ARCHITECTURAL DESIGN

Prepared By: Natabar Khatri

New Summit College

Unit 6

INTRODUCTION TO ARCHITECTURAL DESIGN

- Focuses on organizing a software system and designing its overall structure.
- Defines the **high-level structure** of the software system, including its major components and how they interact
- Bridges **requirements engineering** and **detailed design**.
- Outputs an architectural model that represents the system's organization as interconnected components.
- Typical Diagram includes:
  - Layered Architecture
  - Microservices Map
  - Technology Stack Overview
  - Component Diagrams
  - Deployment Diagrams

THE ROLE OF ARCHITECTURAL DESIGN

- **Early Architecture is Critical:**
  Agile values adaptability but avoids incremental architecture changes (costly!).
- **Challenge:**
  Modifying architecture requires system-wide changes.

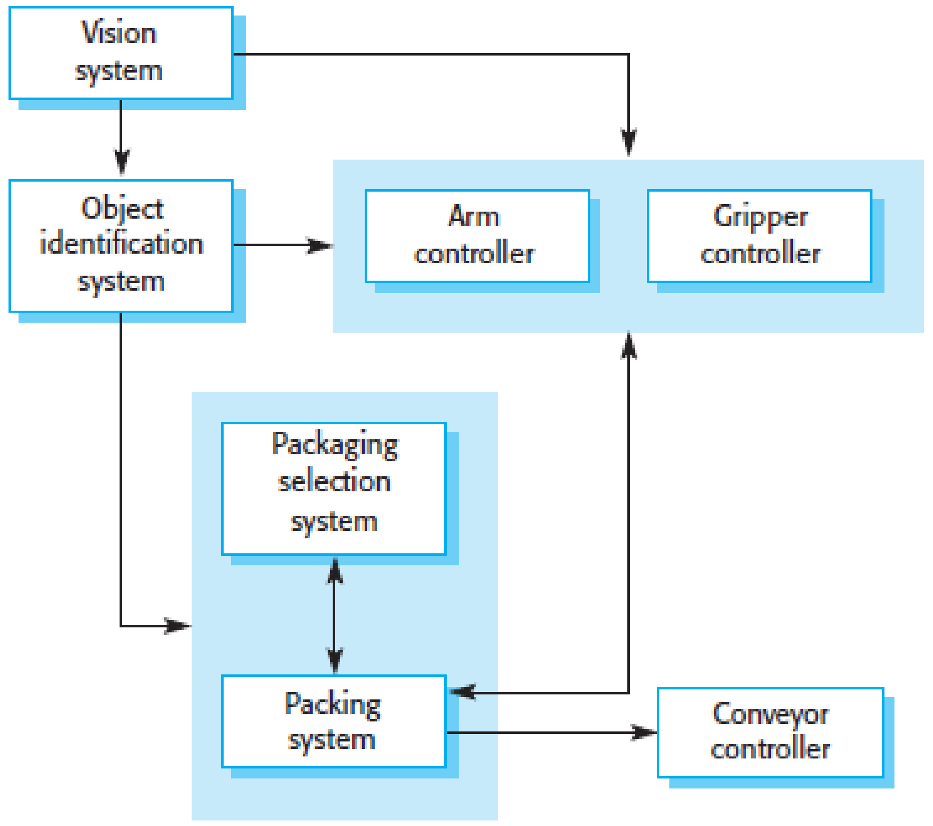

Fig 6.1: Packing robot control system architecture

RELATIONSHIP WITH REQUIREMENTS ENGINEERING

- Requirements often propose abstract architectures.
- However, there is significant overlap between requirements engineering and architectural design.
- Grouping system functions aids discussions with stakeholders about requirements

Levels of Architectural Abstraction

- Architecture operates at two levels:
  - **small (individual programs)**
  - Focuses on individual programs and their decomposition into components.
  - This level deals with how a single program is structured internally.
  - **large (complex enterprise systems).**
  - Concerns complex enterprise systems that may include multiple programs and components
  - often distributed across different computers and potentially managed by different organizations

IMPORTANCE AND BENEFITS

- **Importance of Software Architecture**
- Architecture impacts non-functional characteristics like performance, robustness, and maintainability.
- Non-functional requirements significantly influence architectural decisions.

- **Benefits of Architectural Design**
  - **Stakeholder communication**: The architecture serves as a high-level representation that facilitates discussion among various stakeholders.
  - **System analysis**: Early architectural design requires analysis that helps ensure the system can meet critical requirements.
  - **Large-scale reuse**: Architectural models enable reuse across systems with similar requirements, supporting approaches like product-line architectures.

REPRESENTING SYSTEM ARCHITECTURE

- System architectures are commonly modeled using informal block diagrams like in Packing robot control system architecture
  - Boxes represent components
  - Nested boxes indicate component decomposition
  - Arrows show data or control flow between components

Dual Purpose of Architectural Models

Architectural models serve two primary purposes:

- **Facilitating design discussions**: The high-level abstraction helps stakeholders understand and discuss the system without getting bogged down in details.
- **Documenting the architecture**: More detailed models provide a complete description of components, interfaces, and connections to support system understanding and evolution.

ARCHITECTURAL DESIGN DECISIONS

Architectural design is a creative decision-making process impacting system structure and development.

Key Architectural Decisions

- The architectural design process revolves around answering several fundamental questions:
- **Architecture Reuse**: Whether existing architectural templates can serve as a foundation for the new system. Many systems within the same application domain share similar architectures that reflect core domain concepts. Application product lines exemplify this approach, building variants from a common core architecture.
- **System Distribution**: How the system will be allocated across hardware resources. While embedded systems and personal applications might not require distributed architectures, most large systems must address distribution across multiple computers, significantly impacting performance and reliability.

ARCHITECTURAL DESIGN DECISIONS

- **Architectural Patterns**: The selection of appropriate architectural styles (such as client-server or layered architectures) that provide proven organizational structures for the system. These patterns capture successful architectural approaches used across various systems.
- **Structural Approach**: The fundamental strategy for organizing system components, including how they will be decomposed into subcomponents and how control will be managed between them.
- **Non-functional Requirements Alignment**: Determining the architectural organization best suited to deliver critical non-functional requirements like performance, security, or maintainability.
- **Documentation Strategy**: Deciding how to effectively capture and communicate the architectural design.

ARCHITECTURAL VIEWS

Architectural views are multiple, complementary perspectives of a software system's architecture, each designed to address the concerns of different stakeholders and to represent different aspects of the design.

Since a single diagram cannot capture all necessary information, these views are essential for both facilitating discussion and documenting the design for implementation.

- Figure 6.2 illustrates the core of the **"4+1" view model** by Philippe Kruchten, which is a seminal framework for understanding these perspectives. The four primary views revolve around the system architecture and are connected by use cases or scenarios (the "+1")

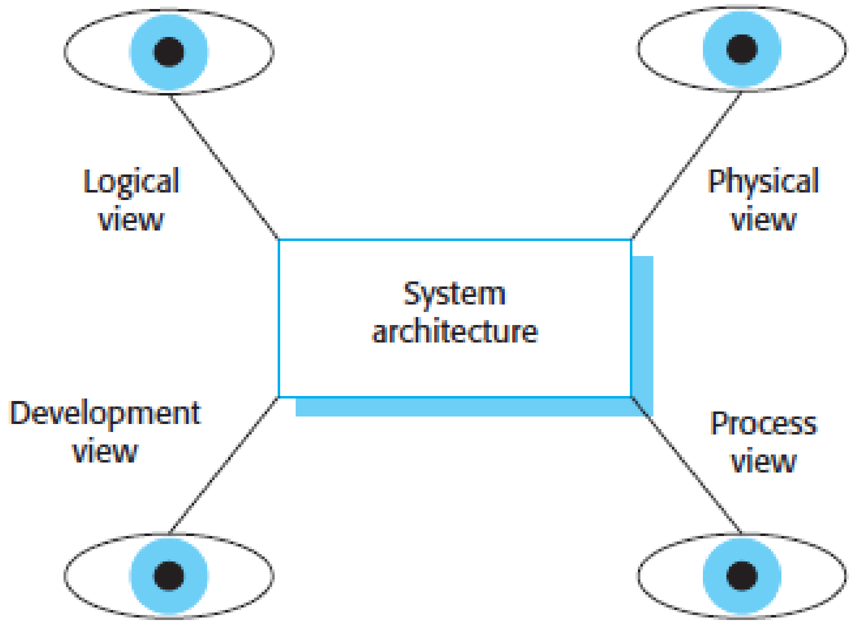

Fig 6.2: Architectural Views

ARCHITECTURAL VIEWS

1. The Four Fundamental Views (Krutchen's 4+1 Model)

- **Logical View:**
  - This view shows the key abstractions in the system as objects or object classes. Its primary purpose is to model the functional requirements of the system - what the system should do. Stakeholders like analysts and designers use this view to relate system requirements to the implementing entities.
  - Sequence, Class and Communication diagram
- **Development View:**
  - This view illustrates how the software is decomposed for development, typically into components, libraries, or modules managed by a single developer or team. It is crucial for software managers and programmers to organize code, manage dependencies, and assign work. This is often the view seen in a version control system.
  - Component and Package diagram

ARCHITECTURAL VIEWS

1. The Four Fundamental Views (Krutchen's 4+1 Model)

- **Process View:**
  - This view focuses on the runtime behavior of the system, showing how it is composed of interacting processes. It deals with dynamics such as communication, synchronization, and concurrency. This view is vital for making judgments about non-functional characteristics like performance, scalability, and availability.
  - Activity diagram
- **Physical View:**
  - Also known as the deployment view, this perspective shows the system's hardware and how software components (processes, objects, etc.) are distributed across that hardware. Systems engineers use this view to plan for deployment, capacity, and hardware provisioning.
  - Deployment Diagram
- 4+1 Scenario: Use Cases

ARCHITECTURAL PATTERNS AND STYLES

- Architectural patterns (or styles) provide templates for system organization that have proven successful in software design. It represents a system organization that has been successfully tried and tested in different systems and environments. Think of it as a reusable, high-level blueprint or template for solving a common problem in software architecture.
- The choice of architectural style directly influences how components are structured, decomposed, and controlled within the system.
- The concept was inspired by the work on object-oriented design patterns and was later developed under the name "architectural styles." An effective pattern description should not only explain the structure but also include crucial contextual information, such as:
- When it is and is not appropriate to use.
- Its strengths and weaknesses.
- Examples of its application.

ARCHITECTURAL PATTERNS AND STYLES

The Model-View-Controller (MVC) Pattern

- MVC separates presentation and interaction from the system data. It structures an application into three interconnected logical components:
  - **Model:** Manages the system data and the core business logic (or operations on that data).
  - **View:** Defines and manages how the data is presented to the user (the UI).
  - **Controller:** Manages user interaction (e.g., clicks, keystrokes) and translates them into commands for the Model and the View.

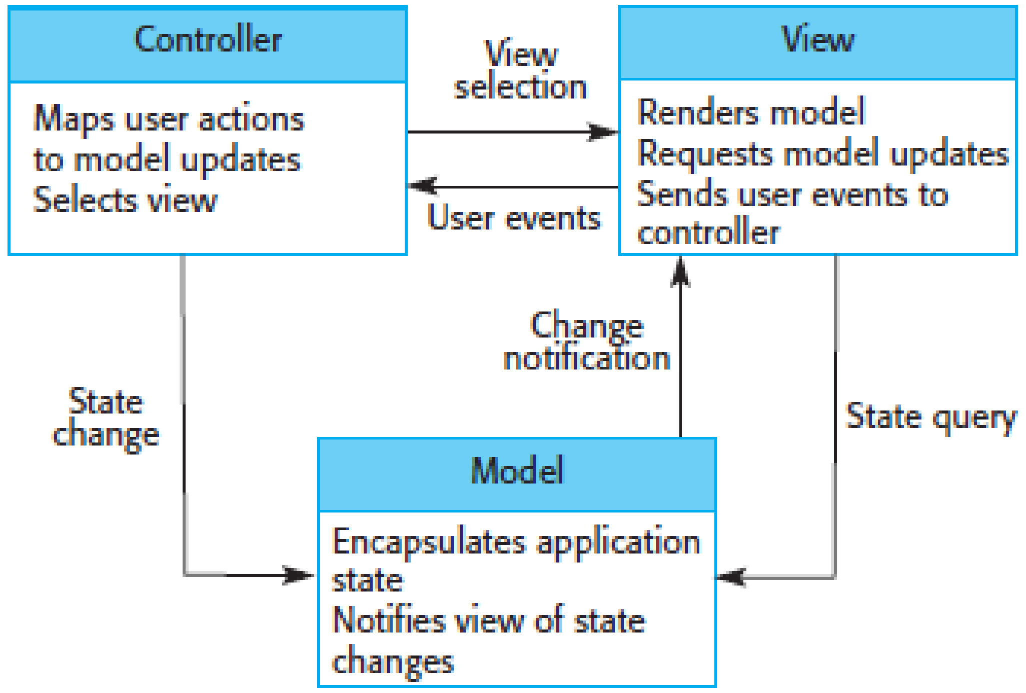

Fig 6.3: The organization of the Model View Controller

ARCHITECTURAL PATTERNS AND STYLES

Conceptual Views

Figure 6.3 illustrates the abstract, conceptual relationships between the components:

- The **Controller** maps user actions to updates in the **Model**.
- The **Model** encapsulates the application state and notifies the **View** when that state changes.
- The **View** renders the model for the user and sends user events to the controller.

ARCHITECTURAL PATTERNS AND STYLES

Practical Application:

Figure 6.4 shows a concrete runtime architecture for a web application using MVC, demonstrating how the abstract pattern is implemented:

- The **Controller** is represented by components handling HTTP requests, application logic, and validation.
- The **View** is responsible for dynamic page generation and form management.
- The **Model** contains the business logic and database interactions.

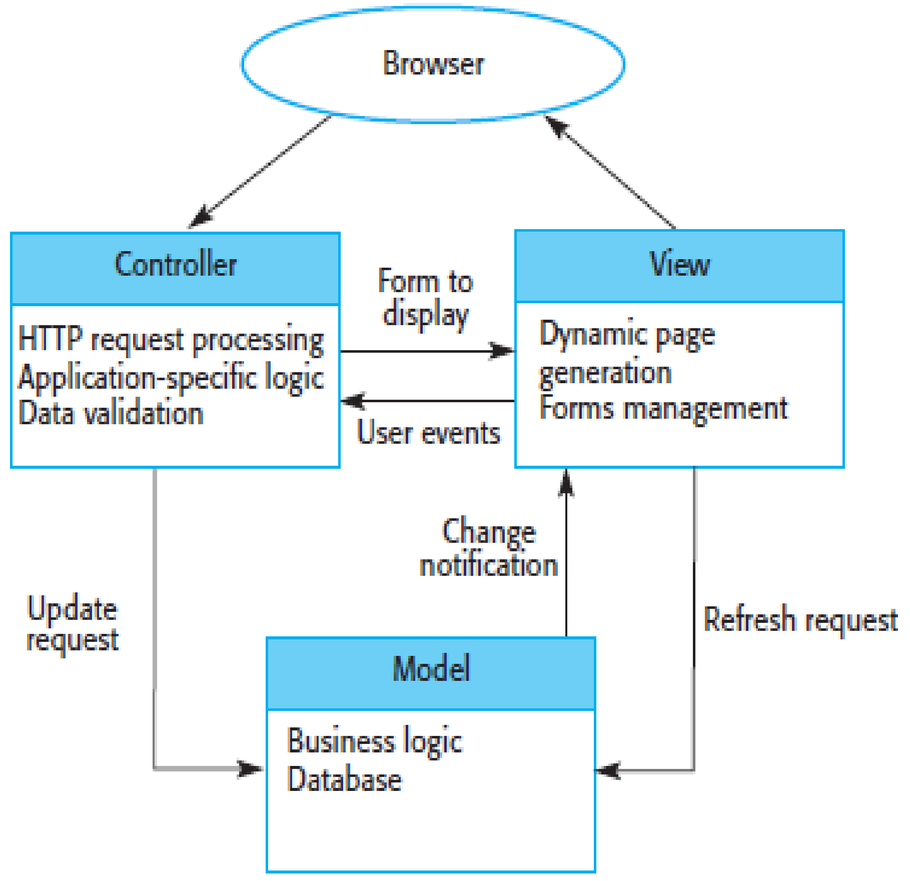

Fig 6.4: Web application architecture using MVC

ARCHITECTURAL PATTERNS AND STYLES

When to Use:

- MVC is ideal when there are multiple ways to view and interact with data, or when future requirements for interaction and presentation are unknown.

Advantages and Disadvantages:

- The key advantage is the **separation of concerns**, which allows the data (Model) to change independently of its presentation (View) and vice versa. It supports presenting the same data in different ways.
- The main disadvantage is that it **can introduce additional code and complexity**, which might be unnecessary for applications with simple data models and interactions.

ARCHITECTURAL PATTERNS AND STYLES

The Role and Importance of Patterns

- Architectural patterns are a powerful method for **presenting, sharing, and reusing knowledge** about successful system designs. They capture the wisdom and experience of the software engineering community, providing a common vocabulary for architects and designers to communicate complex ideas efficiently.
- By applying a known pattern, developers can leverage a proven solution, which reduces risk and accelerates the design process.
- While it's impossible to cover all patterns, they are widely used to capture and promote good architectural design principles across many different types of systems.

ALIGNING ARCHITECTURE WITH NON-FUNCTIONAL REQUIREMENTS

- The architecture must be carefully designed to support the system's non-functional requirements, which often involve trade-offs between competing priorities:
- **Performance**: Favors architectures with:
  - Localized critical operations
  - Fewer, larger components
  - Minimized network distribution
  - Potential for component replication
- **Security**: Requires:
  - Layered structures
  - Protected core components
  - Rigorous validation mechanisms
  - Clear security boundaries

ALIGNING ARCHITECTURE WITH NON-FUNCTIONAL REQUIREMENTS

- **Safety**: Benefits from:
  - Co-located safety-critical operations
  - Minimal safety-critical components
  - Built-in protection systems
  - Clear failure modes
- **Availability**: Needs:
  - Redundant components
  - Hot-swappable elements
  - Minimal single points of failure
- **Maintainability**: Thrives with:
  - Fine-grained, self-contained components
  - Clear separation of concerns
  - Minimal shared state
  - Well-defined interfaces

ARCHITECTURAL EVALUATION

- Evaluating architectural designs presents challenges since the ultimate test occurs during system operation. However, architects can perform preliminary evaluations by:
- Comparing designs against reference architectures
- Assessing alignment with generic architectural patterns
- Reviewing pattern characteristics against system requirements
- Considering documented experiences with similar architectures