Patterns in Software Engineering

Lecturer: Raman Ramsin

Lecture 4

GoF Design Patterns – Behavioral

Part 1

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

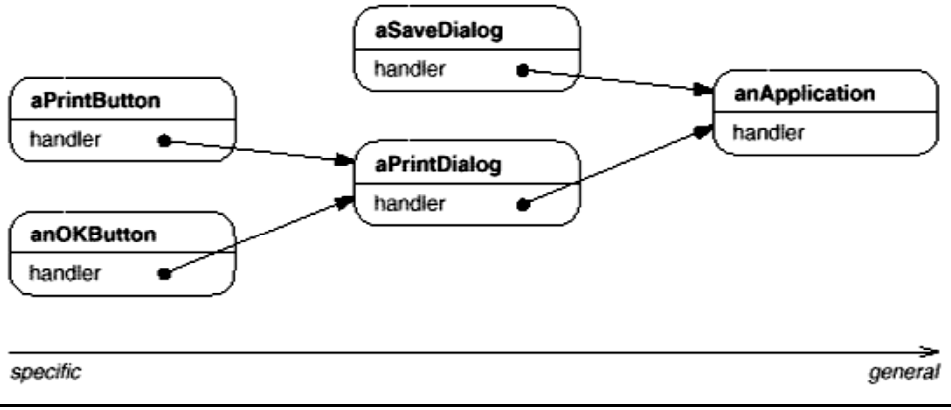
- **Object**
  - ***Chain of Responsibility:*** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
  - ***Command:*** Encapsulate a request as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

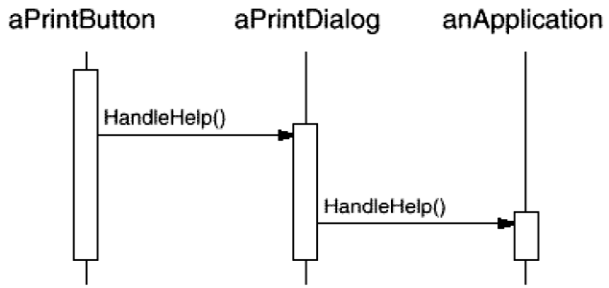
GoF Behavioral Patterns – Object (Contd.)

- **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.

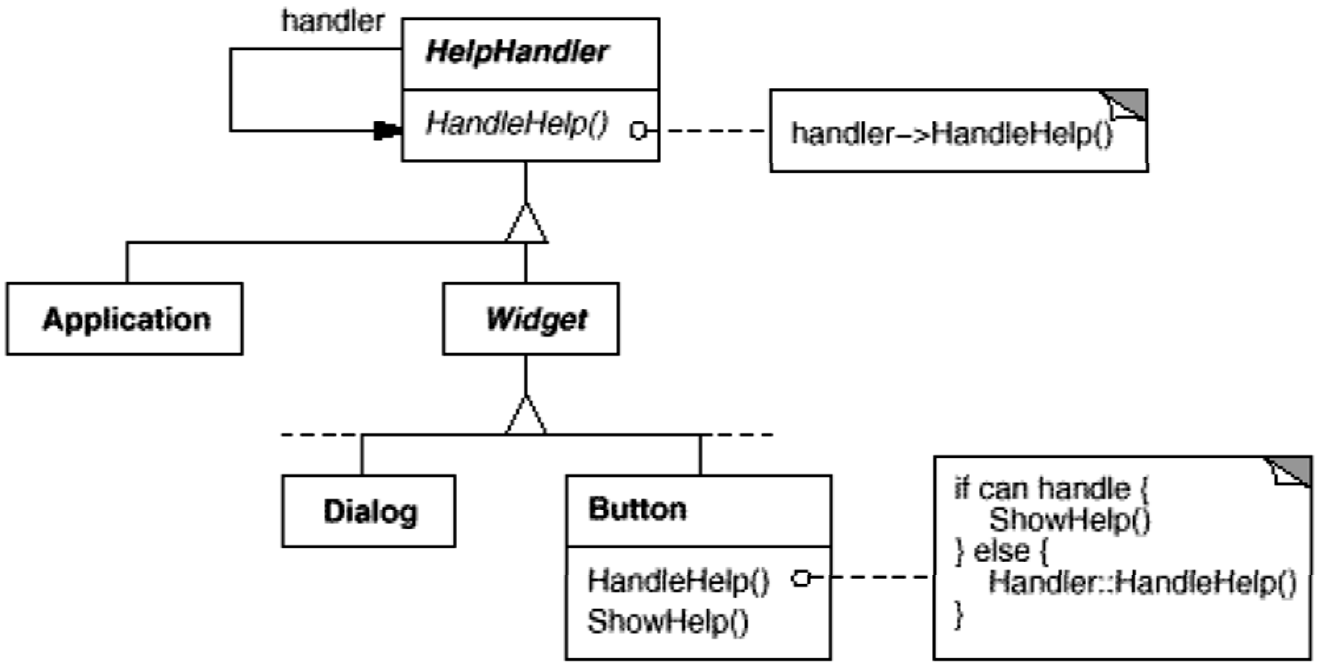
Chain of Responsibility

- 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.





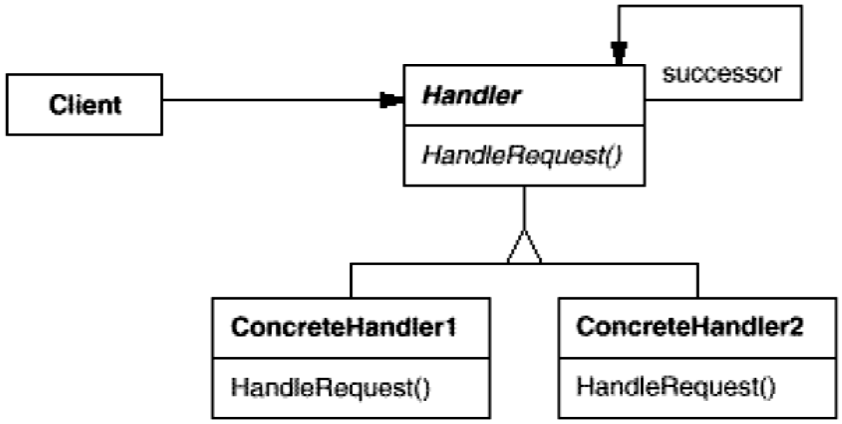
Chain of Responsibility: Class Hierarchy

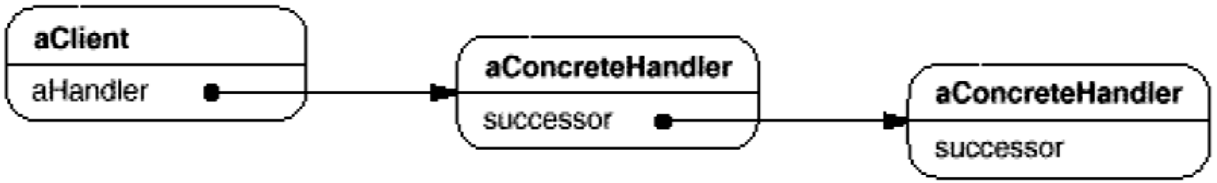


Chain of Responsibility: Applicability

- Use the Chain of Responsibility pattern when
  - more than one object may handle a request, and the handler isn't known *a priori*. The handler should be ascertained automatically.
  - you want to issue a request to one of several objects without specifying the receiver explicitly.
  - the set of objects that can handle a request should be specified dynamically.

Chain of Responsibility: Structure





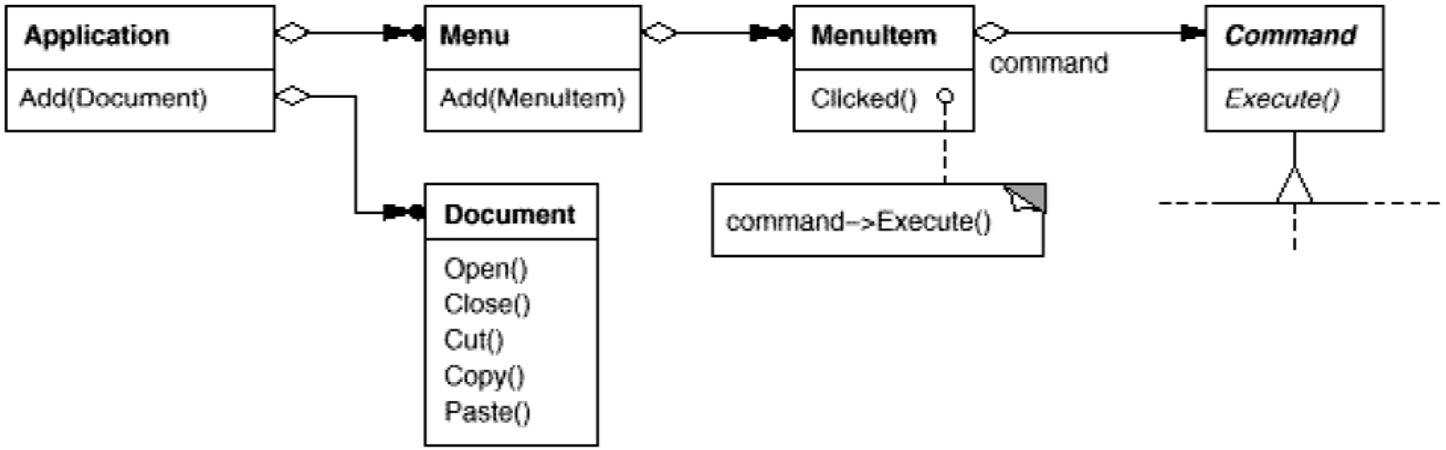
Chain of Responsibility: Consequences

- ✓ *Reduced coupling.*
- ✓ *Added flexibility in assigning responsibilities to objects.*

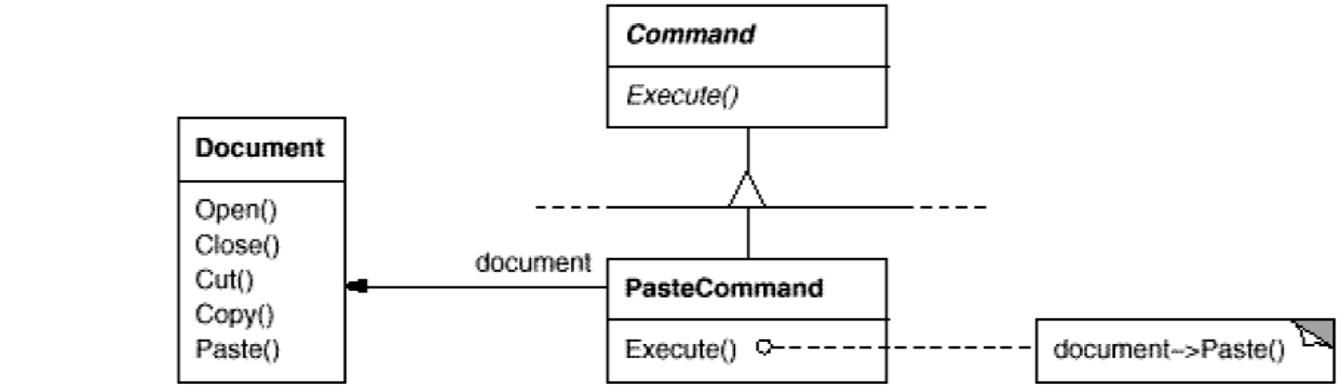
- ✗ *Receipt isn't guaranteed.*

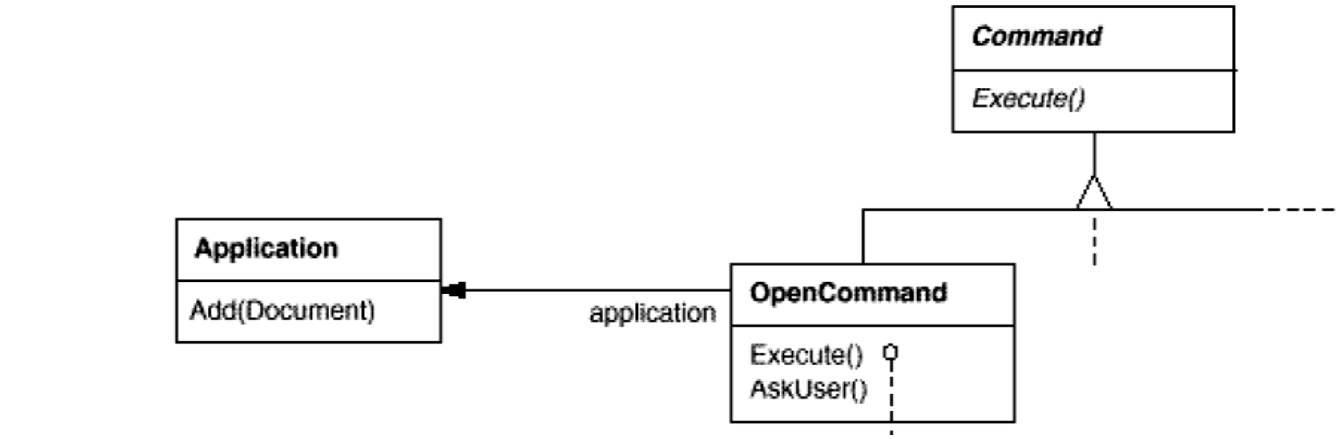
Command

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

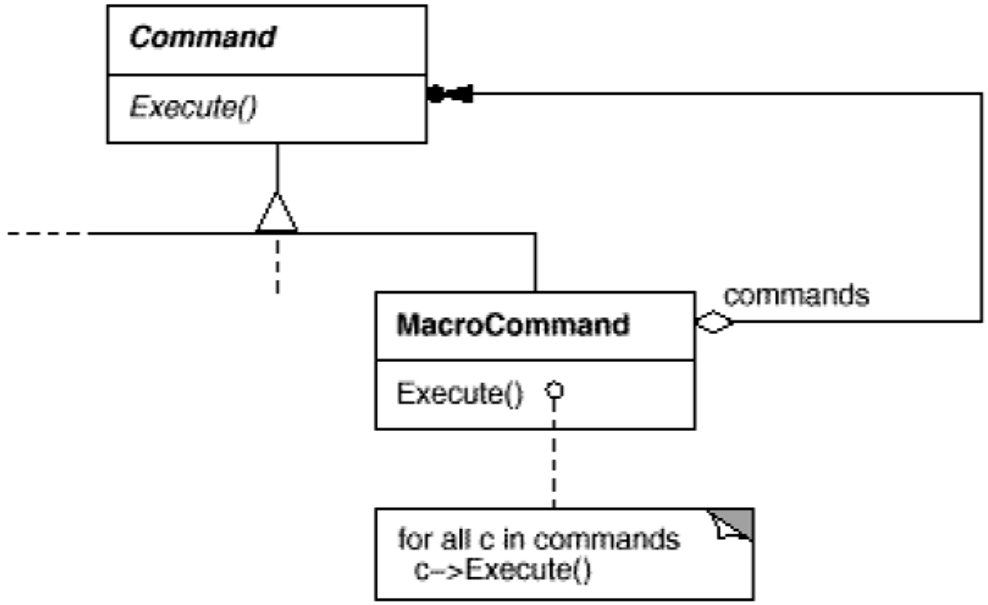


Command: Examples





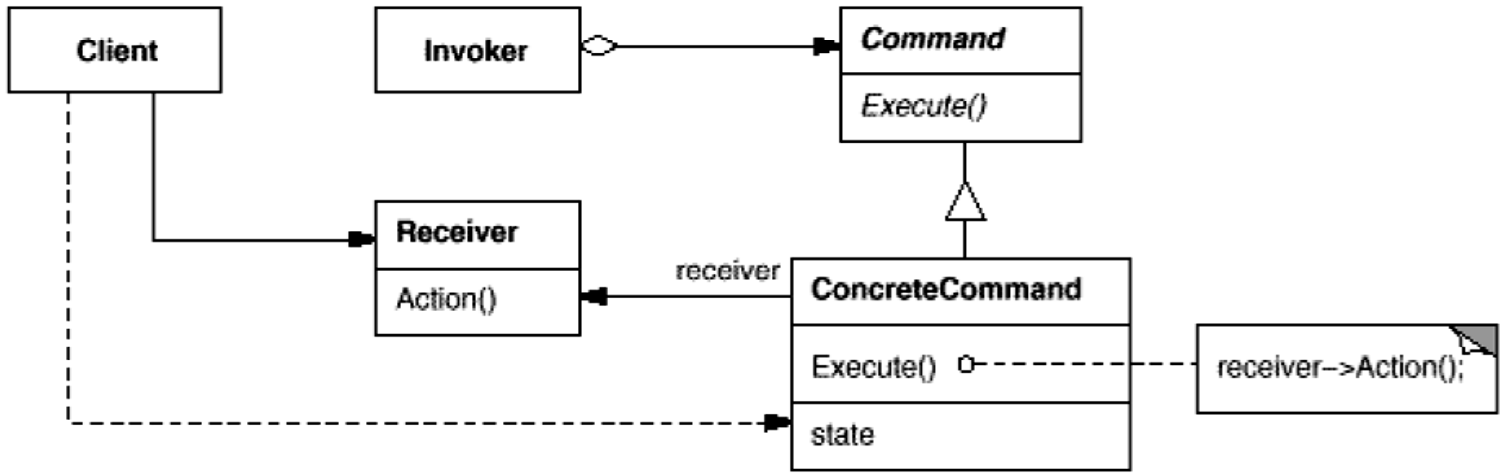
Command: Macro-Command

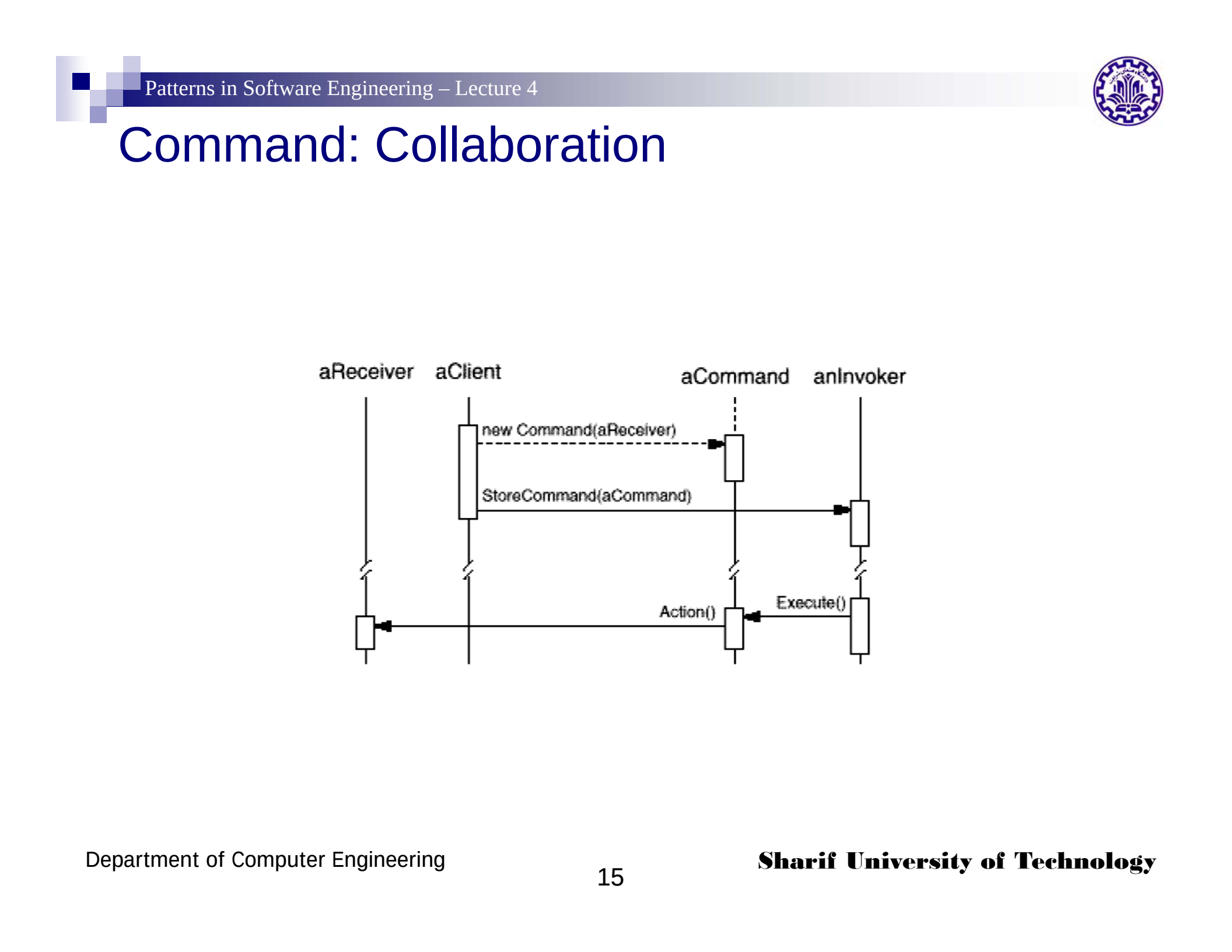


Command: Applicability

- Use the Command pattern when you want to
  - parameterize objects by an action to perform.
  - specify, queue, and execute requests at different times.
  - support undo. The Command's Execute operation can store state for reversing its effects in the command itself. The Command interface must have an added Unexecute operation that reverses the effects of a previous Execute.
  - support logging changes so that they can be reapplied in case of a system crash (by augmenting the Command interface with load and store operations).
  - structure a system around high-level operations built on primitives operations (based on the commands' common interface).

Command: Structure





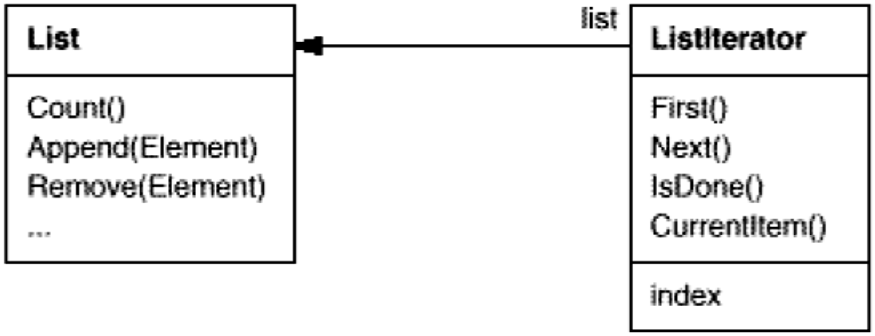
Command: Collaboration

Command: Consequences

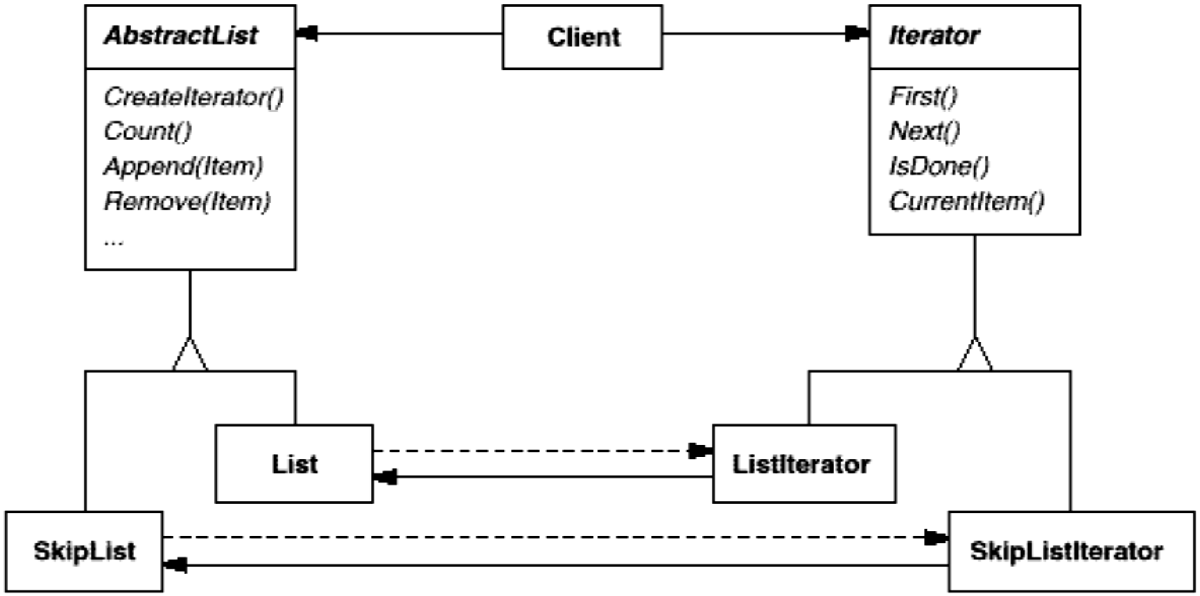
- ✓ *Command decouples the object that invokes the operation from the one that knows how to perform it.*
- ✓ *Commands are first-class objects.* They can be manipulated and extended like any other object.
- ✓ *You can assemble commands into a composite command.*
- ✓ *It's easy to add new Commands, because you don't have to change existing classes.*

Iterator

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



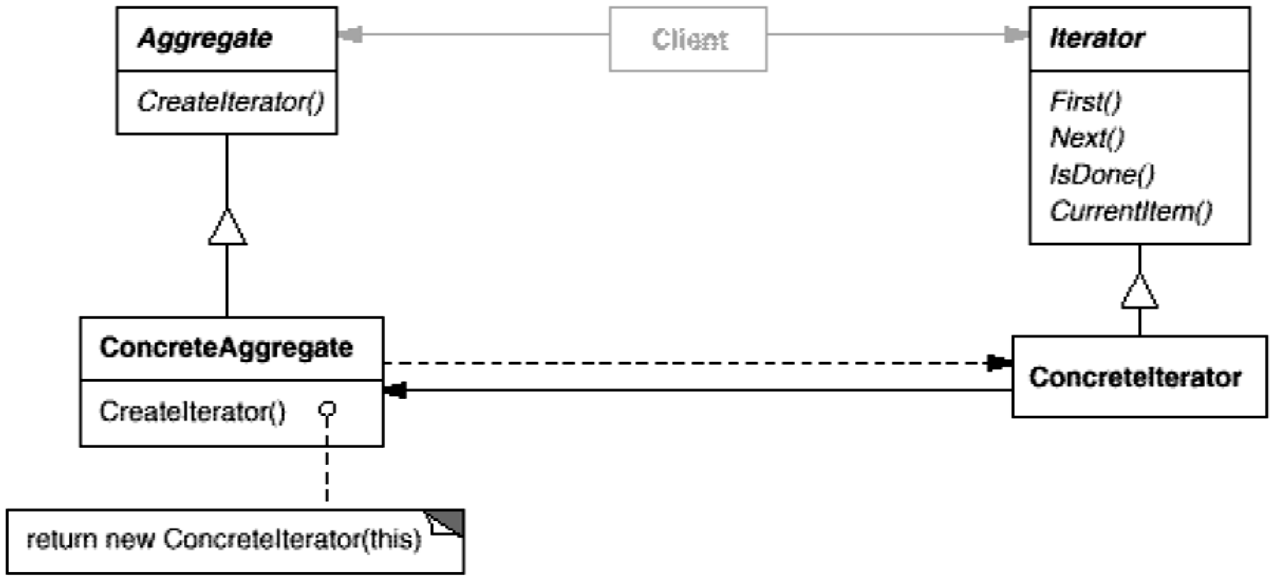
Iterator: Example



Iterator: Applicability

- Use the Iterator pattern
  - to access an aggregate object's contents without exposing its internal representation.
  - to support multiple traversals of aggregate objects.
  - to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Iterator: Structure

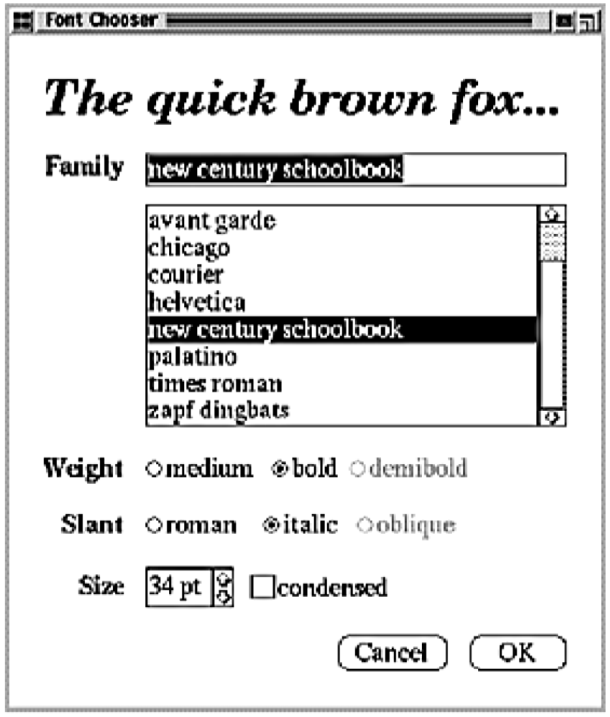


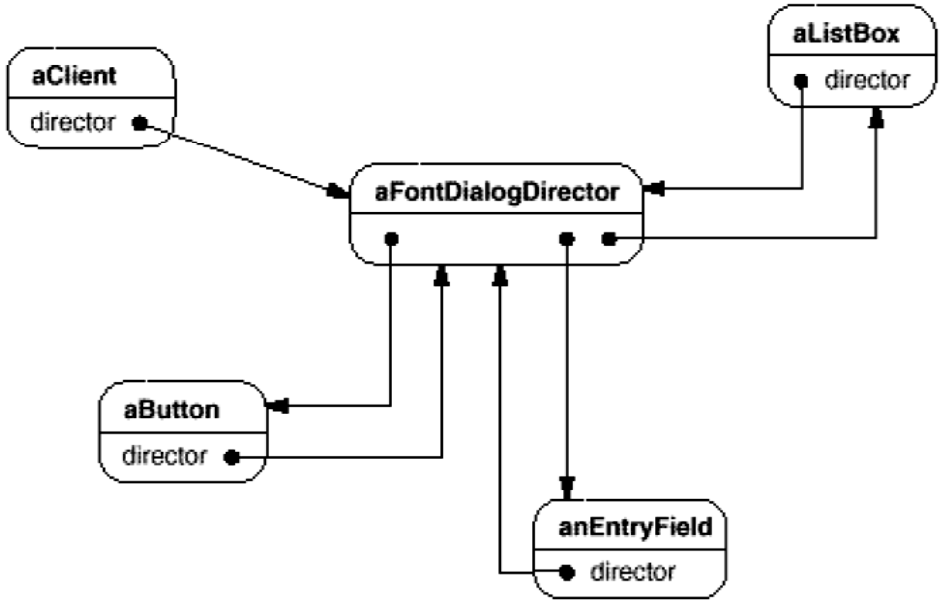
Iterator: Consequences

- ✓ *It supports variations in the traversal of an aggregate.*
- ✓ *Iterators simplify the Aggregate interface.* Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying its interface.
- ✓ *More than one traversal can be pending on an aggregate.* You can have more than one traversal in progress at once.

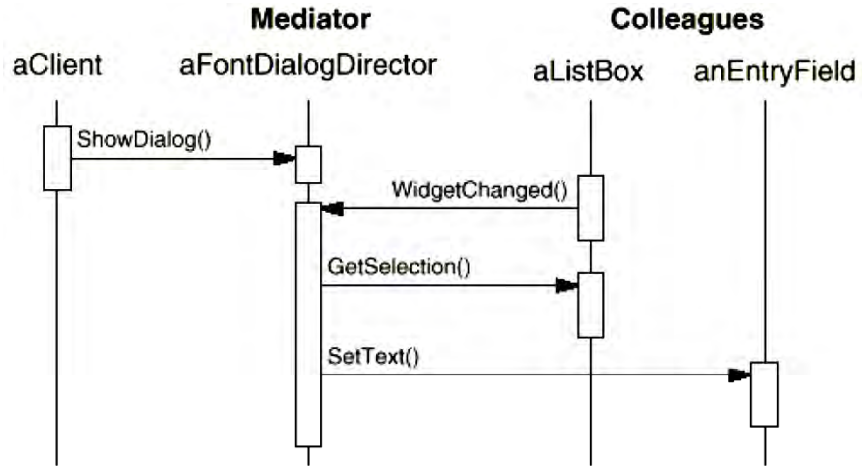
Mediator

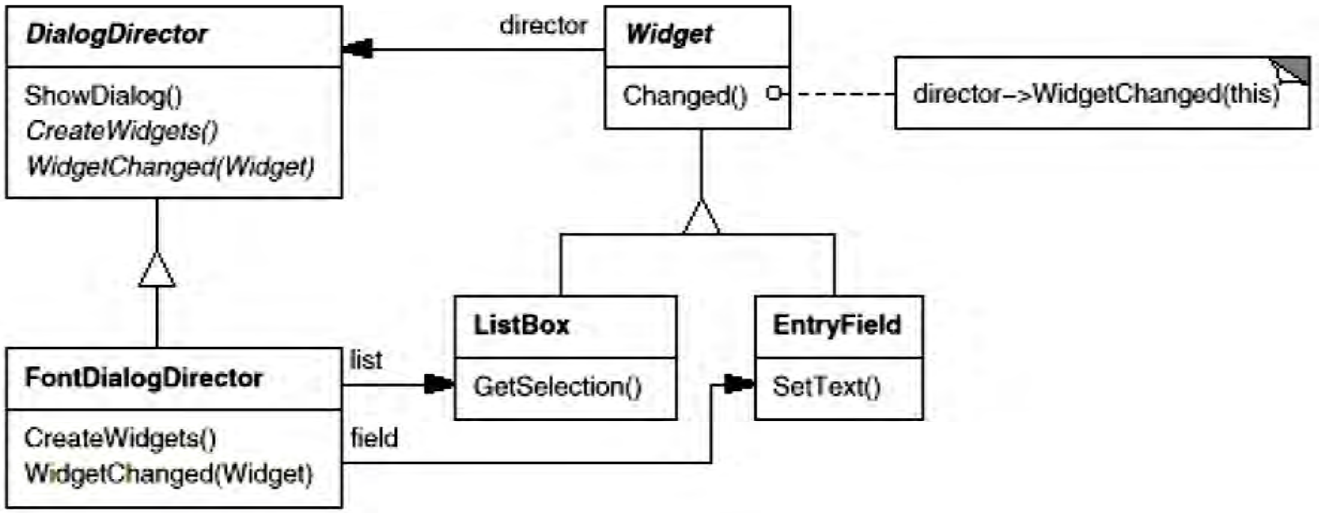
- Intent:
  - Define an object that encapsulates how a set of objects interact: promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.





Mediator: Typical Collaboration and Class Hierarchy

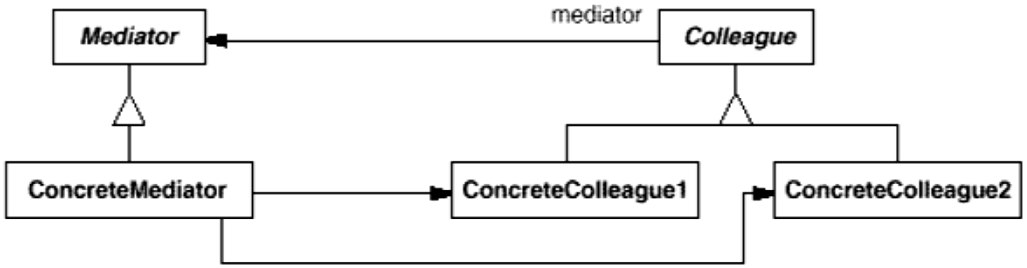


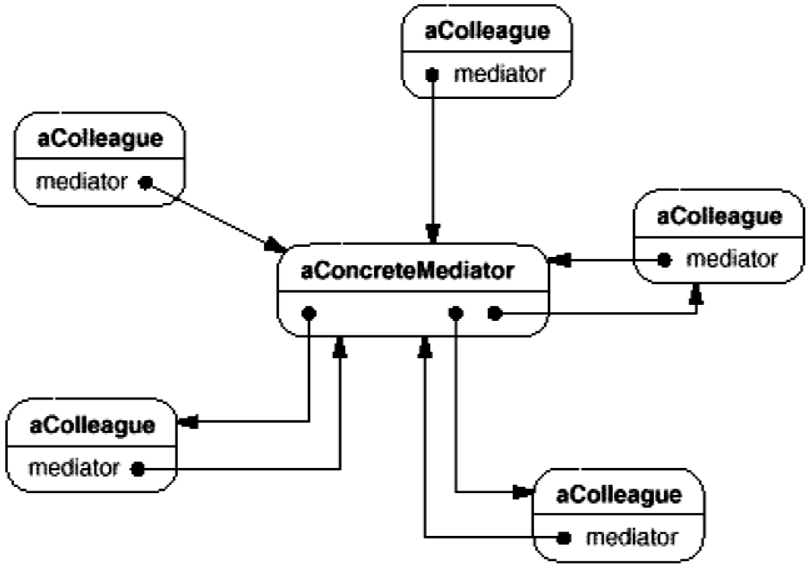


Mediator: Applicability

- Use the Mediator pattern when
  - a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
  - reusing an object is difficult because it refers to and communicates with many other objects.
  - a behavior that's distributed between several classes should be customizable without a lot of subclassing.

Mediator: Structure





Mediator: Consequences

- ✓ *It limits subclassing.* A mediator localizes behavior that otherwise would be distributed among several objects. Changing this behavior requires subclassing Mediator only.
- ✓ *It decouples colleagues.* A mediator promotes loose coupling between colleagues.
- ✓ *It simplifies object protocols.*
- ✓ *It abstracts how objects cooperate.* Making mediation an independent concept and encapsulating it in an object lets you focus on how objects interact apart from their individual behavior.

- ✗ *It centralizes control.* The Mediator pattern trades complexity of interaction for complexity in the mediator.

Reference

- Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley, 1995.