Documentation for Layered, Broker, and Pipe-and-Filter Patterns

---

Introduction to Patterns

Architectural patterns provide a structured approach to solving recurring design problems in software systems. Each pattern addresses a specific problem within a defined context and offers a proven solution.

Pattern Overview

1. **Layered Pattern**
2. **Broker Pattern**
3. **Pipe-and-Filter Pattern**

Layered Pattern

Context

In complex software systems, the development and evolution of system components need to be handled independently. The system must be segmented to allow modules to be developed separately, supporting modularity, flexibility, and reuse.

Problem

How to separate software into independent units that can be developed, tested, and maintained independently, while maintaining clear communication between them?

Solution

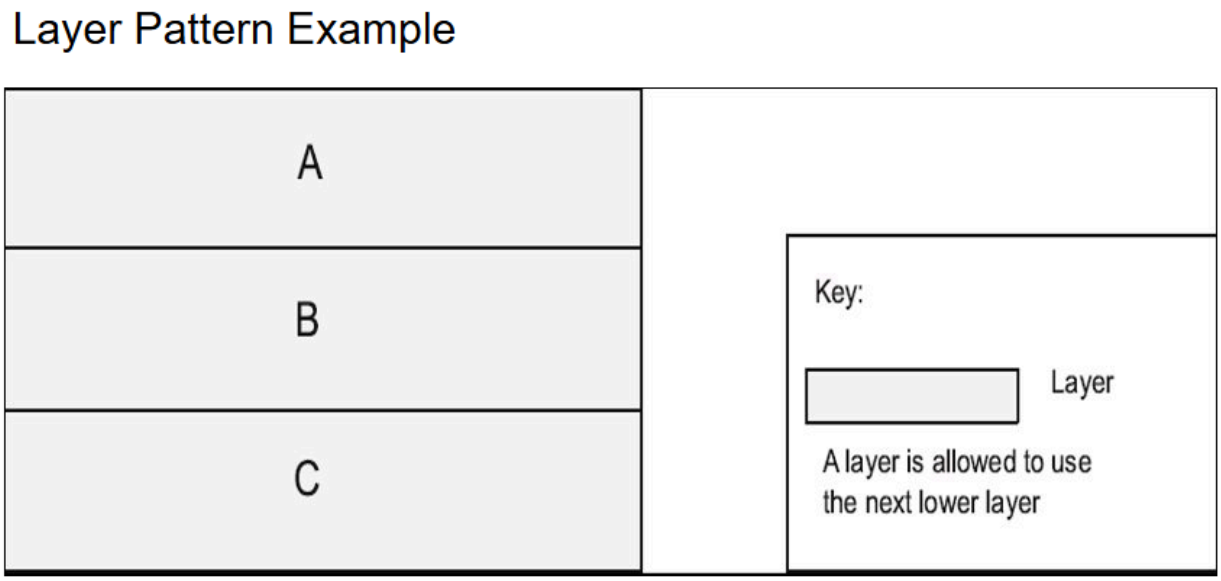
The Layered Pattern divides the software into distinct layers, where each layer performs a specific set of tasks and interacts only with the layer directly below or above it. Each layer offers a cohesive set of services exposed through a public interface, ensuring a unidirectional flow of communication.

Example

- **Use Case**: In a web application, the architecture is divided into three main layers:
    - **Presentation Layer**: Manages user interfaces.
    - **Business Logic Layer**: Handles business rules and operations.

- **Data Access Layer**: Manages data storage and retrieval from the database.

Diagram: Layer Pattern Example



Elements

- **Layers**: Grouping of modules, each offering a set of related services.
- **Relations**: Defines allowed usage between layers, typically in a top-down manner.
- **Constraints**:
  - Each module belongs to one layer.
  - No circular dependencies between layers.

Strengths

- Clear separation of concerns.
- Improved maintainability and scalability.
- Easier to test individual layers.

Weaknesses

- Increases initial complexity and overhead.
- Potential performance penalty due to multiple layers.

Broker Pattern

Context

Distributed systems often consist of services spread across multiple servers. These services must communicate with each other seamlessly, without users needing to know the location or nature of service providers.

Problem

How to structure distributed software so that service users do not need to know the specific details or locations of the service providers, allowing for dynamic service bindings?

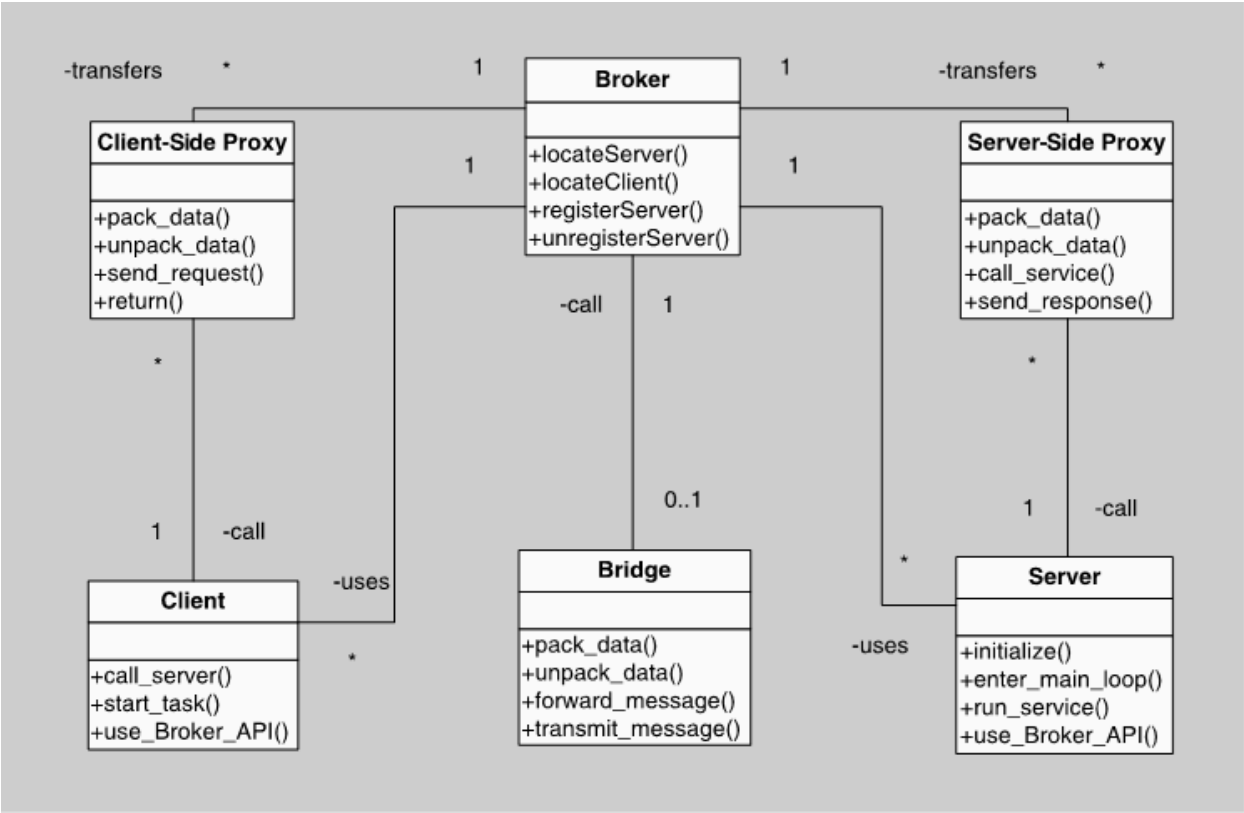
Solution

The Broker Pattern introduces an intermediary called a **broker** that manages communication between service clients and service providers. Clients communicate with the broker, which then forwards requests to the appropriate servers and returns the results to the client.

Example

- **Use Case**: In an e-commerce platform, users interact with a broker to access various services like order processing, payment, and inventory management. The broker handles requests and responses, making service details transparent to users.

Diagram: Broker Pattern



Elements

- **Client**: Requests services.
- **Server**: Provides services.
- **Broker**: Mediates communication between clients and servers.
- **Client-Side Proxy**: Handles client communication with the broker, including message handling.
- **Server-Side Proxy**: Manages server communication with the broker.

Relations

- Clients and servers connect to brokers, possibly through proxies.

Constraints

- Communication must pass through the broker.
- The broker must handle service discovery, forwarding, and result handling.

Strengths

- Decouples clients and servers, allowing for easier modifications.
- Supports dynamic binding and scalability.

Weaknesses

- Adds latency due to the intermediary layer.
- Single point of failure if the broker is not designed for redundancy.
- Can be a target for security attacks.

Pipe-and-Filter Pattern

Context

Many software systems require processing streams of data through successive transformations. These transformations are often repetitive and need to be implemented as reusable components.

Problem

How to create a system with loosely coupled components that can perform data transformations in sequence, allowing for parallel execution and flexibility?

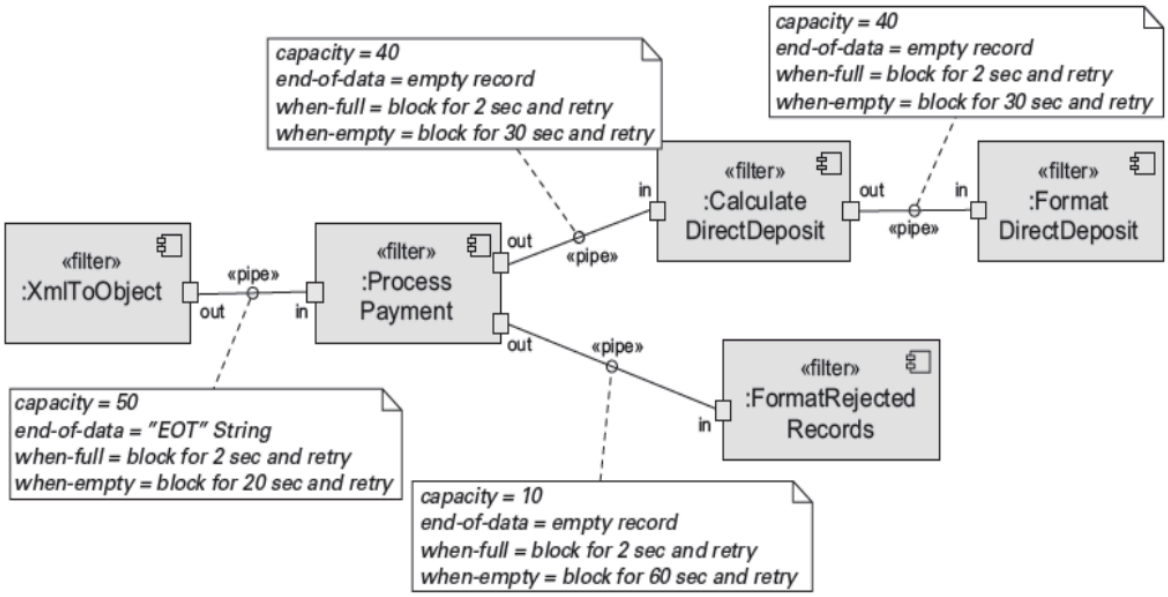
Solution

The Pipe-and-Filter Pattern divides the system into components called **filters**, which perform data processing. Data flows through **pipes** that connect the filters, allowing for independent, parallel, and reusable transformations.

Example

- **Use Case**: A data processing pipeline that reads raw sensor data (input), cleans it, performs analysis, and outputs the results in a report.

Diagram: Pipe-and-Filter Pattern



Elements

- **Filter**: A processing component that transforms data.
- **Pipe**: A connector that transmits data between filters.

Relations

- Pipes connect filter outputs to filter inputs, enabling sequential data flow.

Constraints

- Filters must have well-defined inputs and outputs.
- Filters should not maintain state between invocations.

Strengths

- High reusability of filters.
- Supports parallel processing and dynamic reconfiguration.

Weaknesses

- May introduce performance bottlenecks if filters are not optimized.

- Error handling can be complex in the pipeline.

Use Cases and Scenarios

Layered Pattern Use Case

- **Scenario**: In a banking system, the layered architecture separates the presentation layer (user interface), business layer (loan processing), and data layer (customer data). This separation allows for independent updates and testing of each layer.

Broker Pattern Use Case

- **Scenario**: In a cloud-based service, the broker manages requests from users to different cloud providers for storage, processing, or database services. The broker ensures service continuity even if providers change.

Pipe-and-Filter Pattern Use Case

- **Scenario**: In a video streaming service, data is processed in filters for decoding, buffering, and rendering, connected by pipes. This setup allows for parallel processing of video data for better performance.

Conclusion

The **Layered Pattern** is suitable for modular systems, the **Broker Pattern** is essential for distributed systems, and the **Pipe-and-Filter Pattern** is effective for sequential data processing. Each pattern has its strengths and weaknesses but provides a structured approach to building scalable, maintainable, and adaptable software architectures.