Patterns in Software Engineering

Lecturer: Raman Ramsin

Lecture 8

GoV Patterns – Architectual

Part 2

Architectural Patterns: Categories

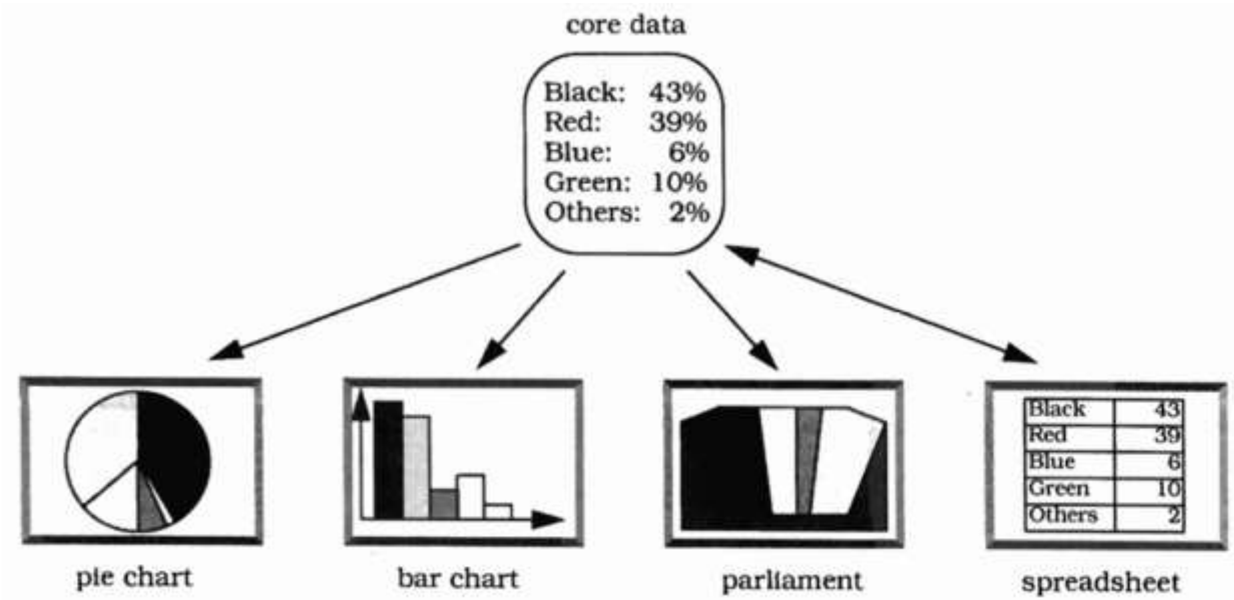
- **From Mud to Structure**
  - *Layers*, *Pipes and Filters*, and *Blackboard*
- **Distributed Systems**
  - *Broker*, also *Microkernel* and *Pipes and Filters*
- **Interactive Systems**
  - Support the structuring of systems that feature human-computer interaction.
  - *Model-View-Controller* and *Presentation-Abstraction-Control*
- **Adaptable Systems**
  - Support extension of applications and their adaptation to evolving technology and changing functional requirements.
  - *Reflection* and *Microkernel*

Architectural: Interactive Systems

- **Model-View-Controller (MVC):** Divides an interactive application into three components: *Model, Views*, and *Controllers.*
  - The *model* contains the core functionality and data.
  - *Views* and *controllers* together comprise the *user interface*.
- **Presentation-Abstraction-Control (PAC):** Defines a structure for interactive software systems in the form of a hierarchy of cooperating agents.
  - Every agent consists of three components: *Presentation*, *Abstraction*, and *Control*.
  - This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

Interactive Systems: Model-View-Controller

- Divides an interactive application into three components.
  - The *Model* contains the core functionality and data.
  - *Views* display information to the user.
  - *Controllers* handle user input.
  - A change-propagation mechanism ensures consistency between the user interface (views and controllers) and the model.



Interactive Systems: Model-View-Controller

- **Context -** Interactive applications with a flexible human-computer interface.

- **Problem -** Forces are as follows:
  - The same information is presented differently in different windows, for example, in a bar or pie chart.
  - The display and behavior of the application must reflect data manipulations immediately.
  - Changes to the user interface should be easy, and even possible at run-time.
  - Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.

Model-View-Controller: Structure – Model

- Contains the data and functional core of the application.
- Provides procedures that perform application-specific processing; controllers call these procedures on behalf of the user.
- Provides functions to access its data; view components use these functions to acquire the data to be displayed.
- Implements the change-propagation mechanism :
  - Maintains a registry of dependent components (all views and selected controllers).
  - Changes to the state of the model trigger the change-propagation mechanism.

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

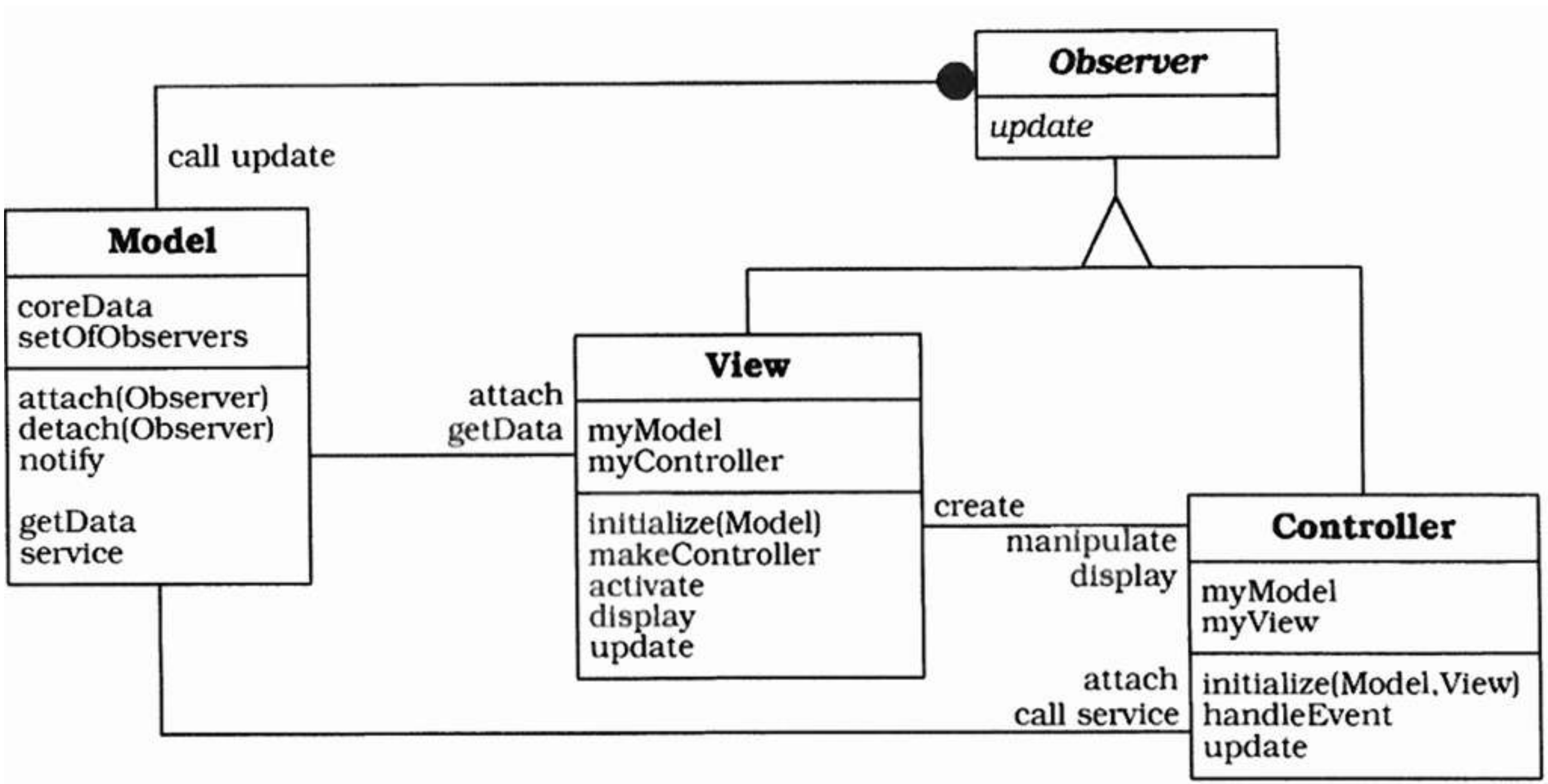
Model-View-Controller: Structure – Views and Controllers

- *View* components present information to the user.
  - Each view defines an update procedure that is activated by the change propagation mechanism and retrieves data from the model.
  - Each view creates a suitable controller.
  - Views often offer functionality that allows controllers to manipulate the display.
- *Controller* components accept user input as events.
  - If the behavior of a controller depends on the state of the model, the controller registers itself with the model and implements an update operation.

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

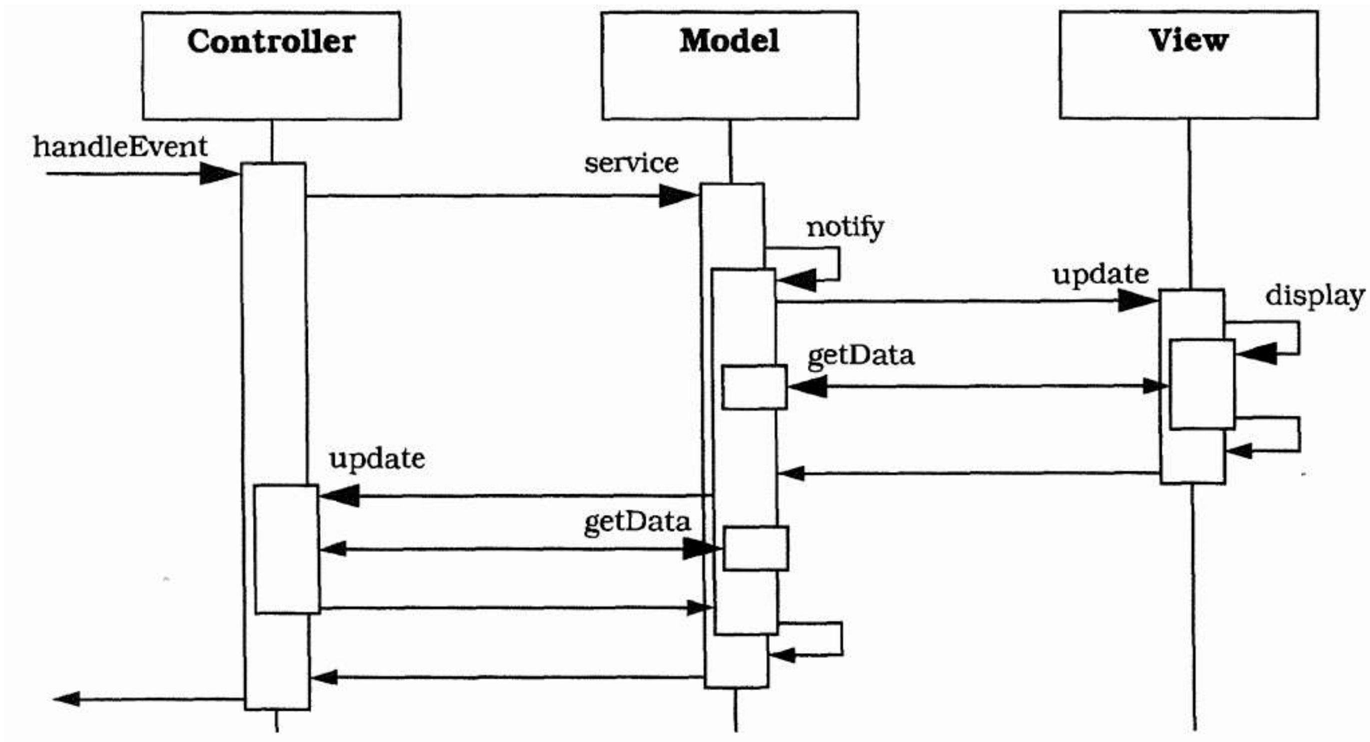
Class Controller	**Collaborators**
Responsibility • Accepts user input as events. • Translates events to service requests for the model or display requests for the view. • Implements the update procedure, if required.	• View • Model

Model-View-Controller: Class Structure



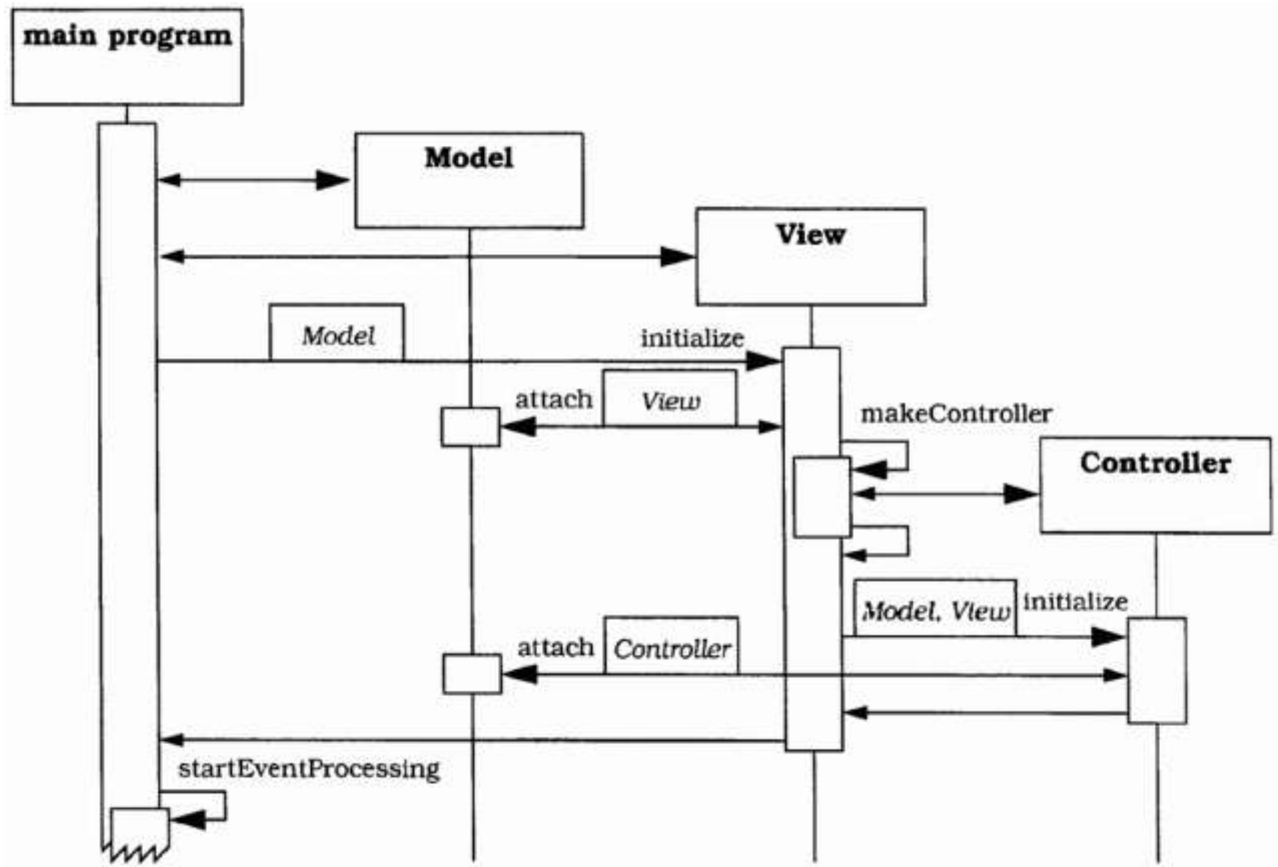
Model-View-Controller: Dynamics – Scenario I

- User input that results in changes to the model triggers the change-propagation mechanism.



Model-View-Controller: Dynamics – Scenario II

- the MVC triad is initialized.



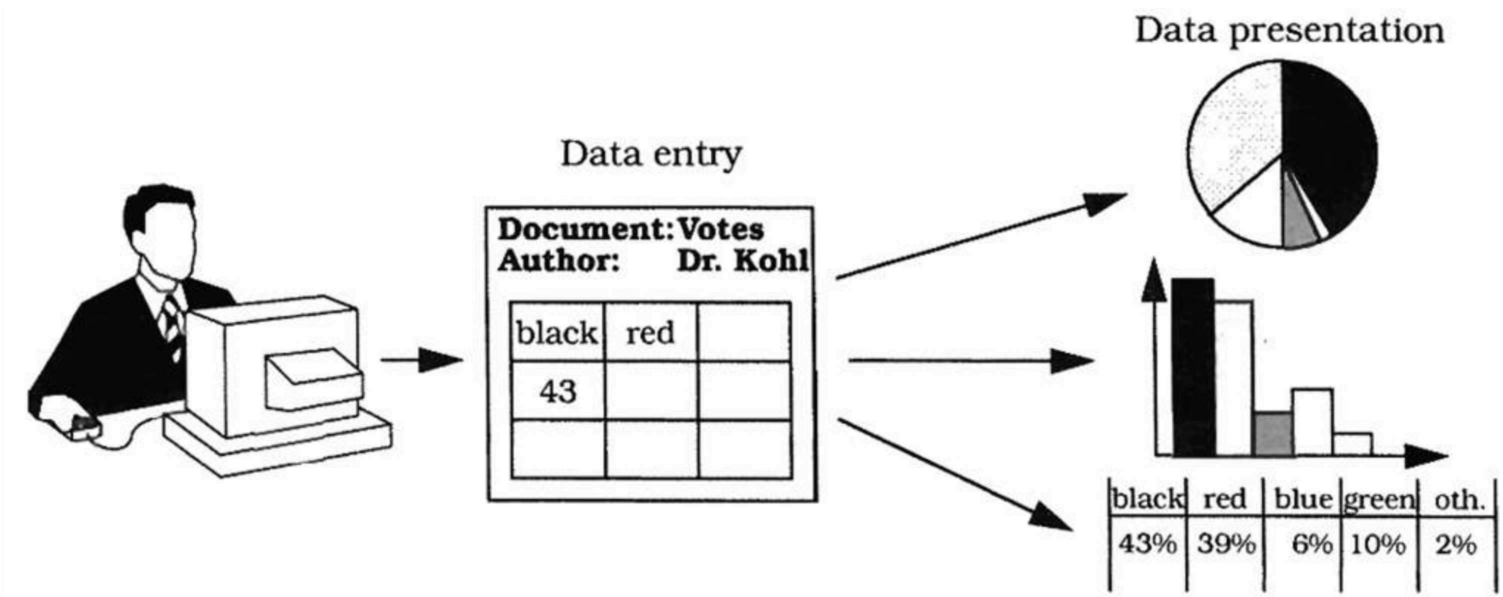
Model-View-Controller: Consequences

- ✓ Multiple views of the same model
- ✓ Synchronized views
- ✓ 'Pluggable' views and controllers
- ✓ Exchangeability of 'look and feel'
- ✓ Framework potential

- ✗ Increased complexity
- ✗ Potential for excessive number of updates
- ✗ Close couplings

Interactive Systems: Presentation-Abstraction-Control

- Defines a structure for interactive software systems in the form of a hierarchy of cooperating agents.
- Every agent:
  - is responsible for a specific aspect of the application's functionality, and
  - consists of three components: *presentation*, *abstraction*, and *control*.



Interactive Systems: Presentation-Abstraction-Control

- **Context -** Development of an interactive application with the help of agents
- **Problem -** Forces are as follows:
  - Agents often maintain their own state and data.
  - Interactive agents provide their own user interface.
  - Systems evolve over time.

Presentation-Abstraction-Control: Structure

- Interactive application is structured as a tree-like hierarchy of PAC agents.
  - There should be one top-level agent, several intermediate level agents, and even more bottom-level agents.
  - Every agent consists of three components: *presentation*, *abstraction*, and *control*.

- The *Top-level* PAC agent:
  - provides the functional core of the system;
  - includes parts of the user interface that cannot be assigned to subtasks.
- *Bottom-level* PAC agents:
  - represent self-contained semantic concepts on which users of the system can act, such as spreadsheets and charts.
- *Intermediate-level* PAC agents:
  - represent either combinations of, or relationships between, lower-level agents.

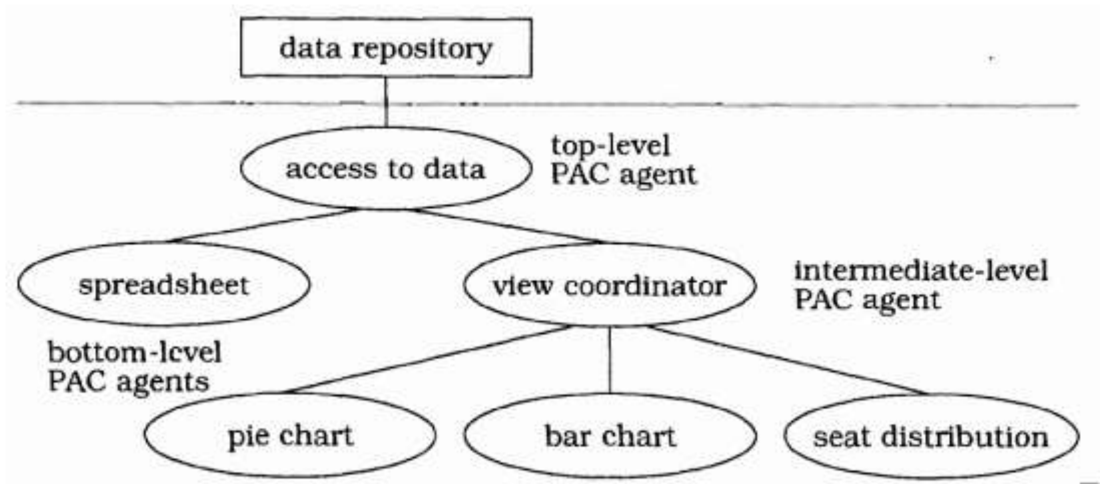
Presentation-Abstraction-Control: Structure – Agents

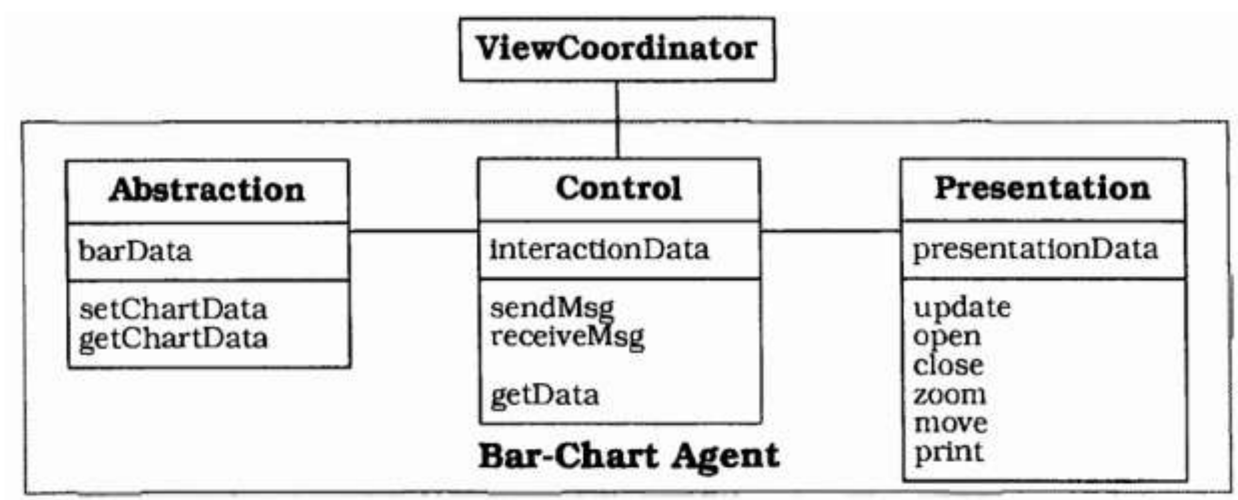
Class Top-level Agent	**Collaborators**
Responsibility • Provides the functional core of the system. • Controls the PAC hierarchy.	• Intermediate-level Agent • Bottom-level Agent

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

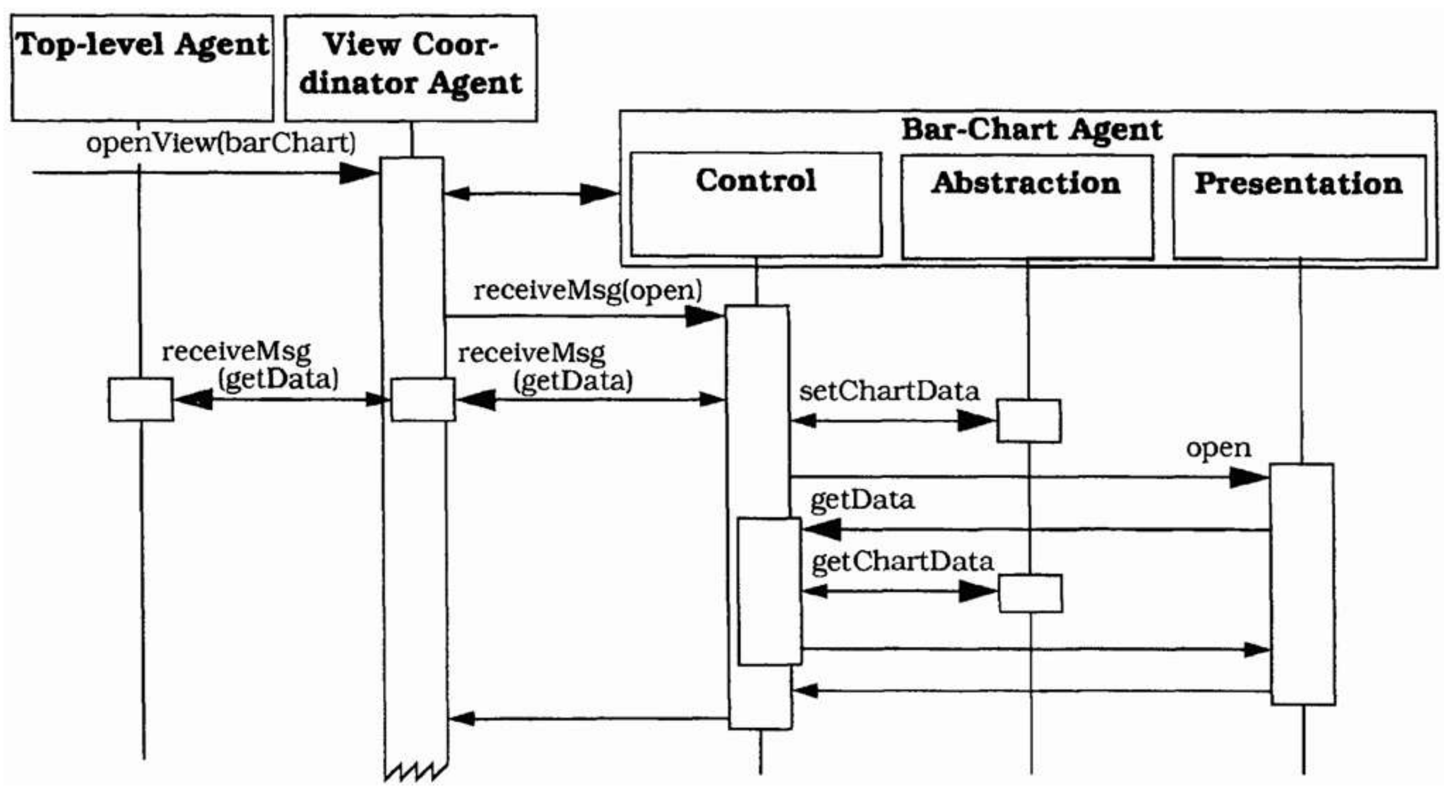
Presentation-Abstraction-Control: Structure – Components





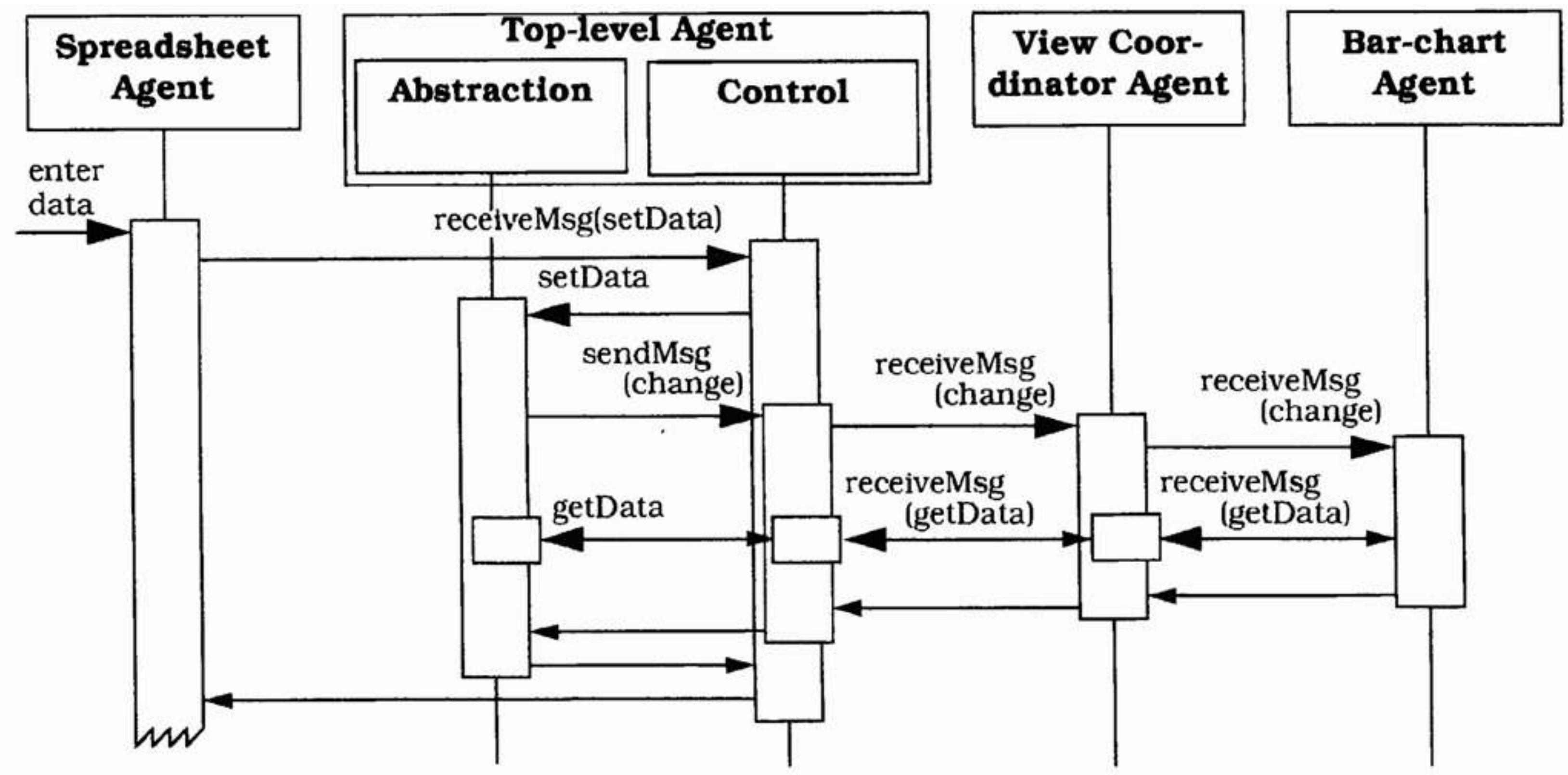
Presentation-Abstraction-Control: Dynamics – Scenario I

- Cooperation between different PAC agents when opening a new bar-chart view of the election data.



Presentation-Abstraction-Control: Dynamics – Scenario II

- Behavior of the system after new election data is entered.



Presentation-Abstraction-Control: Consequences

- ✓ Separation of concerns
- ✓ Support for change and extension
- ✓ Support for multi-tasking

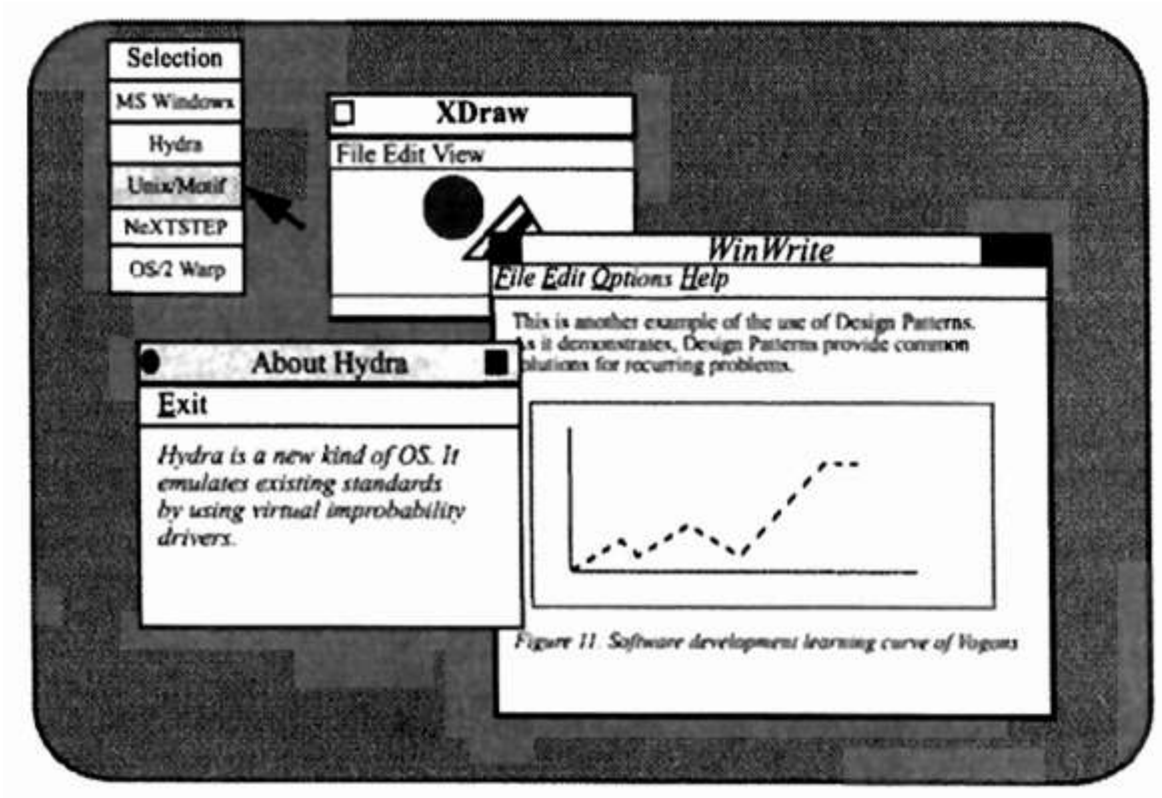
- ✗ Increased system complexity
- ✗ Complex control component
- ✗ Efficiency
- ✗ Applicability

Architectural: Adaptable Systems

- **Reflection:** provides a mechanism for changing structure and behavior of software systems dynamically.
  - Supports the modification of fundamental aspects, such as type structures and function call mechanisms.
  - An application is split into two parts.
    - A *meta level* provides information about selected system properties and makes the software self-aware.
    - A *base level* includes the application logic; its implementation builds on the meta level.
- **Microkernel:** Applies to software systems that must be able to adapt to changing system requirements.
  - separates a minimal functional core from extended functionality and customer-specific parts.
  - serves as a socket for plugging in such extensions and coordinating their collaboration.

Adaptable Systems: Microkernel

- Applies to software systems that must be able to adapt to changing system requirements.
  - Separates a minimal functional core from extended functionality and customer-specific parts.



Adaptable Systems: Microkernel

- **Context -** The development of several applications that use similar programming interfaces that build on the same core functionality.
- **Problem -** Forces are as follows:
  - The application platform must cope with continuous hardware and software evolution.
  - The application platform should be portable, extensible and adaptable to allow easy integration of emerging technologies.
  - The applications in your domain need to support different, but similar, application platforms.
  - The applications may be categorized into groups that use the same functional core in different ways: the underlying platform must emulate existing standards.
  - The functional core of the application platform should be separated into:
    - a component with minimal memory size, and
    - services that consume as little processing power as possible.

Microkernel: Structure – Microkernel

- Fundamental services of the application platform are encapsulated in a *Microkernel* component, which
  - includes functionality that enables other components running in separate processes to communicate with each other.
  - is responsible for maintaining system-wide resources such as files or processes.
  - provides interfaces that enable other components to access its functionality.

Class	***Collaborators***
Microkernel	• Internal Server
Responsibility • Provides core mechanisms. • Offers communi-cation facilities. • Encapsulates sys-tem dependencies. • Manages and controls resources.	

Microkernel: Structure – Servers

- Core functionality that cannot be implemented within the microkernel without unnecessarily increasing its size or complexity is separated in *Internal Servers.*
- *External Servers* are separate processes that represent other application platforms; they implement their own view of the underlying microkernel.

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

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

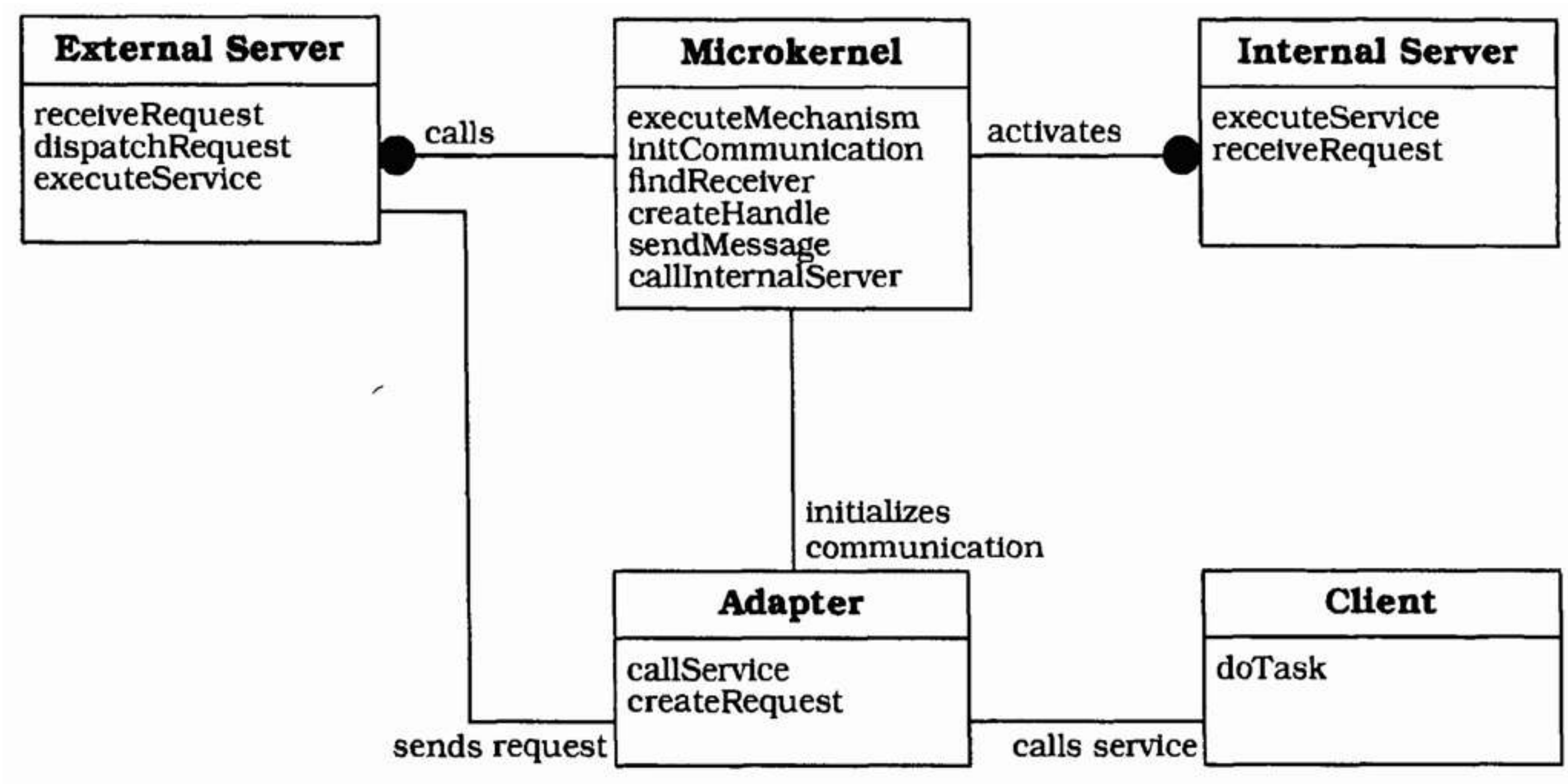
Microkernel: Structure – Clients and Adapters

- *Clients* communicate with external servers by using the communication facilities provided by the microkernel.
- *Adapters* represent interfaces between clients and their external servers, allowing clients to access services of their external server in a portable way.

Class Client	**Collaborators**
Responsibility • Represents an application.	• Adapter

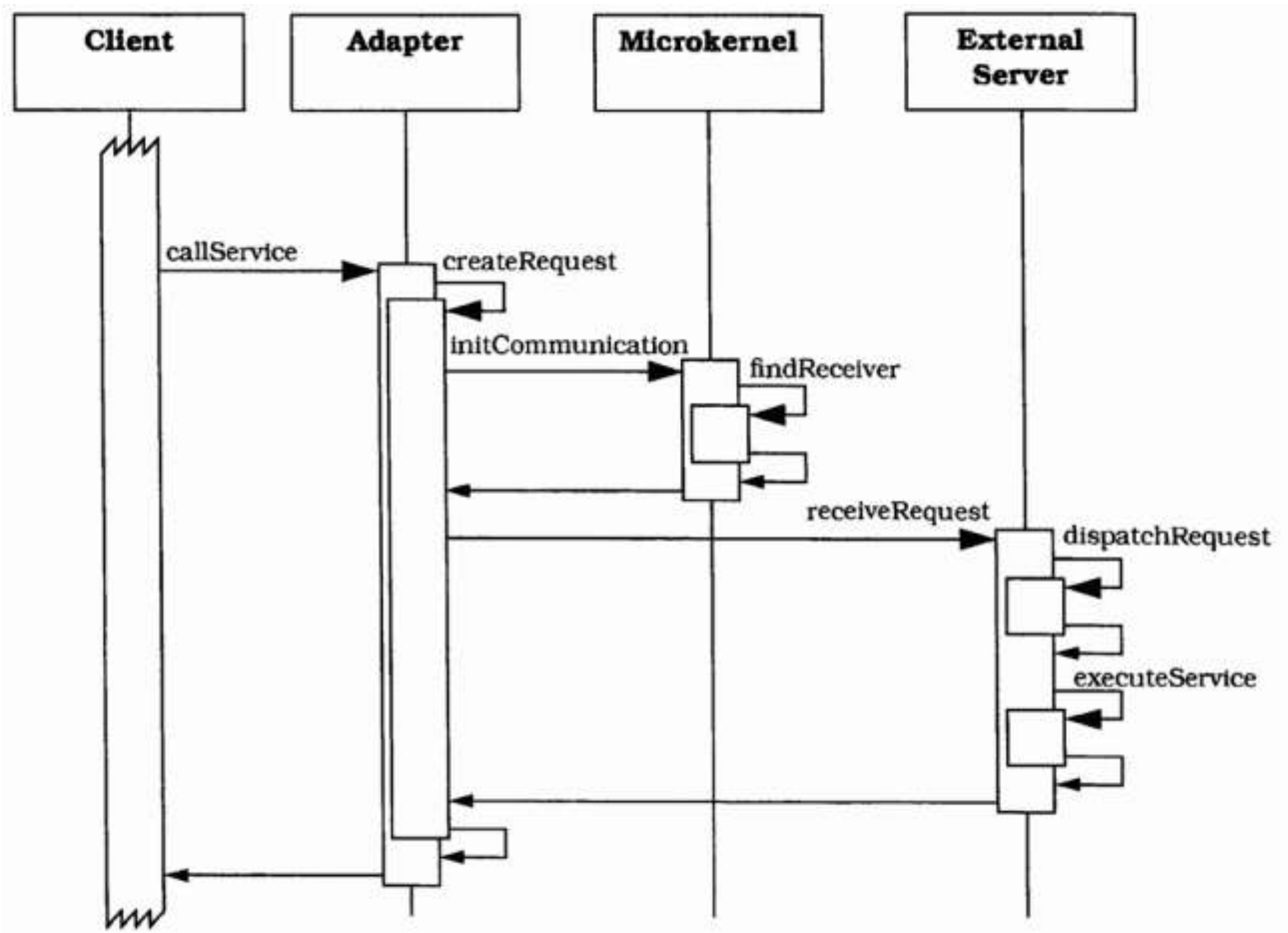
Class Adapter	**Collaborators**
Responsibility • Hides system dependencies such as communication facilities from the client. • Invokes methods of external servers on behalf of clients.	• External Server • Microkernel

Microkernel: Structure – Class Diagram



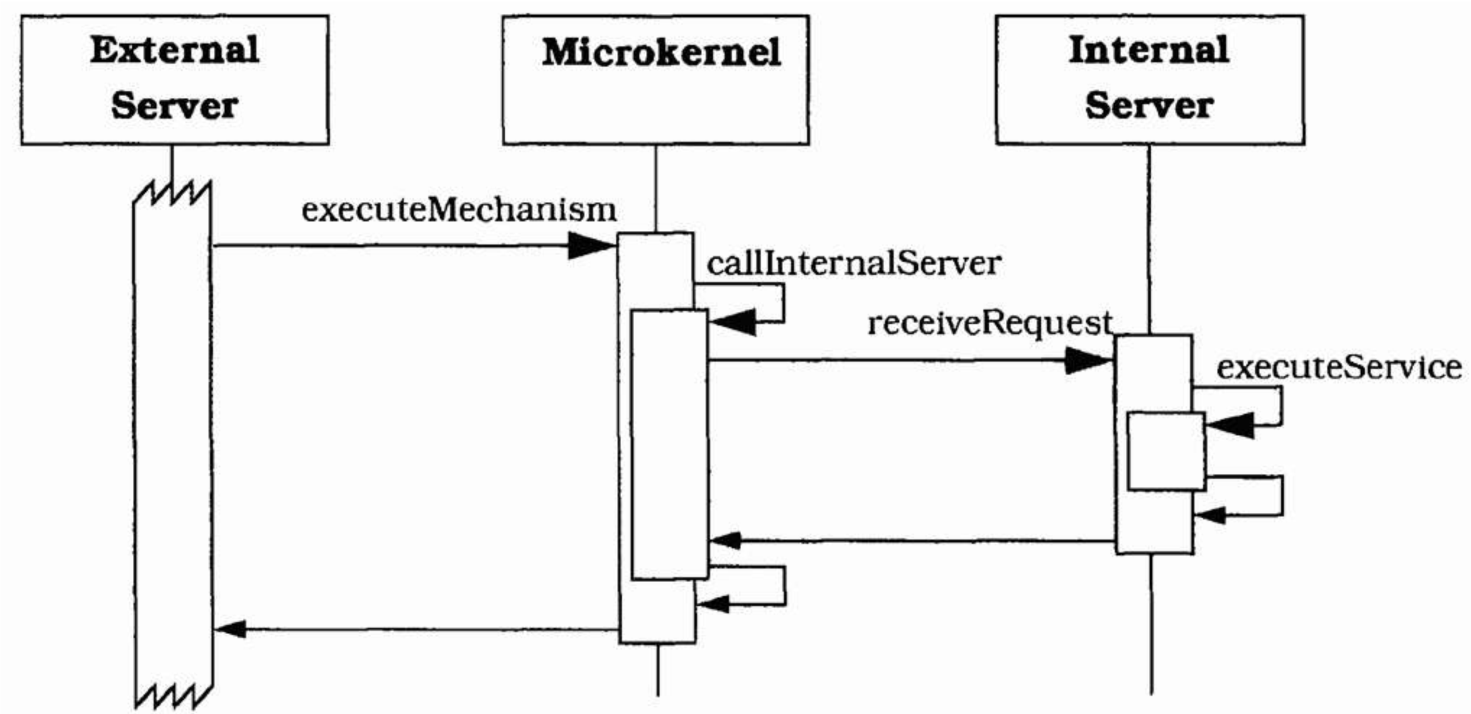
Microkernel: Dynamics – Scenario I

- A client calls a service of its external server



Microkernel: Dynamics – Scenario II

- An external server requests a service that is provided by an internal server.



Microkernel: Consequences

- ✓ Portability
- ✓ Flexibility and Extensibility
- ✓ Separation of policy and mechanism
- ✓ Scalability
- ✓ Reliability
- ✓ Transparency

- ✗ Performance
- ✗ Complexity of design and implementation

Reference

- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., *Pattern-Oriented Software Architecture: A System of Patterns,* Vol. 1. Wiley, 1996.