Documentation for Distributed Systems and Interactive Systems Architecture Patterns

---

1. From Mud to Structure

In complex software systems, there can be an overwhelming number of components and interactions. Patterns help structure these systems by breaking down the overall tasks into smaller, cooperating subtasks. Common patterns that help achieve this are:

- **Layers Pattern**: Divides the system into layers where each layer has a specific role and only interacts with adjacent layers.
- **Pipes and Filters Pattern**: The system is divided into processing steps (filters), connected by pipes that pass data between them.
- **Blackboard Pattern**: Multiple subsystems (agents) share a common knowledge base (the blackboard), collaborating to find a solution to complex problems.

---

2. Distributed Systems

Distributed systems are composed of components that are distributed across different locations and connected via a network. In this category, the **Broker Pattern** is the key pattern used to manage communication between independent, distributed components.

Broker Pattern

- **Context**: Used in distributed systems where independent components need to interact and exchange data.
- **Problem**: If distributed components directly manage communication, it creates dependencies that reduce flexibility and scalability.

Solution: Introduce a **Broker** to act as an intermediary, forwarding client requests to the appropriate server and returning results. The broker decouples clients from servers, allowing each to function independently.

Components of the Broker Pattern

1. **Clients**: Applications that request services from the broker.
2. **Servers**: Applications that register their services with the broker and provide functionality to clients.

3. **Brokers**: Act as intermediaries between clients and servers, managing communication and service coordination.
4. **Proxies**: Handle message exchange, helping to hide details of the communication between the broker, clients, and servers.
5. **Bridges**: Used to enable communication between brokers located on different networks.

Use Case Example: A web browser (client) requests a web page from a web server (server) via the broker. The broker locates the correct server and forwards the request without requiring the client to know the server's physical location.

---

3. Interactive Systems

Interactive systems feature a high level of user interaction, typically using graphical user interfaces (GUIs). The goal is to create usable systems where the functional core is separated from the user interface, allowing for flexibility in modifying the interface without affecting core functionality.

Model-View-Controller (MVC) Pattern

The **MVC Pattern** divides an interactive application into three main components:

1. **Model**: Contains the core functionality and data of the application.
2. **View**: Presents information to the user.
3. **Controller**: Handles user input and updates the model or view accordingly.

Components of MVC

1. **Model**:
    - Encapsulates core data and functionality.
    - The model is independent of the user interface, making it reusable and easier to maintain.
    - Notifies views of any changes to data via a change-propagation mechanism.
2. **Use Case Example**: In a banking application, the model manages account balances and transactions.
   Diagram:

Class Model	Collaborators • View • Controller
Responsibility • Provides functional core of the application. • Registers dependent views and controllers. • Notifies dependent components about data changes.	

3. **View**:
    - Displays information from the model.
    - Multiple views can show the same data differently (e.g., a bar chart or pie chart).
4. **Use Case Example**: A dashboard may display a financial report in multiple views such as graphs or tables.
   Diagram: View Structure

Class View	Collaborators • Controller • Model
Responsibility • Creates and initializes its associated controller. • Displays information to the user. • Implements the update procedure. • Retrieves data from the model.	

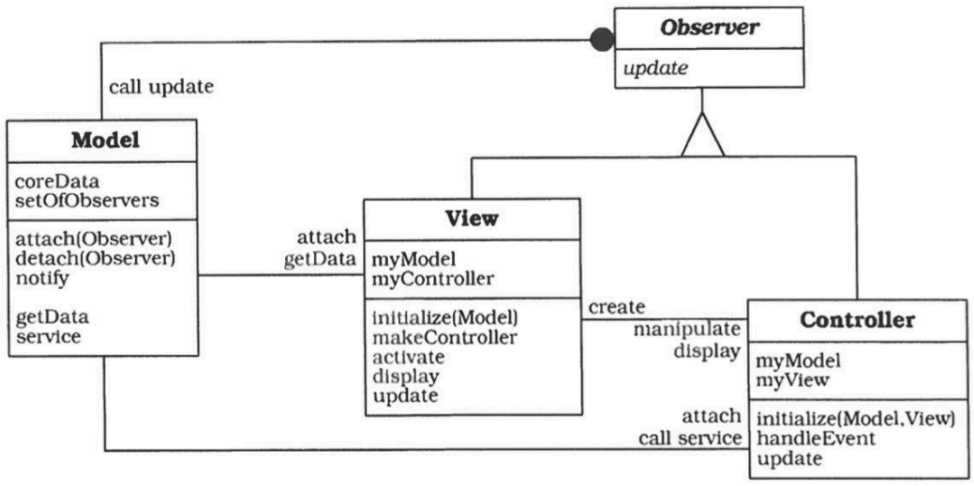
5. **Controller**:
    - Manages user input like mouse clicks or keyboard inputs.
    - Updates the model or view based on user interactions.
6. **Use Case Example**: In a photo editing app, the controller listens for clicks on toolbar icons to apply filters to an image.
   Diagram: Controller Structure

Class Controller	**Collaborators** • View • Model
Responsibility • Accepts user input as events. • Translates events to service requests for the model or display requests for the view. • Implements the update procedure, if required.	

Change-Propagation Mechanism in MVC

The model component notifies all registered views whenever data changes. Each view then retrieves the updated data and refreshes the displayed information. This mechanism ensures consistency between the view and the model across multiple views.

Structure: C++



---

4. Presentation-Abstraction-Control (PAC) Pattern

The **PAC Pattern** is another architectural model used for interactive systems, particularly suited for systems that can be divided into multiple cooperating agents. Each agent is responsible for a specific aspect of the system's functionality and is composed of three components: **presentation**, **abstraction**, and **control**.

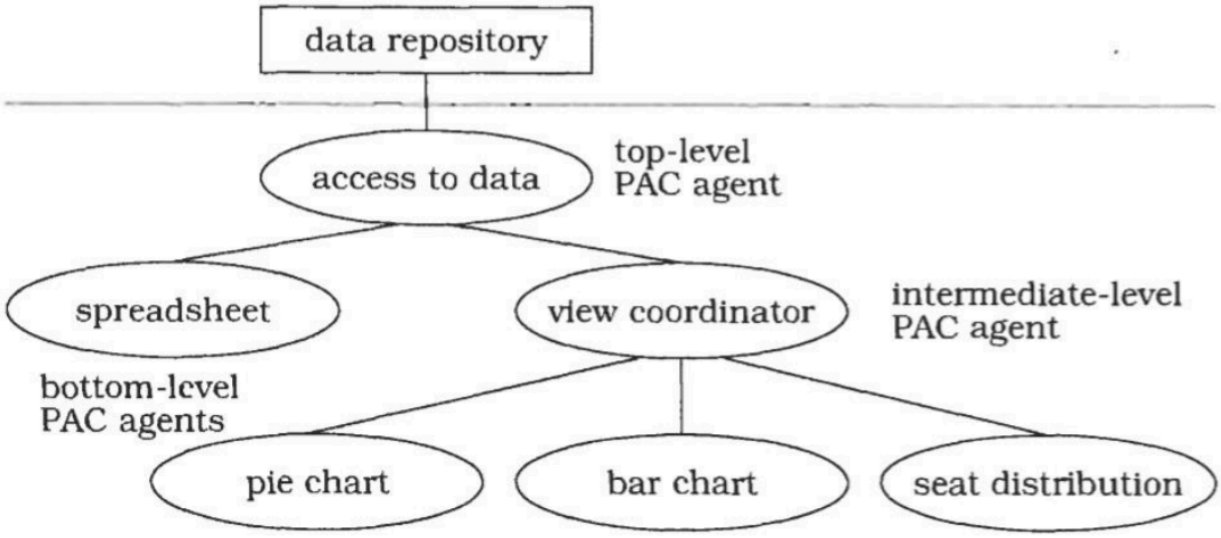
- **Context**: Used in complex interactive systems where different tasks can be broken down into smaller, independent agents.

Solution: Organize the system into a hierarchy of PAC agents. Each agent manages a specific part of the application, with higher-level agents overseeing broader functionality and lower-level agents managing specific, smaller tasks.

Components of PAC Agents

1. **Top-Level Agents**:
    - Represent the core functionality of the system.
    - Provide system-wide services such as the data model or global settings.
2. **Use Case Example**: In a traffic control system, the top-level agent manages the overall flow of traffic data.
3. **Bottom-Level Agents**:
    - Handle specific, self-contained tasks or user interface elements.
    - Present user interaction features, like buttons, charts, or widgets.
4. **Use Case Example**: A bottom-level agent in a CAD system might control the layout of a specific drawing component.
5. **Intermediate-Level Agents**:
    - Coordinate or combine the actions of multiple lower-level agents.
    - Serve as mediators between top-level and bottom-level agents.
6. **Use Case Example**: An intermediate-level agent might manage the coordination of various data views in a dashboard, ensuring consistency across the system.
   Diagram: Top-Level Agent Structure, Bottom-Level Agent Structure, Intermediate-Level Agent Structure

Typical Hierarchy of Agents



Class Top-level Agent	***Collaborators***
Responsibility • Provides the functional core of the system. • Controls the PAC hierarchy.	• Intermediate-level Agent • Bottom-level Agent

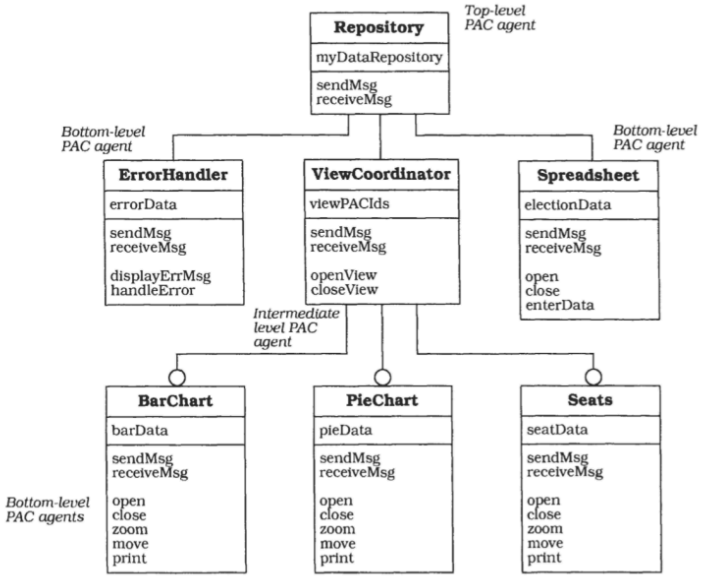
Class Interm. -level Agent	***Collaborators***
Responsibility • Coordinates lower-level PAC agents. • Composes lower-level PAC agents to a single unit of higher abstraction.	• Top-level Agent • Intermediate-level Agent • Bottom-level Agent

Class Bottom-level Agent	***Collaborators***
Responsibility • Provides a specific view of the software or a system service, including its associated human-computer interaction.	• Top-level Agent • Intermediate-level Agent

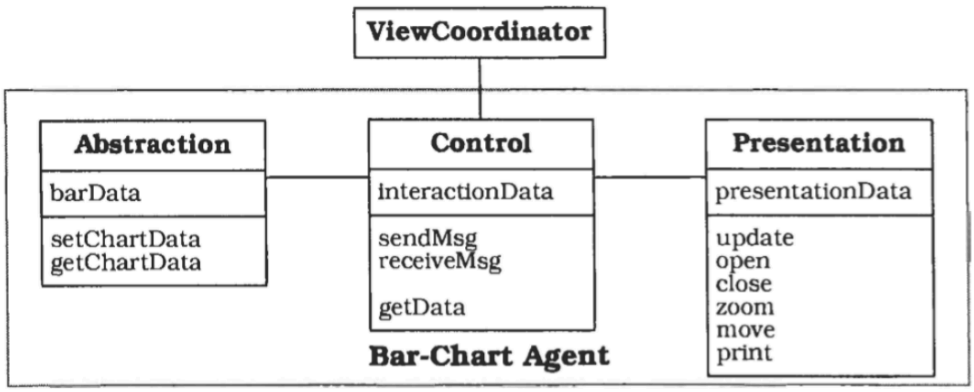
Implementation Steps for MVC and PAC

1. **Separate User Interaction from Core Functionality**: Ensure the user interface can be changed independently of the core functionality.
2. **Implement a Change-Propagation Mechanism (MVC)**: Establish a system where the model notifies all dependent views of changes.
3. **Design and Implement Views and Controllers (MVC)**: Create multiple views and controllers for different ways to interact with the model.
4. **Set Up PAC Agent Hierarchy (PAC)**: Organize PAC agents in a tree structure, with top-level agents overseeing core functionality and lower-level agents managing specific tasks.

Typical PAC Object Model



Internal Structure of a PAC Agent



---

5. Use Cases and Scenarios

Broker Pattern Use Case:

A company uses a cloud service broker to manage their multi-cloud environment. The broker abstracts the cloud services and forwards requests between the internal applications (clients) and various cloud service providers (servers), without the clients needing to know which provider hosts the requested service.

MVC Pattern Use Case:

A task management app uses the MVC pattern. The **model** manages tasks and deadlines, the **view** shows a list of tasks, and the **controller** handles user actions like adding or removing tasks.

PAC Pattern Use Case:

In a network traffic monitoring system, the **top-level agent** controls data flow, the **intermediate-level agents** manage specific networks, and **bottom-level agents** display traffic data for individual users.

Conclusion

The **Broker Pattern** supports distributed systems by decoupling clients and servers, while **MVC** and **PAC** patterns help structure interactive systems. These patterns make systems more flexible, scalable, and maintainable by separating core functionality from the user interface or communication mechanisms.