Software Architectural Patterns - Simple and Understandable Documentation

This document summarizes the architectural patterns used to structure software systems effectively. The key categories include patterns that help manage complexity, distribute components, support interactive systems, and ensure adaptability.

---

1. From Mud to Structure

These patterns help you avoid a disorganized collection of components by breaking down a system into manageable subtasks. Common patterns include:

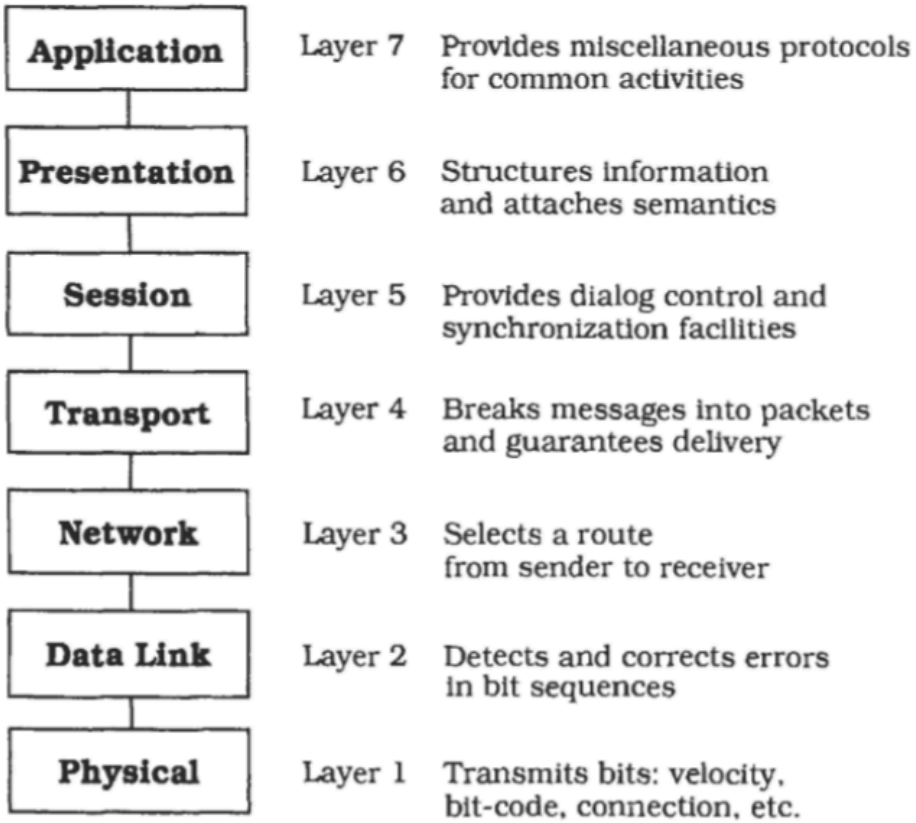
- **Layers Pattern**
- **Pipes and Filters Pattern**
- **Blackboard Pattern**

Layers Pattern

The **Layers Pattern** divides a system into layers, where each layer provides specific services and interacts only with the layer directly below it.

- **Example**: The OSI 7 Layer Model in networking is a classic example of the Layers Pattern.

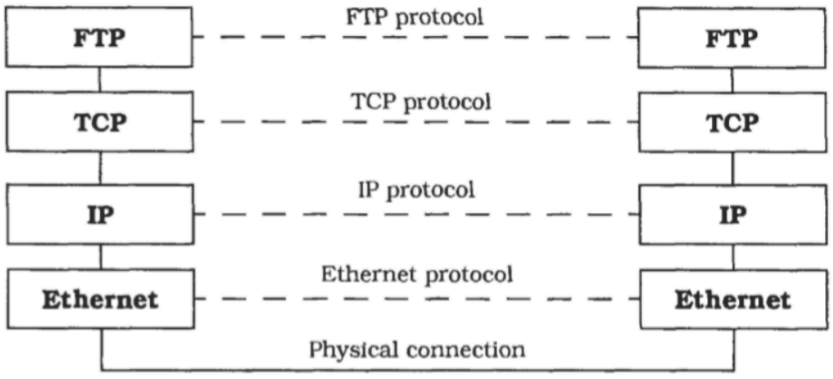
Diagram: OSI 7 Layer Model



Implementation Steps for the Layers Pattern:

1. Define criteria for grouping tasks into layers.
2. Determine the number of abstraction levels needed.
3. Assign tasks and name the layers.
4. Specify services for each layer.
5. Refine the layer structure and interfaces.
6. Ensure adjacent layers communicate properly.
7. Decouple layers for flexibility.

Diagram: TCP/IP Layer Model



Variants of the Layers Pattern

- **Relaxed Layered System**: In this variant, layers can use the services of any layer below them, not just the next lower one. This increases flexibility but reduces maintainability.
- **Layering Through Inheritance**: In object-oriented systems, lower layers can act as base classes for higher layers, which inherit and modify services. However, this can create tight coupling and problems if lower layers change.

---

2. Distributed Systems

Broker Pattern

The **Broker Pattern** provides a structure for distributed applications. It decouples components (clients and servers) and allows them to communicate via a broker, making the system flexible and scalable.

- **Context**: Used in distributed systems where components operate independently and need to interact remotely.
- **Problem**: Direct communication between components leads to dependencies and makes scalability difficult.
- **Solution**: A broker handles all communication between clients and servers, forwarding requests and responses.

---

3. Interactive Systems

These systems need patterns that support user interactions.

Model-View-Controller (MVC) Pattern

The **MVC Pattern** separates the application into three components:

- **Model**: Handles the data and business logic.
- **View**: Manages the user interface.
- **Controller**: Acts as an intermediary between the model and the view, handling user input and updating the view.
- **Example**: In a web application, the MVC pattern separates the presentation layer (HTML/CSS) from the business logic (JavaScript) and the data (backend database).

---

4. Adaptable Systems

These patterns support system adaptability and evolution over time.

Microkernel Pattern

The **Microkernel Pattern** separates the core system from optional services. The core system (or microkernel) manages essential tasks, while additional features are added through plug-ins or modules.

- **Example**: Operating systems like Windows NT use the microkernel architecture to allow for easy updates and system extensions without modifying the core.

Reflection Pattern

The **Reflection Pattern** enables a system to adapt at runtime by modifying its structure and behavior dynamically. This is useful for systems that need to change based on evolving requirements.

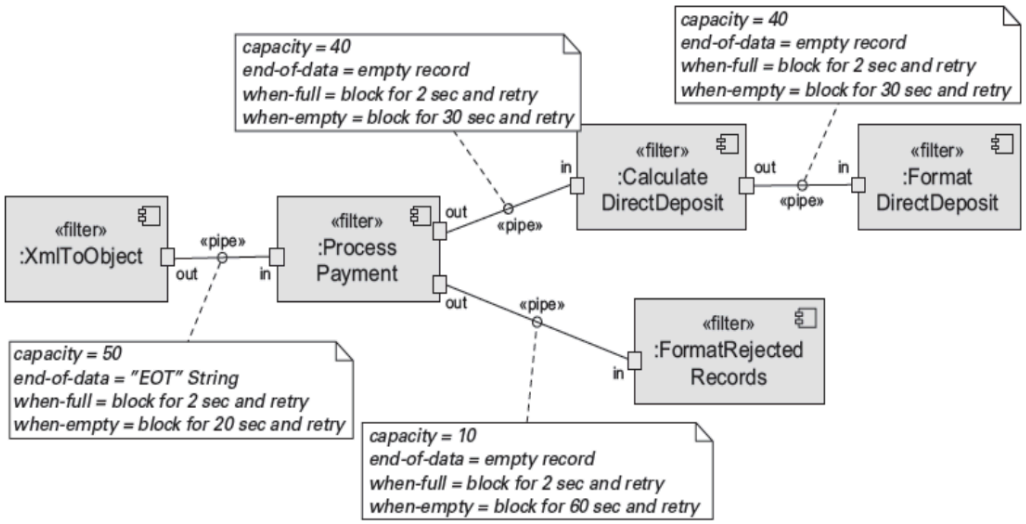
---

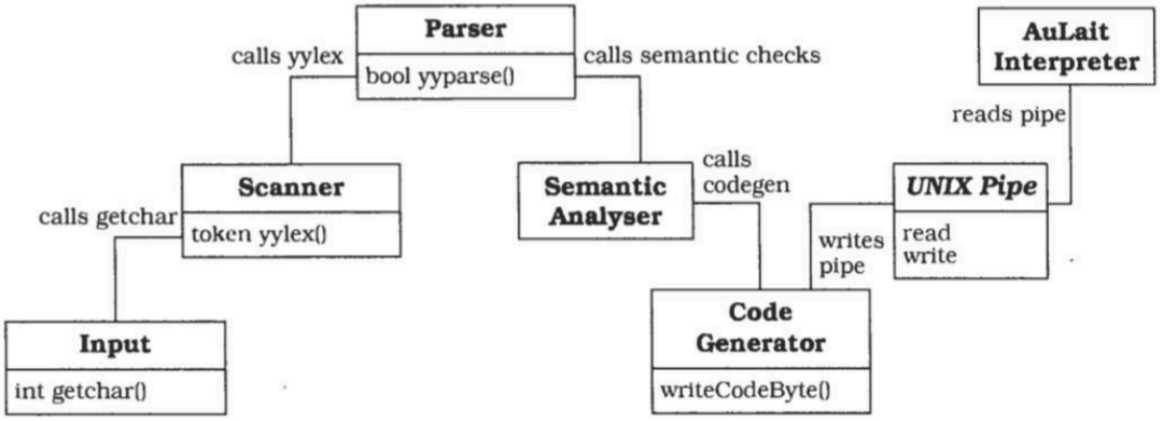
5. Pipes and Filters Pattern

The **Pipes and Filters Pattern** structures a system that processes a stream of data through a sequence of processing steps (filters). Each filter transforms the data and passes it to the next step through pipes.

- **Context**: Systems that process data streams, such as data pipelines.
- **Problem**: Systems need to be broken down into reusable, loosely coupled components.
- **Solution**: The system is divided into independent filters, each processing the data and passing it along to the next filter via pipes.

Diagram: Pipe and Filter Example





Implementation Steps for the Pipes and Filters Pattern:

1. Divide the system's task into processing stages.
2. Define the data format for each pipe.
3. Implement the connections between pipes and filters.
4. Design the filters and error-handling mechanisms.
5. Set up the pipeline and allow flexibility for reordering filters.

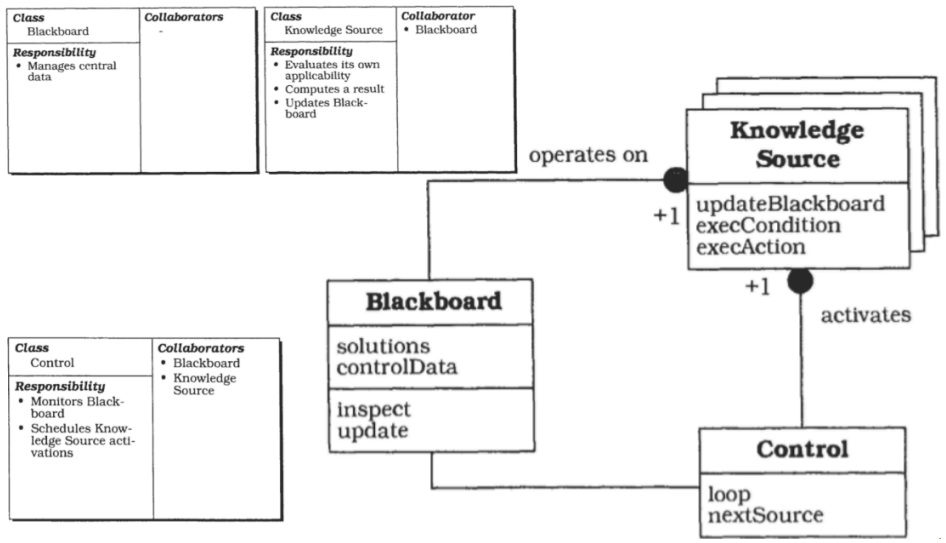
---

6. Blackboard Pattern

The **Blackboard Pattern** is used for complex problems where no deterministic solution exists. Multiple independent subsystems contribute partial solutions to build a complete solution.

- **Context**: Used in systems with no clear solution, such as speech or image recognition.
- **Problem**: The problem requires different algorithms, representations, and solutions to be combined dynamically.
- **Solution**: A central "blackboard" stores partial solutions contributed by independent components, which work together to solve the overall problem.

Diagram: Blackboard Architecture



---

7. Benefits of Architectural Patterns

- **Reuse**: Patterns like Layers and Pipes and Filters promote reusability of components.
- **Scalability**: Broker and Microkernel patterns allow systems to scale by decoupling components and extending functionality easily.
- **Flexibility**: Patterns such as MVC and Blackboard provide flexibility by separating concerns and allowing dynamic problem-solving.

---

8. Liabilities of Architectural Patterns

- **Complexity**: Some patterns, like Broker, add complexity to the system, especially in testing and debugging.
- **Efficiency**: Patterns like Pipes and Filters can introduce overhead, especially when filters are chained together in a sequence.

- **Tight Coupling**: Inheritance-based layering can create tight coupling, making it harder to modify components independently.