Behavioral Patterns

Jim Yu

IBM
China Development Lab
Greater China Group

Why Behavioral Patterns

- Implement program behaviors in an object-oriented and flexible way
- Assign responsibility among classes or objects
- Encapsulate program behaviors that might change
  - e.g. algorithms, state-dependent behaviors, object communications, object traversal
- Reduce coupling in the program
- decouple request sender and receiver

Iterator

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

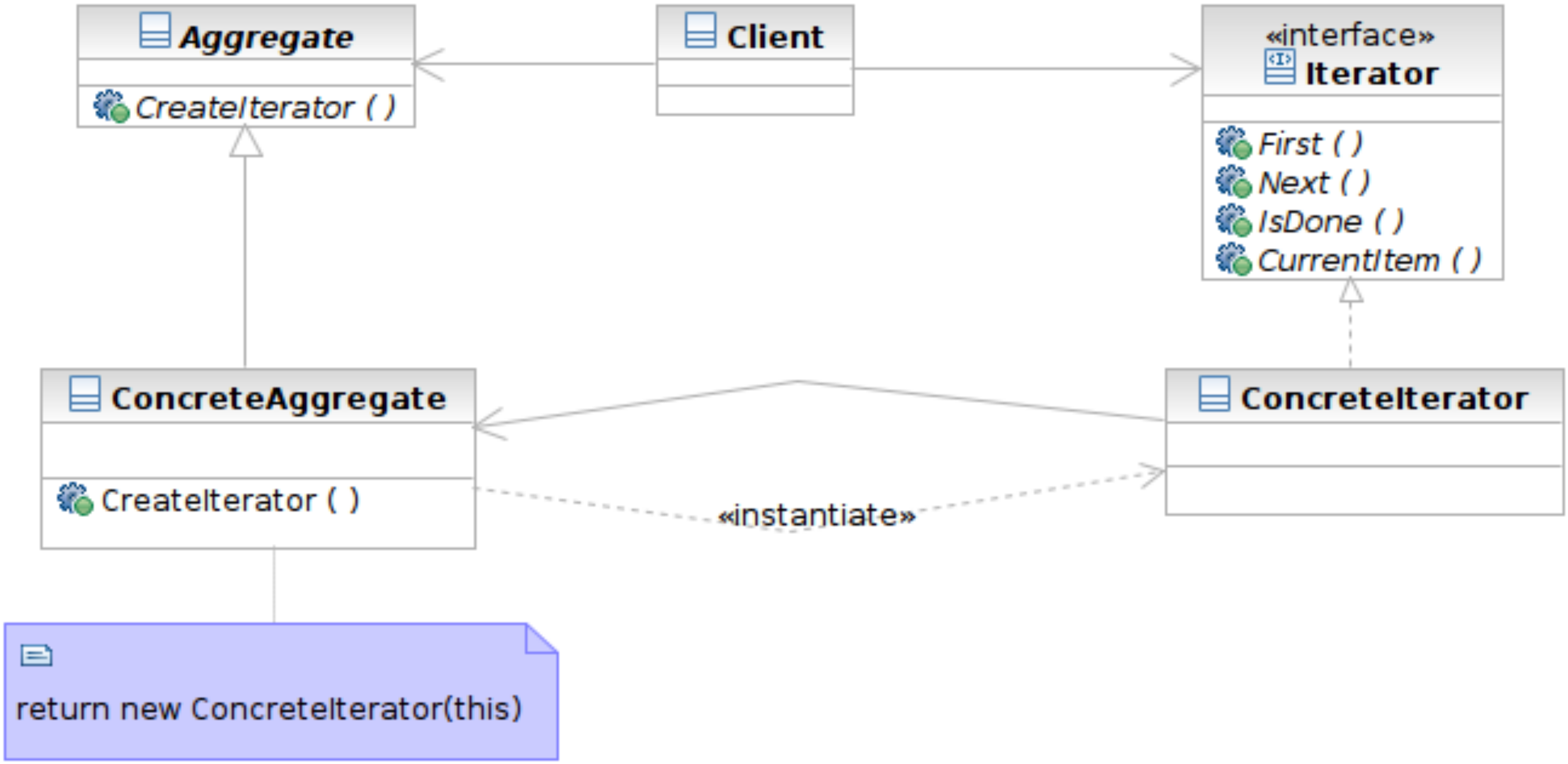
Challenge

- Show your belongings
  - Iterate over the items in you have and display them
- Save the progress
  - Iterate over the player's object graph and save them
- First attempt:
  - Traverse the linked list via each node's next pointer
  - Depth-first traverse the player's object graph

Iterator

- Problem: we often want to iterate over a collection of objects. How can we do this in a flexible way?
- Think: what's the effort if you replace your LinkedList with an ArrayList? Or even a BinarySearchTree? Can you provide multiple traversal methods?
- Target: given an aggregate (collection) class, we want to traverse its elements without knowing how it's implemented.

Structure


Aggregate
CreateIterator ( )
Client
«interface»
Iterator
First ( )
Next ( )
IsDone ( )
CurrentItem ( )
ConcreteAggregate
CreateIterator ( )
«instantiate»
ConcreteIterator
return new ConcreteIterator(this)

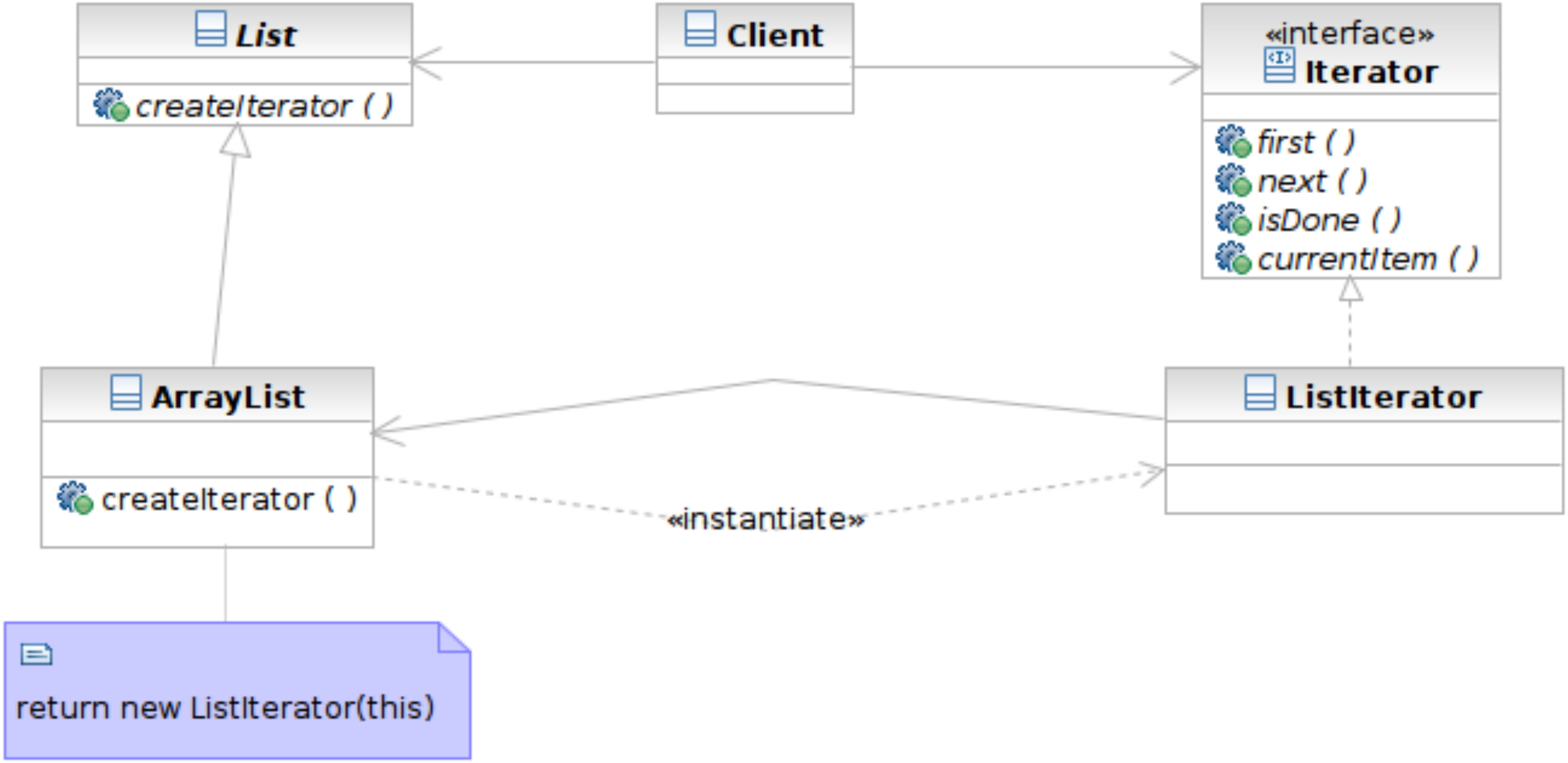
Participants

- Class **Iterator** defines an interface for accessing and traversing elements
- Class **ConcreteIterator** implements the Iterator interface; keeps track of the current position of traversal
- Class **Aggregate** defines an interface for creating an Iterator object
- Class **ConcreteAggregate** implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Applicability

- Use the Iterator pattern
  - to access the elements of an aggregate object
  - to support multiple traversals of aggregate objects
    - forwards, backwards, depth-first, etc.
  - to provide a uniform interface for traversing different Aggregate structures
    - linked lists, array, tree, graph, etc.

Sample Structure



Samples

- List and Iterator:
  - class List and Iterator
- Concrete List and Iterator
  - class ArrayList and ListIterator
- Using Iterator
  - Method PrintUsers.testPrintUsers()
  - Reverse Iterator: method ReverseIterate.testReverseIterator()

Consequences

- It supports variations in the traversal of an aggregate: replace the iterator and the traversal algorithm is changed
- Iterators simplify the Aggregate interface: Iterator methods are not implemented in each concrete Aggregate (you may also reuse concrete Iterators)
- Support for more than one traversal of the Aggregate: just add Iterator factory methods

Related Patterns

- **Composite**: use iterator to traverse the composite object structure
- **Factory Method**: creates the concrete iterator

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.

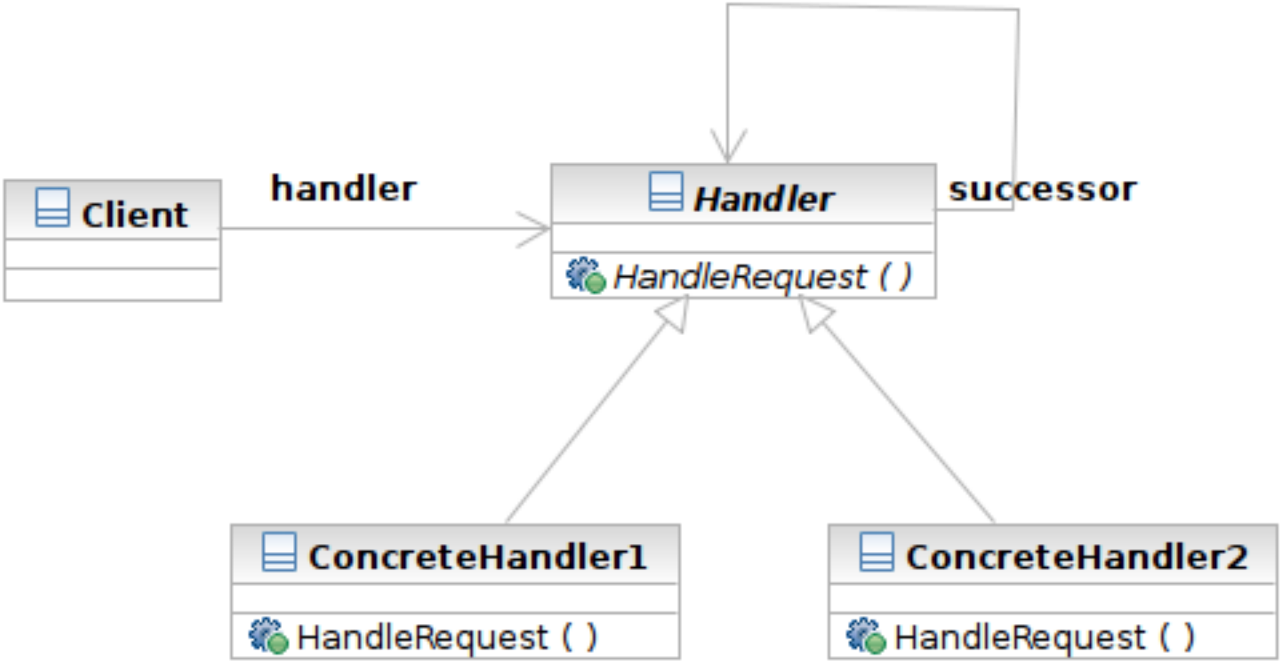
Challenge

- You are implementing the user input handler of the GUI widgets
    - The widgets have parent-children relationships
    - If the object can be selected, then the object takes the focus and performs the action
    - If the object cannot be selected, then try to select the object's parent
- First attempt: code it using if … then …

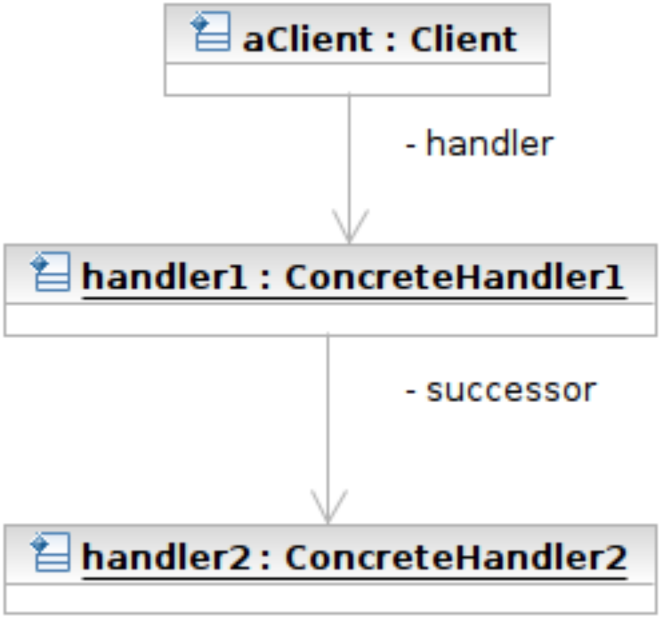
Chain of Responsibility

- Problem: how can you handle a request in a flexible way if multiple objects may take responsibility?
- Think: what is the effort if the widgets are composed differently? What if some widgets are added?
- Target: decouple the request sender and handler by chaining the possible handlers and passing the request along the chain until handled.

Structure


Client
handler
Handler
HandleRequest ( )
successor
ConcreteHandler1
HandleRequest ( )
ConcreteHandler2
HandleRequest ( )

Structure


aClient : Client
- handler
handler1 : ConcreteHandler1
- successor
handler2 : ConcreteHandler2

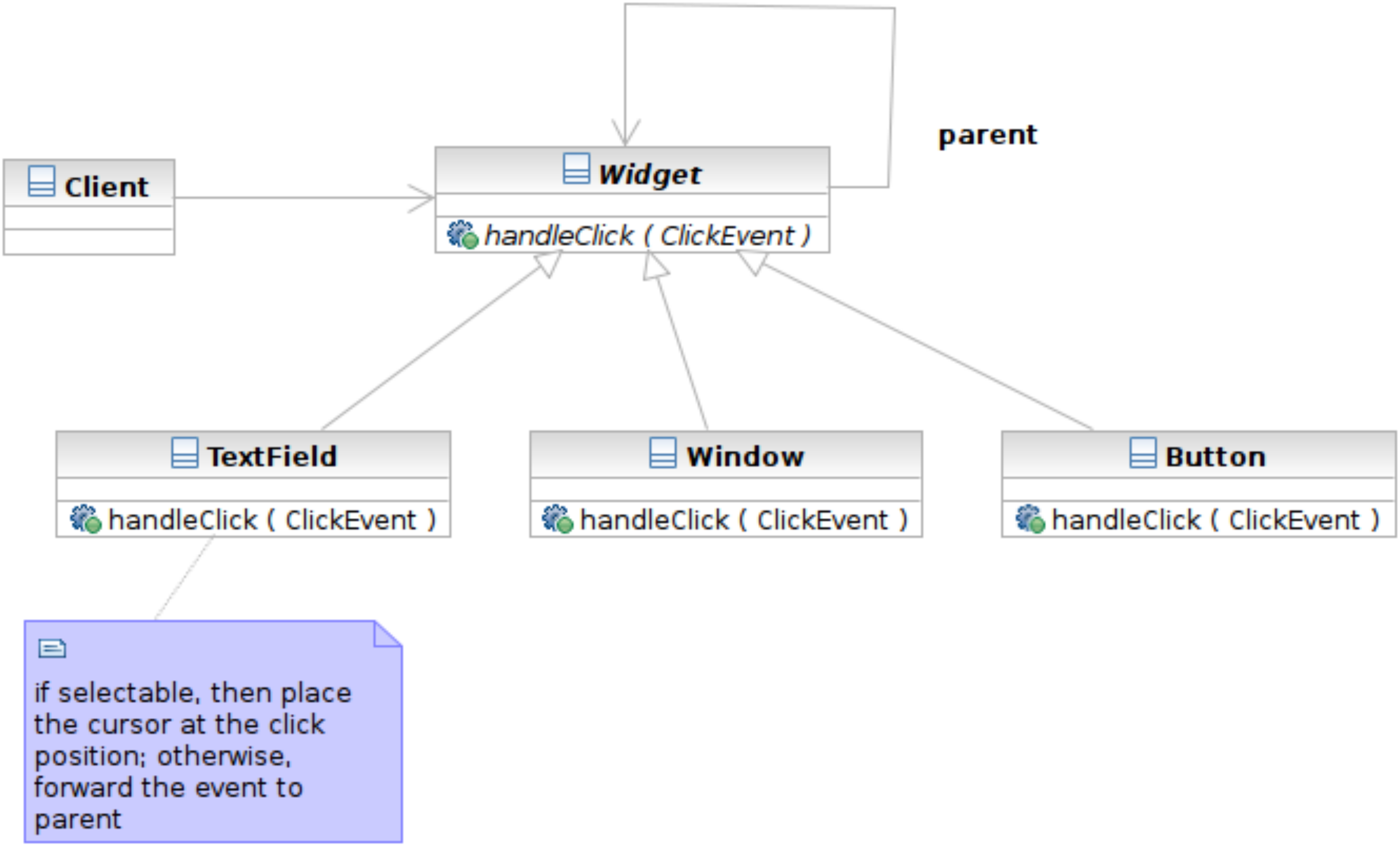
Participants

- Class Handler defines an interface for handling requests
- Class ConcreteHandler handles requests or forwards the request that it cannot handle to its successor
- Class Client initiates the requests to a ConcreteHandler object

Applicability

- Use Chain of Responsibility when
  - more than one object may handle a request, and the handler is not known a priori.
  - 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.
    - by modifying the chain

Sample Structure


parent
Client
Widget
handleClick (ClickEvent)
TextField
handleClick (ClickEvent)
Window
handleClick (ClickEvent)
Button
handleClick (ClickEvent)
if selectable, then place the cursor at the click position; otherwise, forward the event to parent

Samples

- Handler: class Widget
  - defines the request handling interface
  - holds the reference to its successor (parent in this case)
- ConcreteHandlers: class TextField, Window, Button
  - handle or forward the request
- Client

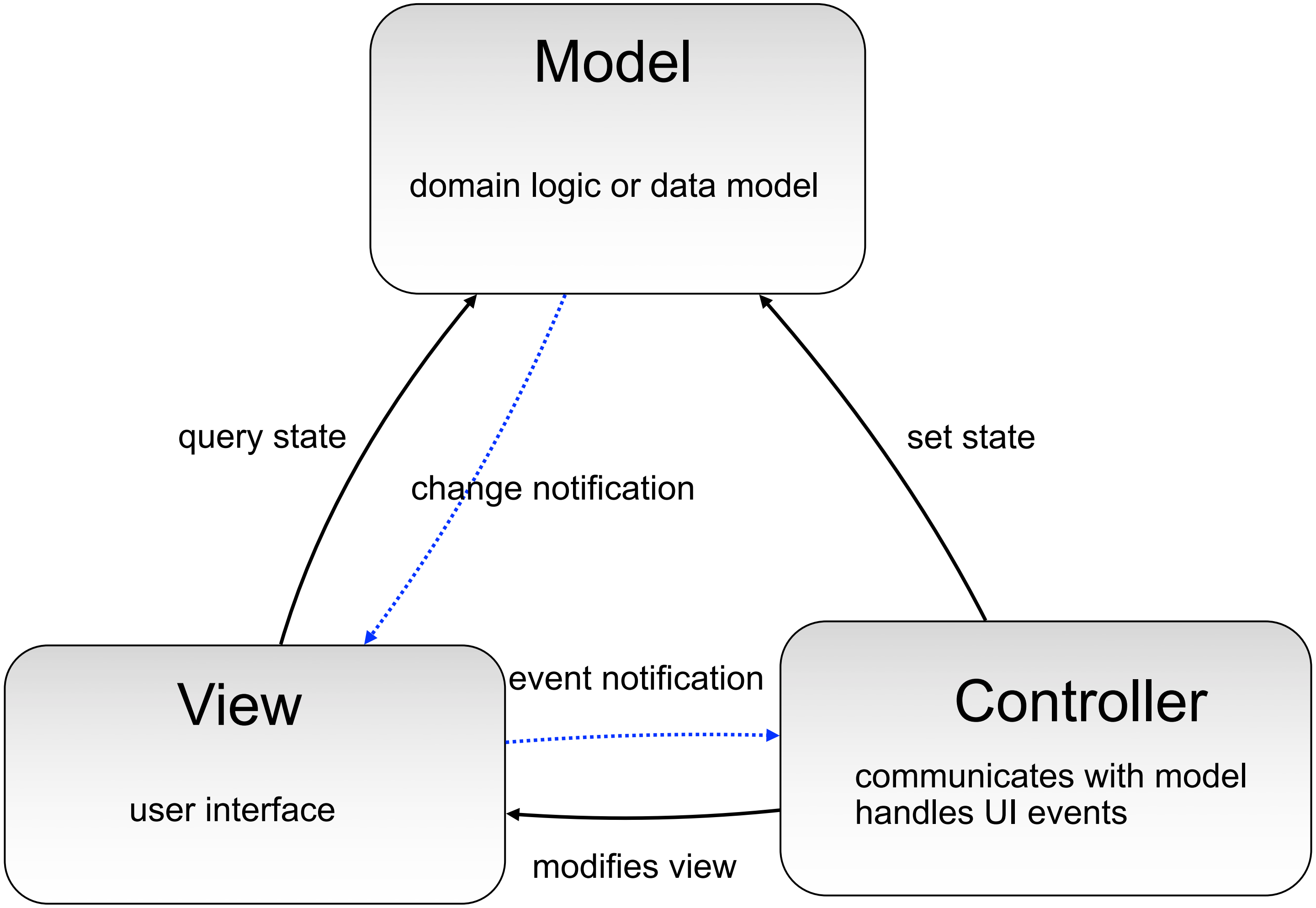
Consequences

- **It reduces coupling.** The pattern frees the client from knowing which handler will handle the request.
- **It adds flexibility in assigning responsibilities to objects.** Just modify the chain at run-time.
- **The receipt is not guaranteed.** The request can fall of the end of the chain without ever being handled.

Related Patterns

- **Composite:** parent node acts as the successor

Model–View–Controller (MVC)



Observer

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

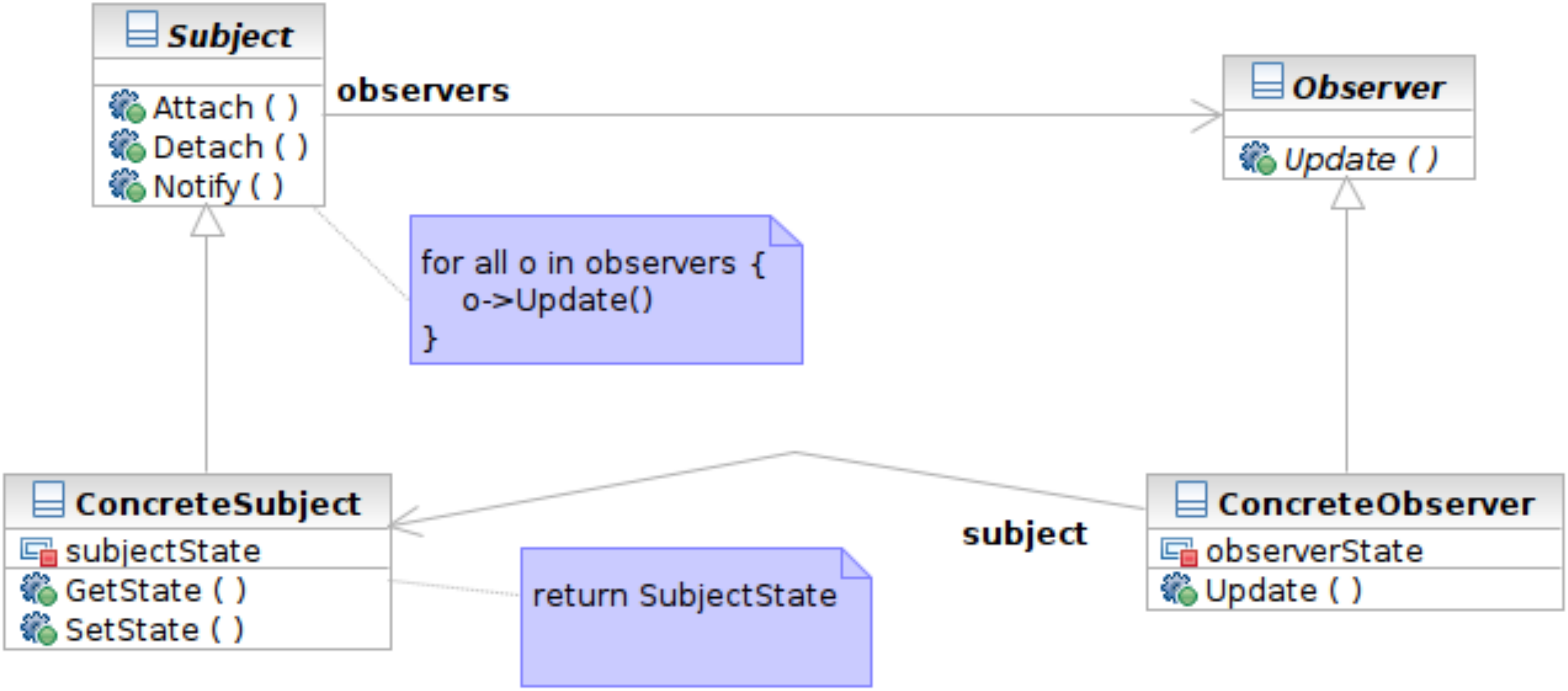
Challenge

- The user interface should listen to events and react to some events
  - Some player sends a message to you
  - You are ambushed by monsters
- First attempt: poll each events in a big event loop
  - Polling wastes CPU cycles when there is no events
  - Spaghetti code of the event loop

Observer

- Problem: we want to listen to events that we are interested in. How can we do this in a flexible way?
- Think: what is the effort if you want to add event types or listeners? Is your implementation extensible and efficient?
- Target: define a relationship between objects so that one (observer) can be notified if another (subject) updates.

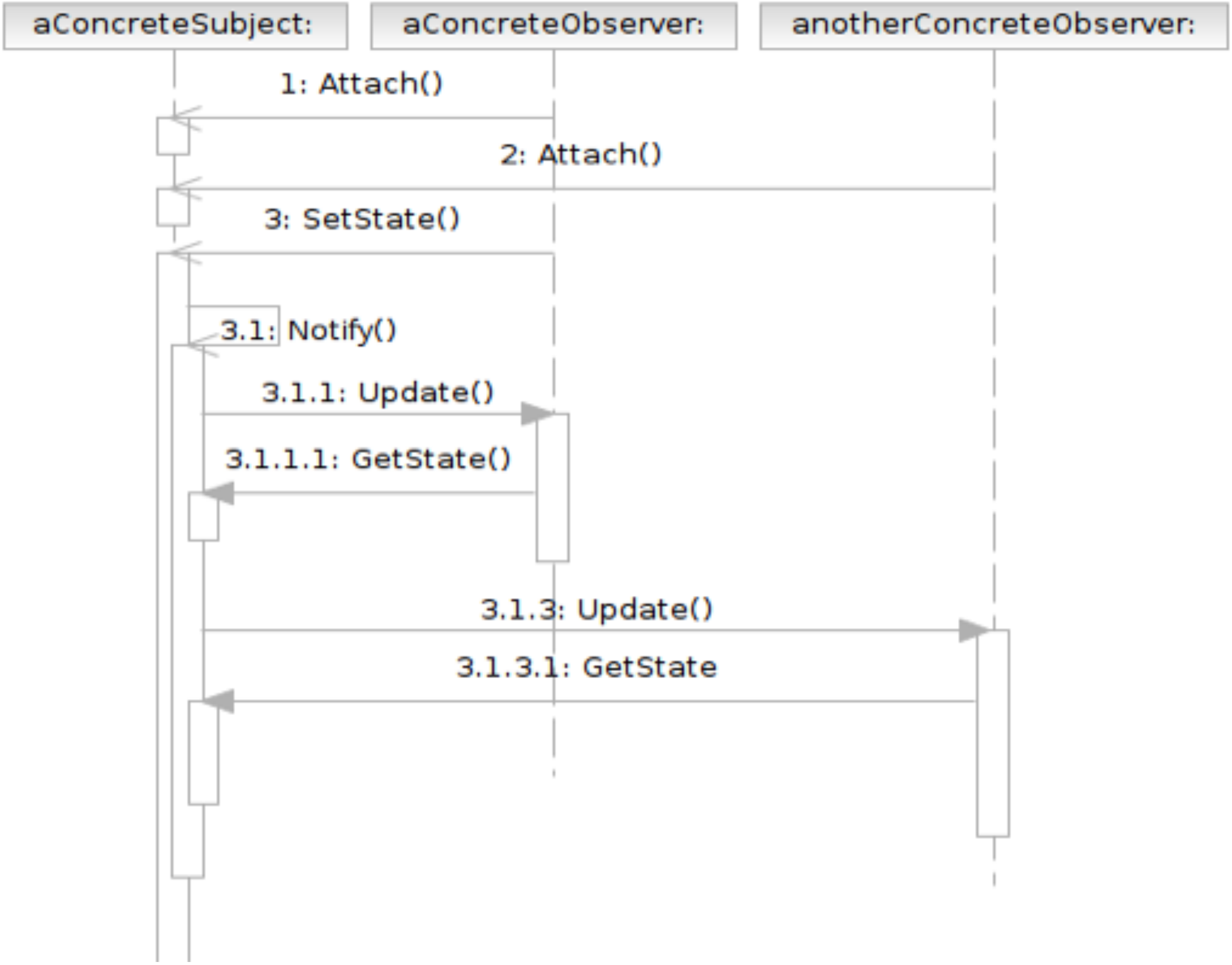
Structure



```
for all o in observers {
    o->Update()
}
```

return SubjectState

Interaction



Participants

- Class **Subject** knows its observers and provides an interface for attaching and detaching Observer objects
  - A.K.A **Publisher**, who generates events and sends notifications
- Class **Observer** defines an updating interface
  - A.K.A. **Subscriber**, who is interested in the events

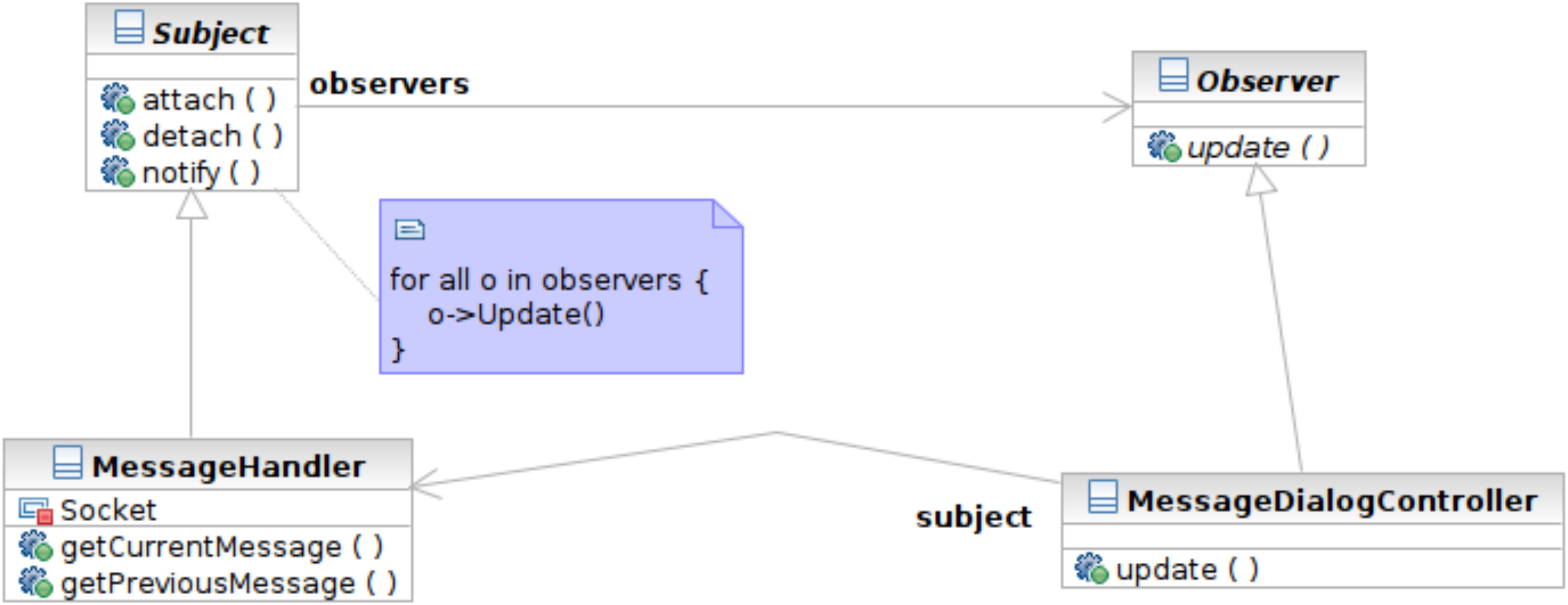
Participants (Cont'd)

- □ Class **ConcreteSubject** stores state and sends notifications to observers
- □ Class **ConcreteObserver** maintains a reference to a ConcreteSubject object; stores states; implements the Observer updating interface

Applicability

- Use the Observer pattern when
  - an abstraction has two aspects, one (observer) dependent on the other (subject).
  - a change to one object (subject) requires changing others (observers), and you don’t know how many objects need to be changed
  - an object (subject) should be able to notify other objects (observers) without making assumptions about who these objects are (the observers' classes).

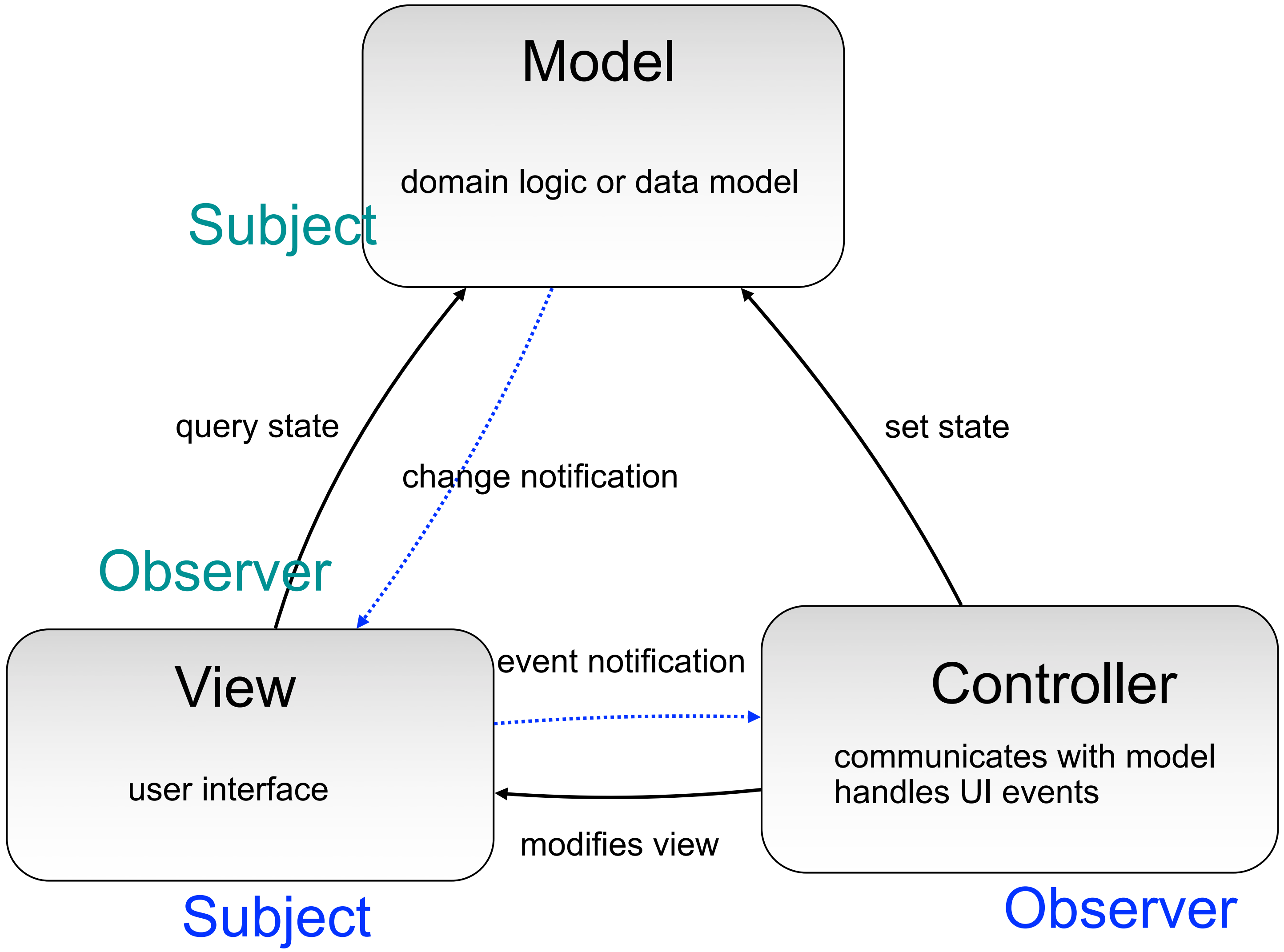
Sample Structure



Samples

- Class Observer
- Class Subject
- Concrete Subject: Class MessageHandler, sends/receives messages to/from network
- Concrete Observers: MessageDialogController, observes the event

MVC and Observer Pattern



Consequences

- Abstract coupling between Subject and Observer. All a subject knows is that it has a list of observers.
- Support for broadcast communication. The notification is broadcast automatically to all interested observers.
- Unexpected notifications. An innocuous operation on the subject may cause all registered observers to be updated.

Related Patterns

- **Mediator**: mediator may receive the communication from the colleagues using the observer pattern

Mediator

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

Challenge

- In your user interface, different widgets should act in response to others
  - click item button, the item list shows up
  - select one friend in the list and detail information is displayed accordingly on another panel

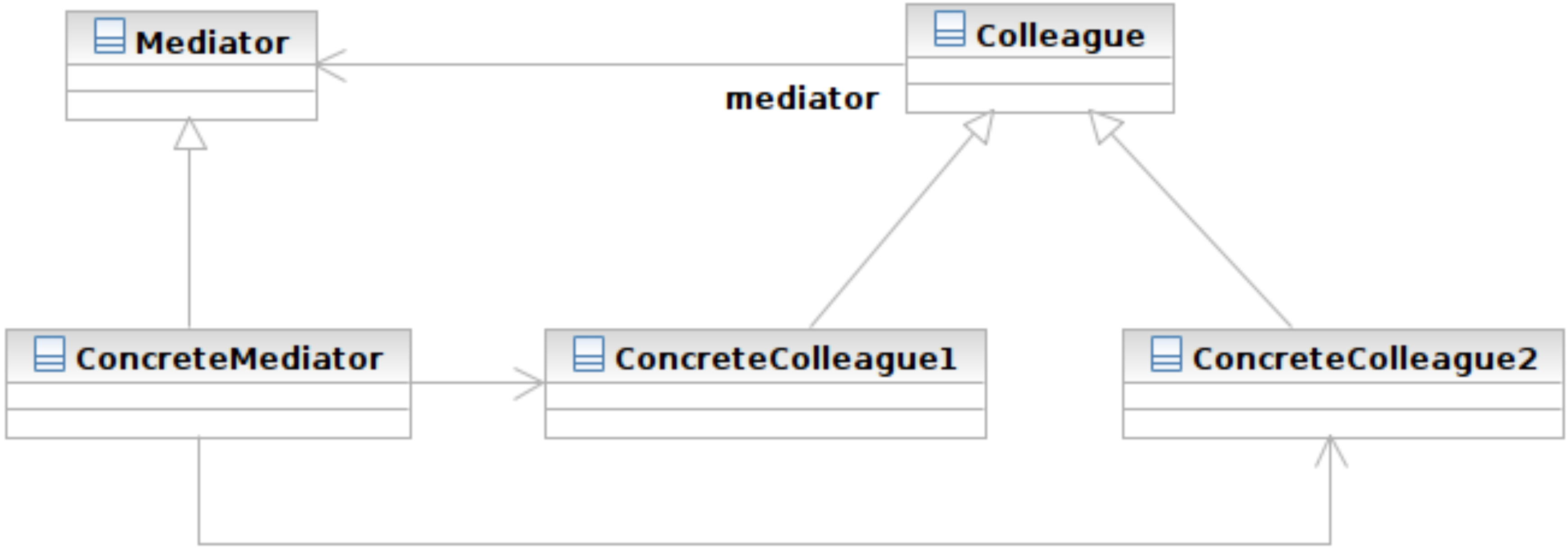
Challenge (Cont'd)

- First attempt:
  - Each widget has references to other widgets and checks other widgets for updates
  - Worst case: each widget knows about all other widgets: O(N^2) complexity of the relationships

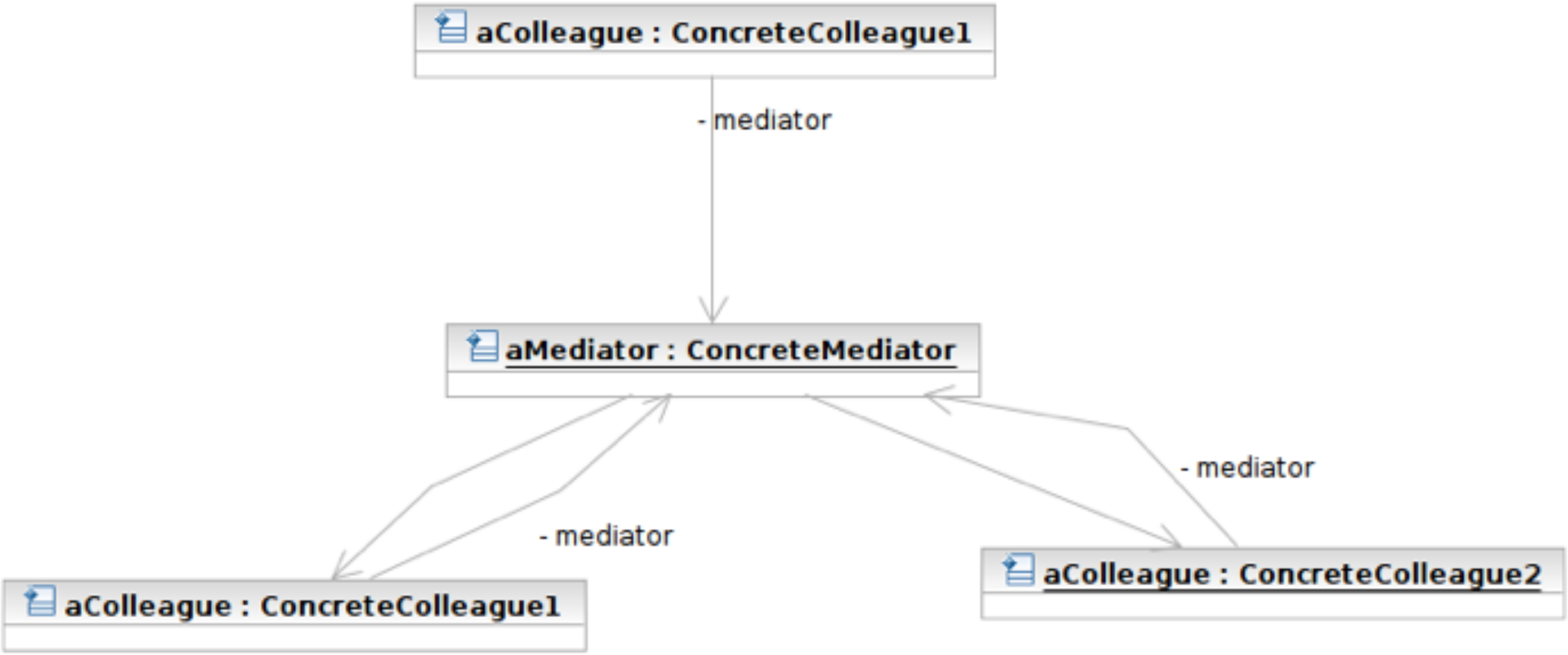
Mediator

- Problem: how can we handle interactions between a set of objects in a flexible way?
- Think: what is the effort if you decide to add one more widgets to the user interface?
- Target: encapsulate the interaction between objects. Objects don't refer to one another and interaction can be varied independently.

Structure


Mediator
Colleague
mediator
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Structure


aColleague : ConcreteColleague1
- mediator
aMediator : ConcreteMediator
- mediator
- mediator
aColleague : ConcreteColleague1
aColleague : ConcreteColleague2

Participants

- Class **Mediator** defines an interface for communicating with Colleague objects
  - Often acts as the **Controller** in the MVC design pattern
  - Often acts as the **Observer** in the Observer pattern
- Class **ConcreteMediator** knows and maintains its colleagues and implements their interactions

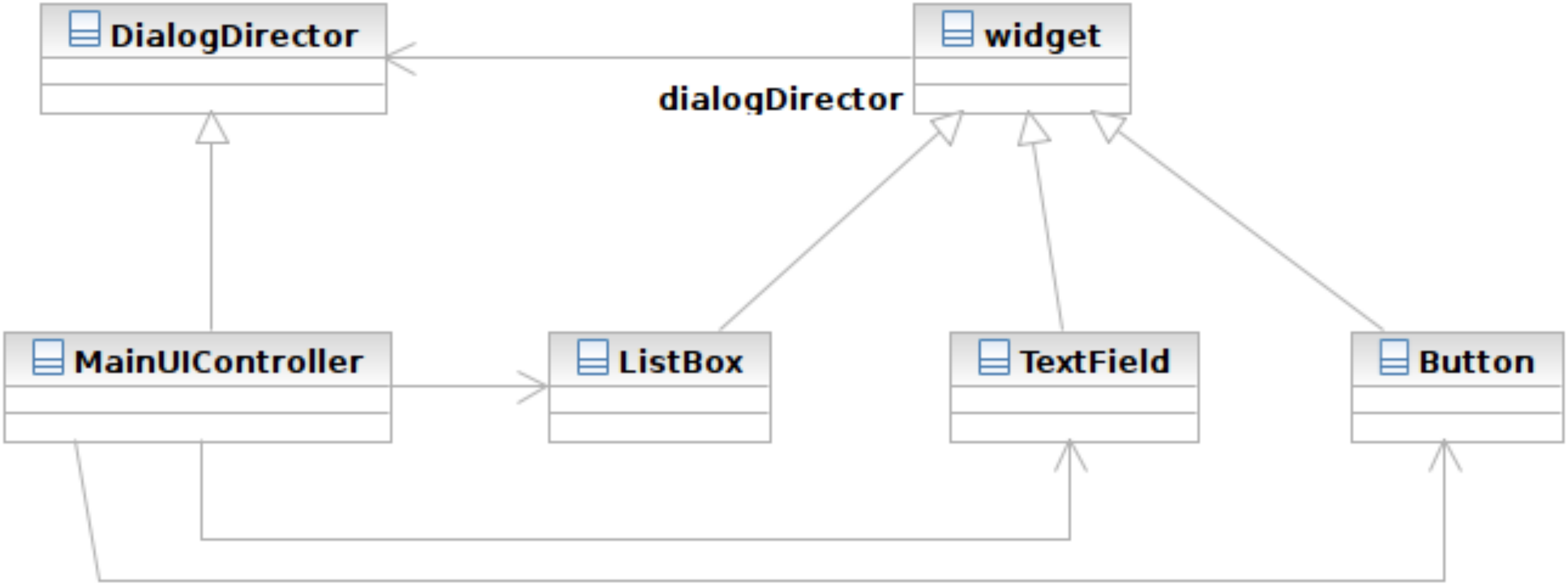
Participants

- Class **Colleague** knows its Mediator and communicates with other colleagues via mediator
  - Often the View components in the MVC pattern
  - The **Subjects** in the Observer pattern

Applicability

- Use the Mediator pattern when
  - a set of colleagues communicate in a well-defined but complex ways.
  - reusing a colleague is difficult because it refers to and communicates with many other objects
  - you want to customize some objects' behaviors and interactions without a lot of subclassing: encapsulate in a mediator

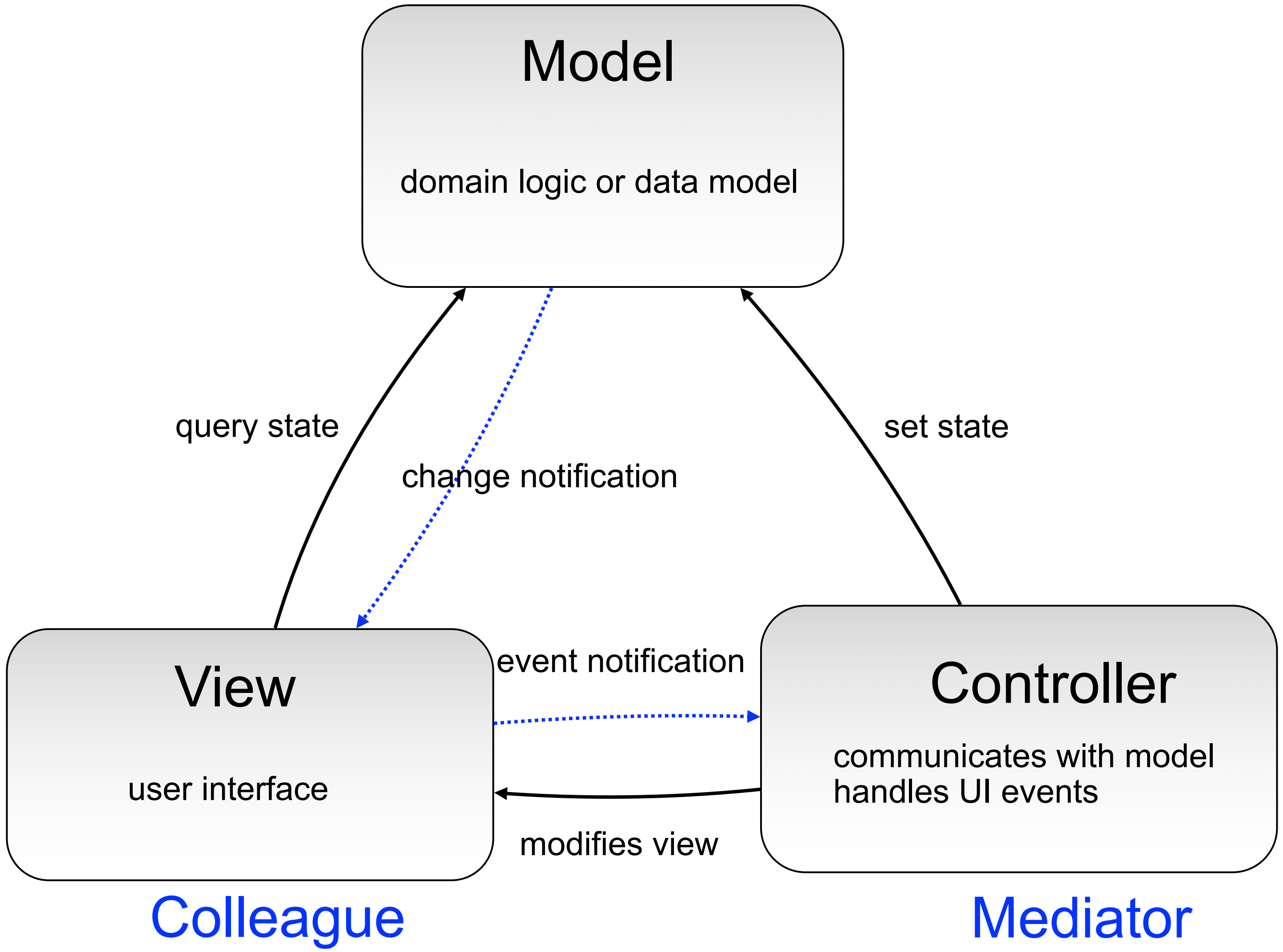
Sample Structure



Samples

- Mediator: class DialogDirector
- Colleague: class Widget
- Concrete Colleagues: ListBox, TextField, Button, and many other GUI components
- Concrete Mediator: MainUIController
  - Implementing DialogDirector::CreateWidgets ()
  - Implementing DialogDirector::update()
    - Observer pattern

MVC and Mediator Pattern



Consequences

- It limits subclassing. A mediator localizes behavior that otherwise would be distributed among several objects.
- It decouples colleagues. Colleagues don't have to know one another
- It simplifies object protocols. Many-to-many interactions between colleges is replaced with one-to-many interactions between the mediator and its colleagues.

Consequences

- It abstracts how objects cooperate. Mediators separate colleagues' interactions from their own behaviors
- It centralizes control. Complexity of interaction is centralized in the mediator.

Related Patterns

- **Facade**: facade provides the interface of the subsystem to the outer world. It's one-way communication. Mediator facilitates two-way communications between colleagues.
- **Observer**: colleagues communicate with the mediator using the Observer pattern

Command

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

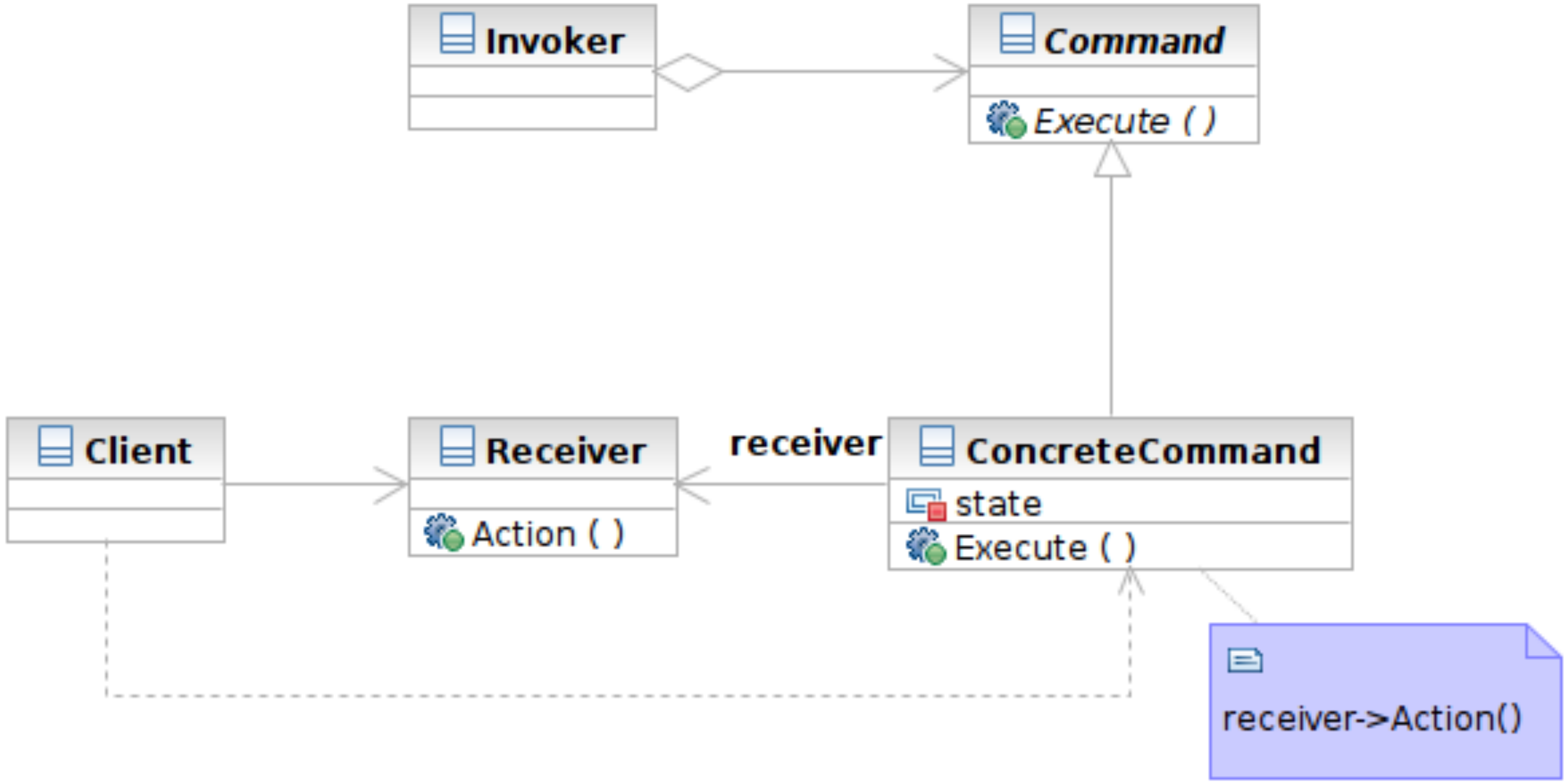
Challenge

- We want to customize the behaviors of the reusable widgets
  - Add a new user when “buy item” button is pushed
  - We have “sell item”, “drop item” and many more widgets performing different actions
  - Widget classes don't know anything about the action, but has to execute it
    - perform the action when the button is pushed
- First attempt: subclassing the widgets

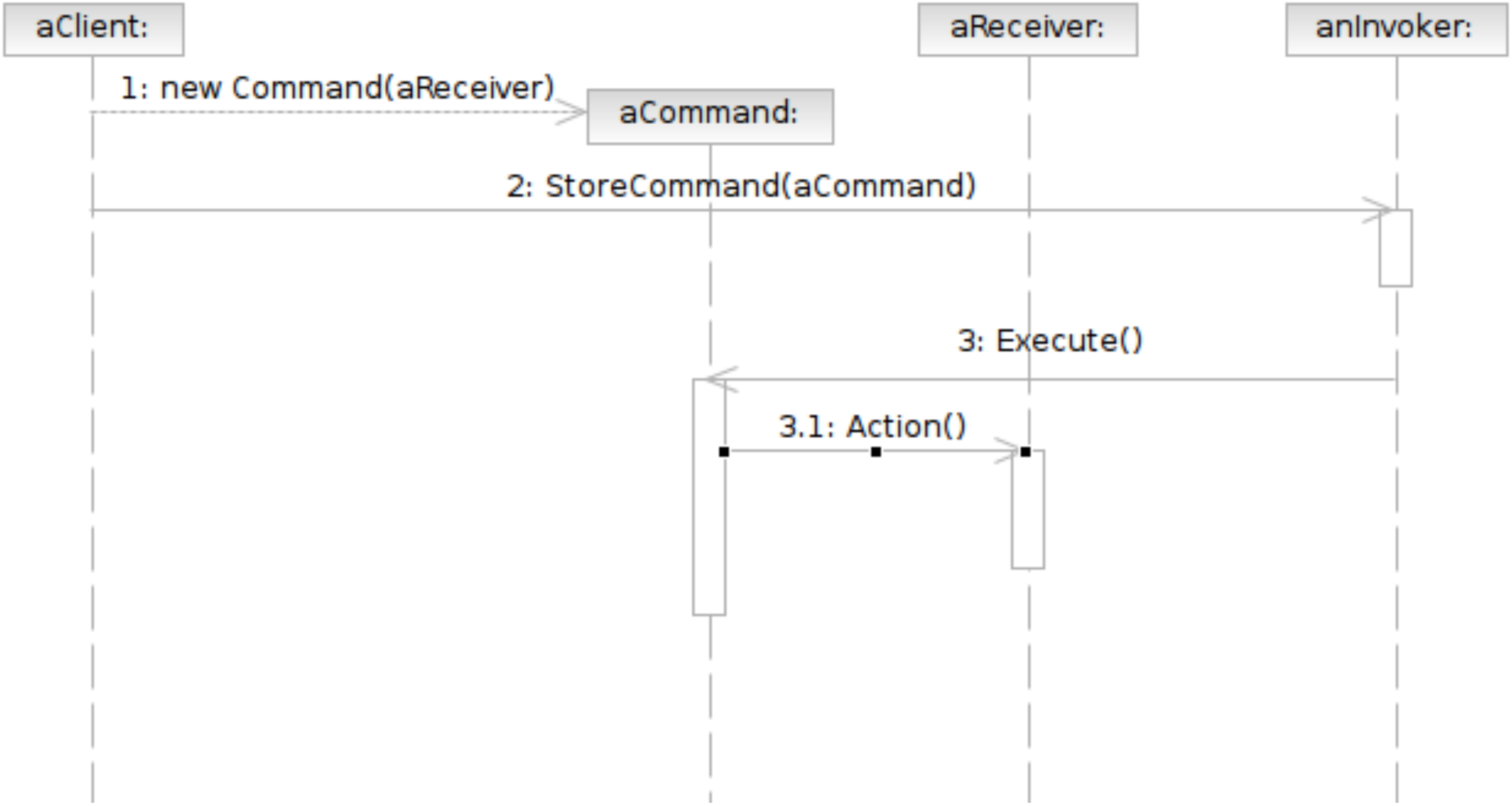
Command

- Problem: how can we define actions that can be invoked by other objects at later times
- Think: is subclassing flexible? What if you have many actions to perform or you are not allowed to subclass the invokers?
- Target: encapsulate actions as objects such that the actions can be passed to invokers, be queued and invoked later, and be undone

Structure


Invoker
Command
Execute ( )
Client
Receiver
Action ( )
receiver
ConcreteCommand
state
Execute ( )
receiver->Action()

Interaction



Participants

- Class **Command** declares an interface for executing an operation.
- Class **ConcreteCommand** defines a binding between a Receiver object and an action; implements Execute by invoking the corresponding operations on Receiver
  - note that there hasn't to be only one receiver used in a command
  - a receiver isn't always necessary for a command to execute, either

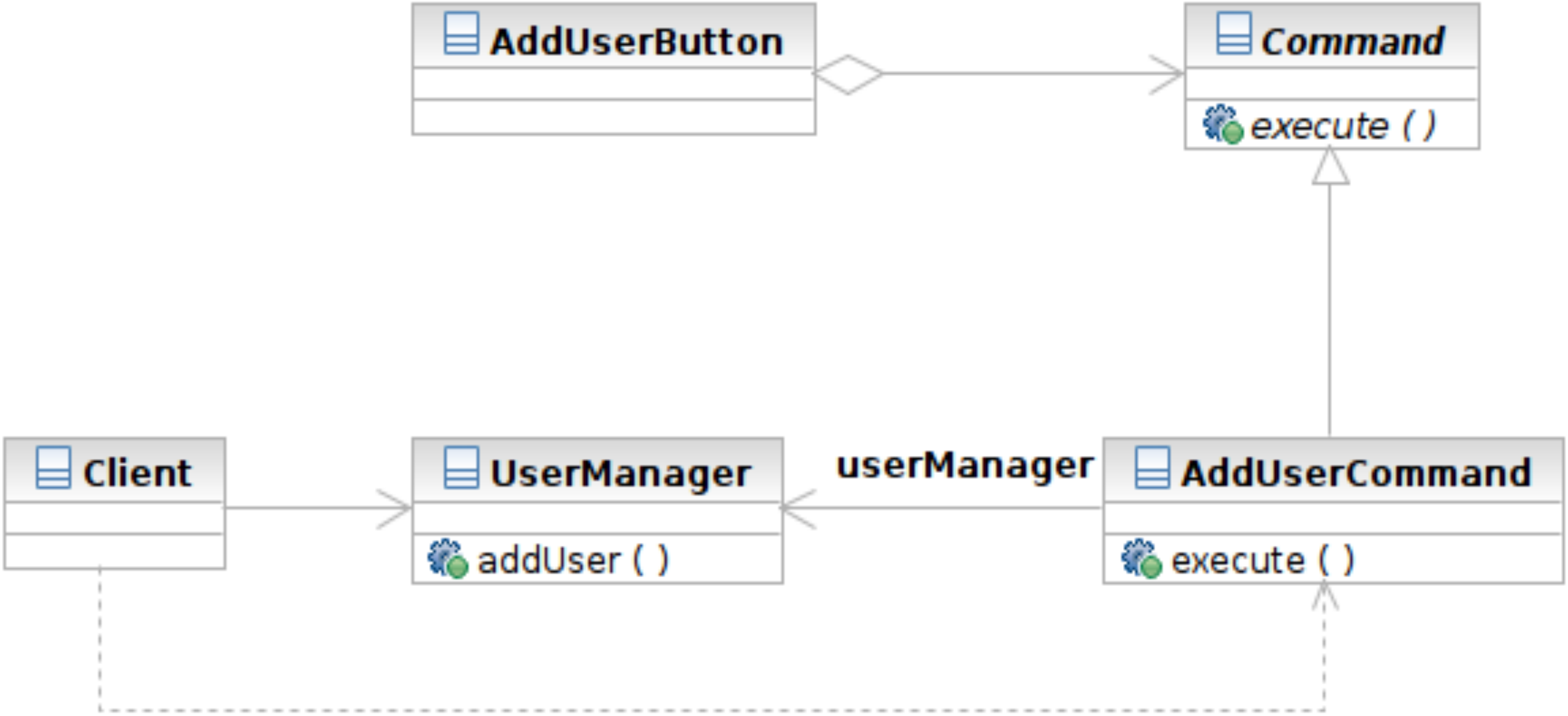
Participants (Cont'd)

- Class **Client** creates a ConcreteCommand object and sets its receiver
- Class **Invoker** asks the command to carry out the request
- Class **Receiver** knows how to perform the operations

Applicability

- Use the Command pattern
  - to parameterize objects (e.g. widgets) with an action (command) to perform.
  - instead of subclassing
  - to specify, queue and execute requests at different times.
  - to support undo.
  - to support macro commands (commands composed of other commands)

Sample Structure



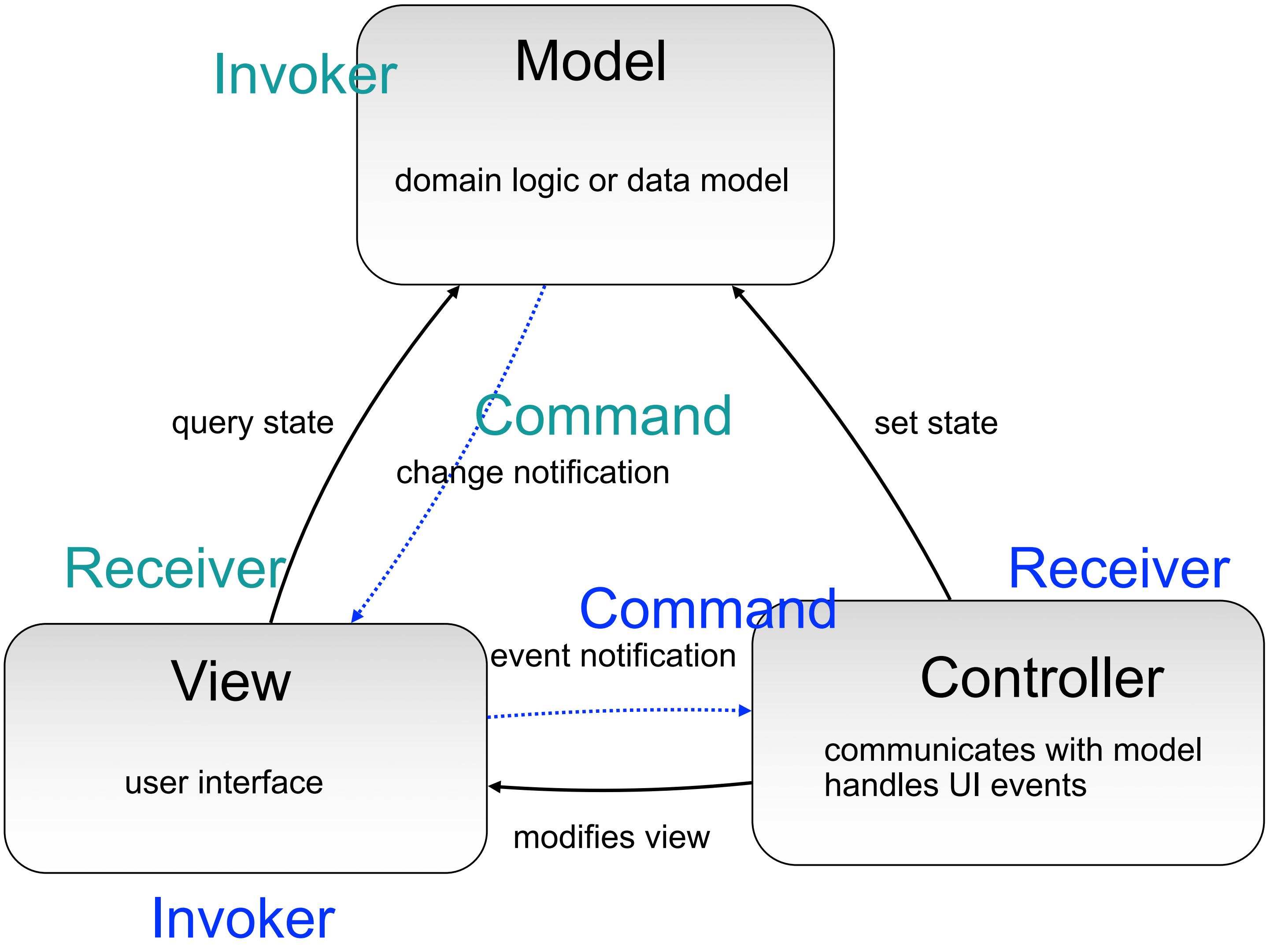
Samples

- Command: class Command
  - defines the interface
- ConcreteCommand: class AddUserCommand
  - implements execute()
- Receiver: class UserManager
  - who receives the command
- Client: class Client
  - creates the command
  - associates the command with the receiver

Samples (Cont'd)

- Invoker: class AddUserButton
  - who triggers the execution of the command
  - e.g. user pushed the button
- Composite Command: class MacroCommand
  - the composite pattern
  - is composed of other commands

MVC and Mediator Pattern



Consequences

- It decouples the invoker from the receiver.
- Commands are first-class objects. They can be assembled into a composite (macro) command.
- They can be extended easily.

Related Patterns

- **Composite**: used to implement MacroCommands
- **Memento**: used to remember the state the command requires for undoing the operation
- **Prototype**: cloning a command before putting on the command history list

Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

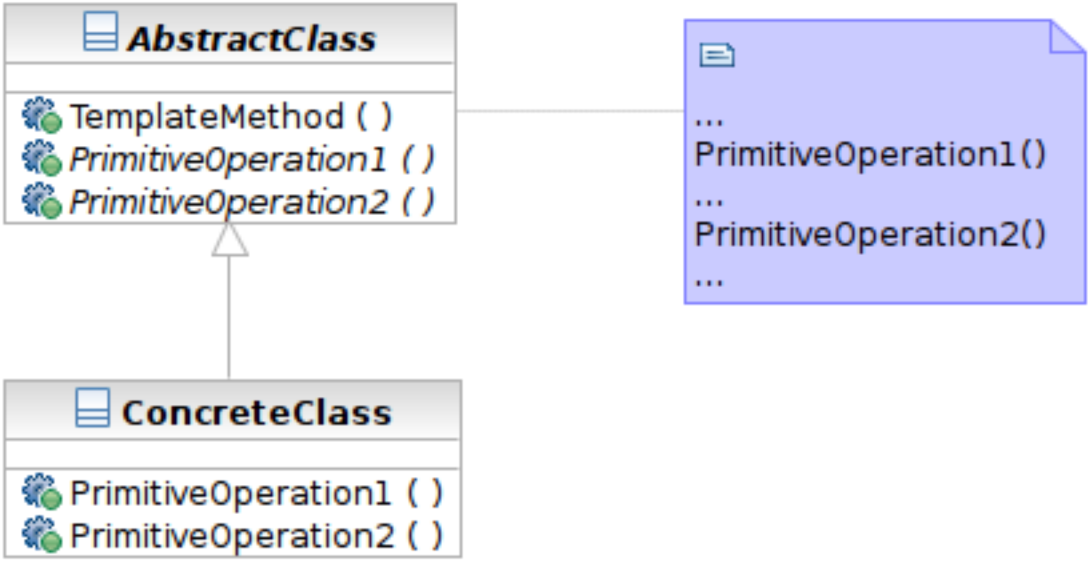
Challenge

- Validating user account on registration
  - check registered account ID
  - validate address, phone number in multiple countries
  - validate credit card
- First attempt: one concrete validator for each country. Each validator performs all validations.
  - some logic are the same for all countries and can be shared

Template Method

- Problem: how can we do both code reuse and customization of one algorithm?
- Think: how much code is redundant in the big validation method? What is the effort to change the validation logic?
- Target: define the skeleton of an algorithm in an operation and defer some steps to subclasses.

Structure


AbstractClass
TemplateMethod ( )
PrimitiveOperation1 ( )
PrimitiveOperation2 ( )
...
PrimitiveOperation1()
...
PrimitiveOperation2()
...
ConcreteClass
PrimitiveOperation1 ( )
PrimitiveOperation2 ( )

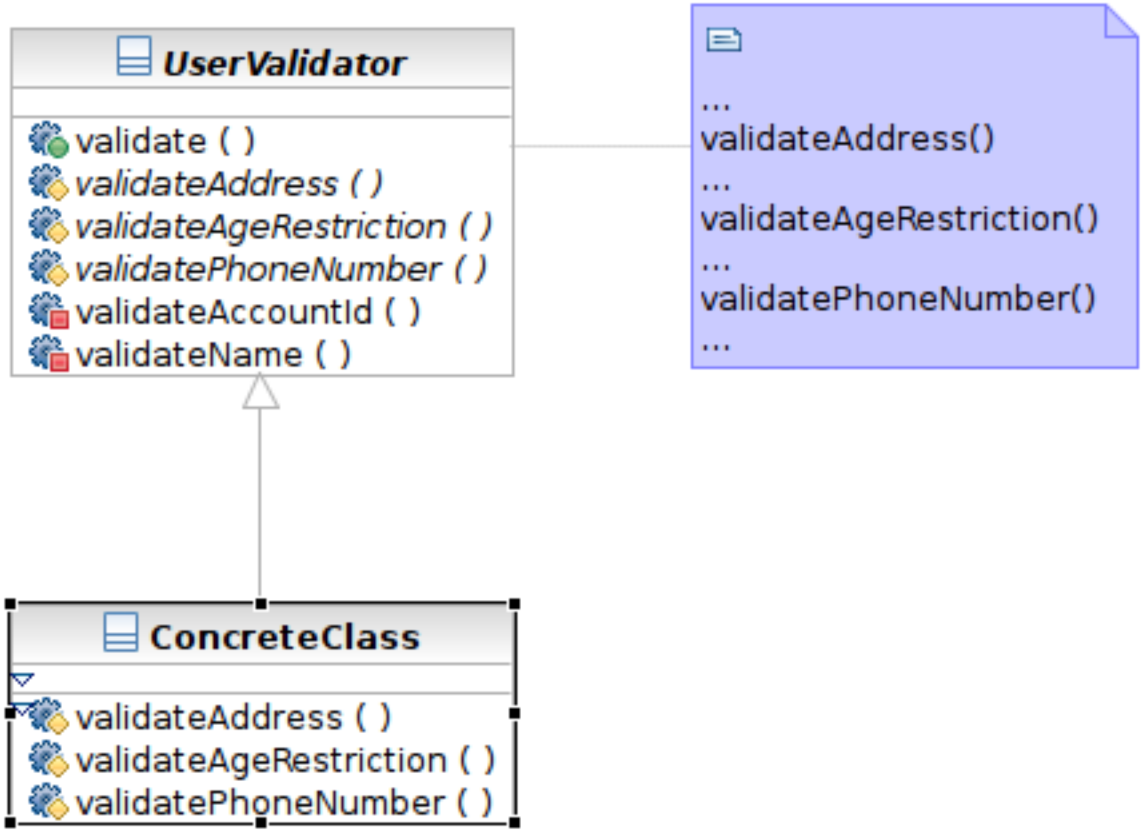
Participants

- Class **AbstractClass** defines abstract primitive operations (steps) of an algorithm; implements a template method defining the skeleton of an algorithm.
- Class **ConcreteClass** implements the primitive operations.

Applicability

- The Template Method pattern should be used
  - to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
  - when common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
  - to control subclasses extensions. Extensions are permitted in implementations of primitive operations.

Sample Structure



Samples

- AbstractClass: class UserValidator
- ConcreteClass:
  - class TaiwauUserValidator and USUserValidator

Consequences

- The Hollywood principle. Don’t call us, we’ll call you.
  - why calling from parent class?
- Template methods call the following kinds of operations:
  - concrete operations
  - concrete AbstractClass operations
  - primitive operations
  - factory methods
  - hook operations

Related Patterns

- **Factory Method**: often acts as the primitive operation that is called by a template method
- **Strategy**: template method varies part of the algorithm via inheritance. Strategy delegates the entire algorithm to another object.