Documentation for Patterns and Architectural Solutions

From Mud to Structure

Overview

Patterns in this category help to prevent a 'sea' of unstructured components by breaking down a system task into smaller, cooperating subtasks. These patterns are used to manage complexity and establish clearer system structure.

Patterns Included

1. **Layers Pattern**
    - Divides software into multiple layers, each providing a specific set of services.
2. **Pipes and Filters Pattern**
    - Processes data streams step-by-step through sequential filters.
3. **Blackboard Pattern**
    - Uses a central data repository (blackboard) to coordinate independent components solving a problem.

Distributed Systems

Overview

This category focuses on systems distributed across multiple servers, emphasizing communication and coordination between services.

Patterns Included

1. **Broker Pattern**
    - Provides an infrastructure for distributed applications, enabling seamless communication between clients and servers.
2. **Microkernel**
    - Primarily focuses on adaptability, supporting distribution as a secondary concern.
3. **Pipes and Filters**
    - Supports data transformation and distribution as a secondary aspect.

Interactive Systems

Overview

Interactive systems require patterns that enable structured human-computer interactions.

Patterns Included

1. **Model-View-Controller (MVC) Pattern**
    - Separates the application into three parts: model (data), view (UI), and controller (logic).
2. **Presentation-Abstraction-Control (PAC) Pattern**
    - Organizes the system as a hierarchy of agents, each with its own presentation, abstraction, and control components.

Adaptable Systems

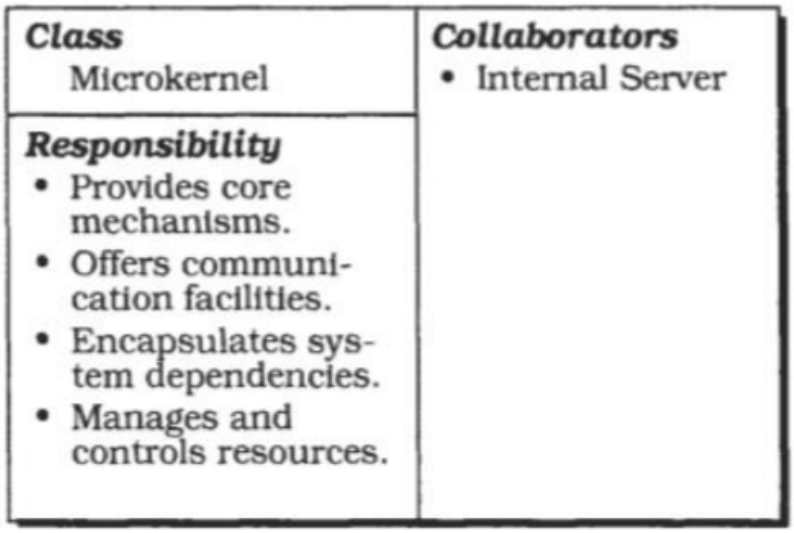
Overview

Adaptable systems focus on supporting extension and evolution to accommodate new functionality, standards, and changing requirements.

Patterns Included

1. **Reflection Pattern**
    - Allows software to change its behavior dynamically.
2. **Microkernel Pattern**
    - Separates a minimal core from additional functionality, supporting modifications and extensions.

Diagram: Microkernel Component



Adaptable Systems Context

Adaptable systems must support new versions of operating systems, user interfaces, and third-party components. These systems need to be flexible and prepared for changing requirements, even during later stages of development.

Design for Change

- Systems should be designed with change in mind, allowing modifications without affecting core functionality. This approach helps maintain system stability while accommodating changes efficiently.

Microkernel Pattern

Context

The Microkernel pattern applies to systems that must adapt to evolving requirements. It is useful for software that supports similar interfaces but different extensions.

Problem

Building software for varying standards and technologies can be challenging. Examples include operating systems and graphical user interfaces.

Solution

The Microkernel separates minimal core functionality from extensions and customizations. It coordinates communication among components, maintains system resources, and allows extensions through internal and external servers.

Structure

The Microkernel pattern includes:

- **Internal Servers**: Extend core functionality.
- **External Servers**: Add specialized services.
- **Adapters**: Connect clients to external servers.
- **Clients**: Request services via adapters.
- **Microkernel**: Central component managing resources and communication.

Diagram:

Internal Server Component

Class Internal Server	Collaborators • Microkernel
Responsibility • Implements additional services. • Encapsulates some system specifics.	

External Server Component

Class External Server	Collaborators • Microkernel
Responsibility • Provides programming interfaces for its clients.	

Client & Adapter

Class Client	Collaborators • Adapter
Responsibility • Represents an application.	

Class Adapter	Collaborators • External Server • Microkernel
Responsibility • Hides system dependencies such as communication facilities from the client. • Invokes methods of external servers on behalf of clients.	

Reflection Pattern

Context

The Reflection pattern allows systems to change their structure and behavior dynamically. It supports evolving requirements by enabling modifications without altering core components.

Problem

Software systems must be flexible to accommodate changes in technology and user requirements. Designing for all possible changes upfront is difficult, but a reflection-based architecture enables changes as needed.

Solution

The Reflection pattern splits the system into:

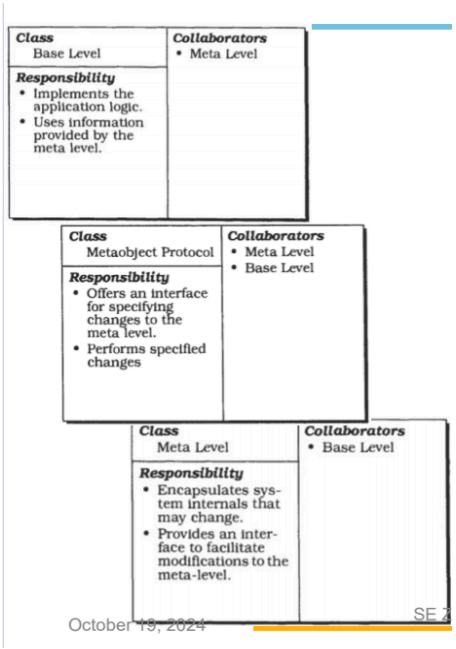
- **Meta Level**: Manages system self-awareness and change.
- **Base Level**: Implements core application logic, relying on meta-level information.

Metaobject Protocol (MOP)

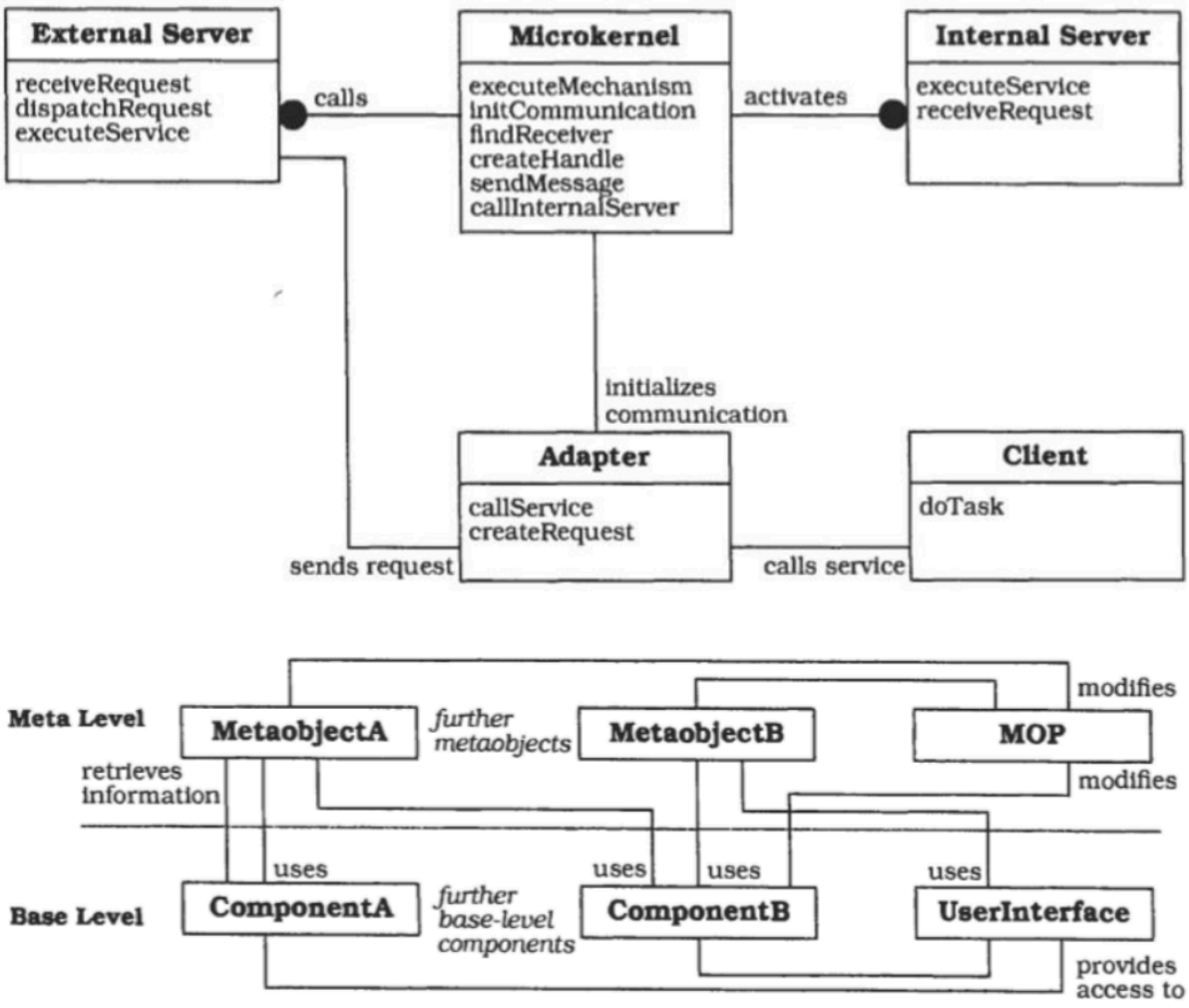
Defines how metaobjects can be manipulated, allowing changes to affect base-level behavior.

Diagram:

Structure



OMT Diagram



Use Cases, Scenarios, and Examples

From Mud to Structure

- **Use Case**: In a document processing system, the Layers pattern can be used to separate text formatting, file handling, and user interface into different layers.

Distributed Systems

- **Scenario**: In an e-commerce platform, the Broker pattern helps manage communication between customer services, payment processing, and inventory management, ensuring seamless transactions.

Interactive Systems

- **Example**: In a customer relationship management (CRM) application, the MVC pattern allows sales representatives to view customer data in different formats (list, card view) while maintaining synchronized updates.

Adaptable Systems

- **Scenario**: In an IoT platform, the Microkernel pattern supports adding new device drivers (internal servers) without changing the core system.

Reflection Pattern

- **Example**: In a data analytics system, the Reflection pattern allows modifying the data aggregation method dynamically based on new algorithm requirements.

Conclusion

The patterns covered—Layers, Pipes and Filters, Broker, MVC, Microkernel, and Reflection—are essential for building structured, distributed, interactive, and adaptable software systems. Each pattern has distinct roles, strengths, and challenges but contributes to creating flexible, scalable, and maintainable software architectures.