Agile Software Development

Lecturer: **Raman Ramsin**

Lecture 17

Patterns for Agile Development

Software Patterns

- Software Patterns support reuse of software architecture and design.
  - Patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a particular domain.
- Patterns represent solutions to problems that arise when developing software within a particular context.
  - i.e., "Pattern == problem/solution pair in a context"

GoF Design Patterns – Principles

- Emphasis on flexibility and reuse through decoupling of classes.

- The underlying principles:
  - program to an interface, not to an implementation.
  - favor composition over class inheritance.
  - find what varies and encapsulate it.

GoF Design Patterns: General Categories

- 23 patterns are divided into three separate categories:
  - **Creational** patterns
    - Deal with initializing and configuring classes and objects.
  - **Structural** patterns
    - Deal with decoupling interface and implementation of classes and objects.
  - **Behavioral** patterns
    - Deal with dynamic interactions among societies of classes and objects.

GoF Design Patterns: Purpose and Scope

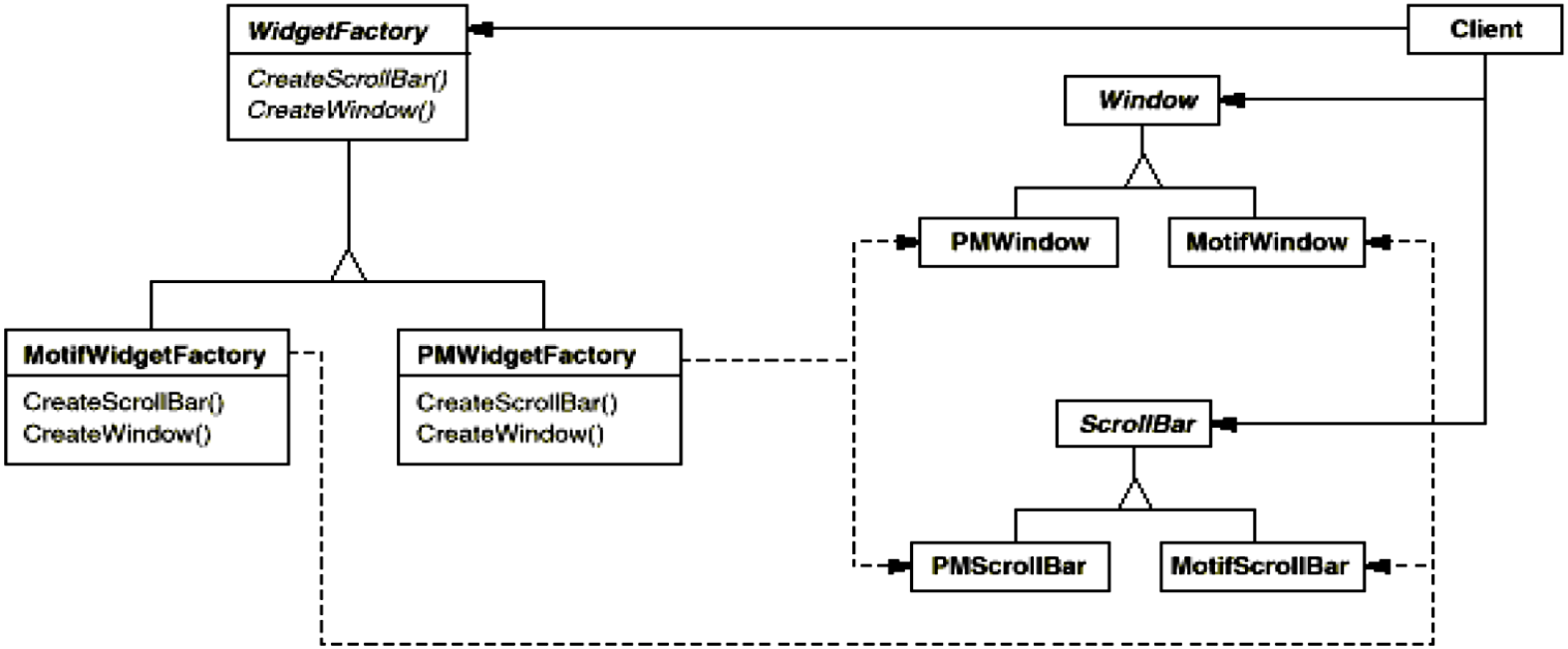
		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter
				Template Method
	Object	Abstract Factory	Adapter (object)	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

GoF Creational Patterns

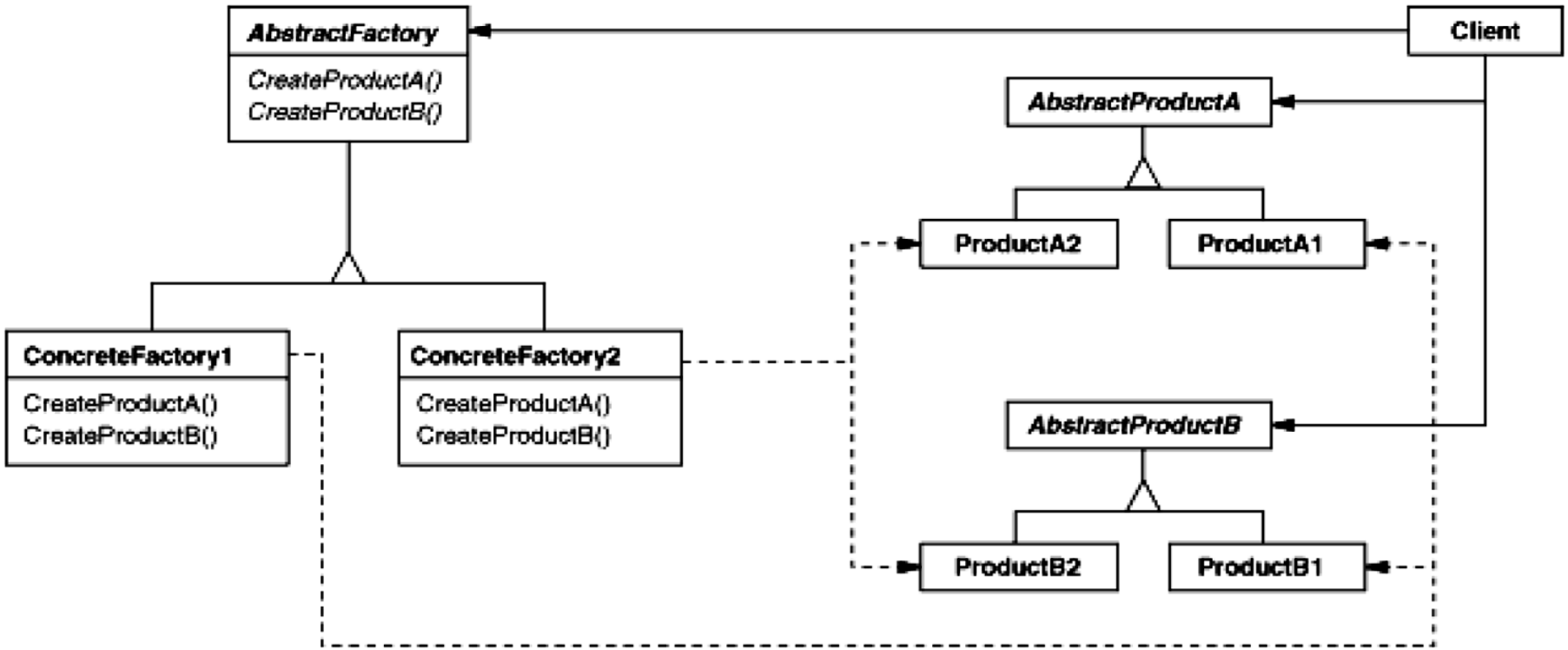
- **Class**
  - ***Factory Method:*** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
- **Object**
  - ***Abstract Factory:*** Provide an interface for creating families of related or dependent objects without specifying their concrete class.
  - ***Builder:*** Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - ***Prototype:*** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - ***Singleton:*** Ensure a class only has one instance, and provide a global point of access to it.

Abstract Factory

- Intent:
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

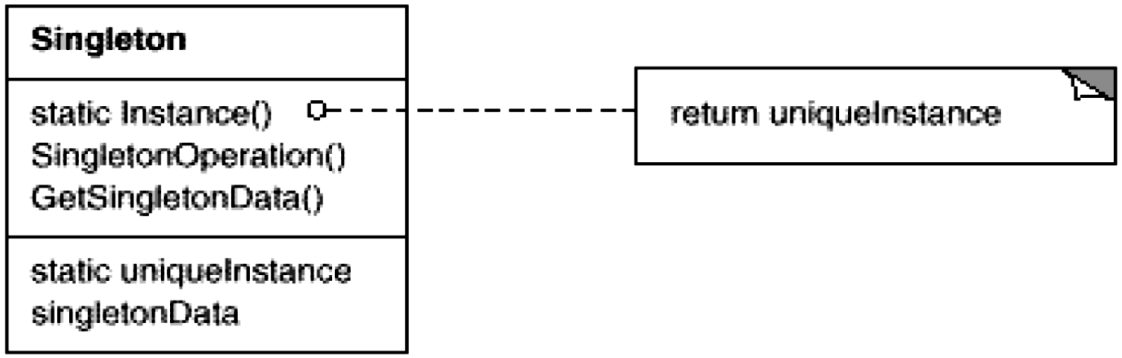


Abstract Factory: Structure



Singleton

- Intent:
  - Ensure a class only has one instance, and provide a global point of access to it.



Singleton: Applicability

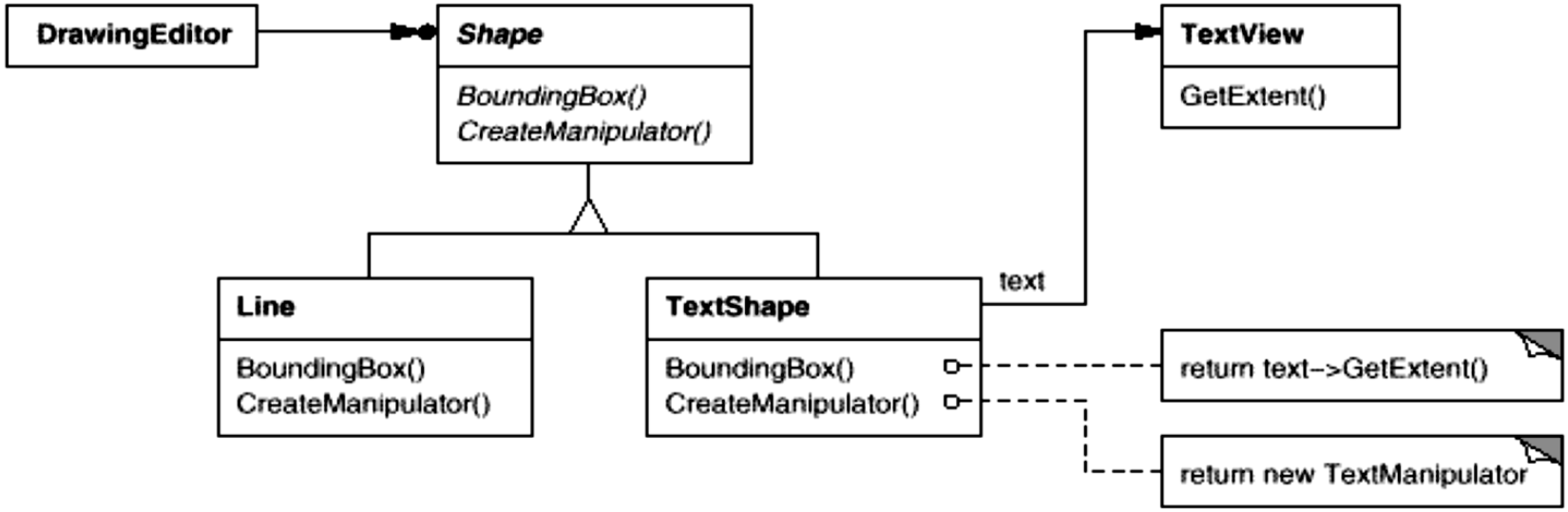
- Use the Singleton pattern when
  - there must be exactly one instance of a class, and it must be accessible to clients from a well known access point.
  - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

GoF Structural Patterns

- **Class/Object**
  - ***Adapter:*** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Object**
  - ***Bridge:*** Decouple an abstraction from its implementation so that the two can vary independently.
  - ***Composite:*** Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
  - ***Decorator:*** Attach additional responsibilities to an object dynamically.
  - ***Façade:*** Provide a unified interface to a set of interfaces in a subsystem.
  - ***Flyweight:*** Use sharing to support large numbers of fine-grained objects efficiently.
  - ***Proxy:*** Provide a surrogate or placeholder for another object to control access to it.

Adapter

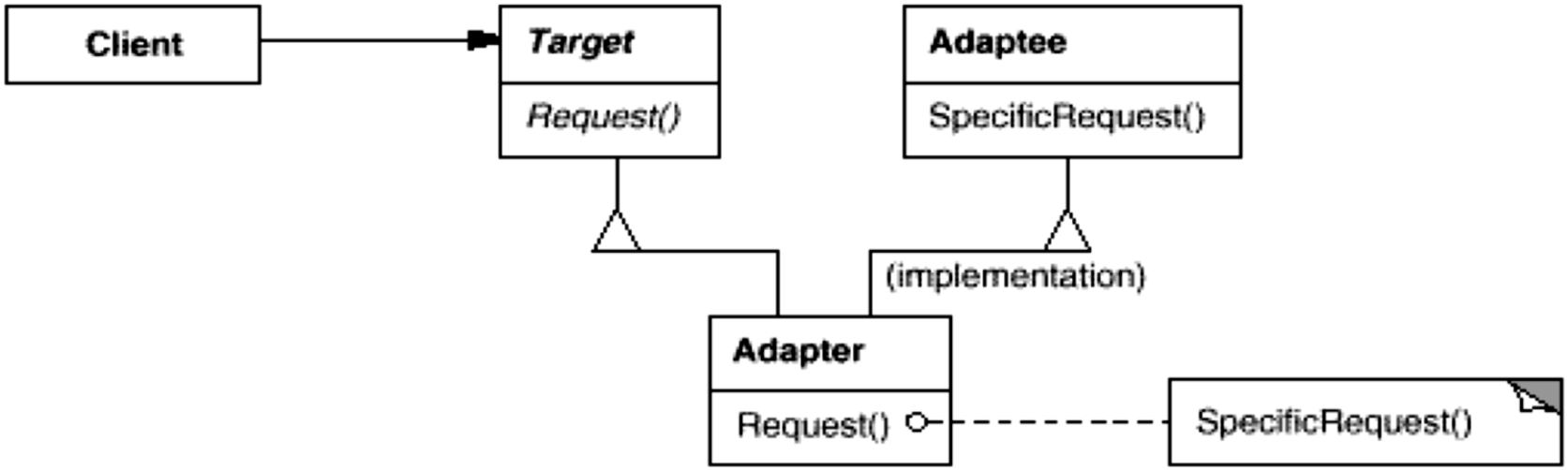
- Intent:
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



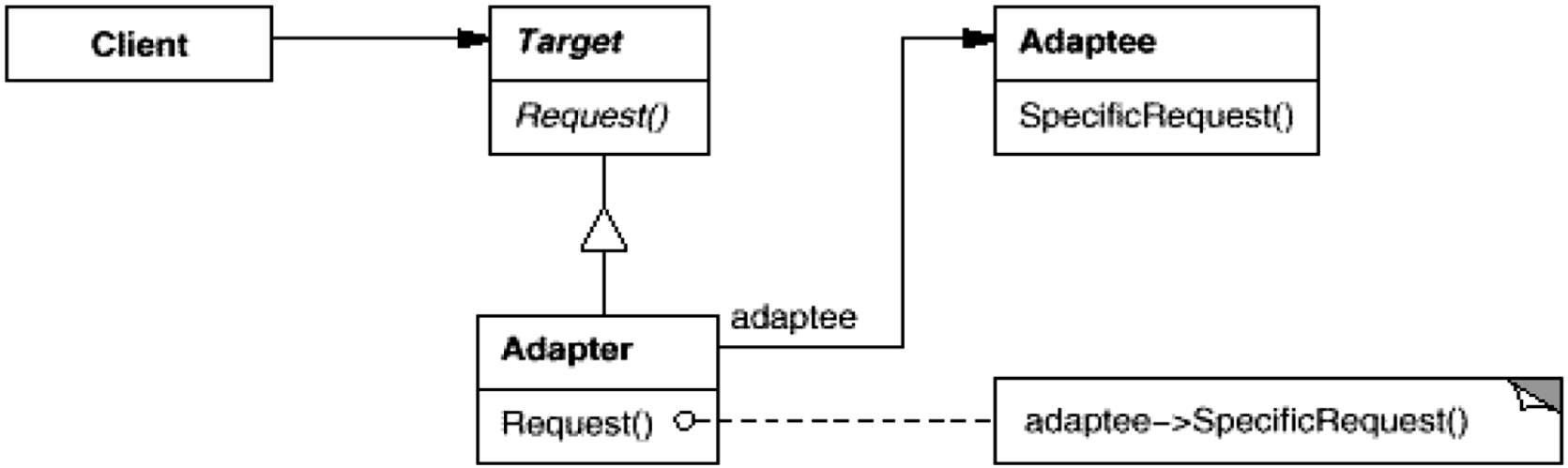
Adapter: Applicability

- Use the Adapter pattern when
  - you want to use an existing class, and its interface does not match the one you need.
  - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
  - *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Adapter (Class): Structure

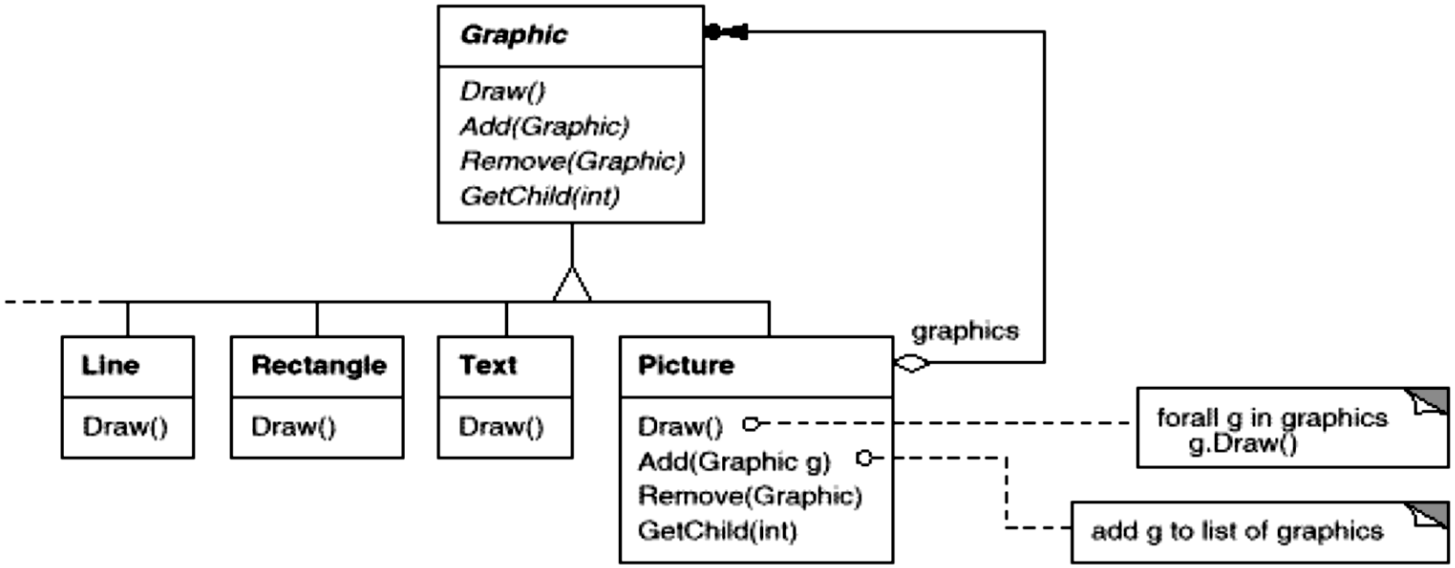


Adapter (Object): Structure

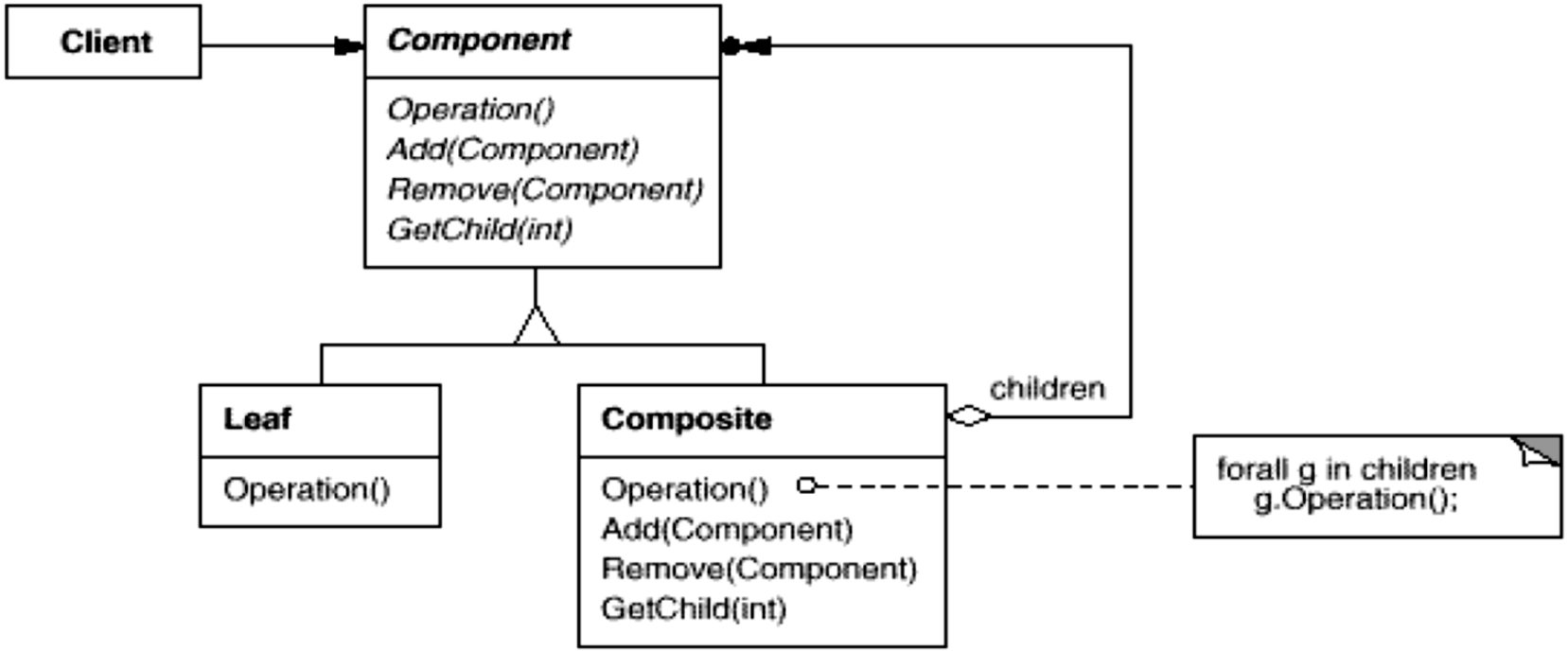


Composite

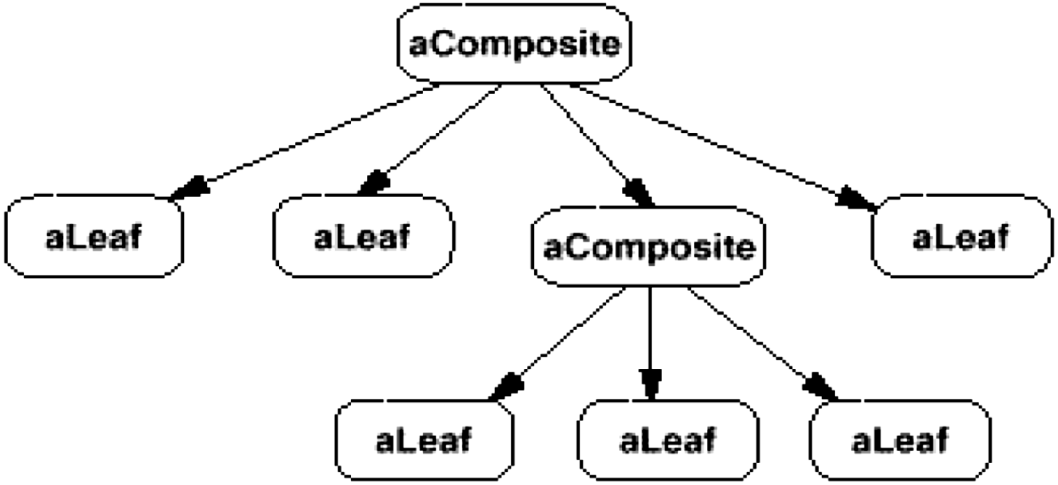
- Intent:
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



Composite: Structure

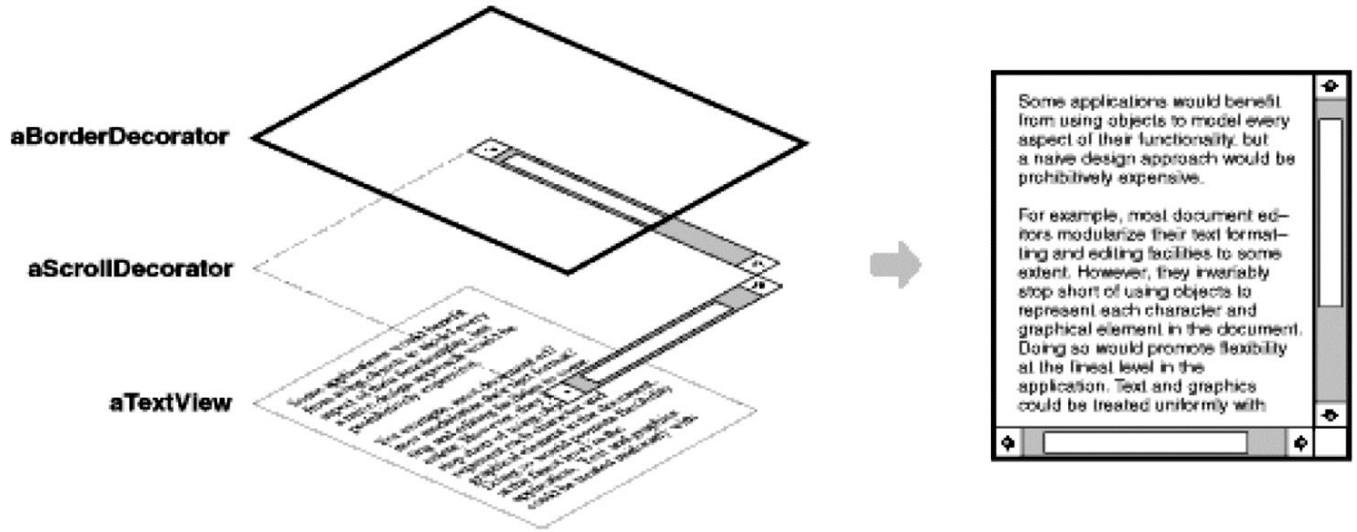


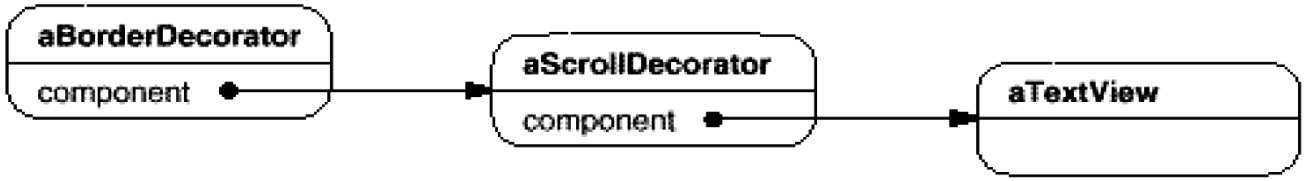
Composite: Typical Object Structure



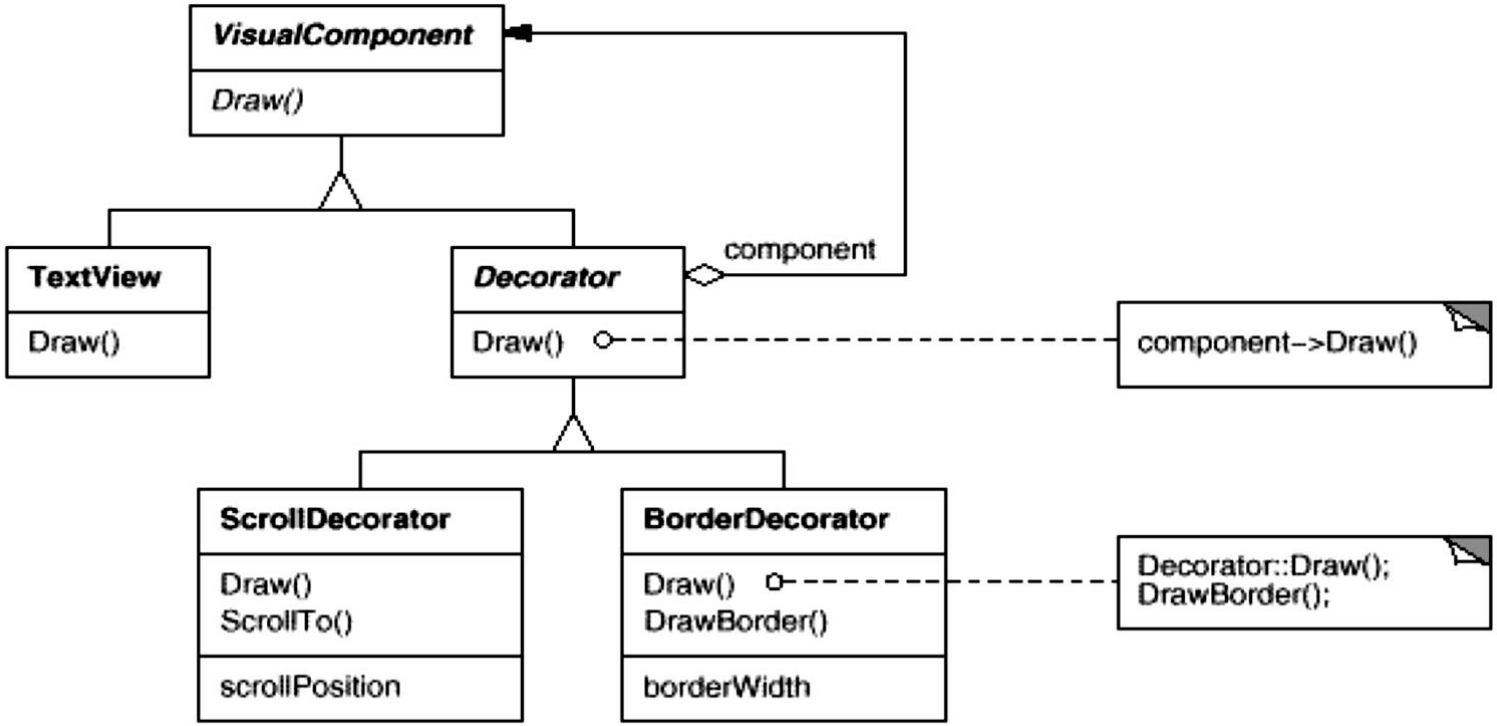
Decorator

- Intent:
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

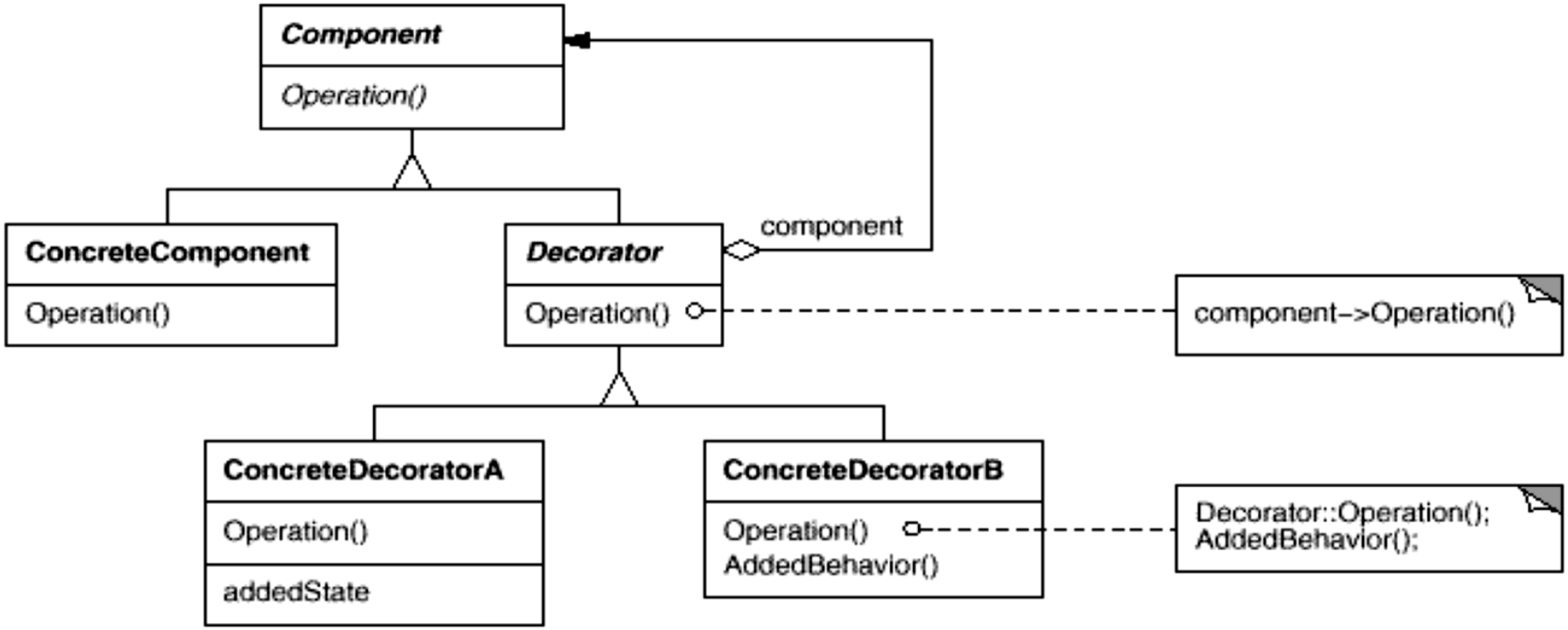




Decorator: Class Hierarchy

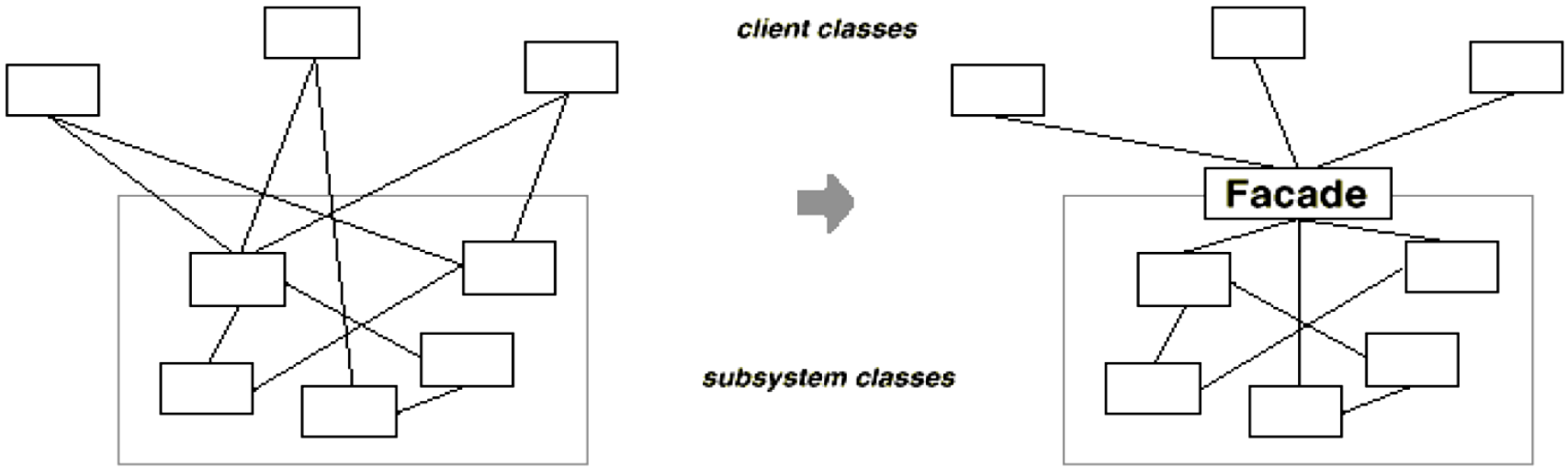


Decorator: Structure

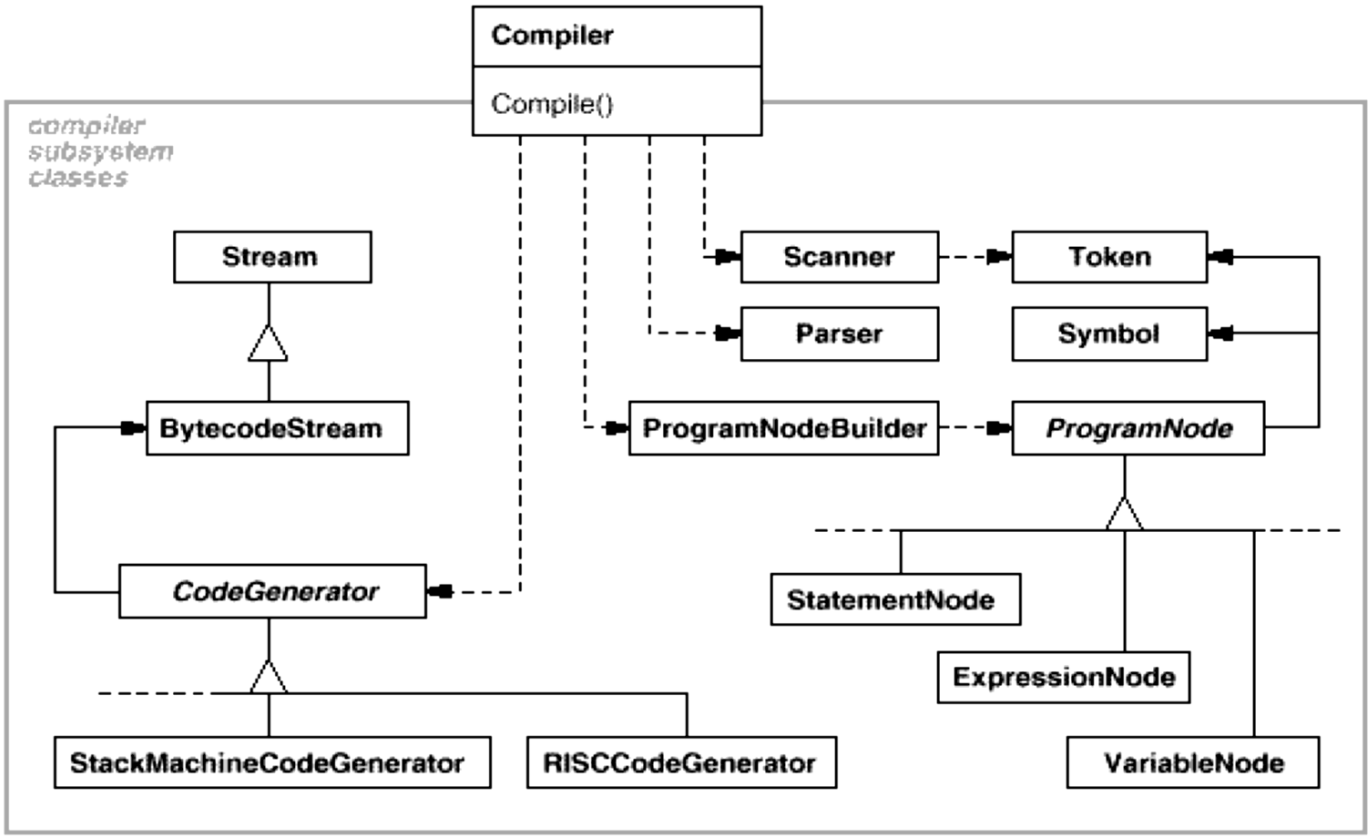


Façade

- Intent:
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

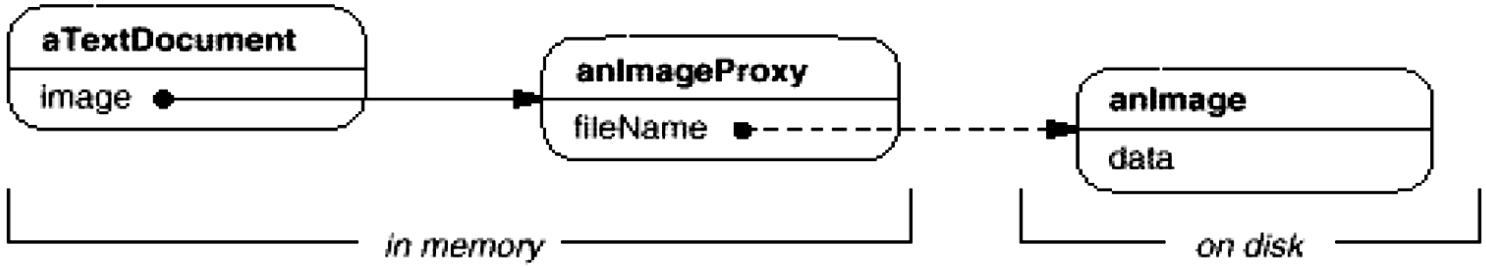


Façade: Class Hierarchy

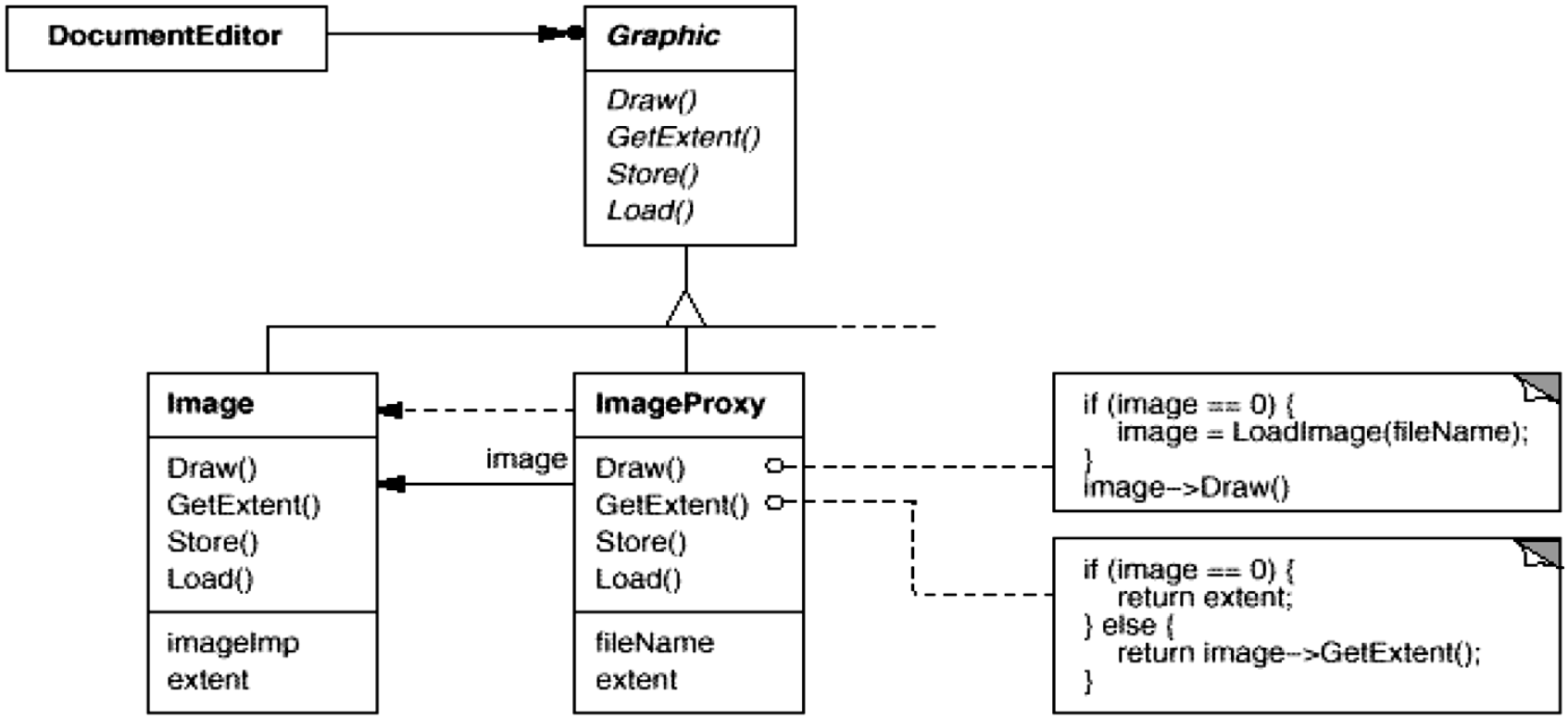


Proxy

- Intent:
  - Provide a surrogate or placeholder for another object to control access to it.



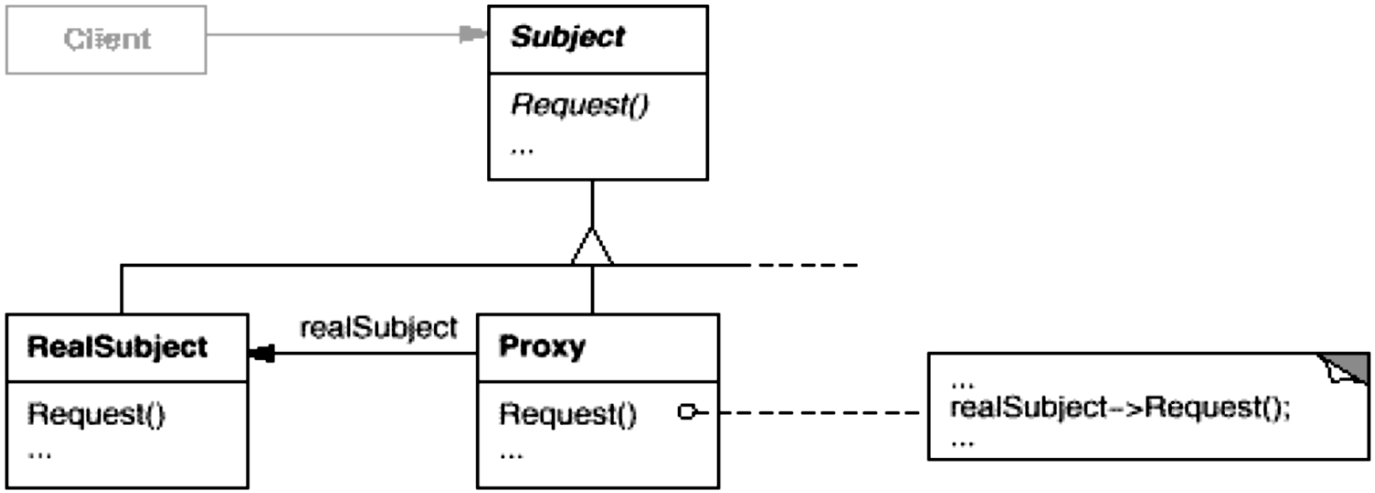
Proxy: Class Hierarchy

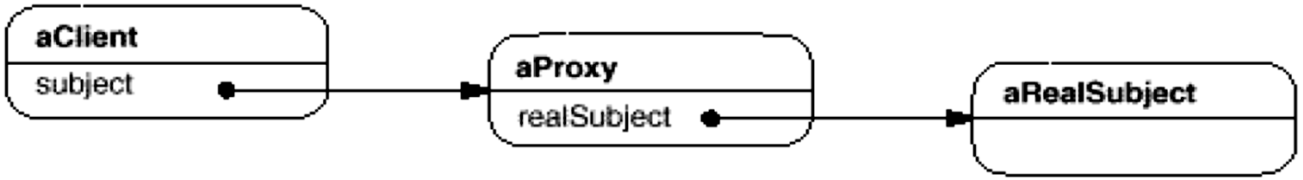


Proxy: Applicability

- Use the Proxy pattern when a surrogate is needed:
  - **Remote proxy:** provides a local representative for an object in a different address space.
  - **Virtual proxy:** creates expensive objects on demand.
  - **Protection proxy:** controls access to the original object.
  - **Smart reference:** a replacement for a bare pointer that performs additional actions when an object is accessed:
    - counting the number of references to the real object so that it can be freed when there are no more references.
    - loading a persistent object into memory when it's first referenced.
    - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Proxy: Structure





GoF Behavioral Patterns – Class

- **Class**
  - ***Interpreter:*** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
  - ***Template Method:*** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses; lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

GoF Behavioral Patterns – Object

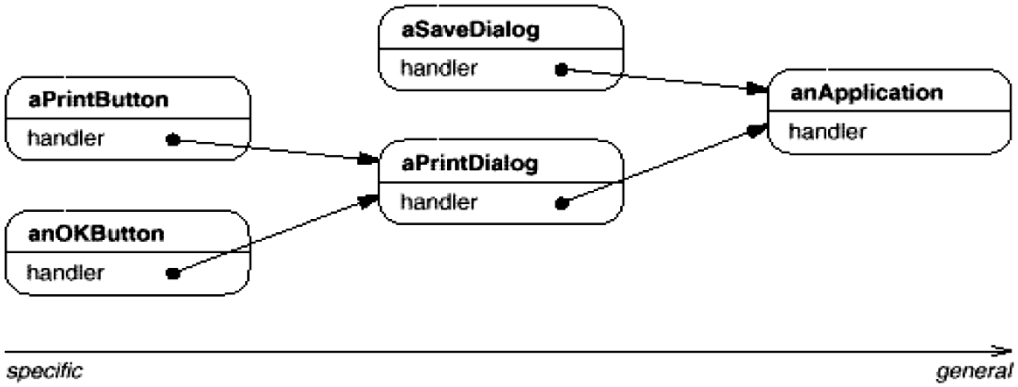
- **Object**
  - ***Chain of Responsibility:*** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
  - ***Command:*** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
  - ***Iterator:*** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
  - ***Mediator:*** Define an object that encapsulates how a set of objects interact; promotes loose coupling by keeping objects from referring to each other explicitly.

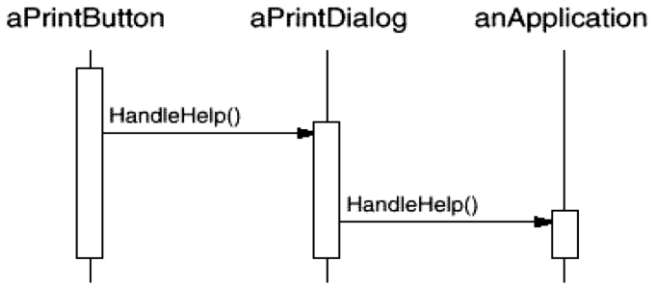
GoF Behavioral Patterns – Object (Contd.)

- **Object (Contd.)**
  - ***Memento:*** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
  - ***Observer:*** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  - ***State:*** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
  - ***Strategy:*** Define a family of algorithms, encapsulate each one, and make them interchangeable; lets the algorithm vary independently from clients that use it.
  - ***Visitor:*** Represent an operation to be performed on the elements of an object structure; lets you define a new operation without changing the classes of the elements.

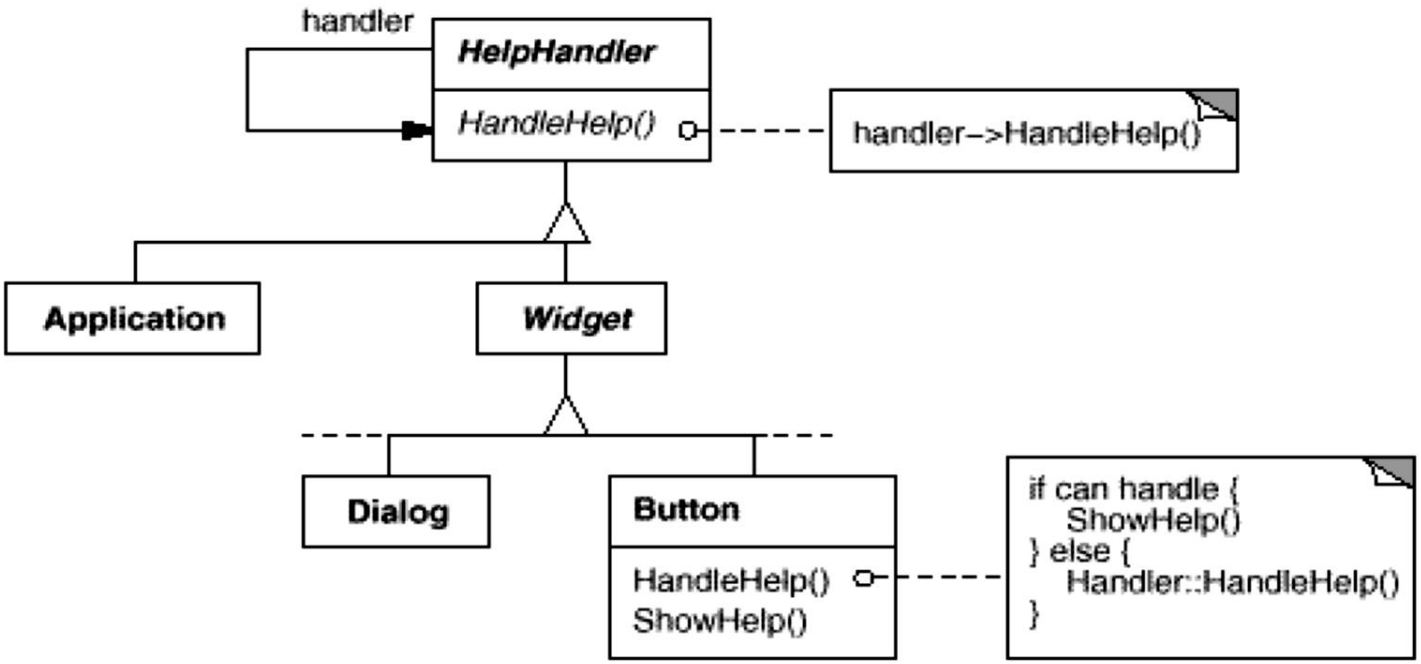
Chain of Responsibility

- Intent:
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

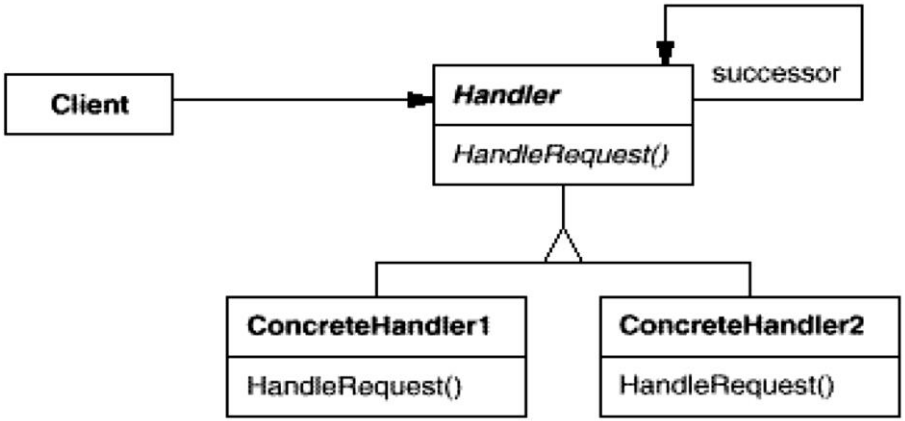




Chain of Responsibility: Class Hierarchy

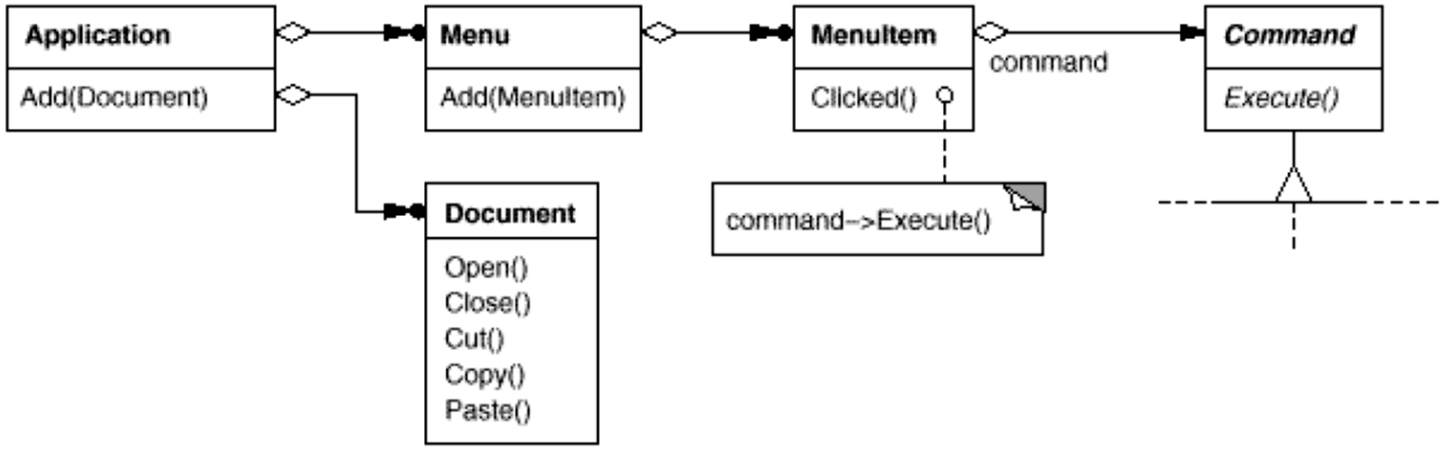


Chain of Responsibility: Structure

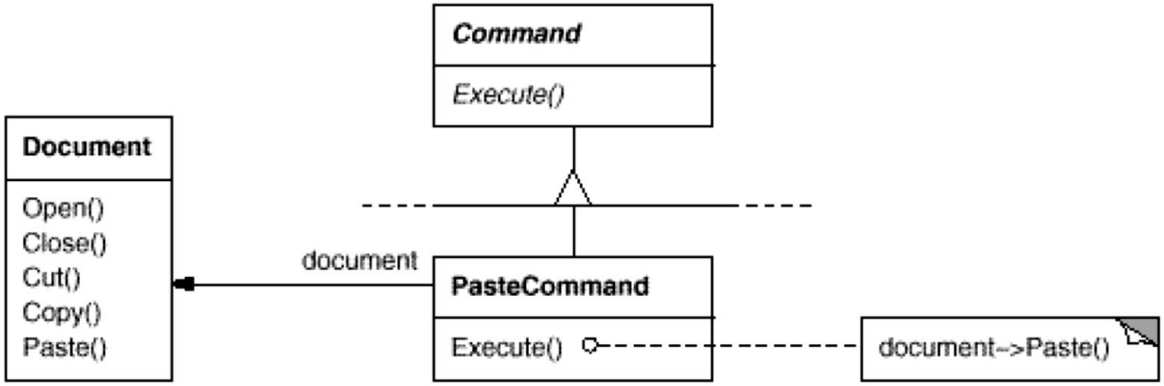


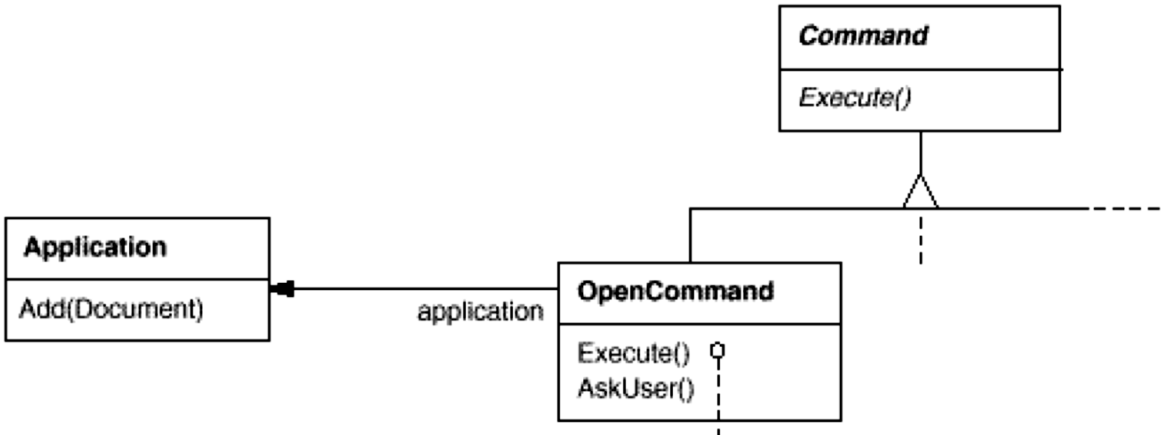
Command

- Intent:
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

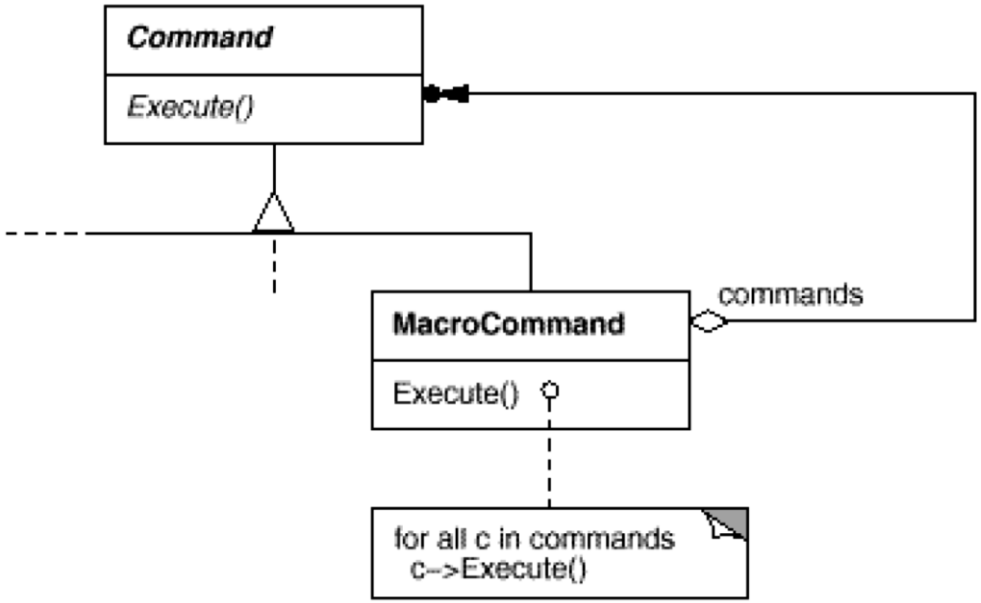


Command: Examples

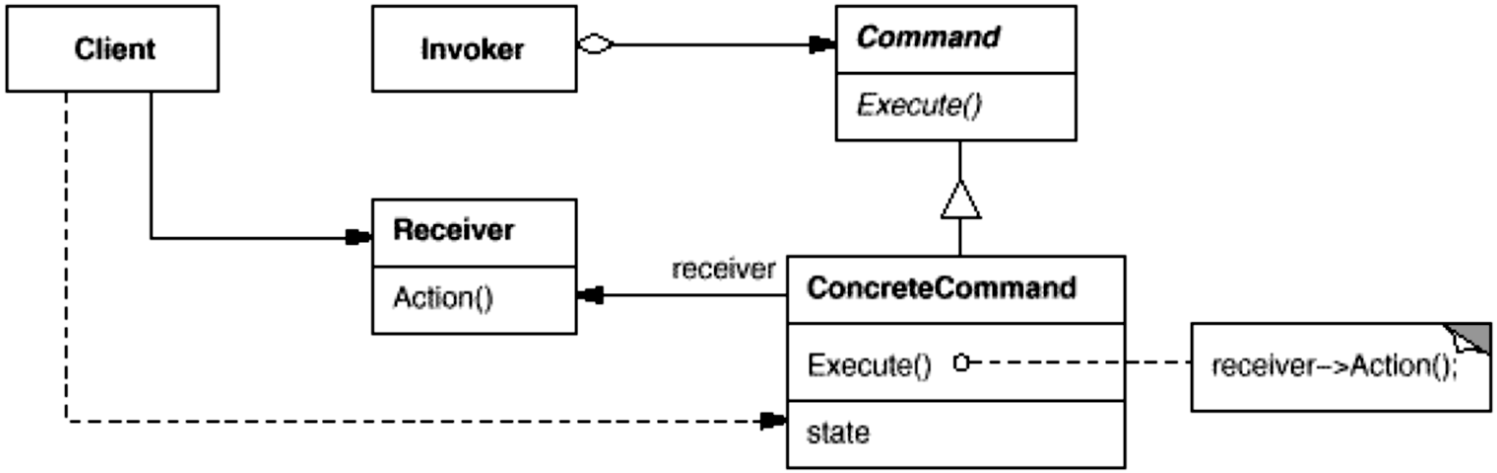




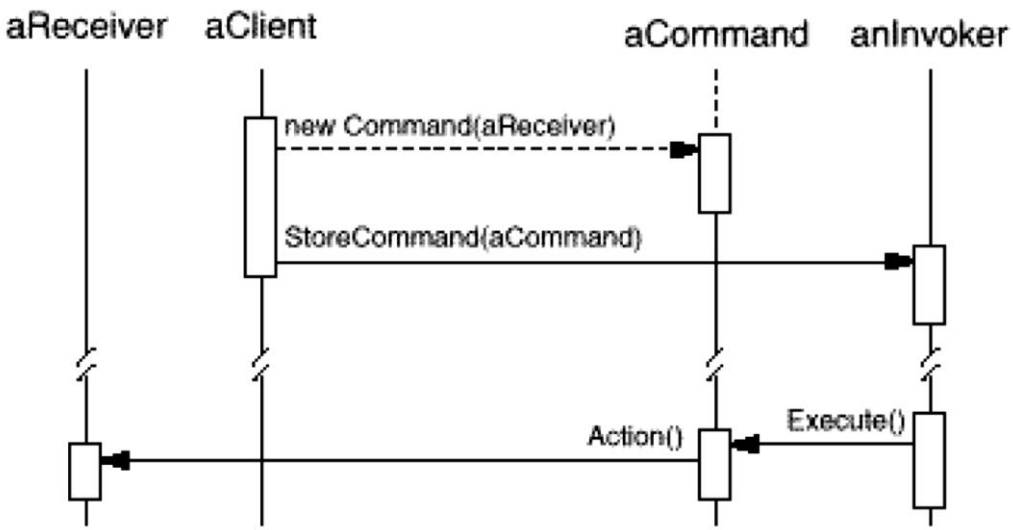
Command: Macro-Command



Command: Structure

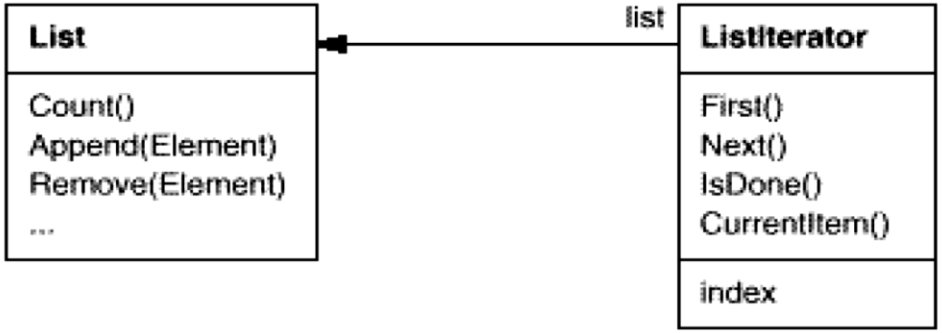


Command: Collaboration

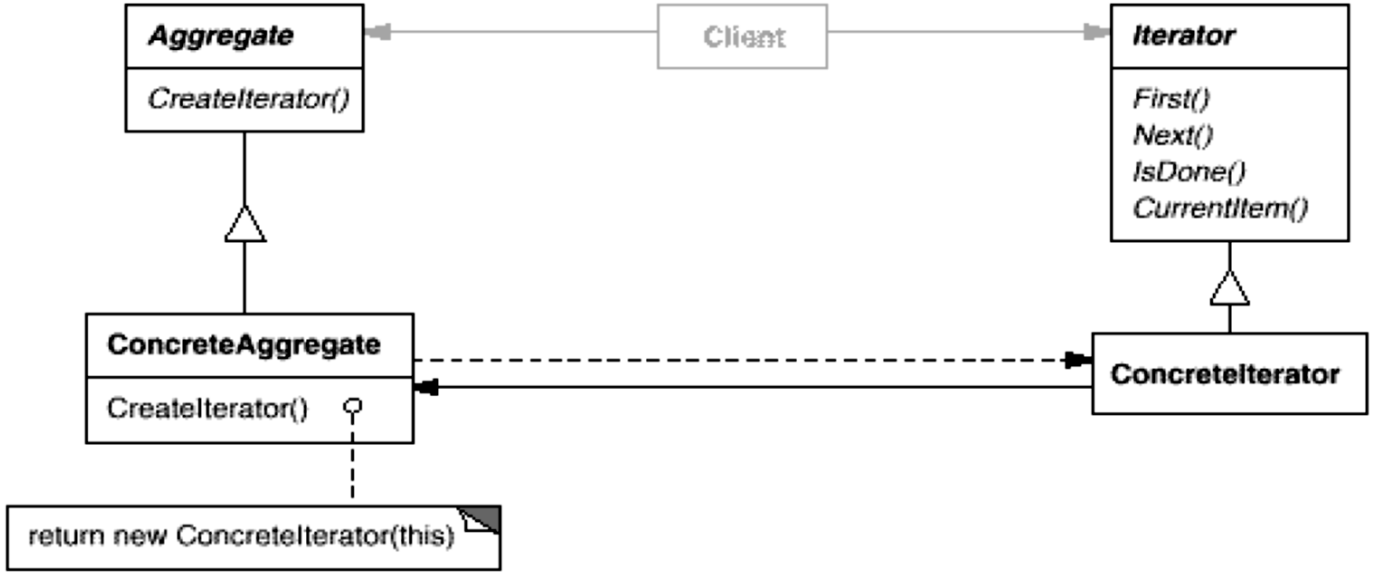


Iterator

- Intent:
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

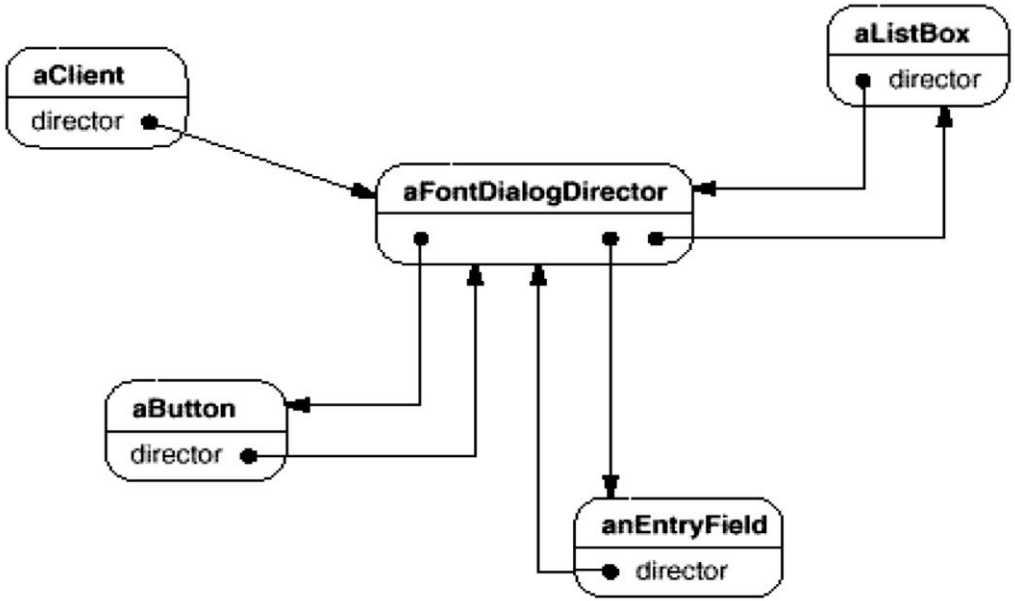


Iterator: Structure

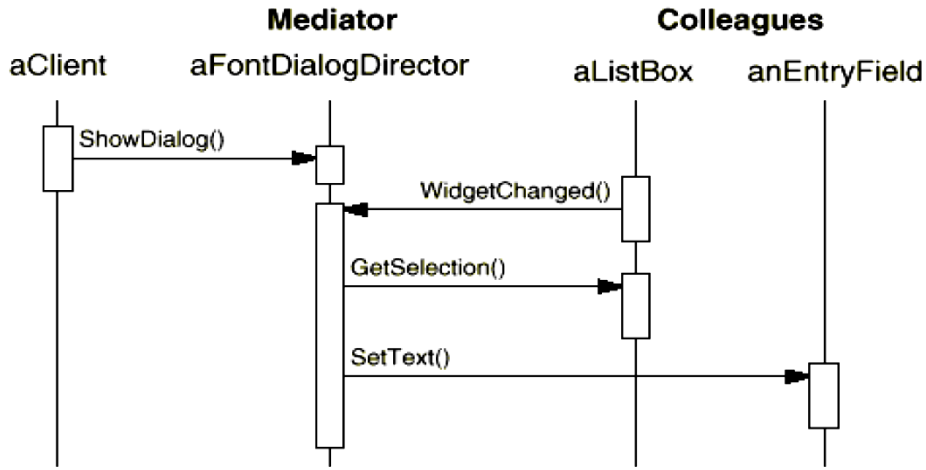


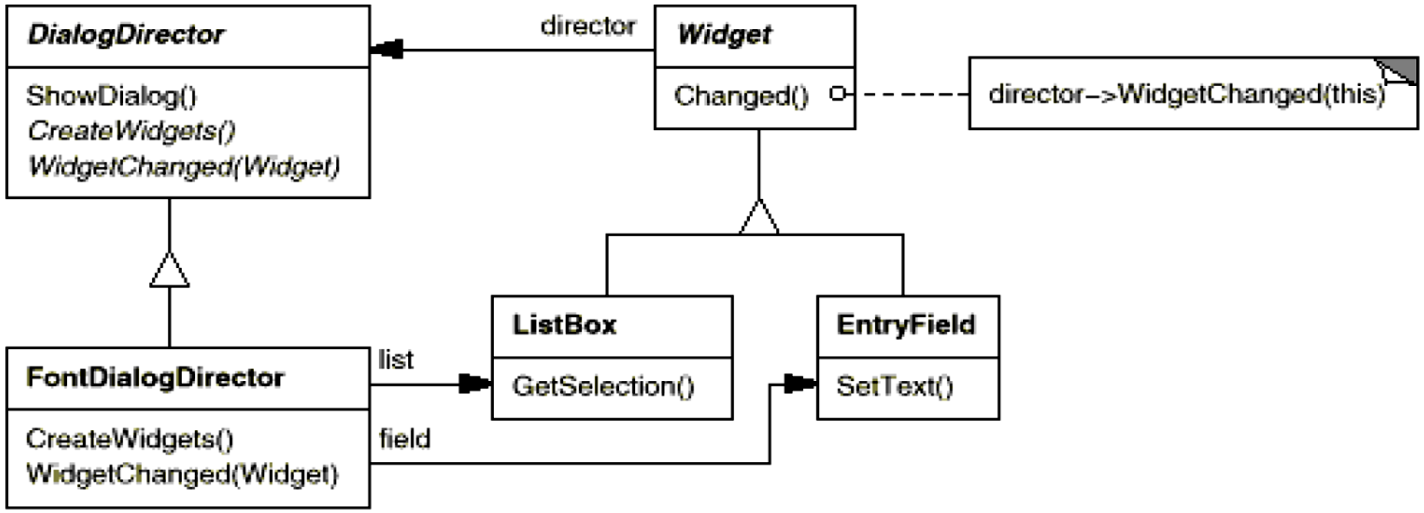
Mediator

- Intent:
  - Define an object that encapsulates how a set of objects interact: promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.

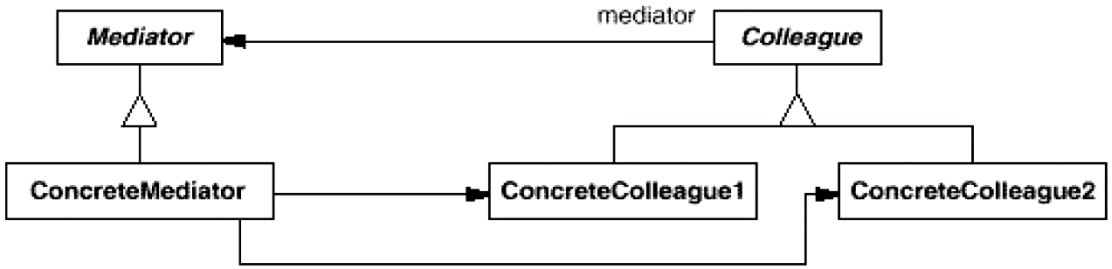


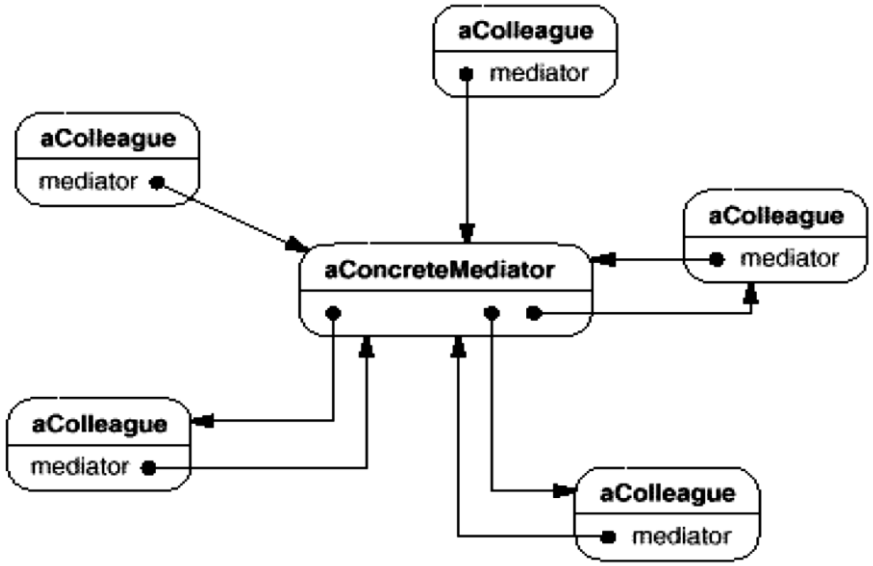
Mediator: Typical Collaboration and Class Hierarchy





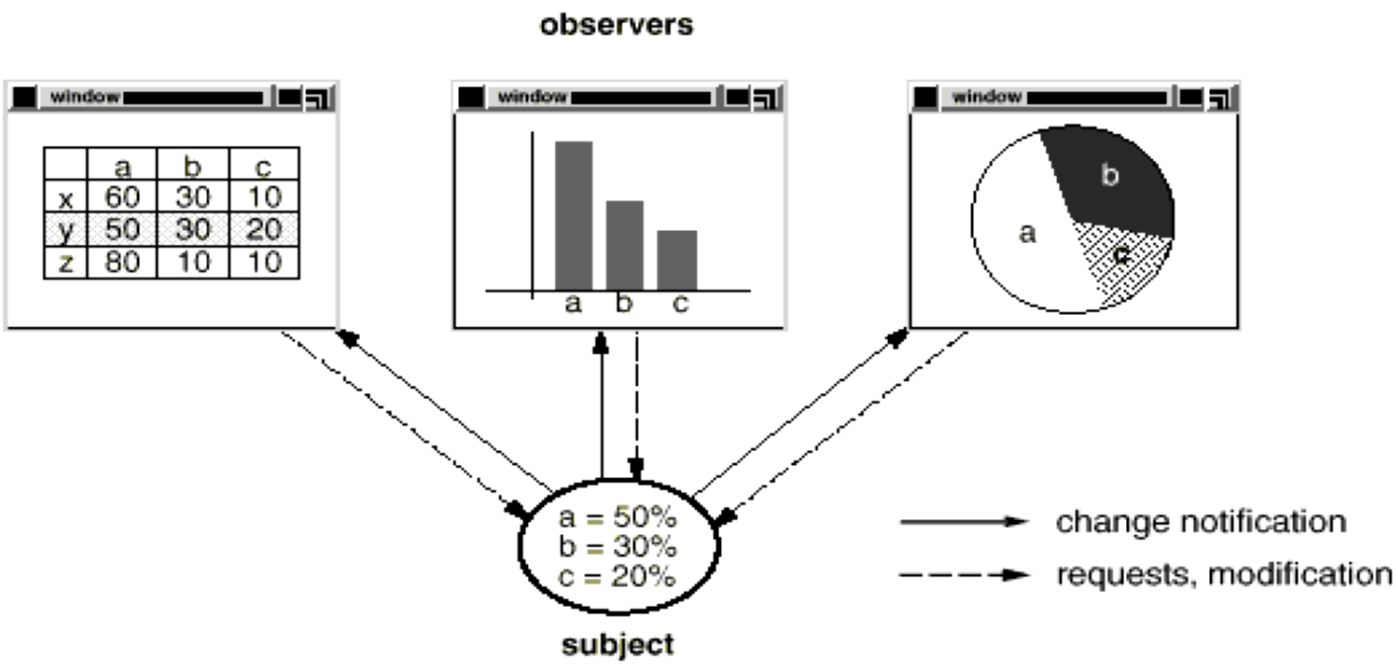
Mediator: Structure



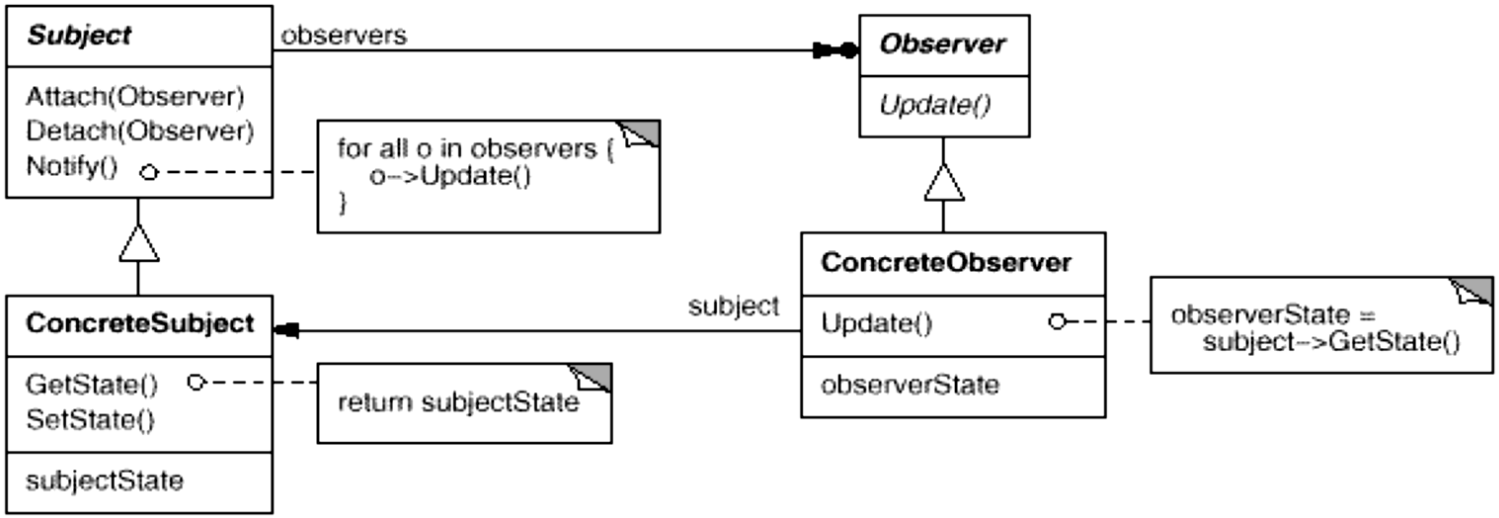


Observer

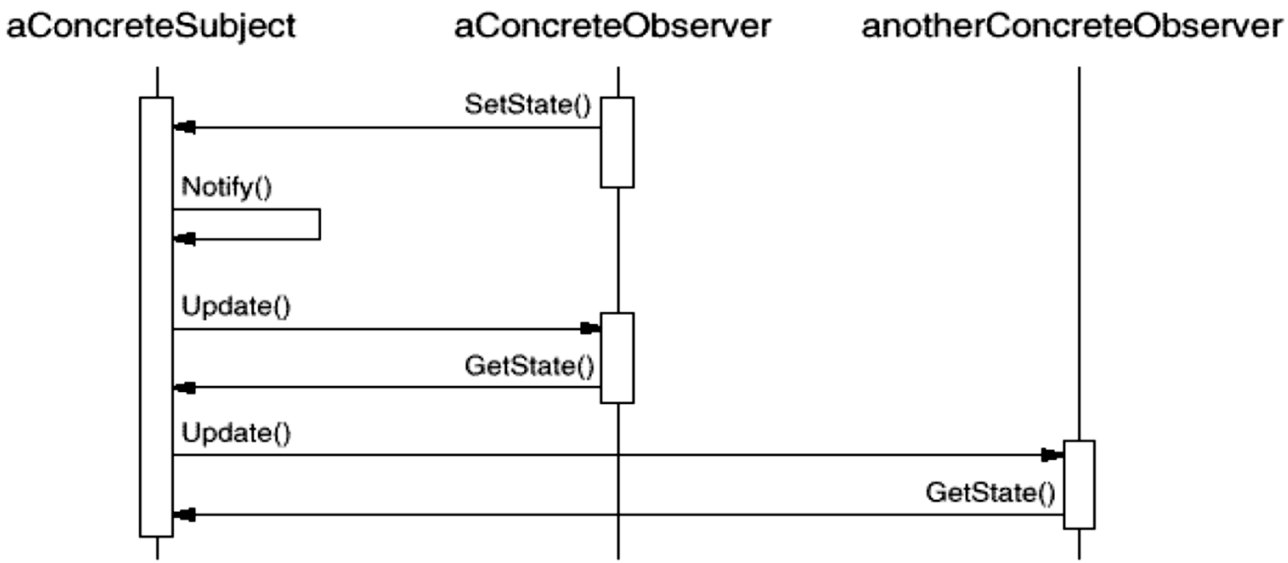
- Intent:
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Observer: Structure

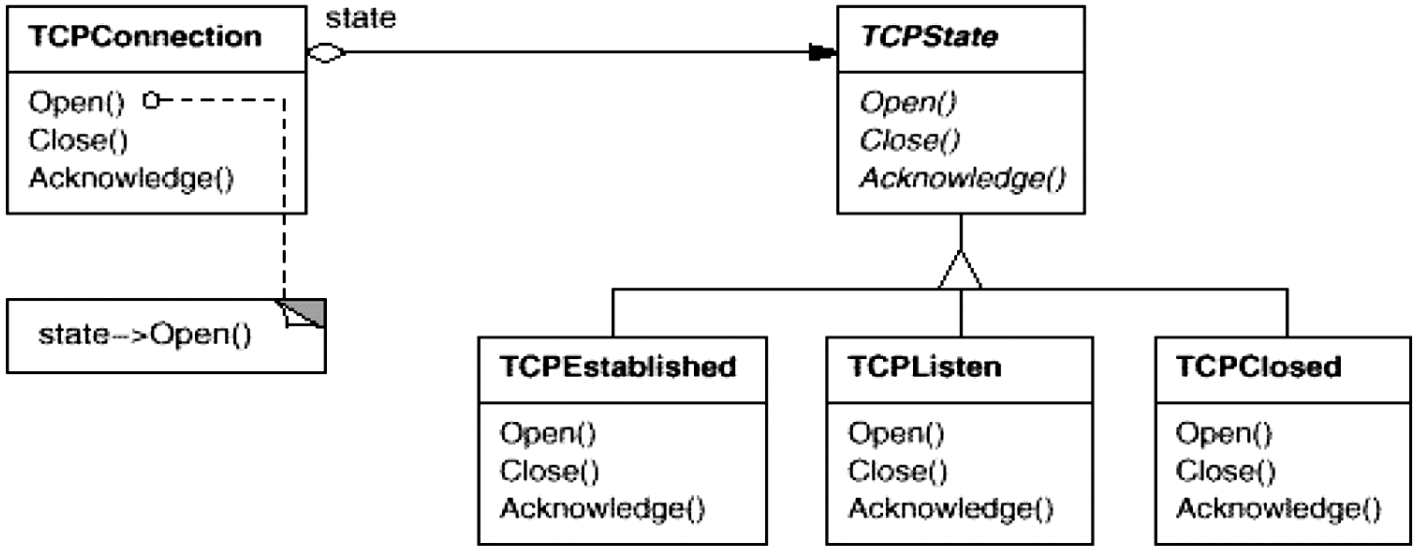


Observer: Collaboration

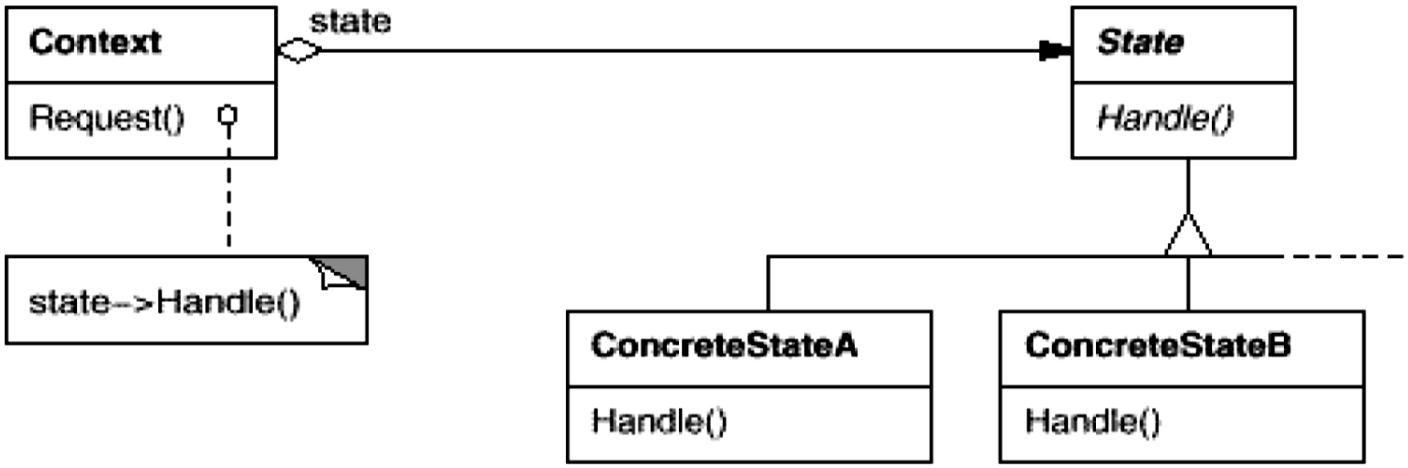


State

- Intent:
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

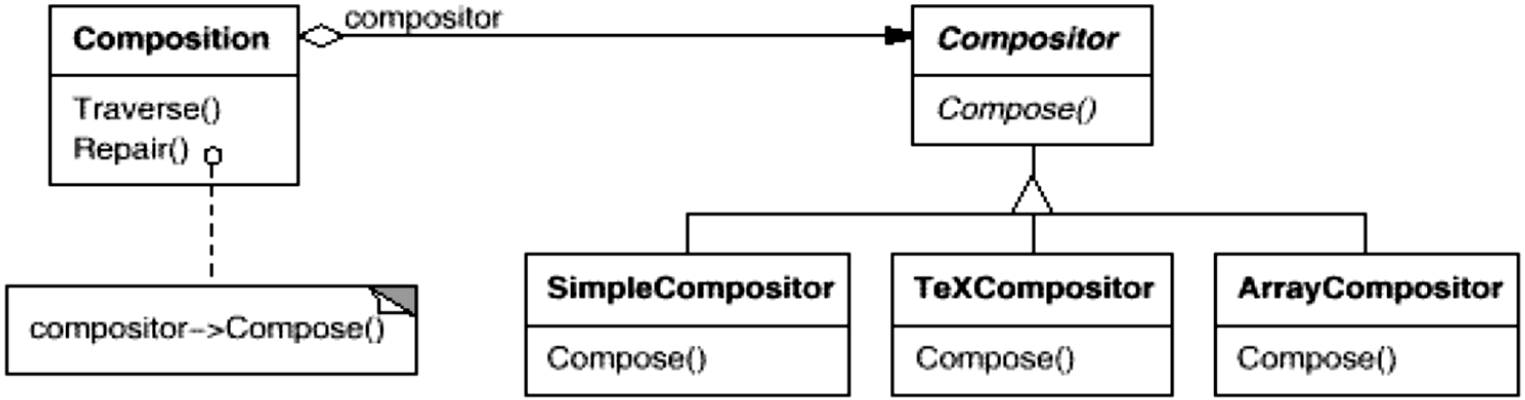


State: Structure

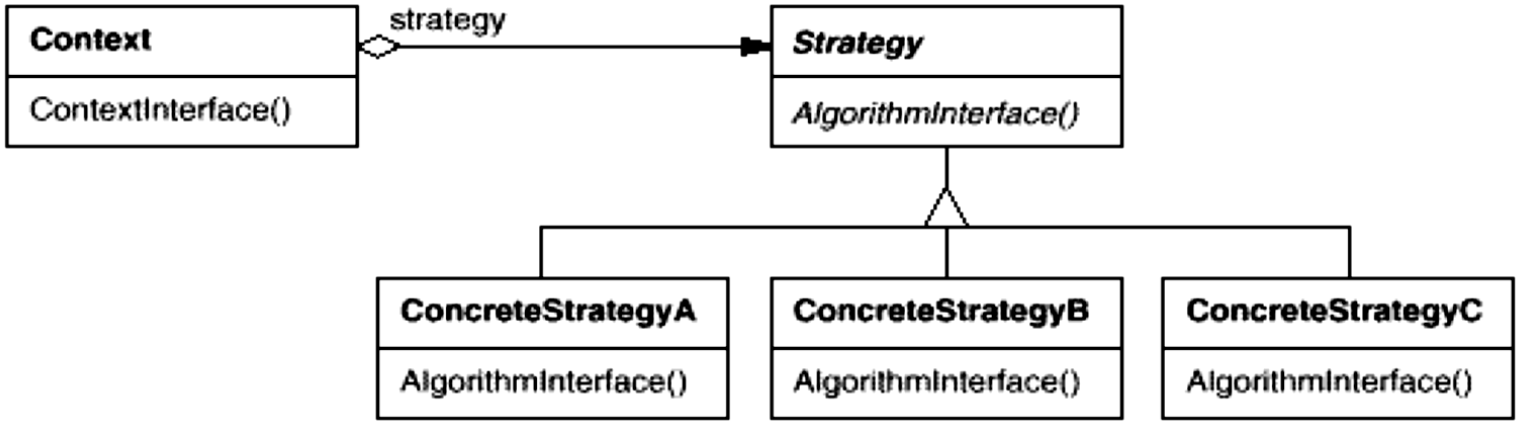


Strategy

- Intent:
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



Strategy: Structure



GoV Patterns for Software Architecture

- According to Buschmann et al.:
  - A *pattern for software architecture* describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution.
  - The solution scheme is specified by describing
    - the constituent components
    - The responsibilities and relationships of the components
    - the ways in which the components collaborate.

GoV Patterns: Categories

- **Architectural**
  - Expresses a fundamental structural organization schema for software systems.
    - Provides a set of predefined subsystems (or components), specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- **Design**
  - Provides a scheme for refining the subsystems or components of a software system, or the relationships between them.
    - Describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.
- **Idiom**
  - Low-level pattern specific to a programming language.
    - Describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Architectural Patterns: Categories

- **From Mud to Structure**
  - Support a controlled decomposition of a system task into cooperating subtasks.
  - *Layers*, *Pipes and Filters*, and *Blackboard*
- **Distributed Systems**
  - Deal with the infrastructure of distributed applications.
  - *Broker*, also *Microkernel* and *Pipes and Filters*, which only consider distribution as a secondary concern.
- **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: From Mud to Structure

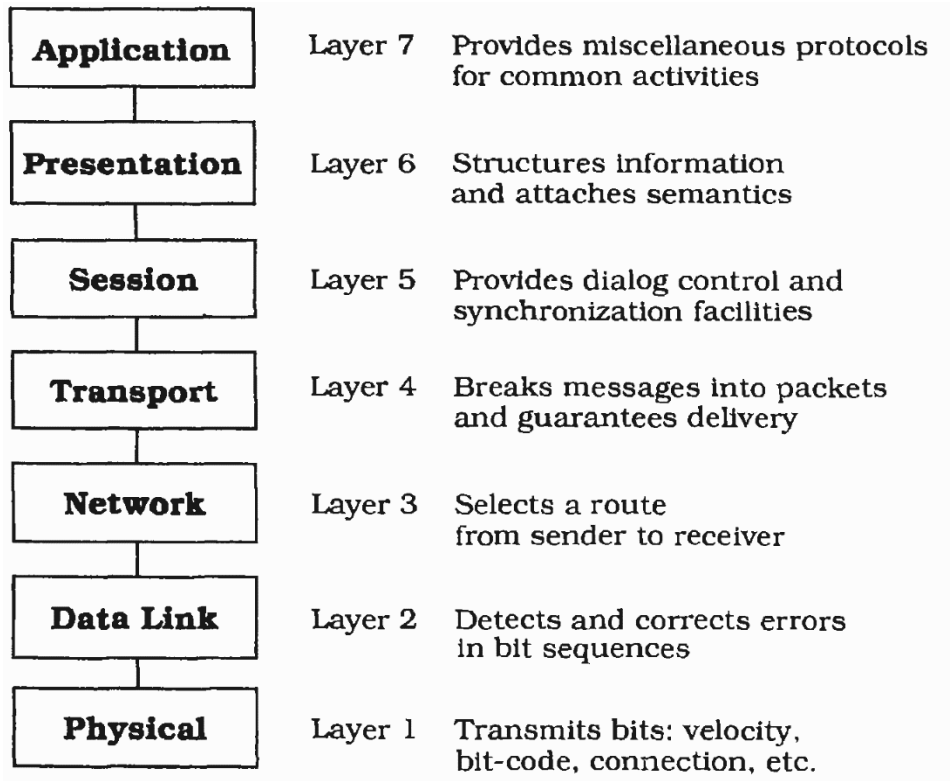
- **Layers:** Helps to structure applications that can be decomposed into groups of subtasks.
  - Each group of subtasks is at a particular level of abstraction.

- **Pipes and Filters:** Provides a structure for systems that process a stream of data.
  - Each processing step is encapsulated in a filter component.
  - Data is passed through pipes between adjacent filters.
  - Recombining filters allows you to build families of related systems.

- **Blackboard:** Useful for problems for which no deterministic solution strategies are known.
  - Several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

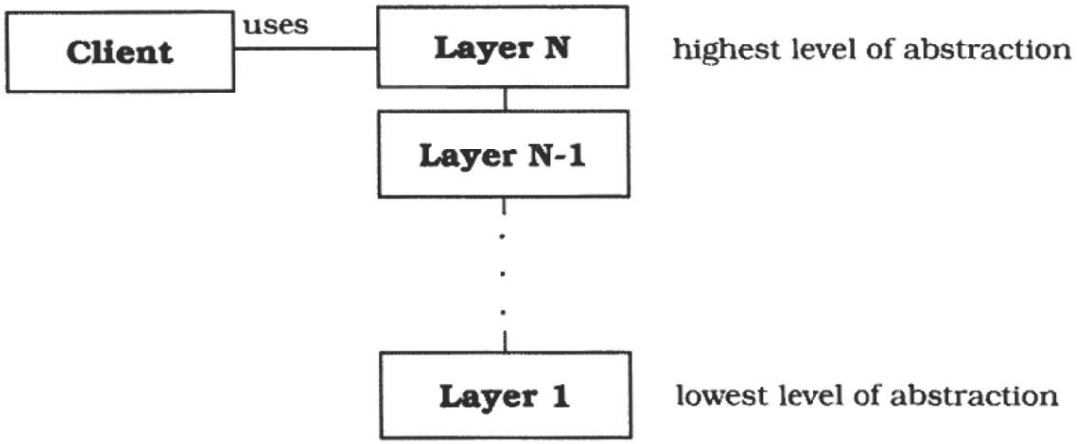
From Mud to Structure: Layers

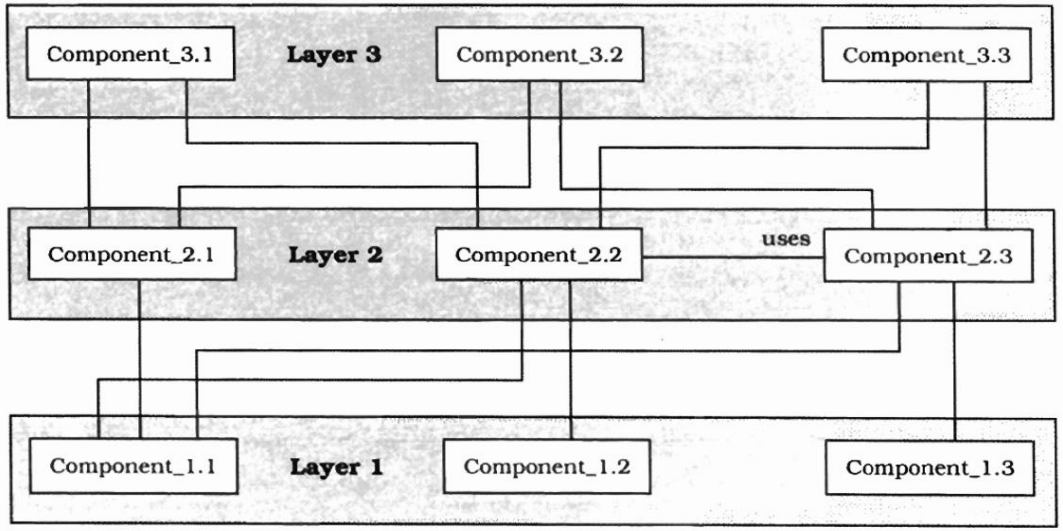
- Helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.



Layers: Structure

Class Layer J	***Collaborator***
Responsibility • Provides services used by Layer J+1. • Delegates subtasks to Layer J-1.	• Layer J-1





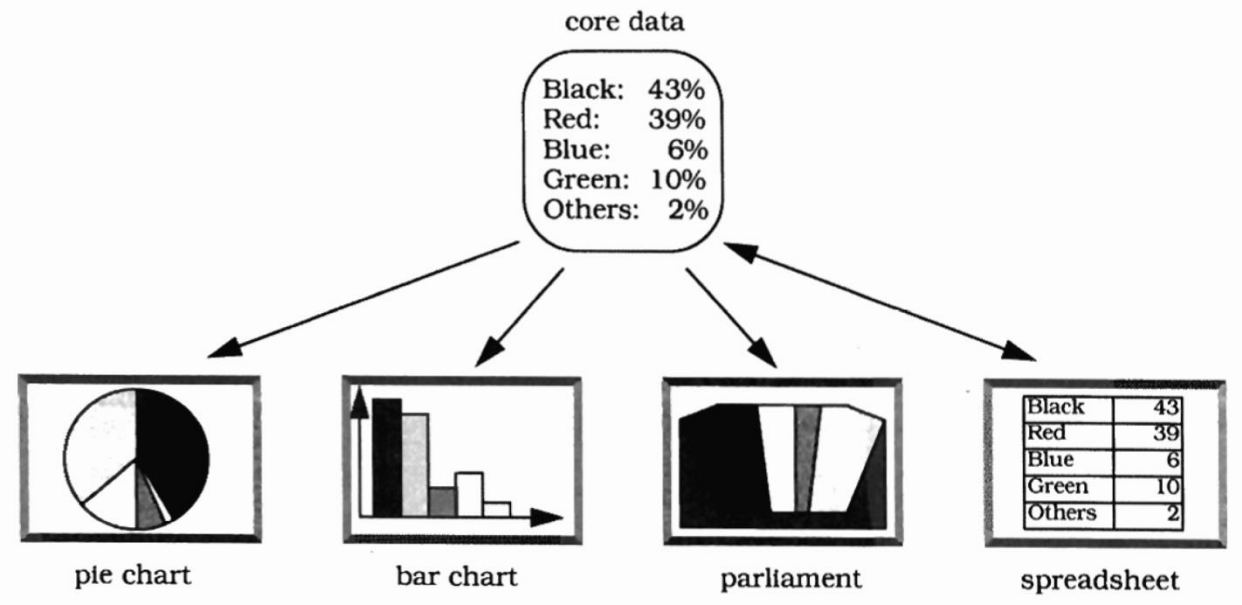
Architectural: Interactive Systems

- **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.



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

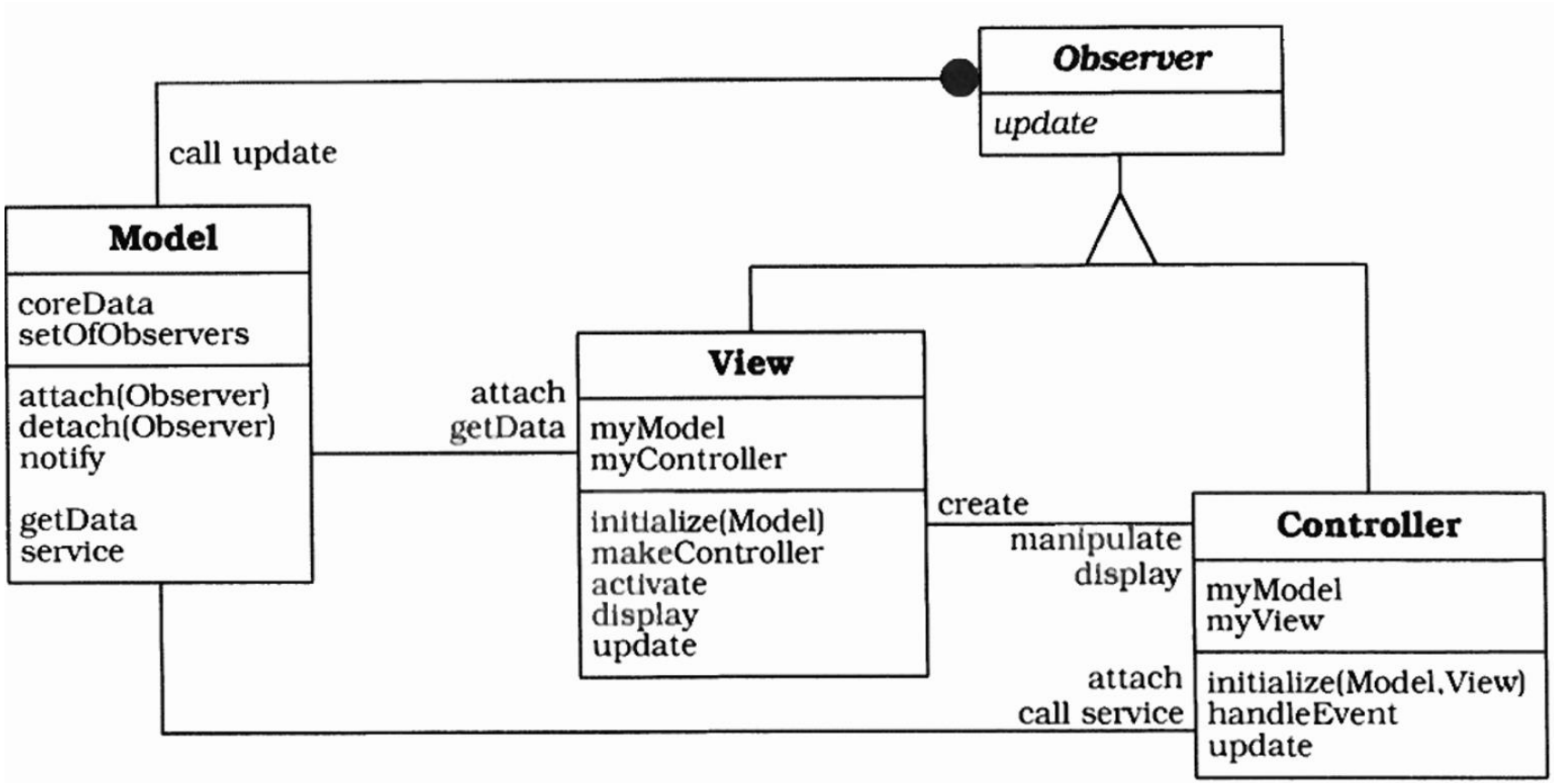
Model-View-Controller: Structure – Views and Controllers

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

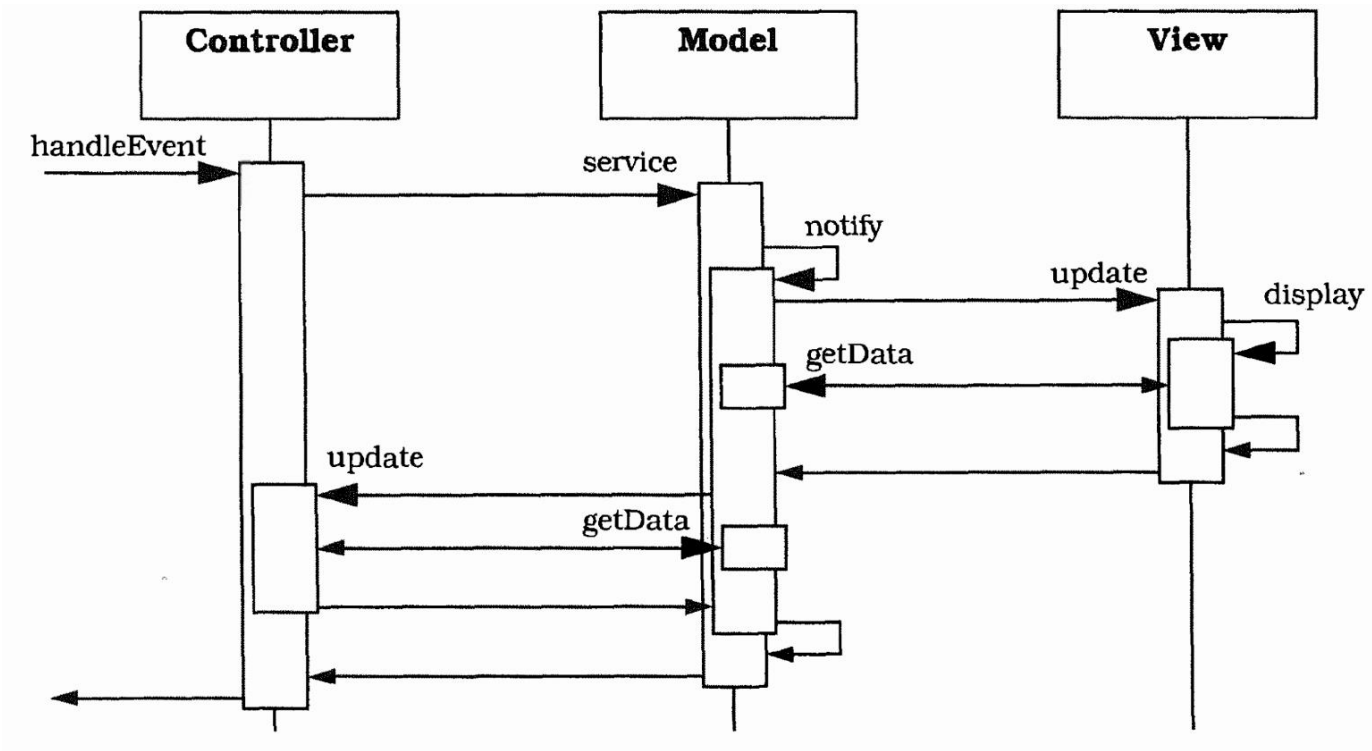
Class Controller	***Collaborators***
Responsibility	• View • Model
• Accepts user input as events. • Translates events to service requests for the model or display requests for the view. • Implements the update procedure, if required.	

Model-View-Controller: Class Structure



Model-View-Controller: Dynamics

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



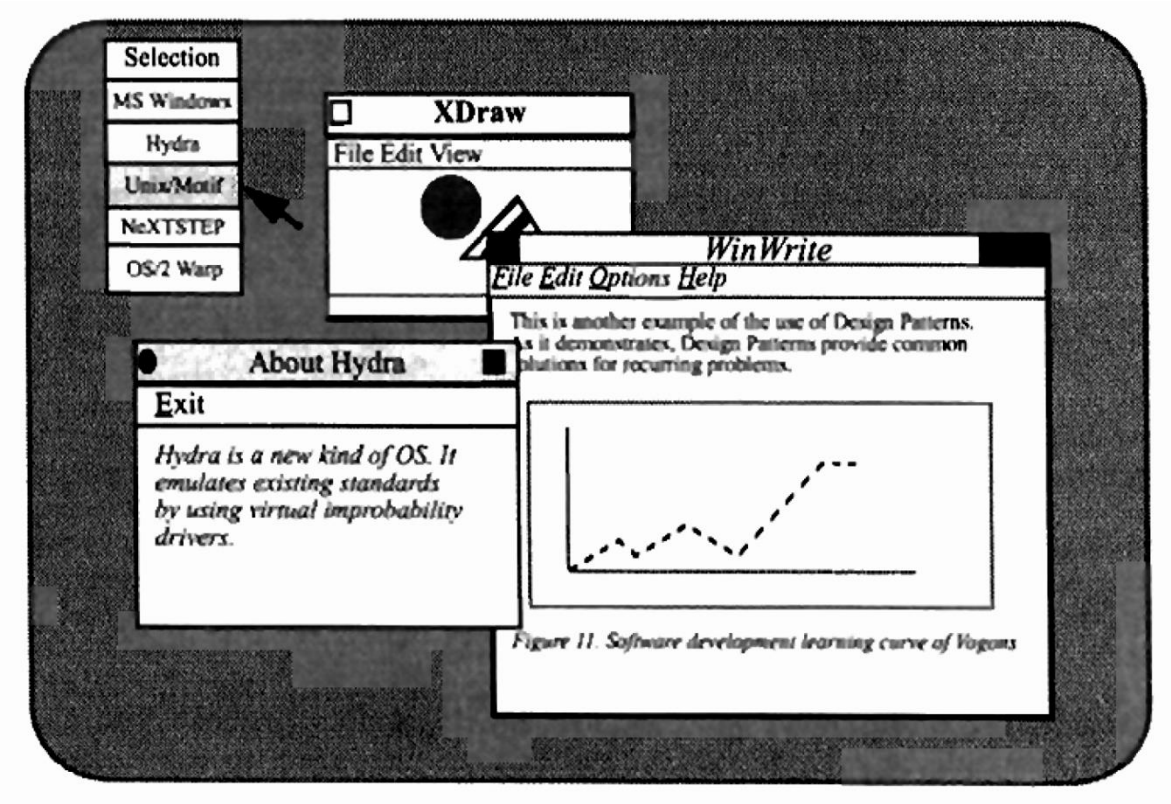
Architectural: Adaptable Systems

- **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.



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 communication facilities. • Encapsulates system 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

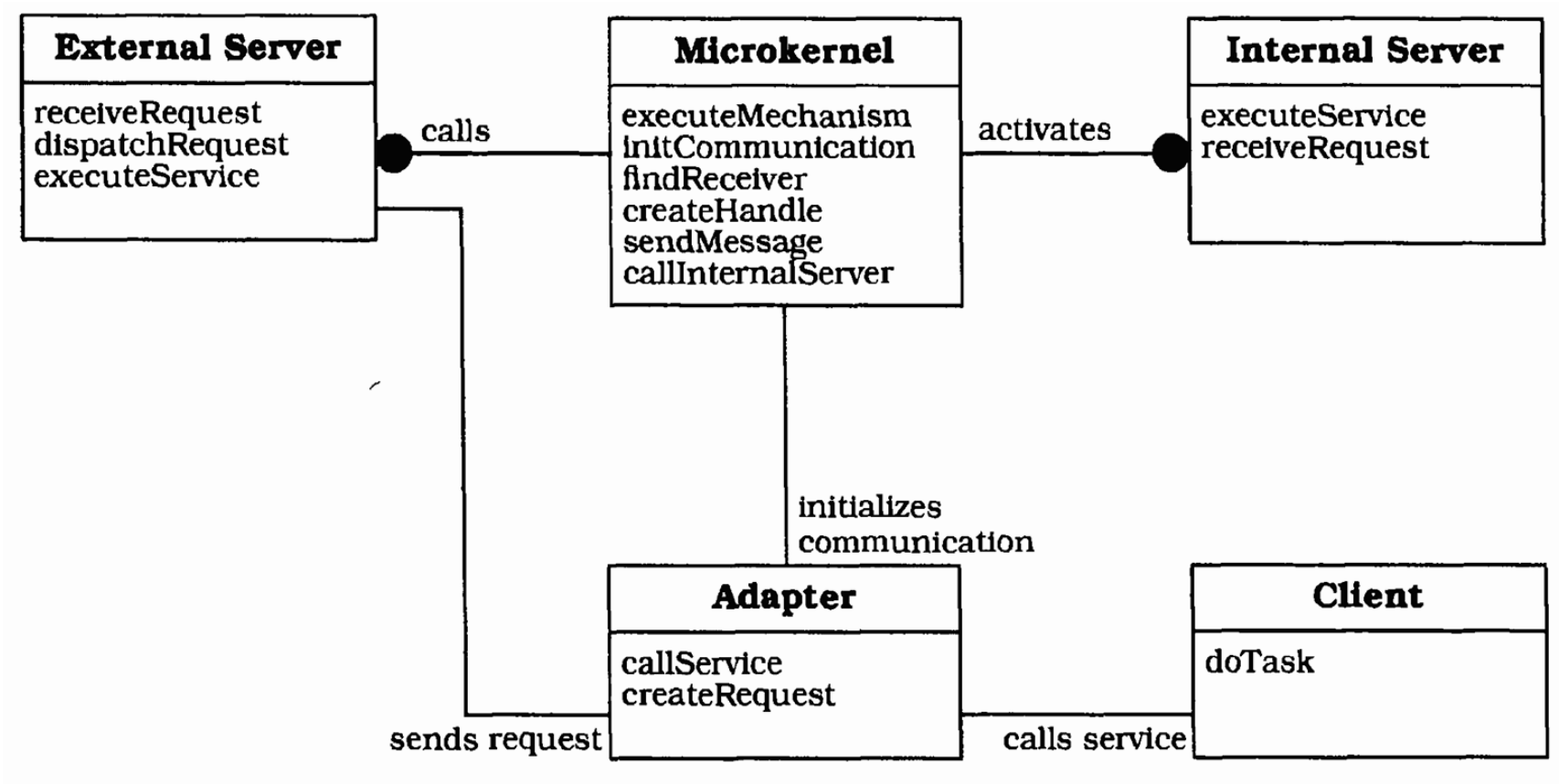
Microkernel: Structure – Clients and Adapters

- *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** • Adapter
Responsibility • Represents an application.	

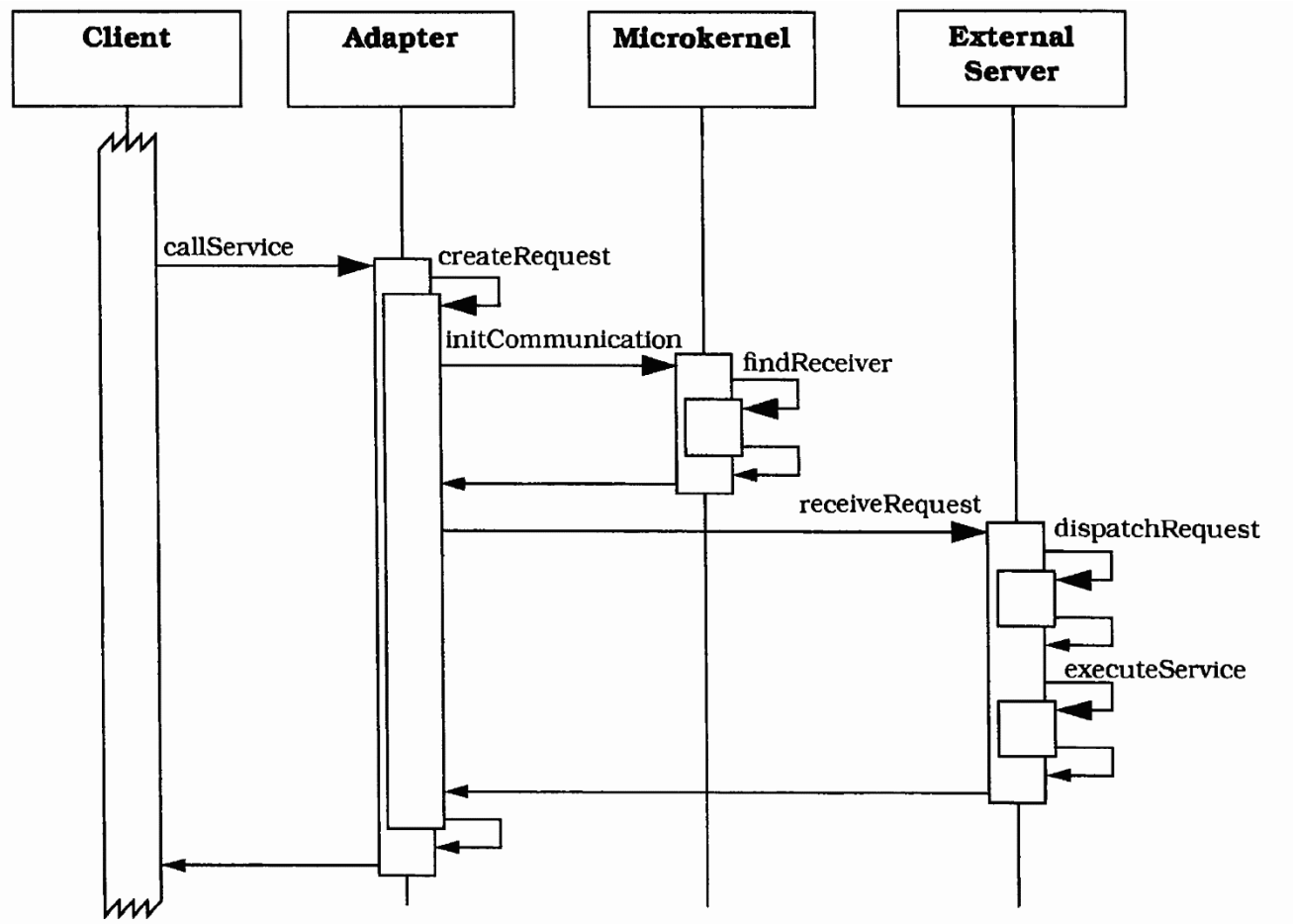
Class Adapter	**Collaborators** • External Server • Microkernel
Responsibility • Hides system dependencies such as communication facilities from the client. • Invokes methods of external servers on behalf of clients.	

Microkernel: Structure – Class Diagram



Microkernel: Dynamics

- A client calls a service of its external server



References

- Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley, 1995.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., *Pattern-Oriented Software Architecture: A System of Patterns,* Vol. 1. Wiley, 1996.
- Freeman, E., and Robson, E., *Head First Design Patterns.* O'Reilly Media, 2020.