
- **Idea:** System decomposed into interacting objects encapsulating data and behavior.
- **Pros/Cons:** High modularity and information hiding, but complex interaction chains can be hard to trace.



ENCAPSULATED OBJECTS & DIRECT INTERACTIONS

Event-Driven Architectures:

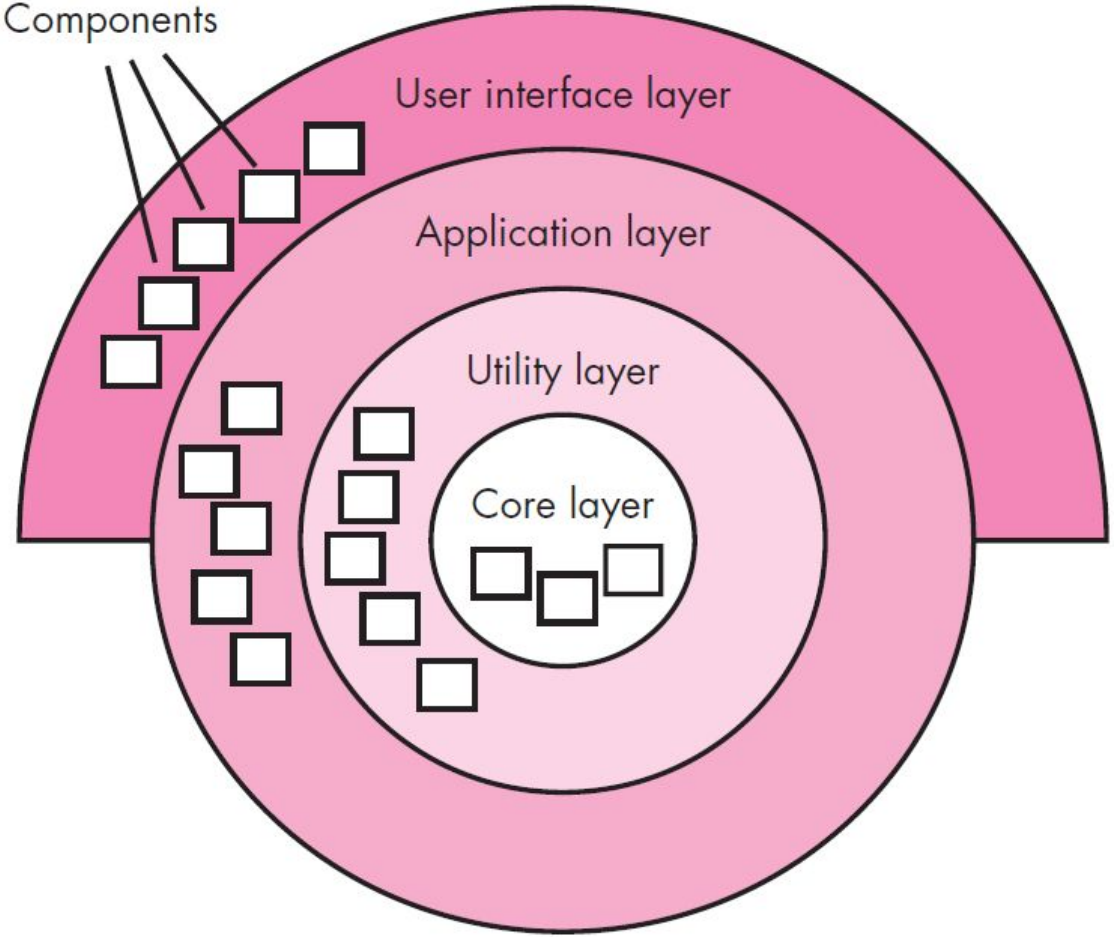
- **Idea:** Components communicate via event announcements (Publish-Subscribe).
- **Pros:** Highly decoupled and easily extensible.
- **Cons:** Control flow is implicit and hard to follow; debugging is challenging.



PUBLISH-SUBSCRIBER MODEL (DECOUPLED)

FIGURE 9.4

Layered architecture

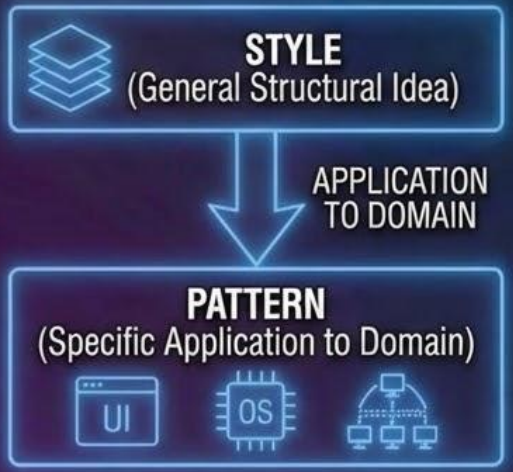


Architectural Patterns vs. Styles

Section 9.3.2: Refining the Concept

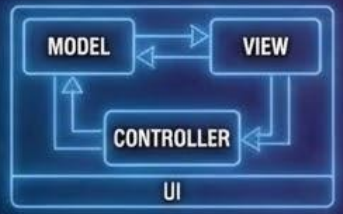
The Distinction:

- **Style**: A general structural idea (e.g., Layered).
- **Pattern**: A specific application of that idea to a problem domain.



Examples:

- **MVC (Model-View-Controller)**: A specific layered pattern for UI applications.
- **Microkernel**: A pattern for Operating Systems.



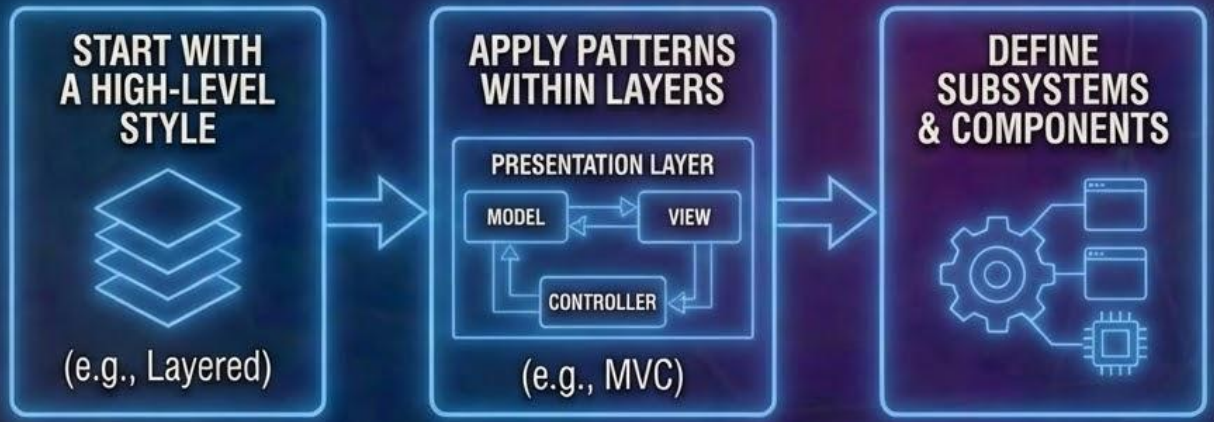
- **Broker:** A pattern for distributed systems (e.g., CORBA).

Organization & Refinement

Section 9.3.3: The Design Process

- **Iterative Approach:**
  Architecture is not formed in a single step.

- **The Flow:**



Part 3: Initiating Architectural Design

Section 9.4: The First Steps of Architecture

- **Focus: Context & Core Abstractions**

Step 1: Representing the System in Context

Section 9.4.1: Defining Boundaries

The Goal: To understand the system's external environment.

Key Question: What external entities (people, systems, devices) does this system interact with?

The Tool: Architectural Context Diagram (ACD) (often synonymous with a Level 0 DFD).

Purpose: Clearly defines the system boundary and the external interfaces required.

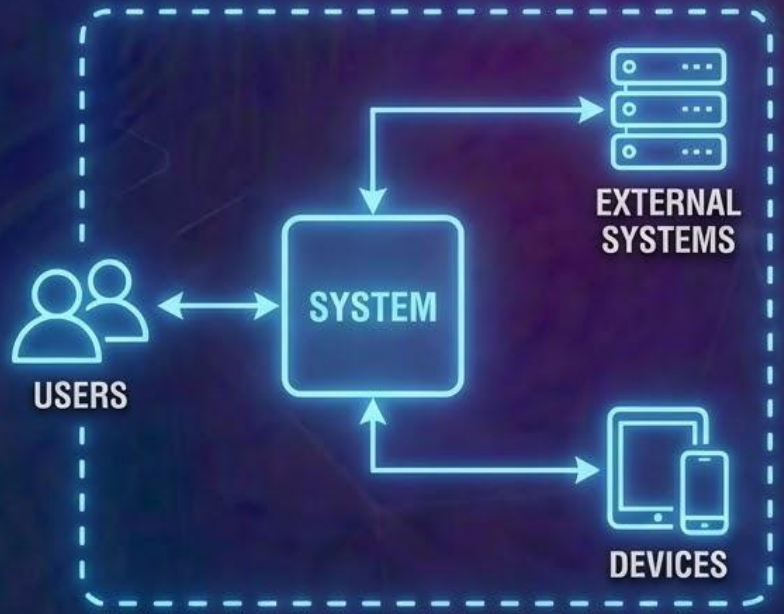


FIGURE 9.5

Architectural context diagram

Source: Adapted from [Bos00].

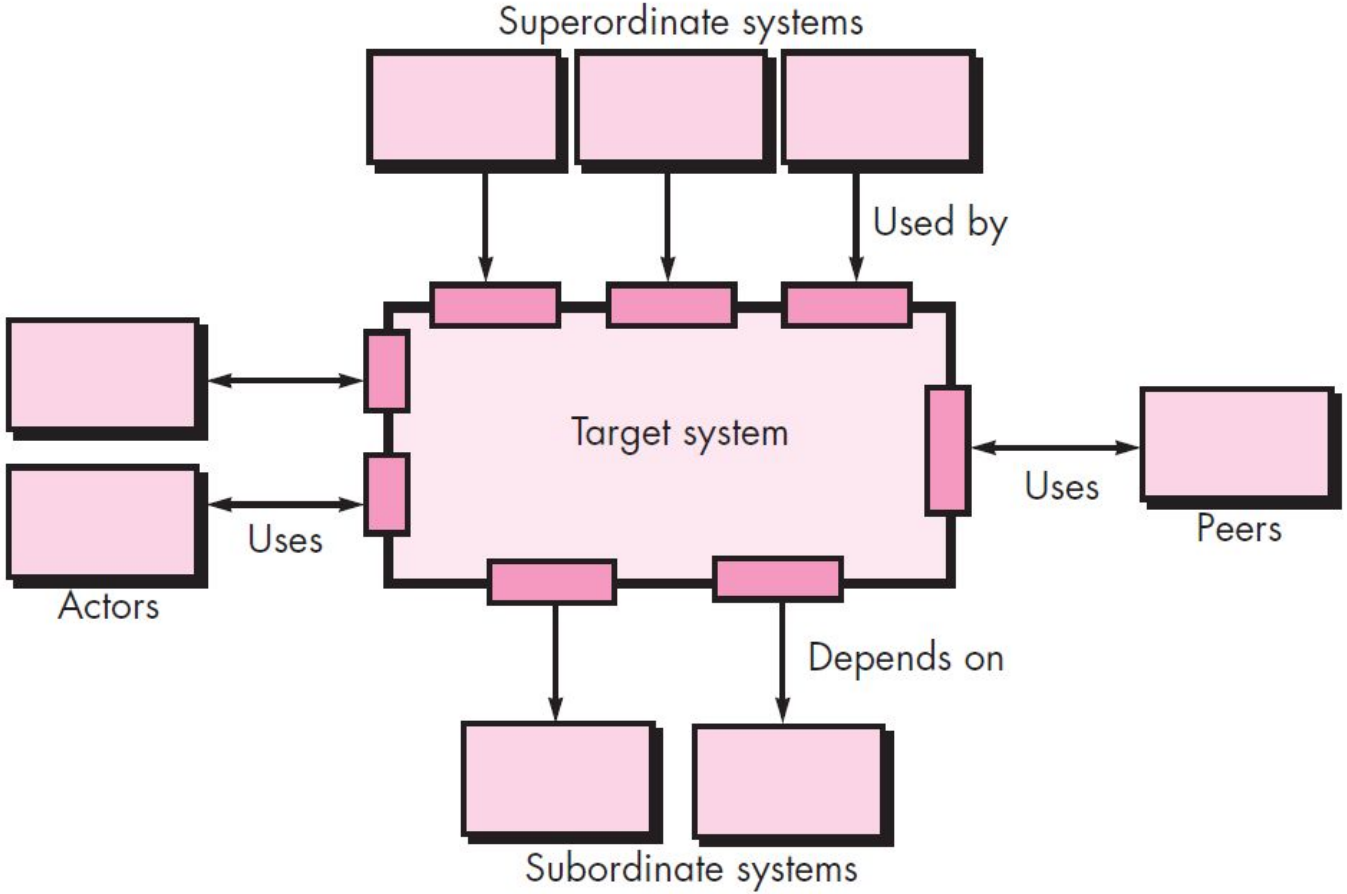
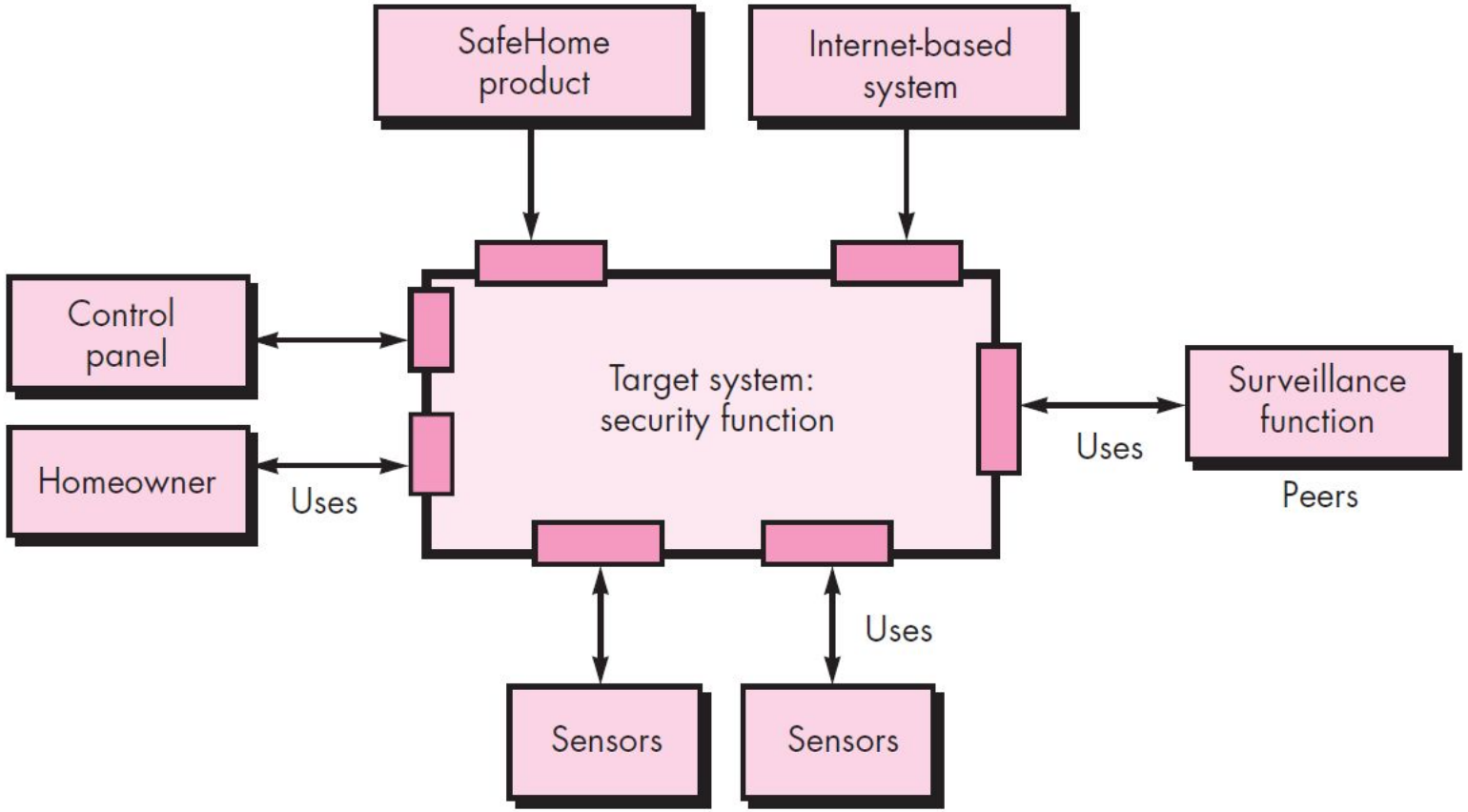


FIGURE 9.6

Architectural context diagram for the *SafeHome* security function

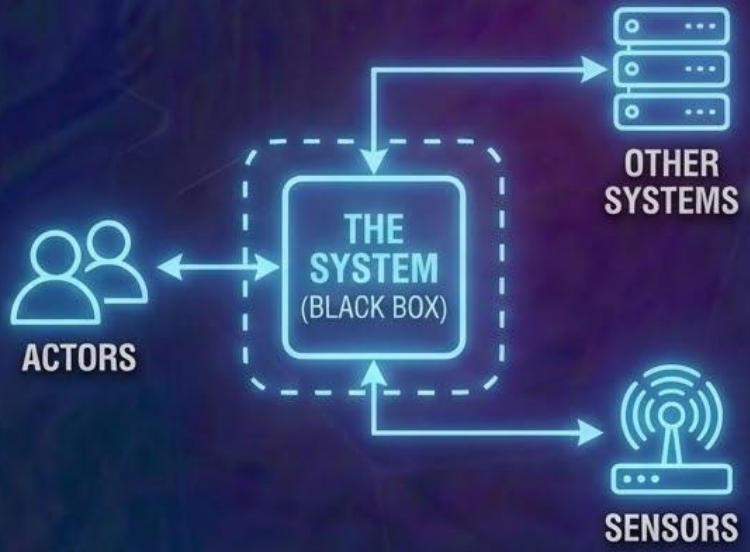


Visualizing the Context

The Architectural Context Diagram (ACD)

Structure:

- **Center:** The System (represented as a single ‘black box’).
- **Surroundings:** External Entities (Actors, other systems, sensors).
- **Connections:** Arrows showing data or control flows between the system and entities.
- **Why it matters:** You cannot design the internal structure until you define exactly what crosses the boundary into the system.



Step 2: Defining Archetypes

Section 9.4.2: Finding the Core

Definition: Archetypes are the most abstract, fundamental abstractions critical to the system's architecture.

Nature: They represent the stable 'stable core' conceptual elements that rarely change.

Discovery Process:

- Look at the Analysis Class Model.
- Identify classes that are central, pervasive, and structurally essential.

Architectural Role: Archetypes form the foundation and often evolve into major subsystems or frameworks.

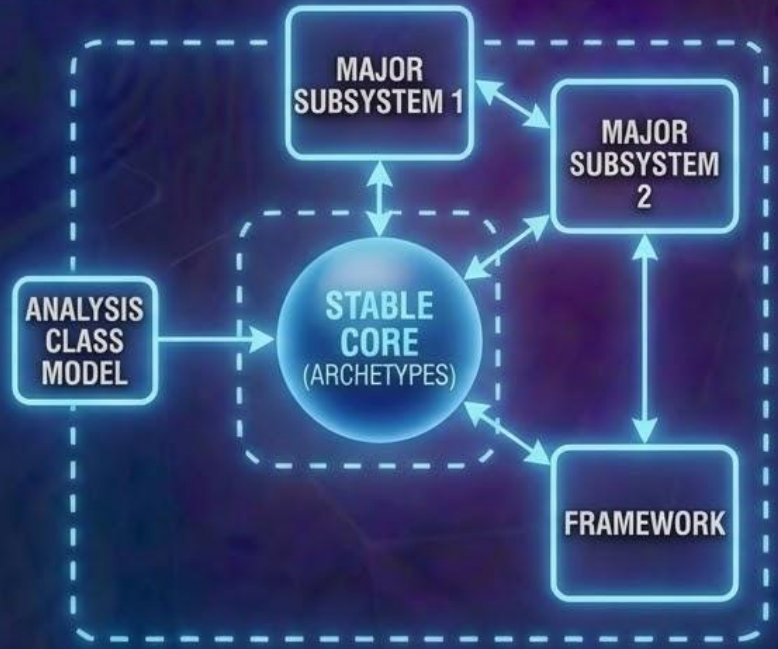
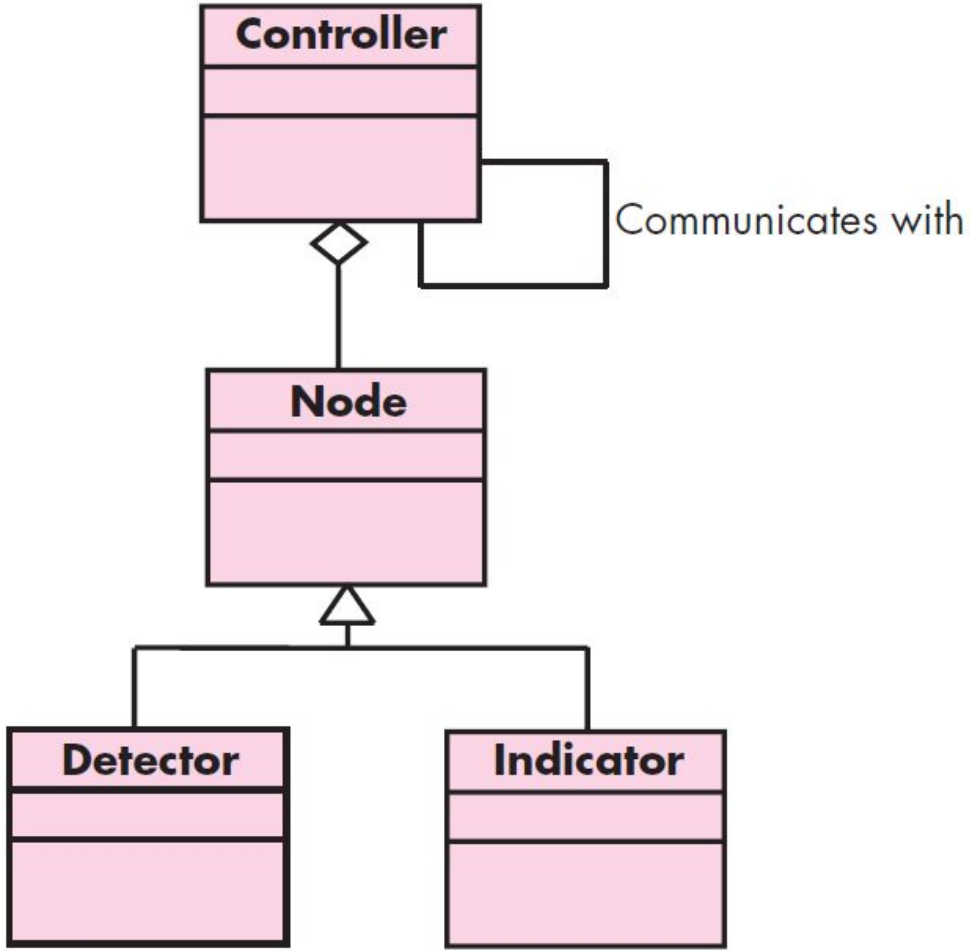


FIGURE 9.7

UML relationships for *SafeHome* security function archetypes

Source: Adapted from [Bos00].



Archetype Examples

From Concept to Subsystem

Example 1: Game Engine

- GameObject
- Renderer
- PhysicsEngine
- Asset

Example 2: E-commerce System

- Customer
- Product
- Order
- Inventory
- ShoppingCart

Note: These aren't just "classes"; they are the pillars upon which the entire architecture rests.