

# Design Patterns

**Ching-Lin Yu**

**Mozilla**



# Contents

2

- Why Design Patterns
- What is a Design Pattern
- GoF Design Patterns
  - ▣ Creational Patterns
  - ▣ Structural Patterns
  - ▣ Behavioral Pattern

# Why Design Patterns

3

- It's all about software complexity
  - ▣ <http://www.informationisbeautiful.net/visualizations/million-lines-of-code/>
- Naive changes tends to deteriorate the software
  - ▣ “Code smells”
    - Duplicated code
    - Long method
    - Complex control structure
    - Large class
    - Code depending on implementation
    - etc.

# Why Design Patterns

4

- Life is hard when you continue to work on the software
- Example
  - ▣ A cloud file system client that is too intimate to the implementation
    - Concrete class names are seen throughout the code
  - ▣ Hard to maintain when a new cloud file system needs to be supported
  - ▣ Solution: abstract factory

# What is a Design Pattern

5

- A general **repeatable solution** to a commonly-occurring **problem** in **software design**.
- With design patterns, you don't have to reinvent the wheel
- Design patterns provide good solutions, not functionally correct solutions

# What is a Design Pattern

6

- So you think you can write good OO programs?
- To reuse ancient's wisdom on software design
  - ▣ More flexible code
  - ▣ Avoid the pitfalls
- To communicate more effectively



# Design Patterns and Object Orientation

7

- Design patterns show how to put good use of OO constructs in designing software
  - ▣ Encapsulation
  - ▣ polymorphism
  - ▣ Inheritance

# What to Expect from Design Patterns

8

- A common design vocabulary
  - ▣ just like Linked Lists in data structures or Quick Sort in algorithms
- A documentation and learning aid
  - ▣ learning design patterns help you understand designs in real systems and make better design
  - ▣ documentation using design patterns are easier to write and understand



# What to Expect from Design Patterns

9

- An adjunct to existing methods
  - ▣ design patterns show how to use OO constructs effectively
  - ▣ provide a smooth transition from analysis to design and then to implementation
- A target for refactoring
  - ▣ refactor to patterns

# GoF and Design Patterns

10

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, the so called “Gang of four”
- As of Mar. 2012, the book was in the 40<sup>th</sup> print since 1994

# Creational Patterns

11

- Creational design patterns abstract the **instantiation process**.
- They help make a system independent of how its objects are created, composed, and represented
  - ▣ They all encapsulate knowledge about which concrete classes the system uses
  - ▣ They hide how instances of these classes are created and put together

# Structural Patterns

12

- A better way for different entities to work together
- Focus on higher level interface composition and integration.
- Particularly useful for making independently developed libraries to work together

# Behavioral Patterns

13

- 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

# GoF Design Patterns

14

- Abstract factory
- Adapter & Facade
- Iterator
- Singleton
- Template method & factory method
- Model/View/Controller
- Command & Observer & Mediator

# GoF Design Patterns

15

- Proxy & Decorator
- State
- Chain of Responsibility
- Prototype
- Builder & Composite & Visitor

# Abstract Factory

16

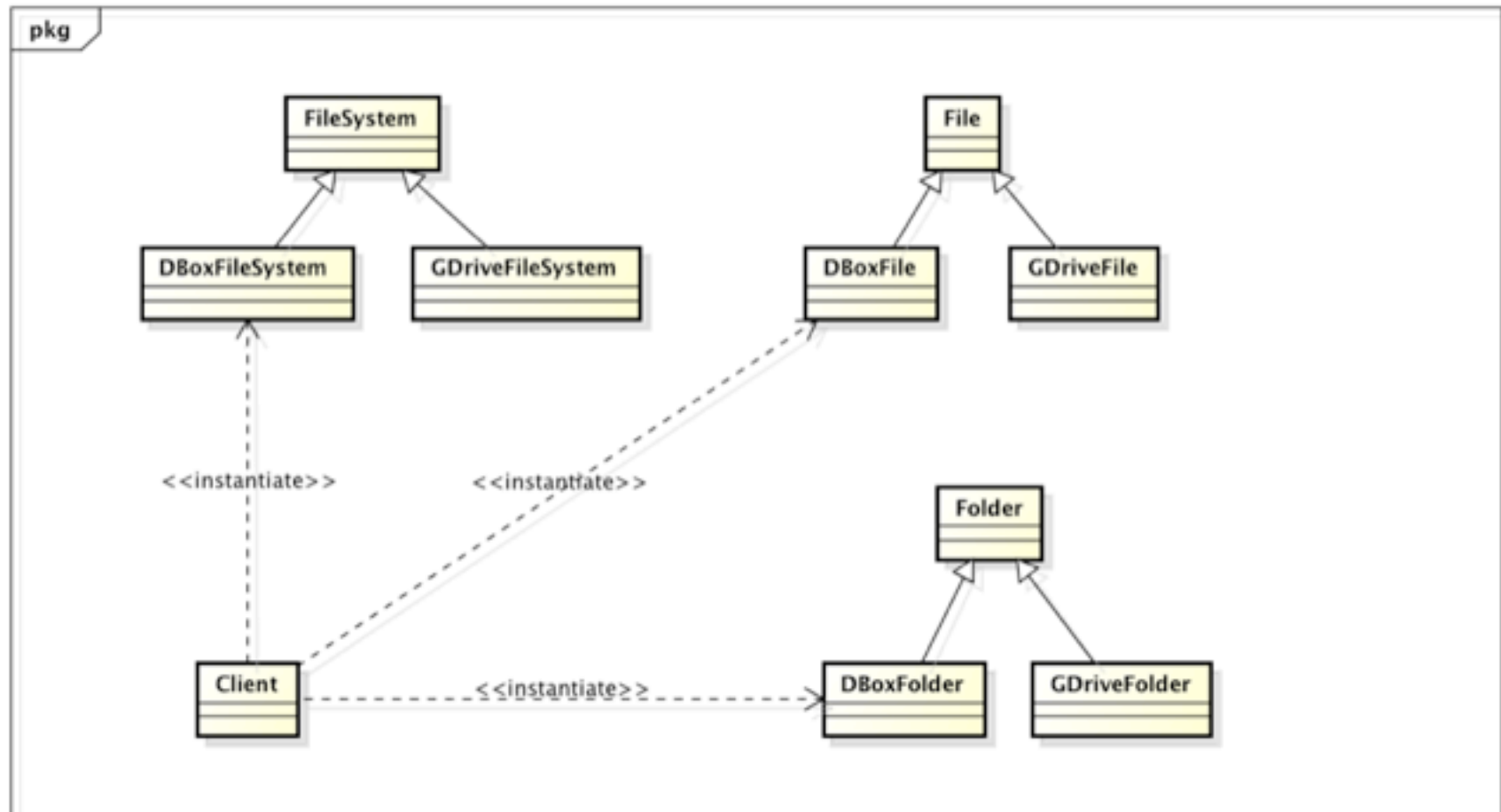
- What it is
  - ▣ An interface for creating families of related or dependent objects
    - Without specifying their concrete classes
- Target Problem
  - ▣ Cloud drive client needs to instantiate different FileSystem, File and Folder objects
    - Without needing to know the concrete classes for different storage providers
  - ▣ Cross platform GUI programming



# Without the Abstract Factory Pattern

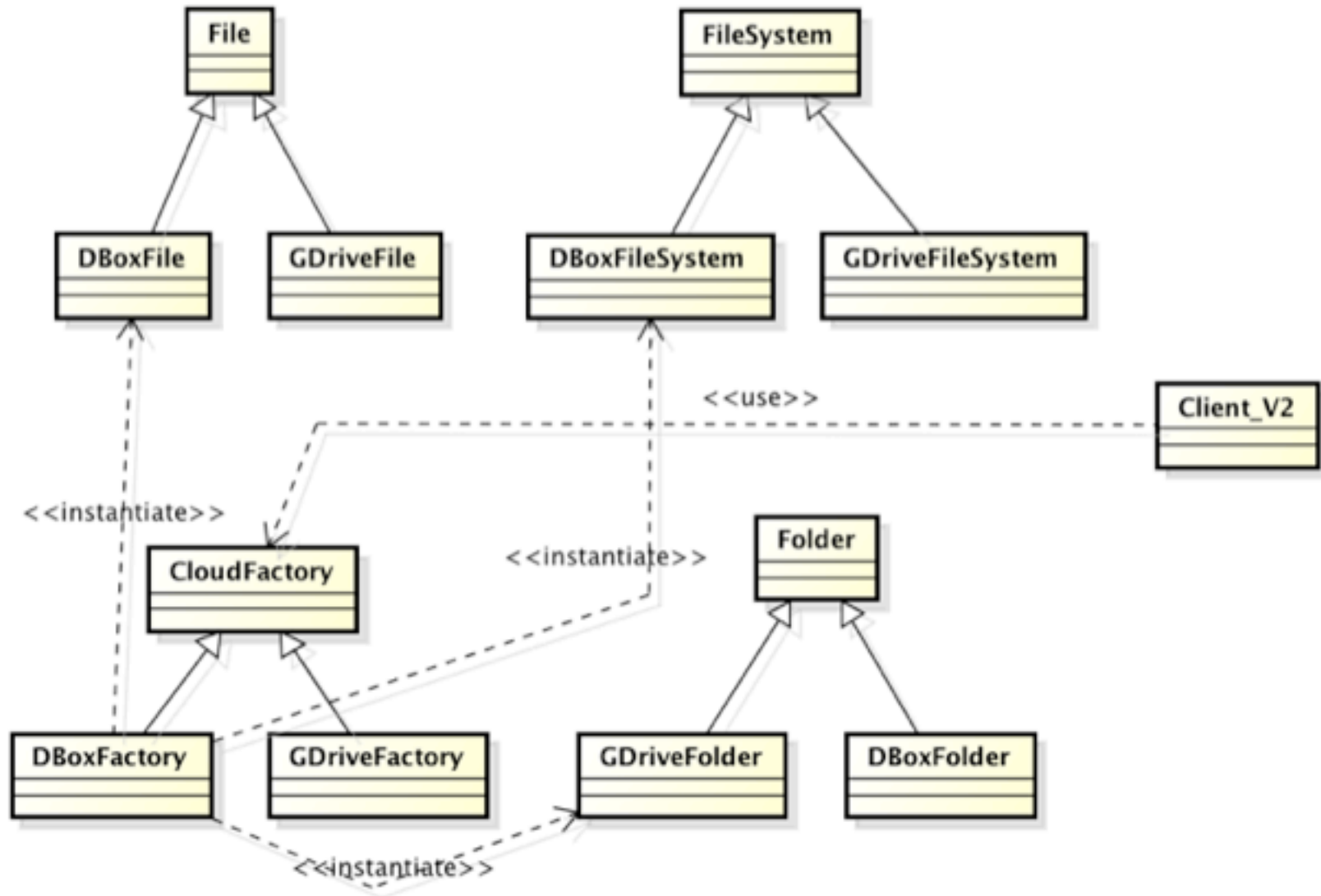
17

- Client has to instantiate the concrete classes of the product family



# Applying the Pattern

18

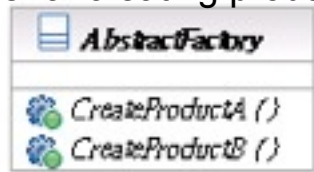


# Structure

19

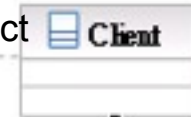
## Abstract Factory

declares an interface for creating product objects

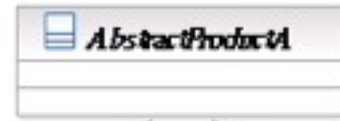


## Client

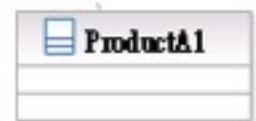
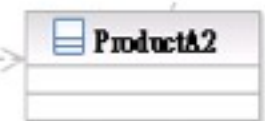
uses only the interface defined by AbstractFactory and AbstractProduct



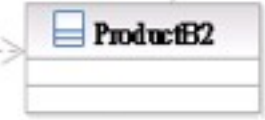
<<use>>



<<use>>



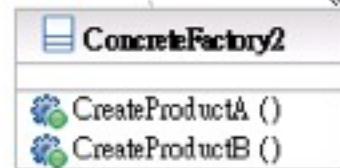
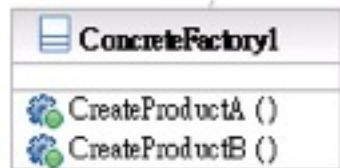
<<instantiate>>



<<instantiate>>

## Concrete Factory

implements the interface



<<instantiate>>

<<instantiate>>

## Concrete Product

defines a product object

## Abstract Product

declares an interface for product object

# Participants

20

- Class **AbstractFactory** declares an interface for creating product objects;
- Class **ConcreteFactory** implements the interface;
- Class **AbstractProduct** declares an interface for product objects;
- Class **ConcreteProduct** defines a product object;
- Class **Client** uses only the interface defined by **AbstractFactory** and **AbstractProduct**

# Interface Change: Adapter & Facade

21

- They both change the interface seen by the using class
- Adapter converts an interface
- Facade simplifies an interface

# Adapter

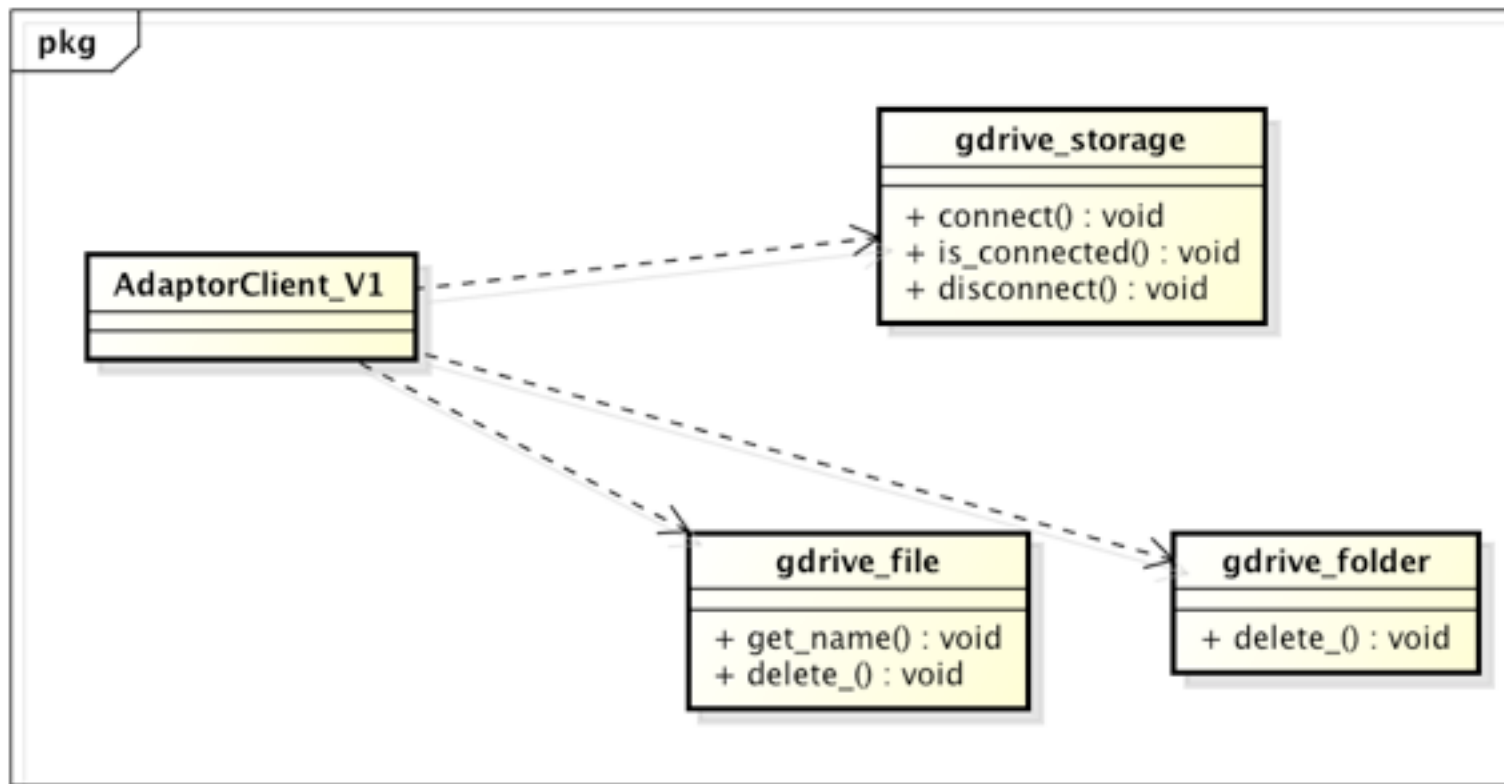
22

- What it is
  - ▣ Conversion of the interface of one class into another the client expects
- Target Problem
  - ▣ Integrate a library into your system but the interface is incompatible
  - ▣ The interface of the library may change in subsequent versions
  - ▣ Replace existing library with another one without impacting existing code

# Without the Adapter Pattern

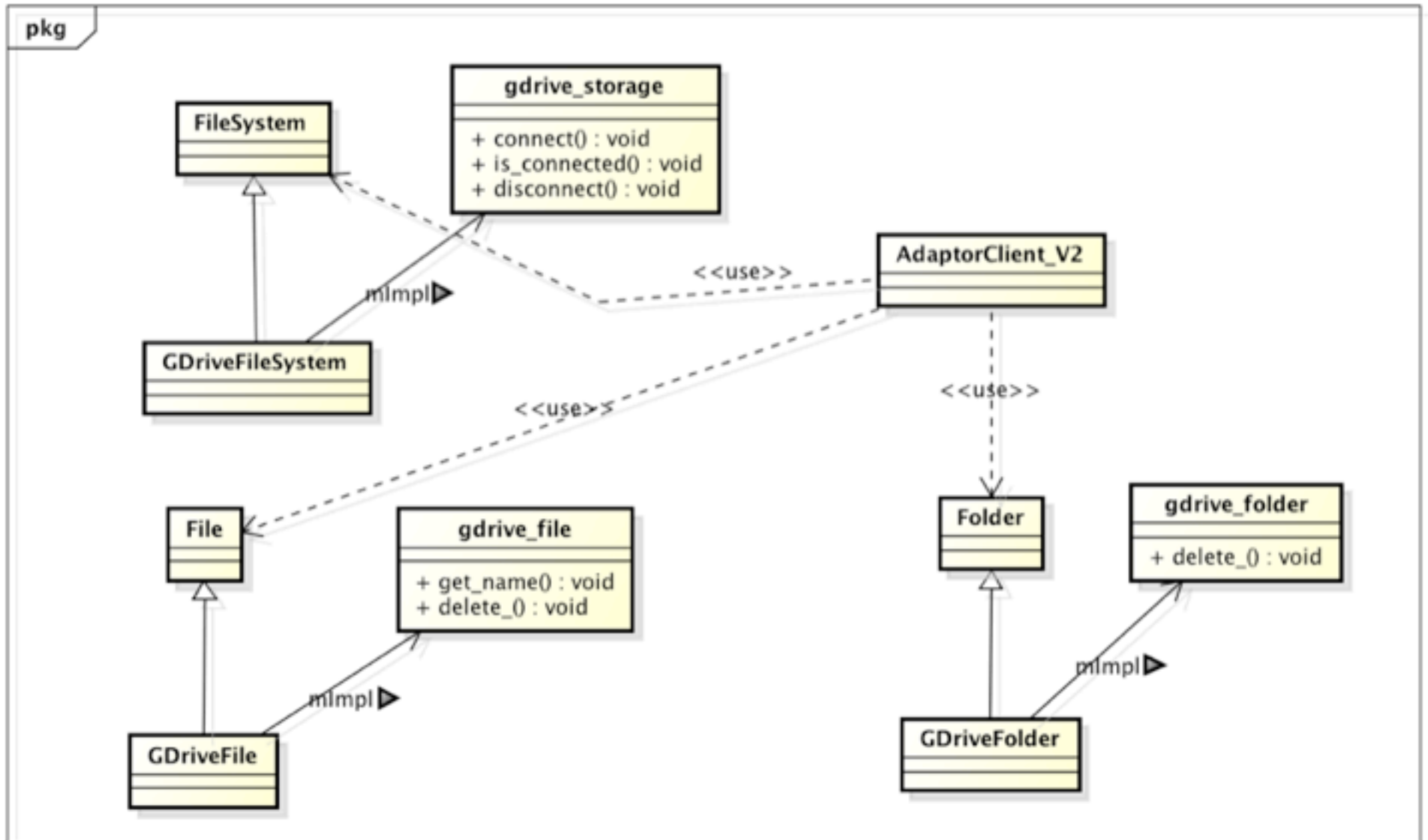
23

- Client is bound to the interface of the library



# Applying the Pattern

24

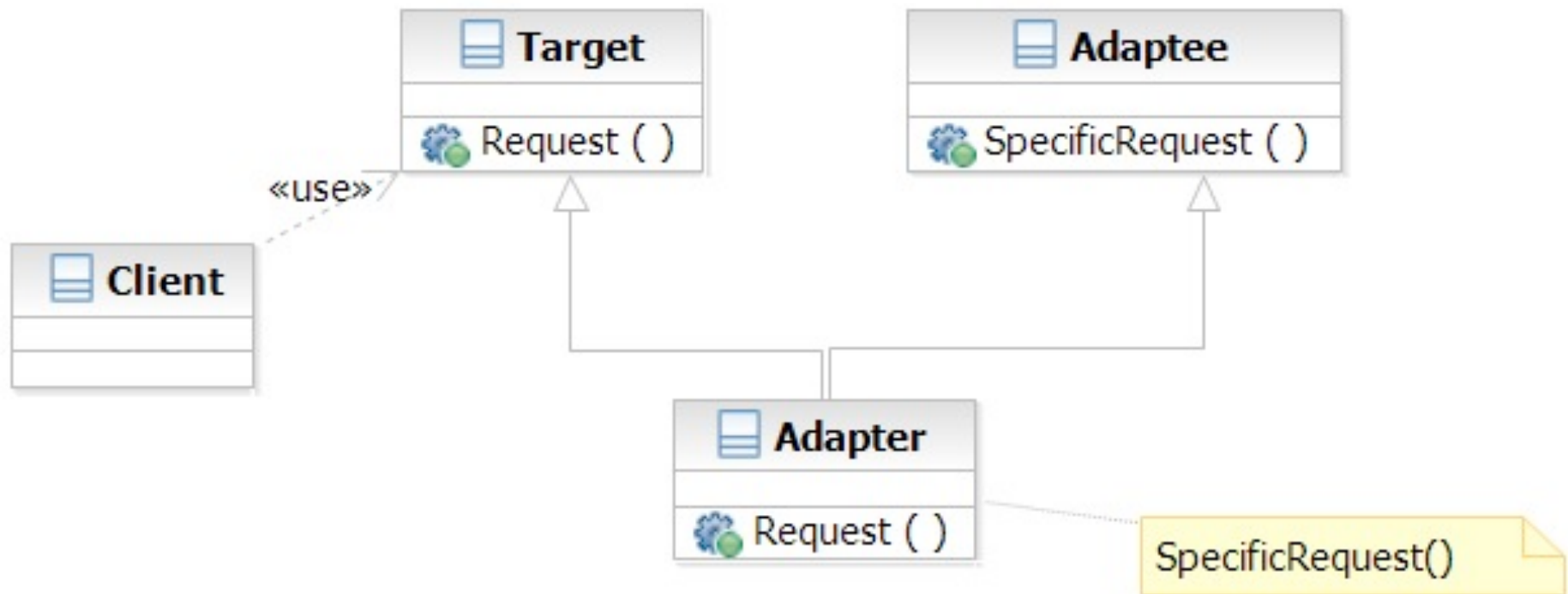




# Structure

25

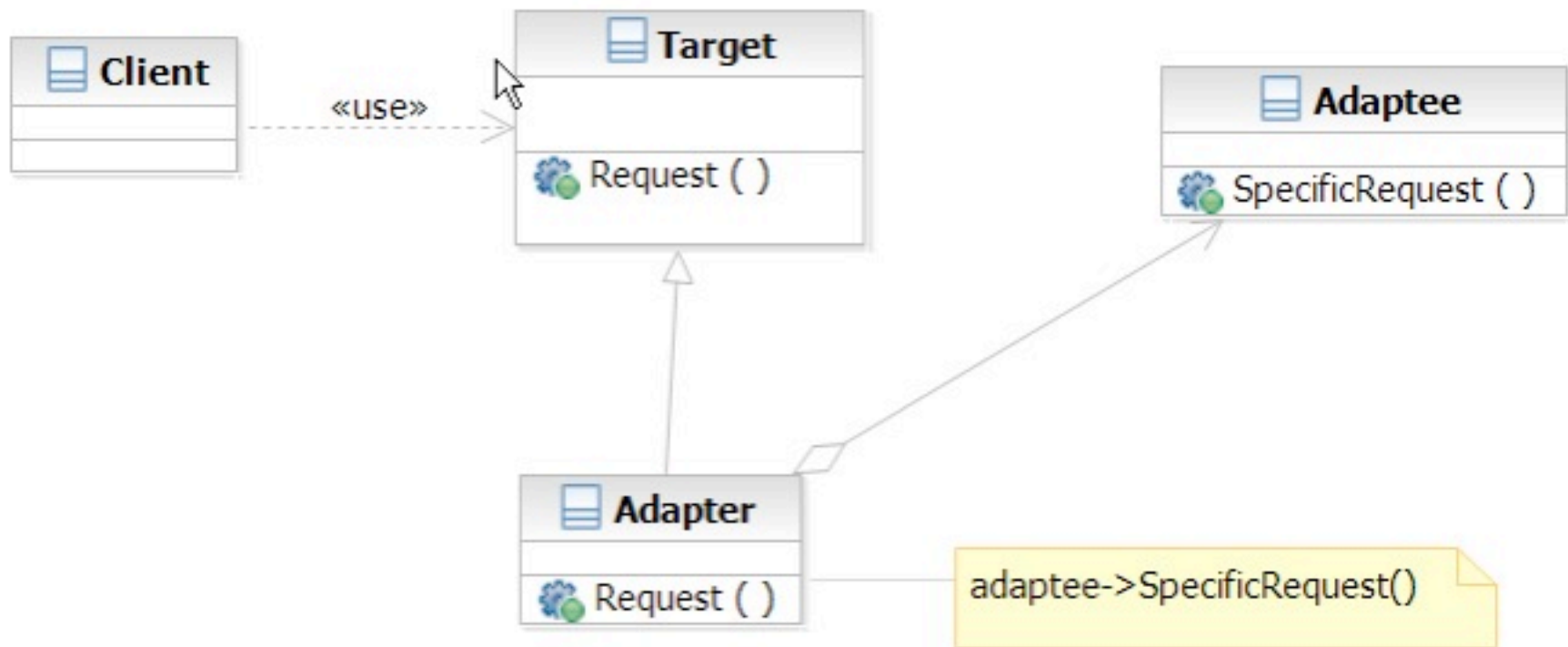
## Class Adapter



# Structure

26

## Object Adapter



# Facade

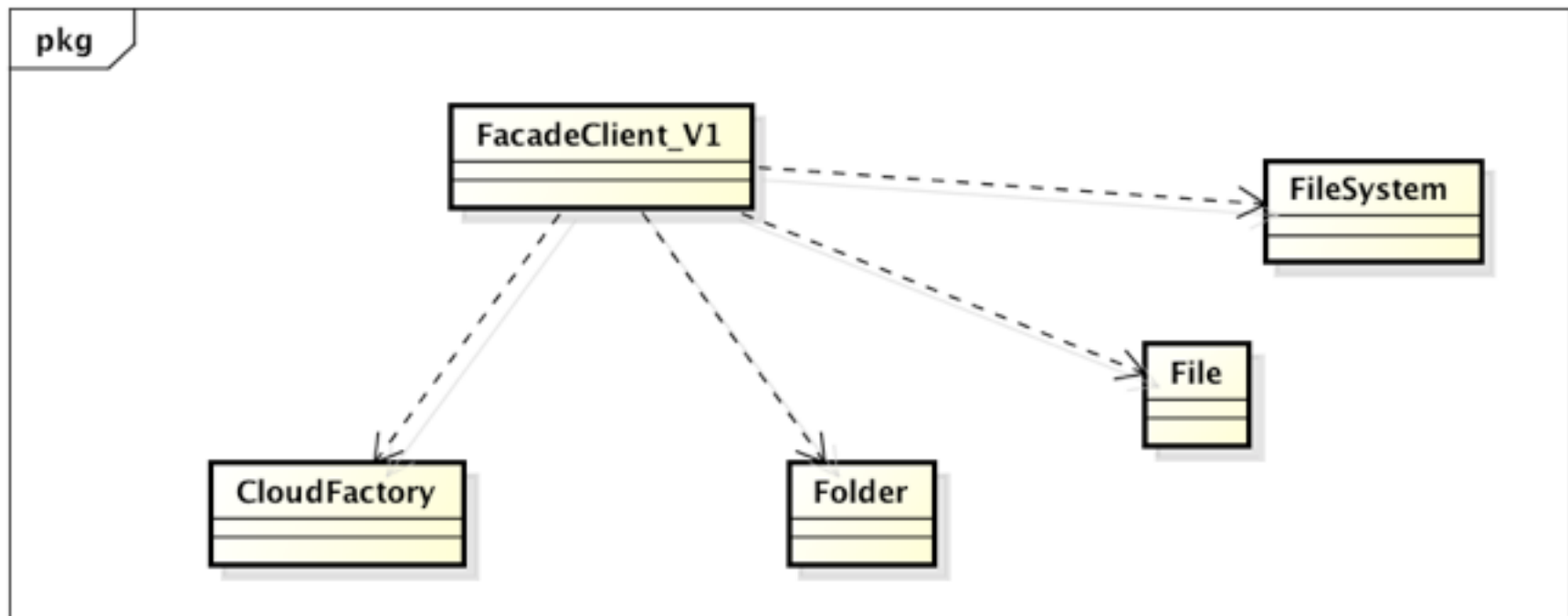
27

- What it is
  - ▣ A high level interface to a set of interfaces in a subsystem
- Target Problem
  - ▣ Providing a simplified interface to the low-level, fine-grained subsystems
    - GCC -> scanner, parser, optimizer, code gen, linker
  - ▣ Unify the access to subsystems
    - e.g. account manager -> database, ldap, remote systems

# Without the Facade Pattern

28

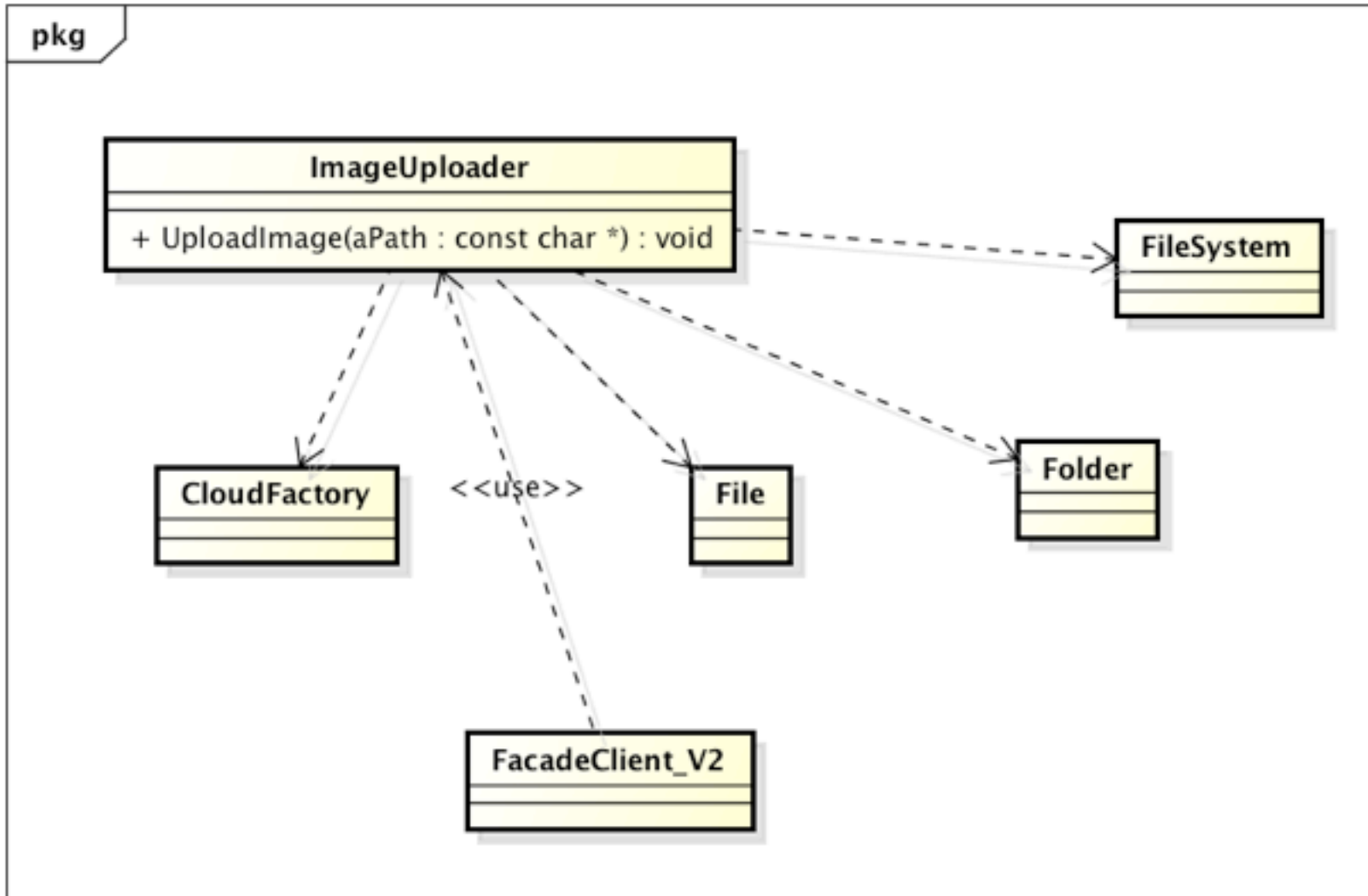
- Client directly uses the interface of the lower-level, fine-grained classes



powered by Astah

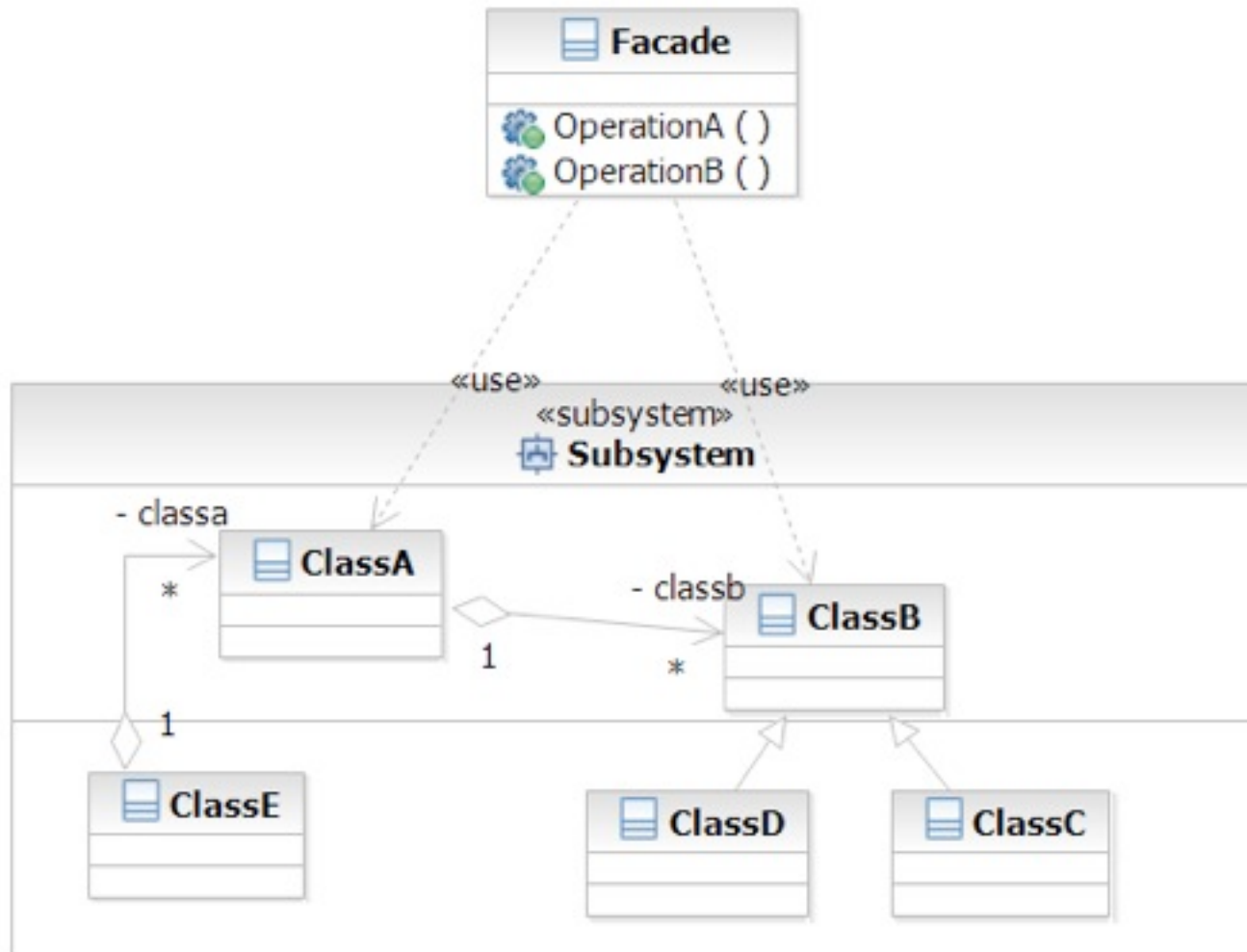
# Apply the Pattern

29



# Structure

30



# Iterator

31

- What it is
  - ▣ A way to access the elements of an aggregate objects sequentially
  - ▣ Without exposing its internal details
- Target Problem
  - ▣ Accessing ‘collection classes’
    - List, Vector, Tree, Sets, etc.
  - ▣ You don’t want your code heavily impacted just because you want to replace a list with a tree

# Without the Iterator Pattern

32

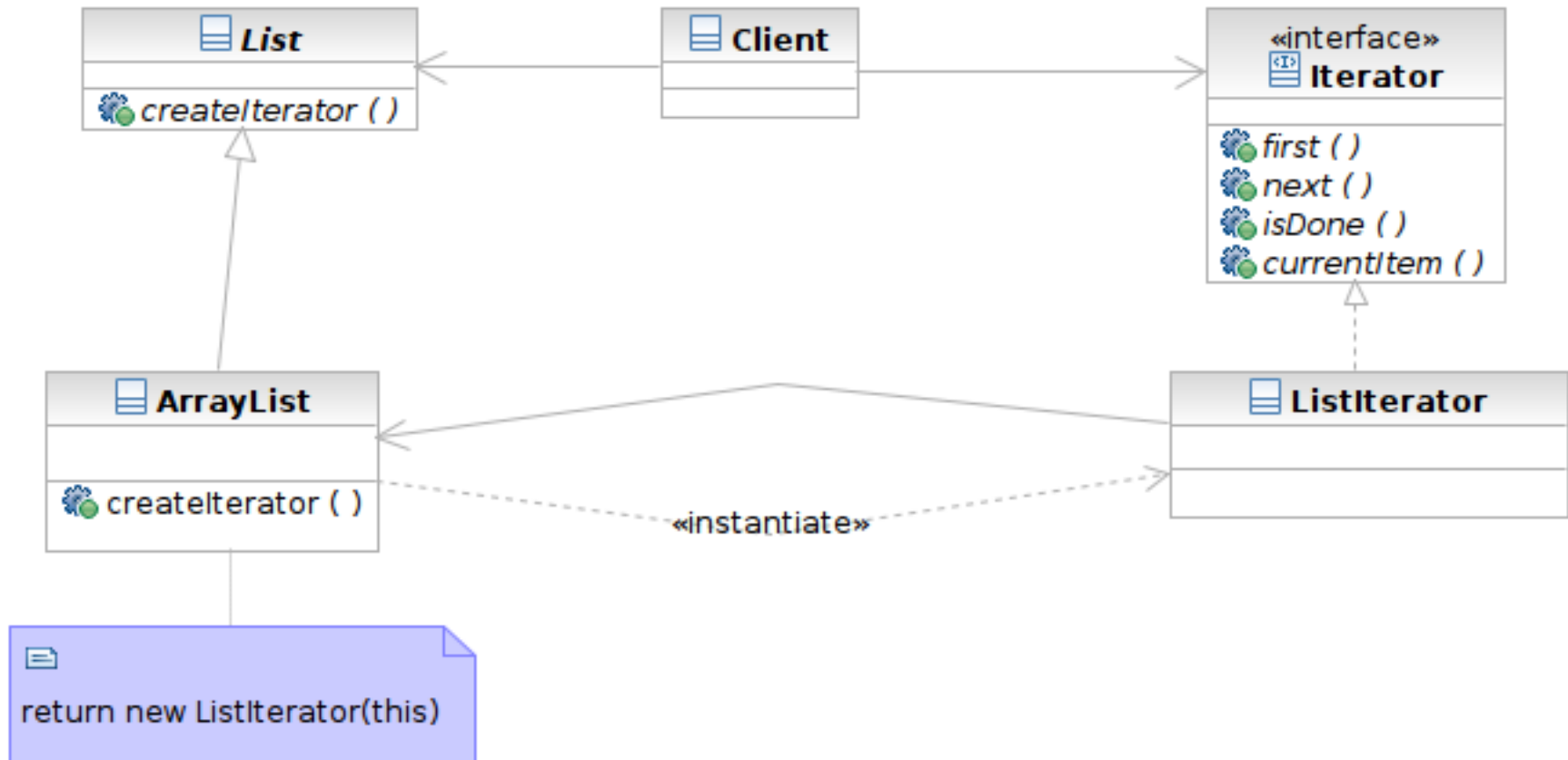
- Client is dependent on the interface of the aggregate classes





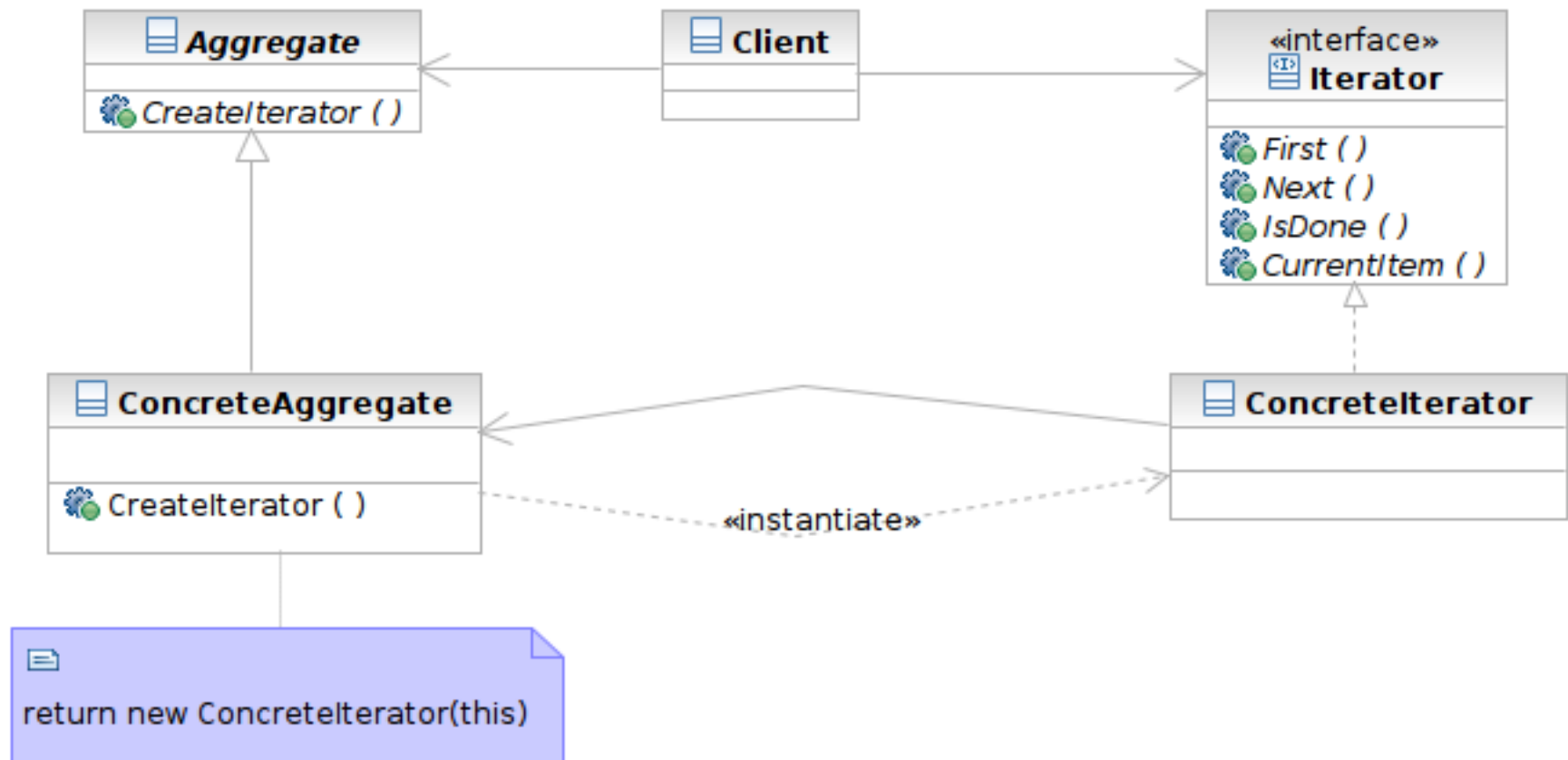
# Applying the Pattern

33



# Structure

34



# Participants

35

- 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

# Beyond Iterator

36

- Iterator provides an universal interface to aggregate classes in an OO way
- Some programming languages solve this problem in language level
  - ▣ Java: foreach style of loop
    - `for (Object element: anArray) { }`
    - Syntactic sugar
  - ▣ Ruby: code block invoked for each element
    - `anArray.each { |element| print element }`

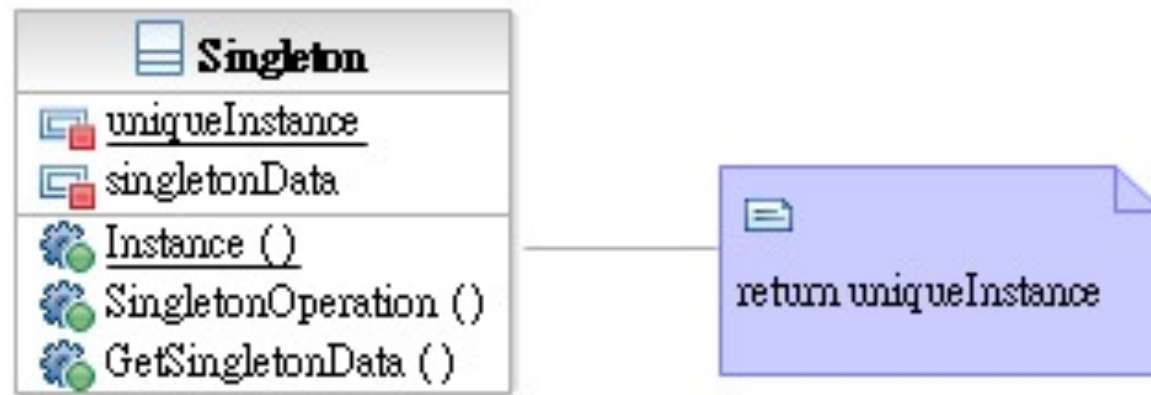
# Singleton

37

- What it is
  - ▣ A class that creates only one instance
  - ▣ The only instance is often globally accessible
- Target Problem
  - ▣ Some classes only need one instance in the system
  - ▣ Multiple instances is either unnecessary or worse, an error in the system
    - Database driver, and abstract factory, connection pool

# Structure

38



## Singleton

defines a static member function that lets clients access its unique instance

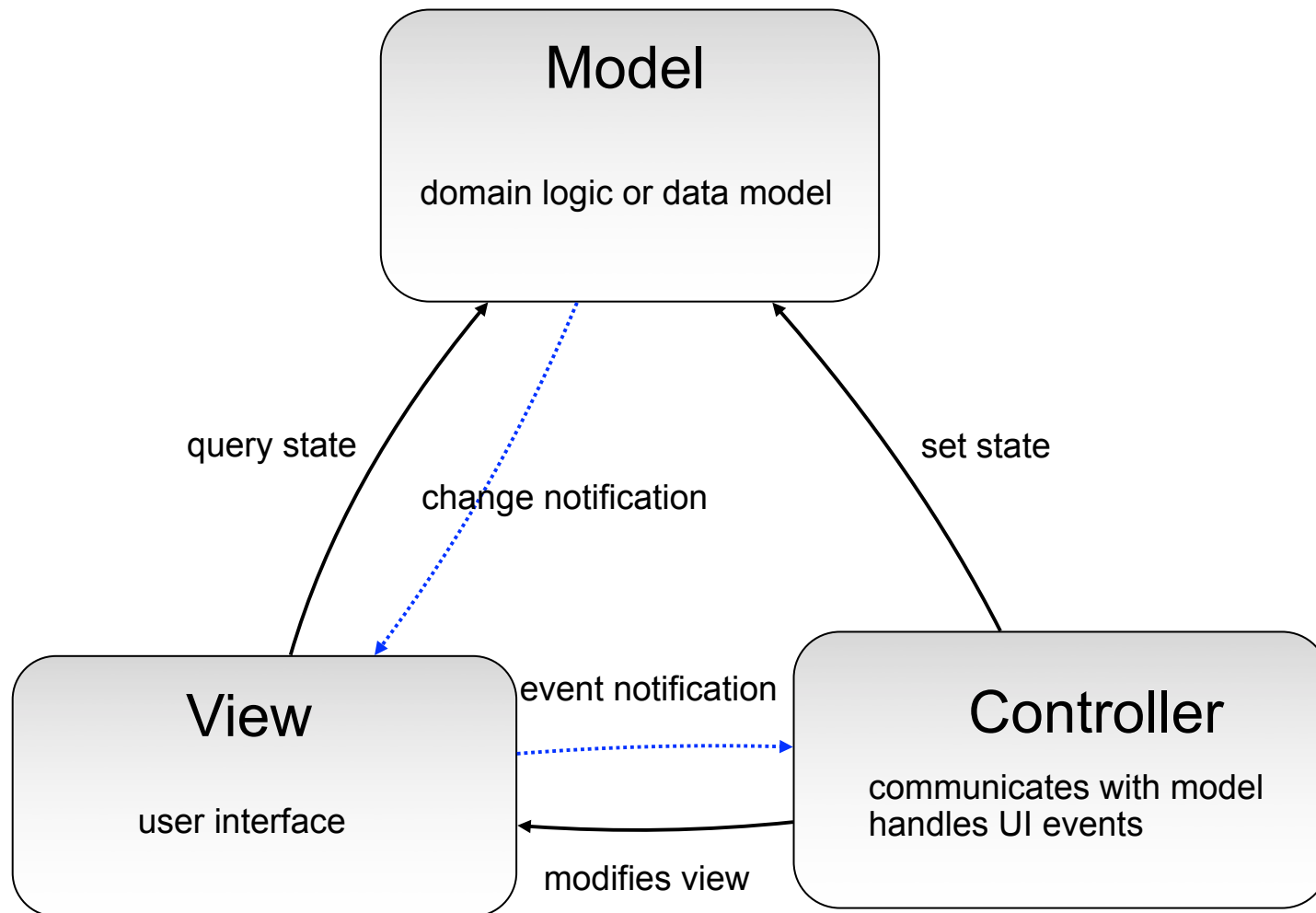
# Participants

39

- Class **Singleton** defines a static member function that lets clients access its unique instance.

# Model-View-Controller (MVC)

40





# Patterns Used in MVC

41

- Mediator: to mediate the communications of widgets
  - ▣ The controller
- Observer: to receive event notifications
  - ▣ Model to View, View to Controller
  - ▣ Async in nature
- Command: to encapsulate the action as objects
  - ▣ Action taken on event notifications

# Mediator

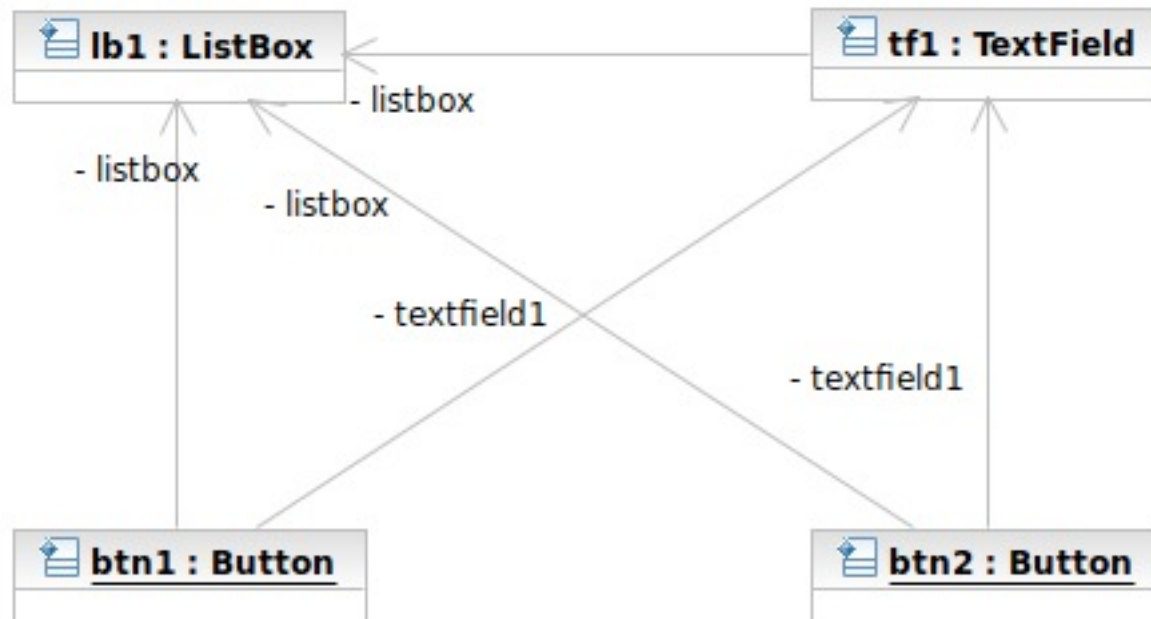
42

- What it is
  - ▣ An object acting as a “hub”
  - ▣ Defines how a set of objects (colleagues) interacts
  - ▣ So colleagues don't have to refer to each other
- Target problem
  - ▣ Different widgets have to act in response to each other
  - ▣ Storing references in widgets is inflexible

# Without the Mediator Pattern

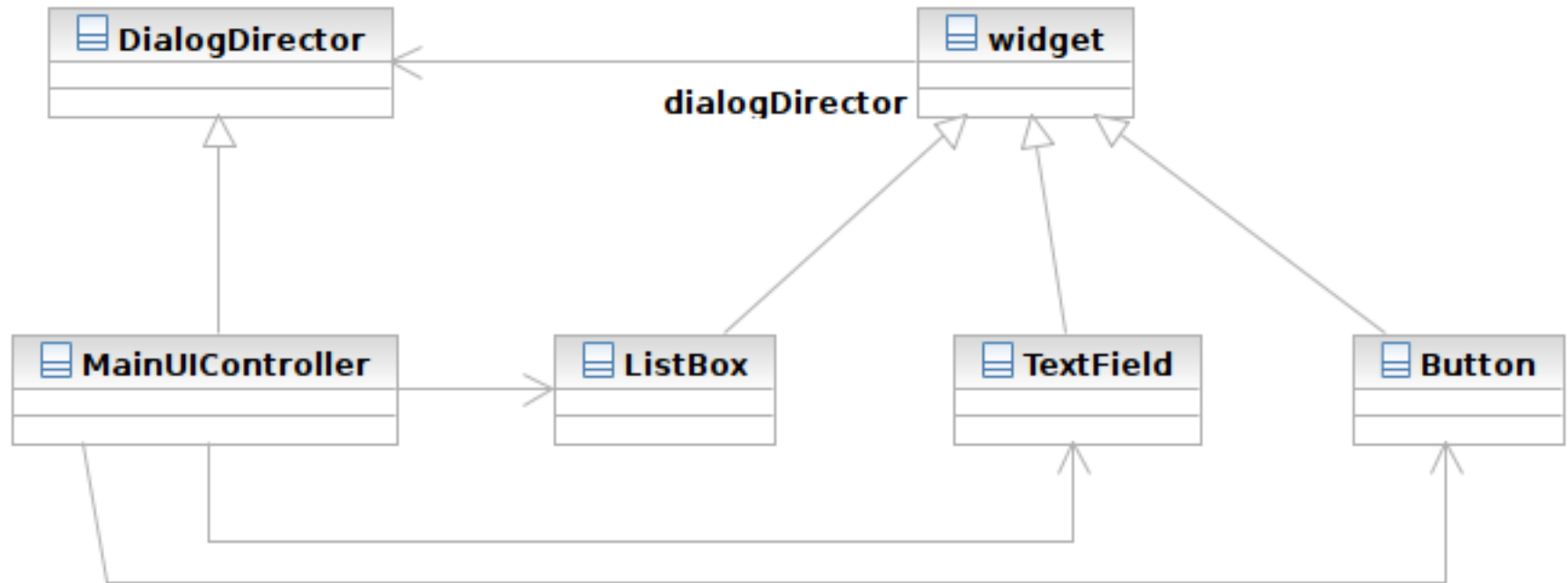
43

- Each concrete widget refers to other widgets to interact with



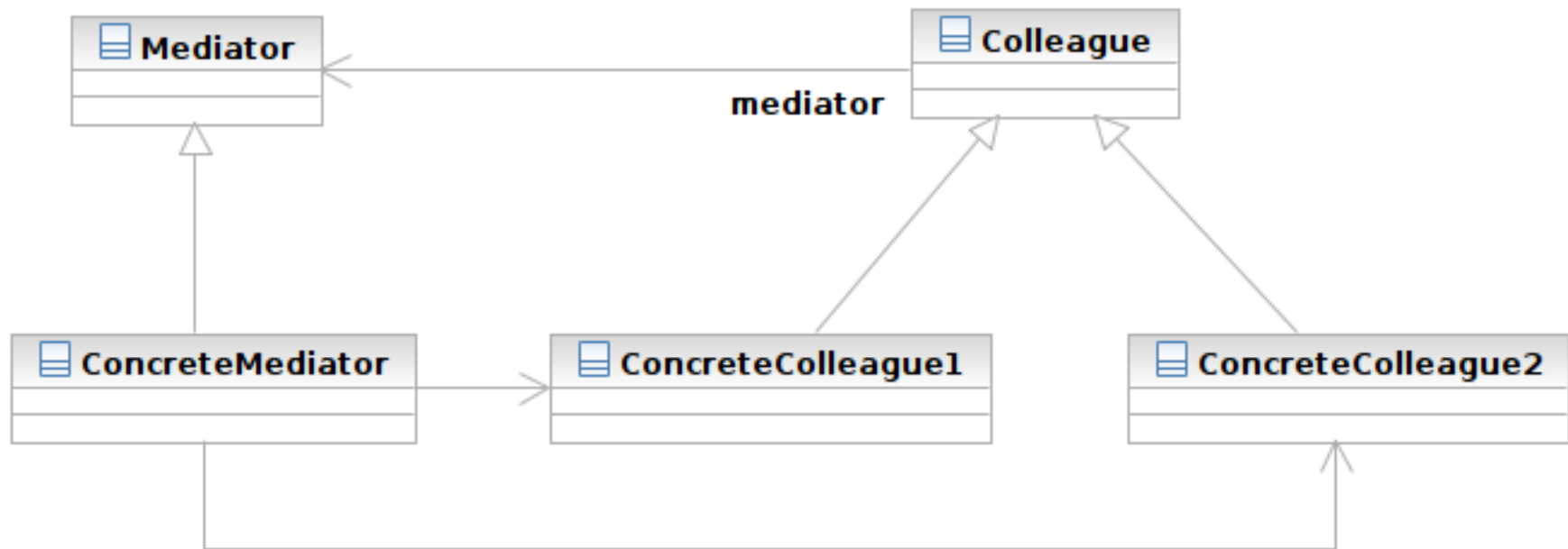
# Applying the Pattern

44



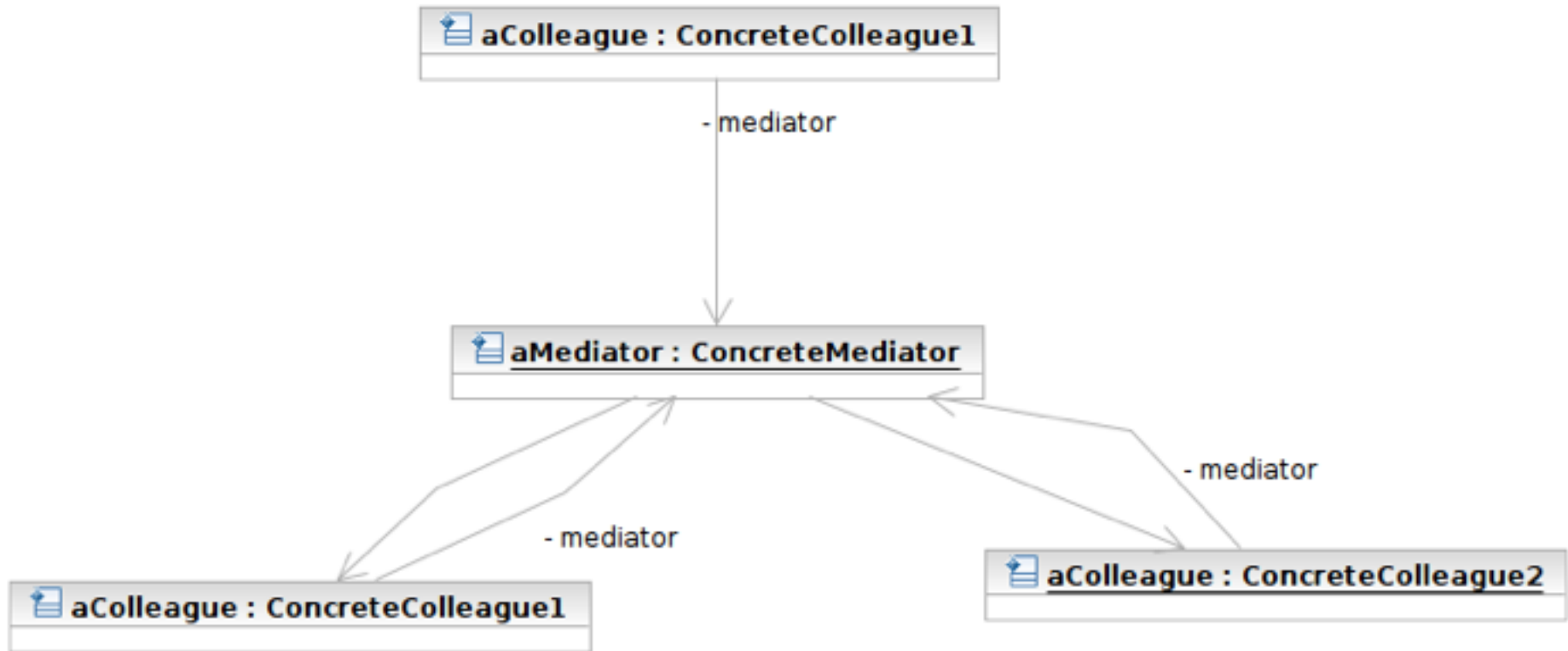
# Structure

45



# Structure

46



# Participants

47

- 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

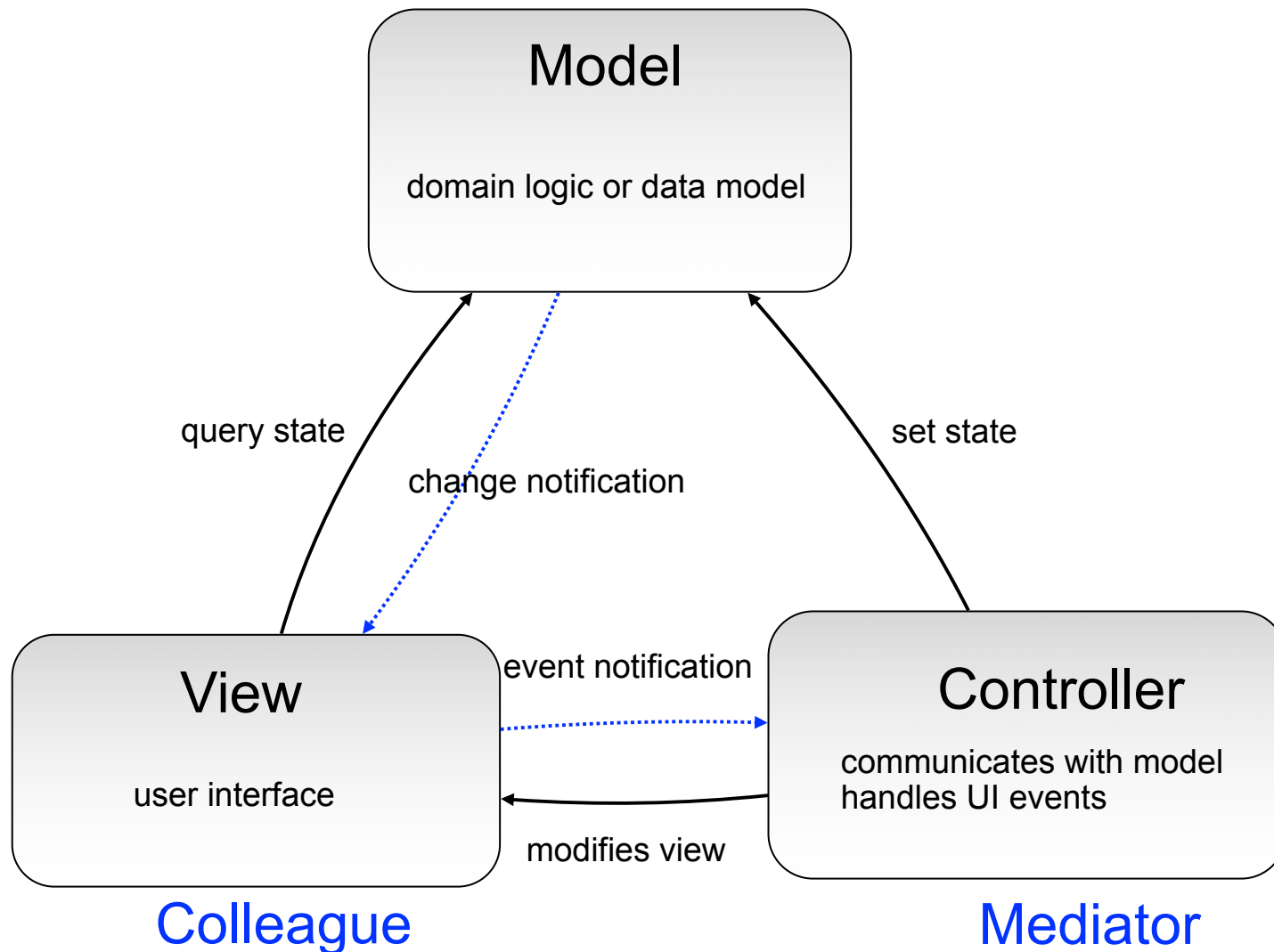
48

- 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



# MVC and Mediator Pattern

49



# Observer

50

- What it is
  - ▣ A one-to-many dependency between objects
  - ▣ Allowing the registrant objects (observers) to be notified
  - ▣ When the something interesting to them happens in the notifier (subject)
- Target Problem
  - ▣ An object should react to some (often async) event
  - ▣ e.g. instant message dialog
  - ▣ Polling is a not a good solution

# Without the Observer Pattern

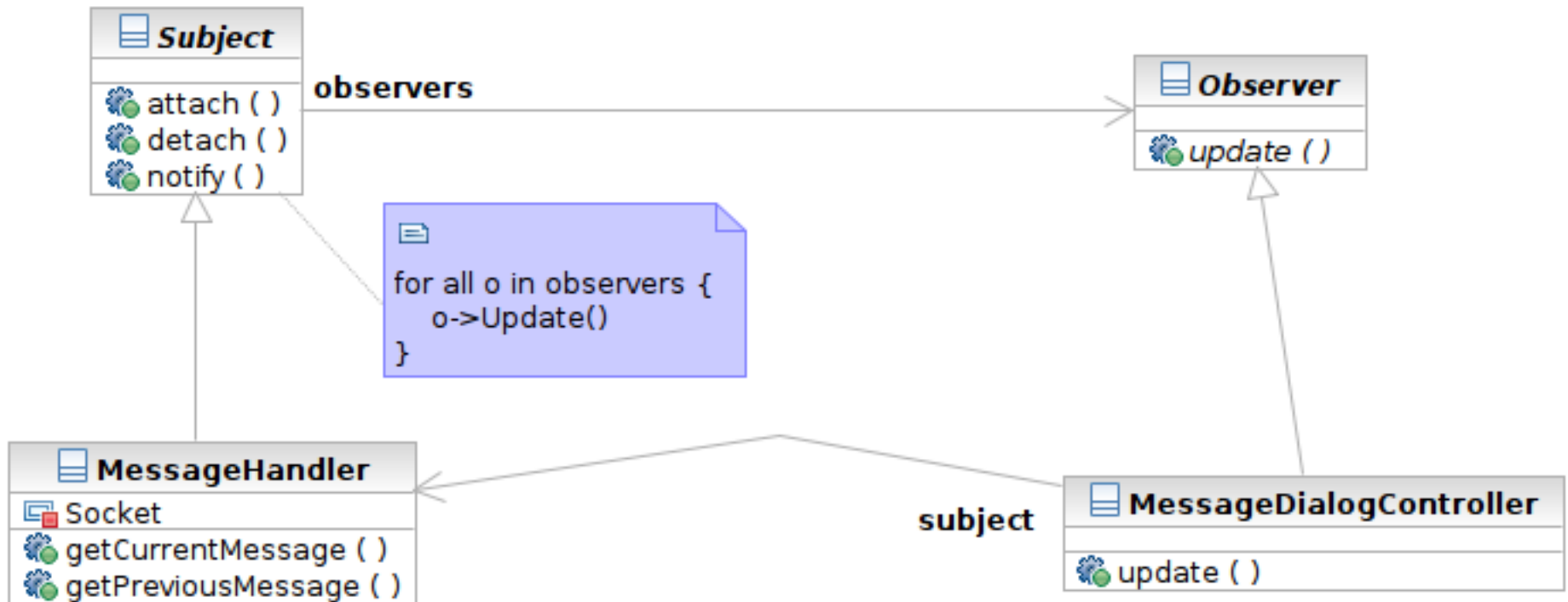
51

- The observer has to continuously query the subject
- The polling approach

```
While (! aSubject.hasChangedState()) {  
  
}  
// now aSubject has changed its state
```

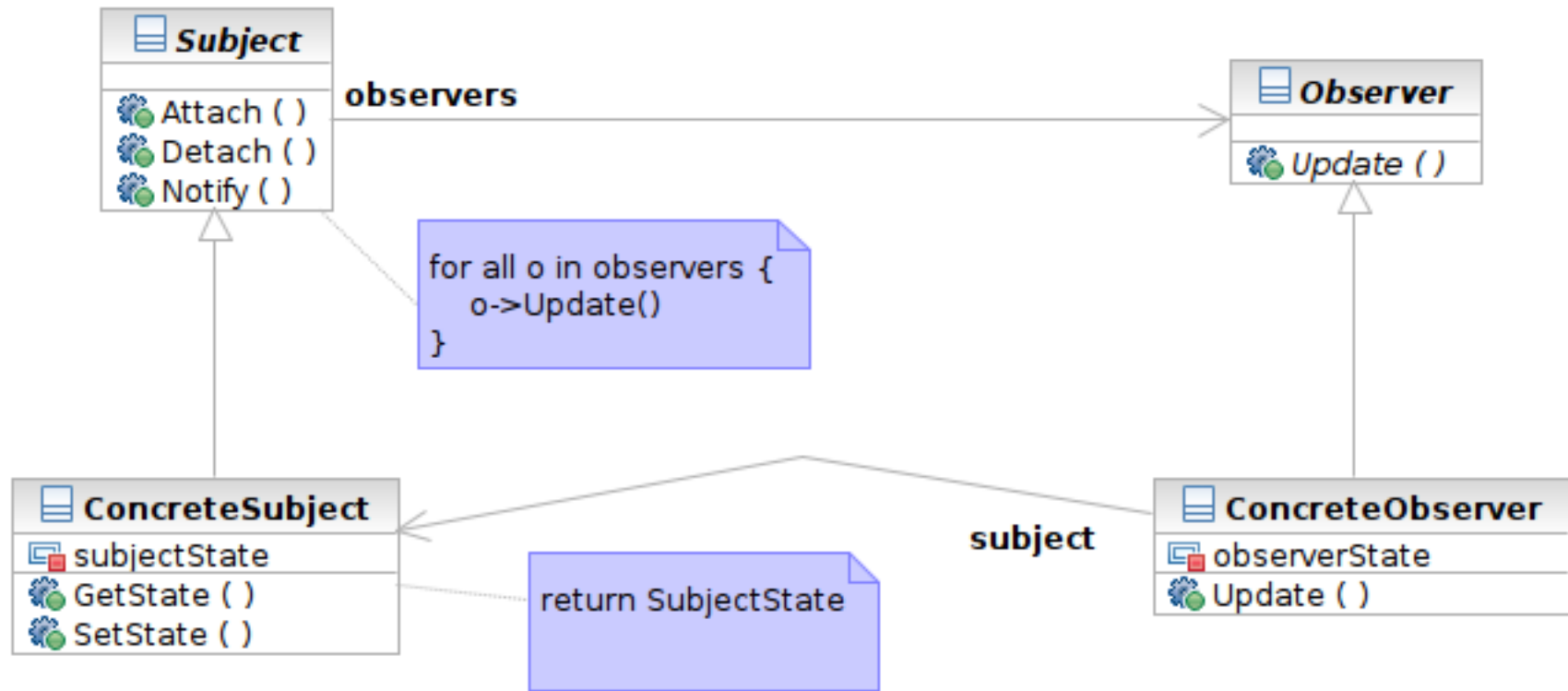
# Applying the Pattern

52



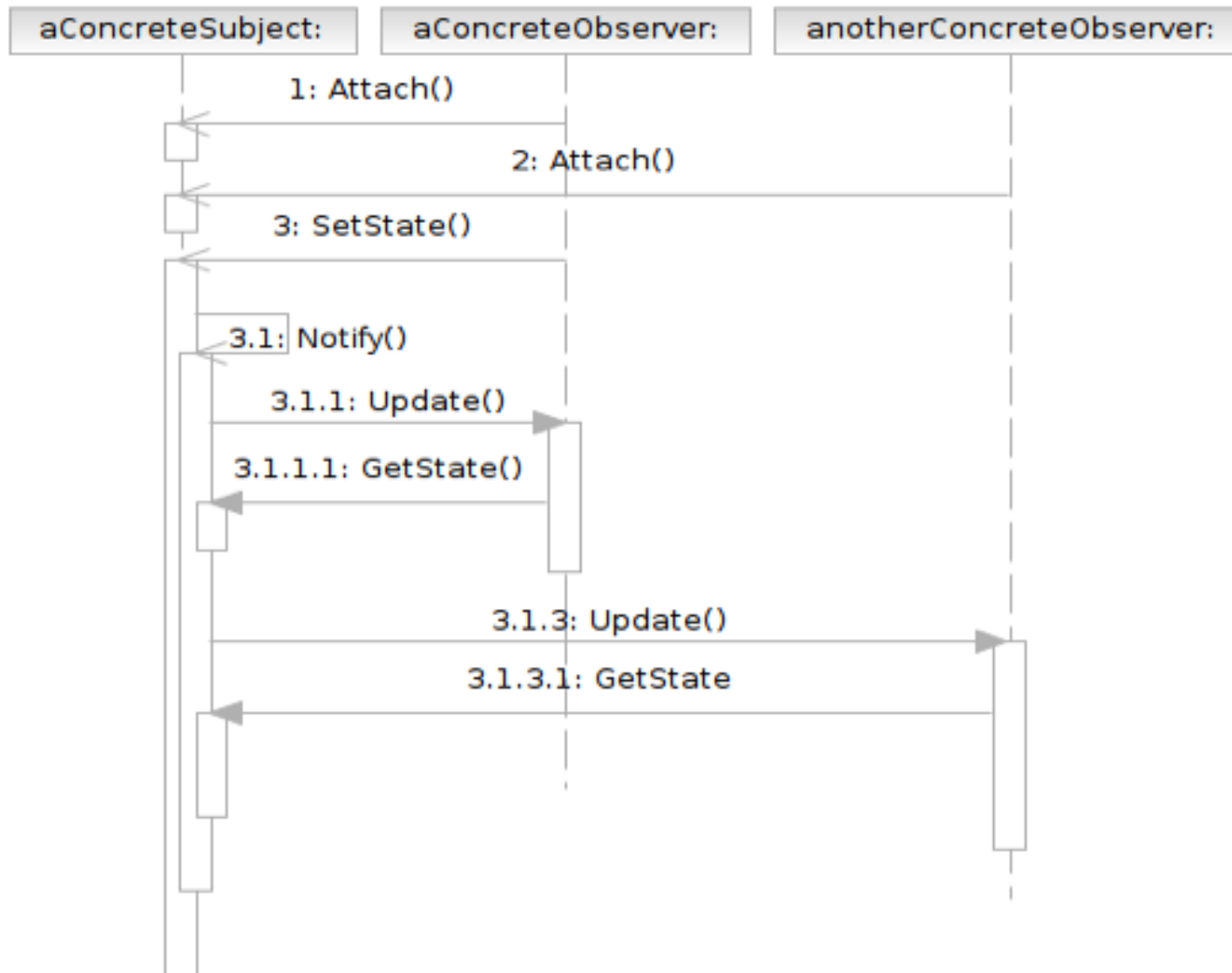
# Structure

53



# Interaction

54



# Participants

55

- 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

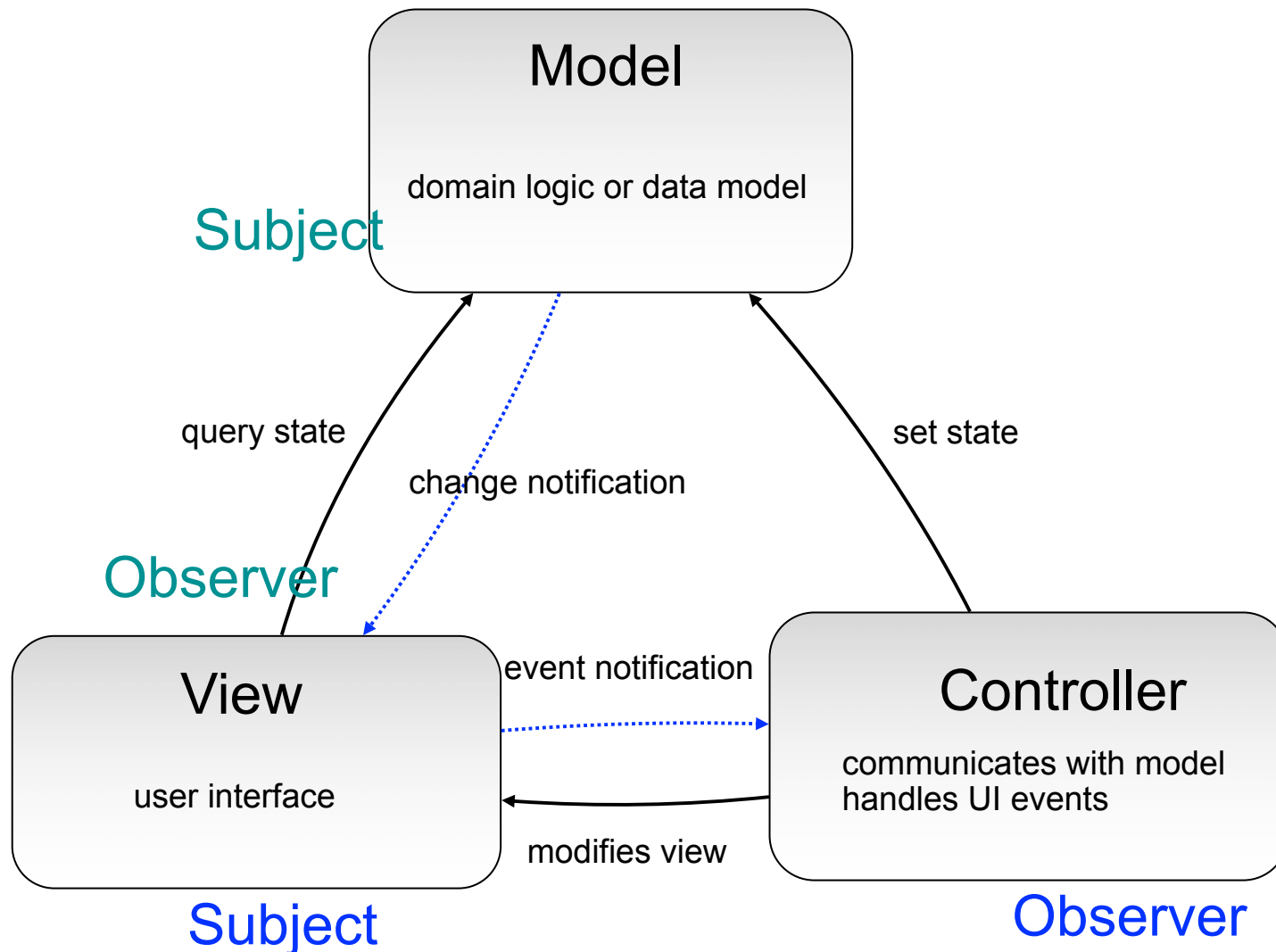
56

- 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



# MVC and Observer Pattern

57



# Command

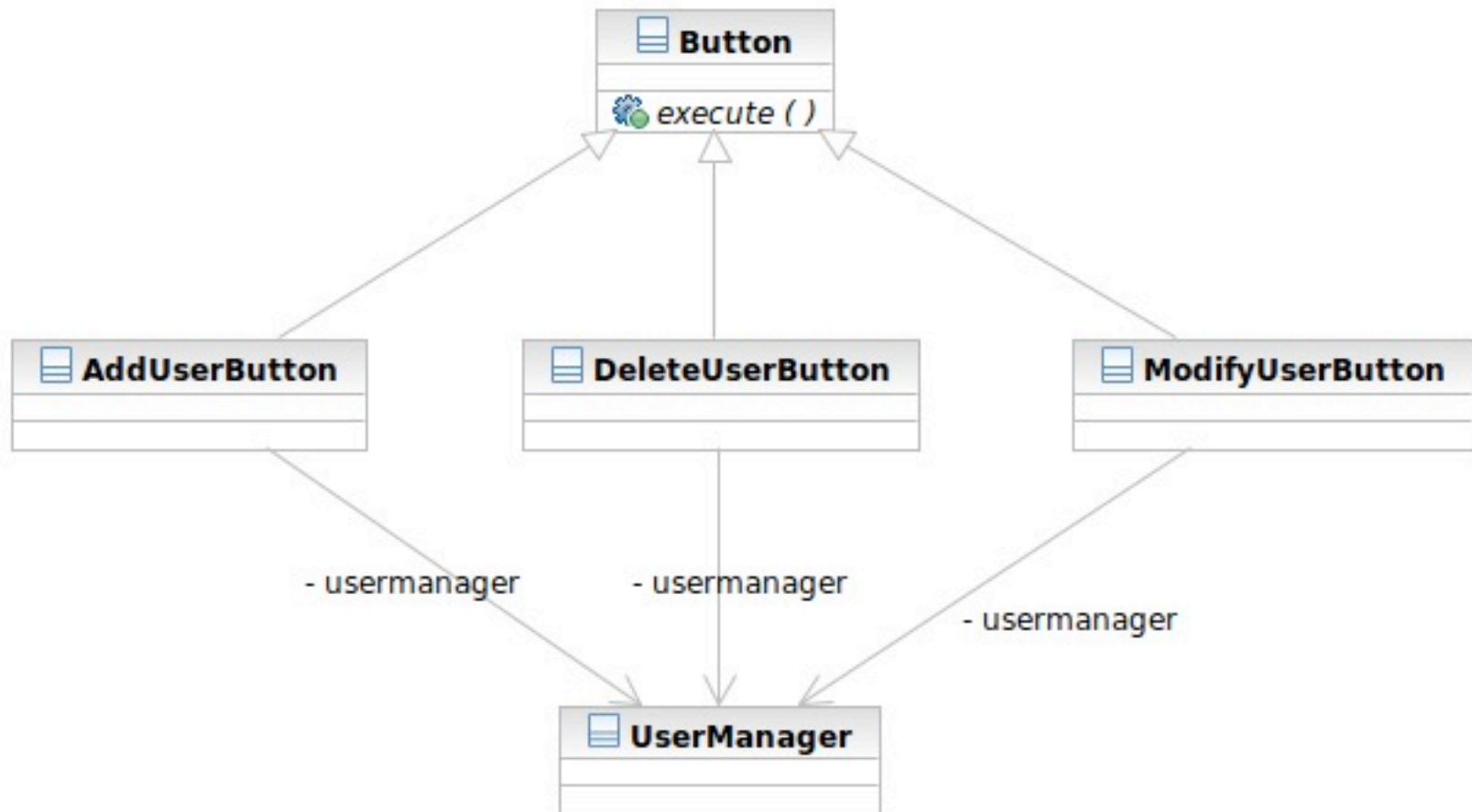
58

- What it is
  - ▣ An **action** encapsulated as an **object**
  - ▣ To be executed later by another client
  - ▣ Can be queued or composed
- Target problem
  - ▣ Customize the behavior of reusable widgets
  - ▣ Subclassing is not a good solution
    - You will have many derived class only to define custom behavior
    - classes for Delete Button, Delete Menu Item, Add Button, Add Menu Item

# Without the Command Pattern

59

- A subclass for each widget instance



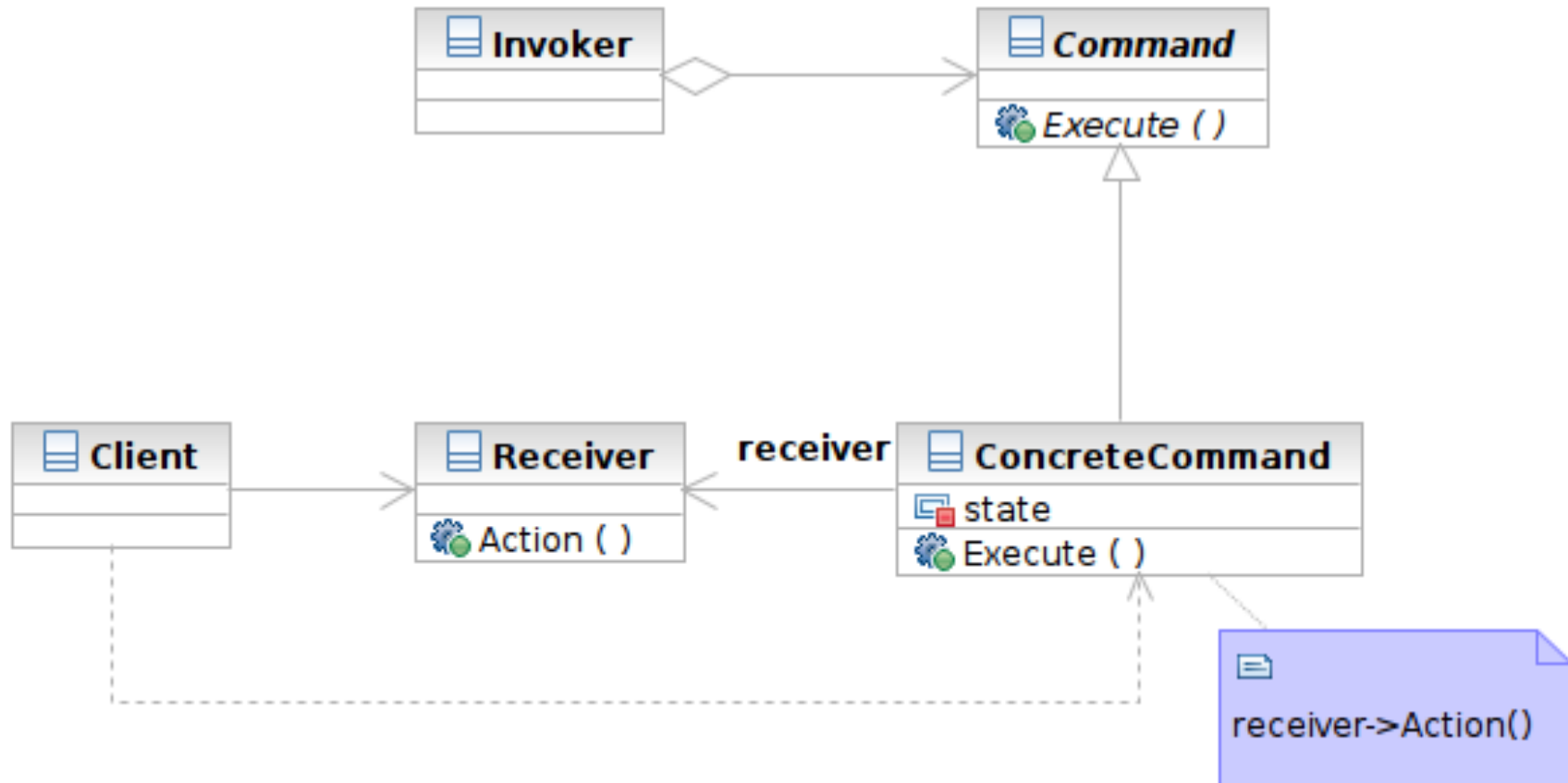
# Applying the Pattern

60



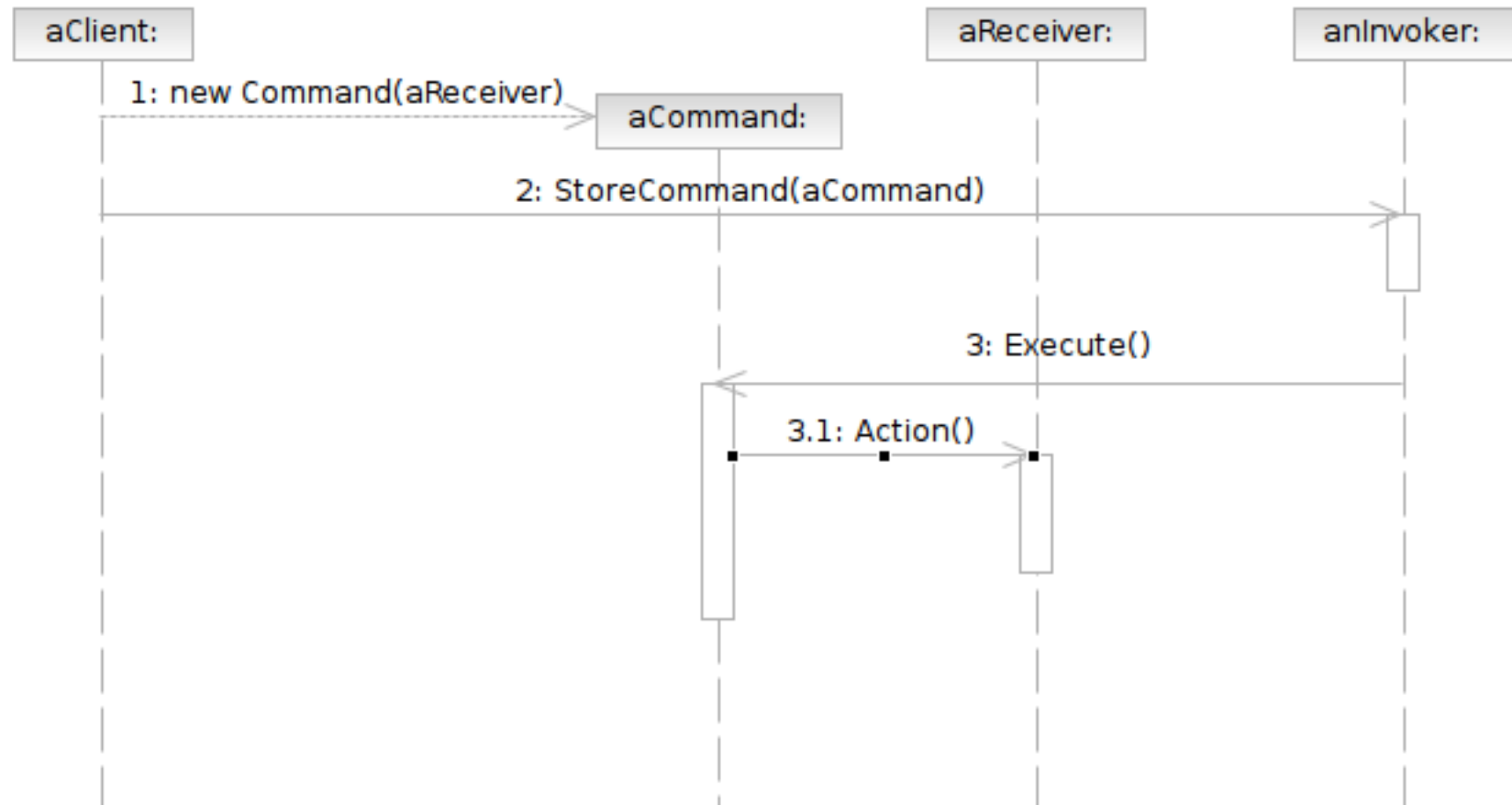
# Structure

61



# Interaction

62



# Participants

63

- 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

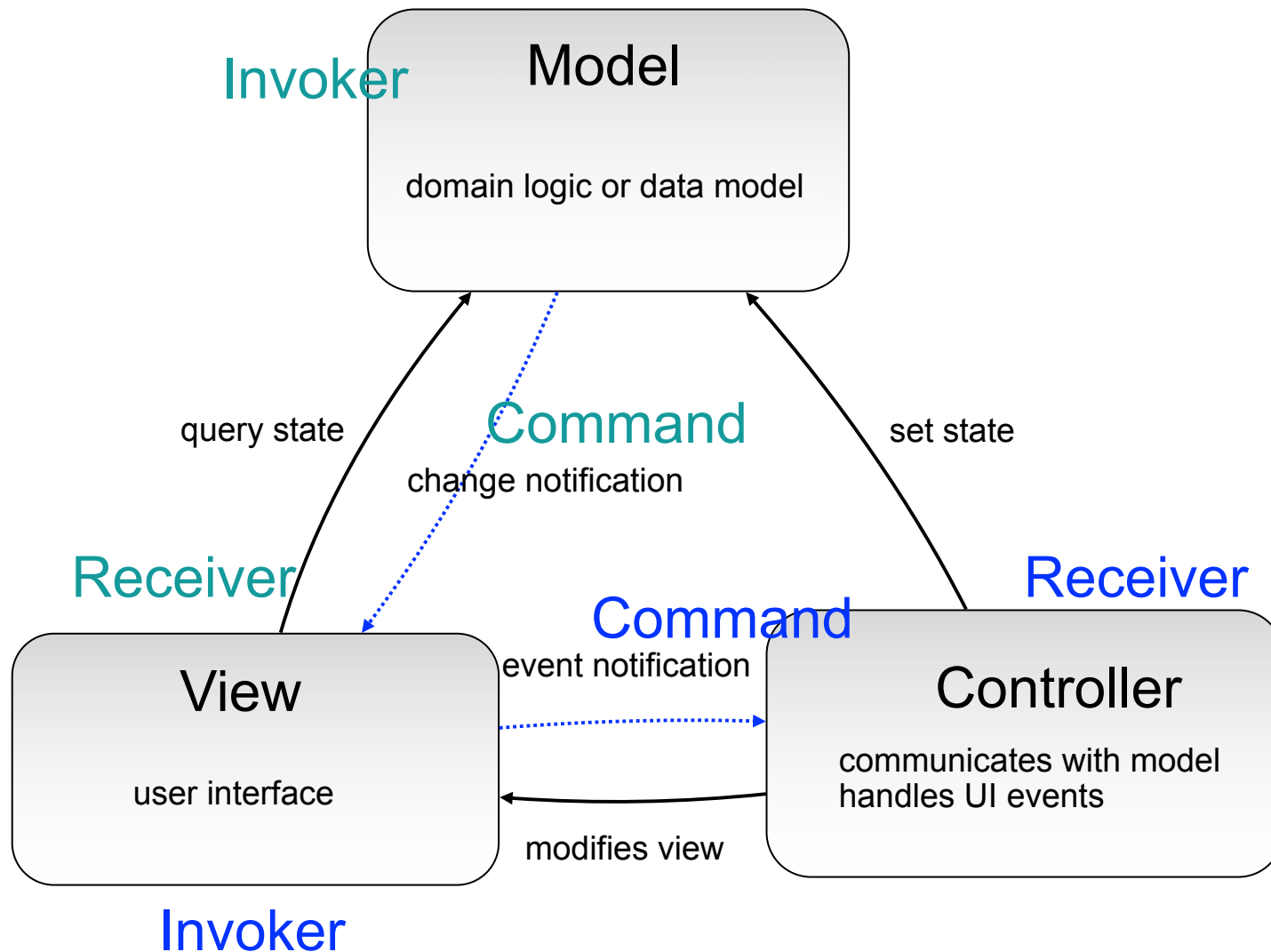
64

- 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



# MVC and Command Pattern

65



# Template Method & Factory Method

66

- What Template Method is
  - ▣ A method that serves as the ‘skeleton’ or structure of an algorithm
  - ▣ Abstract methods called by the template method is implemented in derived classes
- Target problems
  - ▣ Client profile validators for different countries
  - ▣ The generic quick sort algorithms for user-defined classes

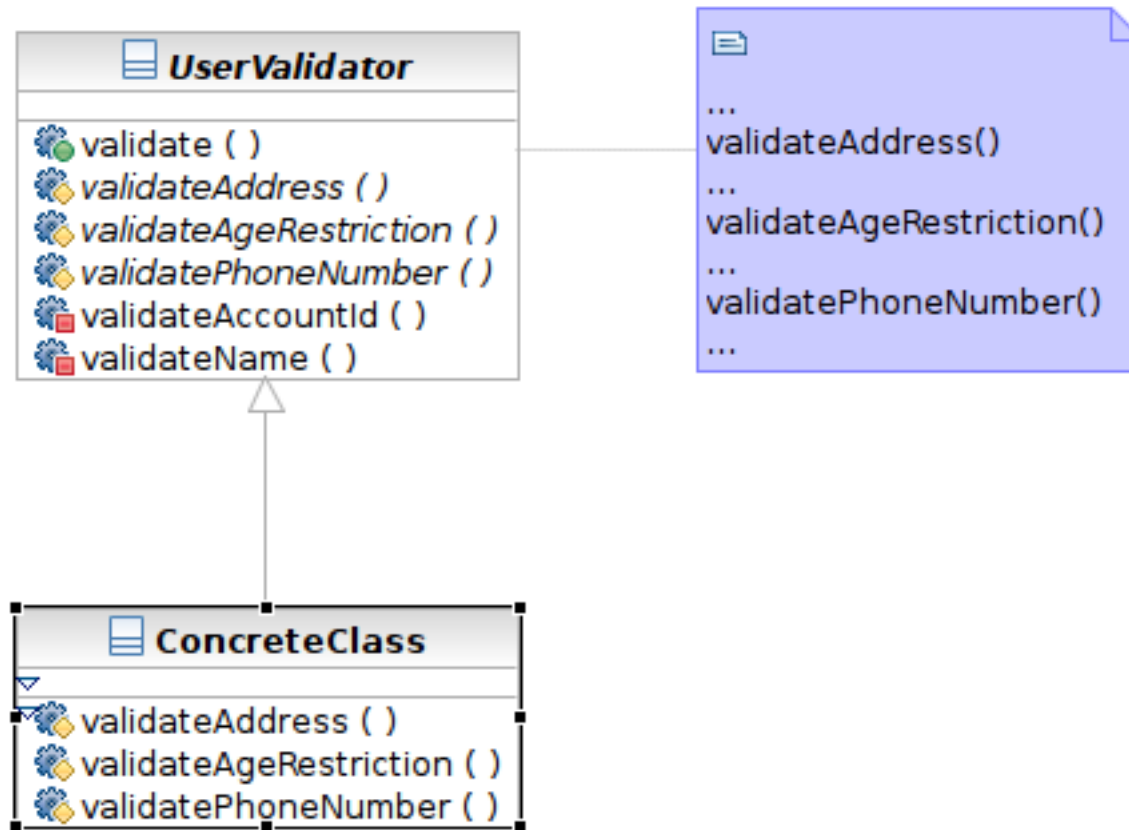
# Without the Template Method Pattern

```
ValidateUSUser () {  
    // validate account id  
    // validate name  
    // validate age restriction (US)  
    // validate phone number (US)  
    // validate address (US)  
}
```

```
ValidateTWUser () {  
    // validate account id  
    // validate name  
    // validate age restriction (TW)  
    // validate phone number (TW)  
    // validate address (TW)  
}
```

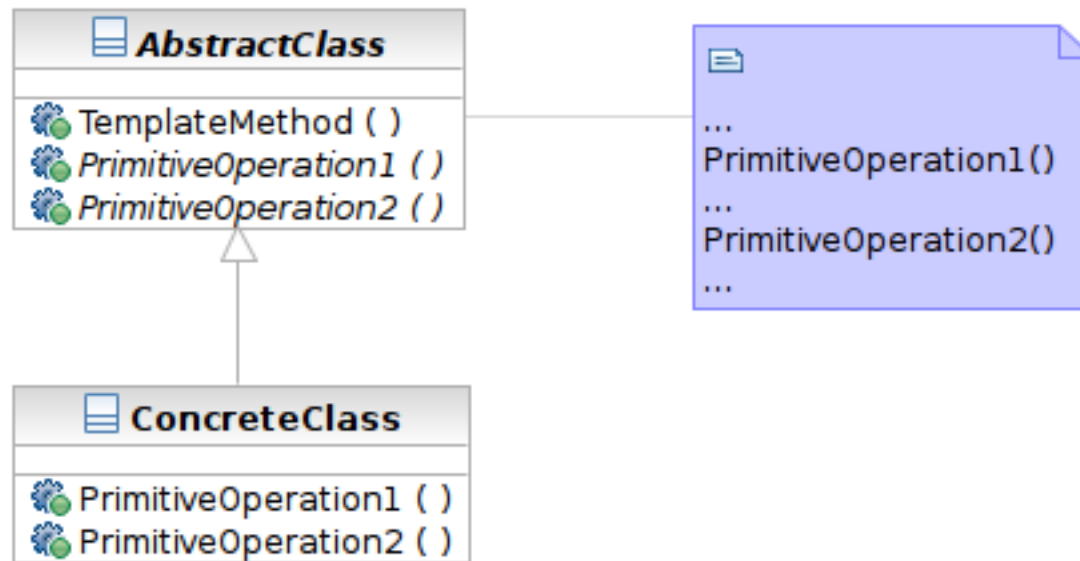
# Applying the Pattern

68



# Structure

69



# 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.

# Factory Method

- What it is
  - ▣ A method that instantiates a concrete class when called
  - ▣ Often called in template method

# Structure

## Product

defines the interface of objects created by factory method

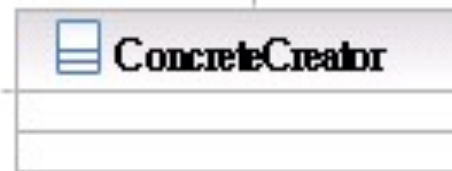
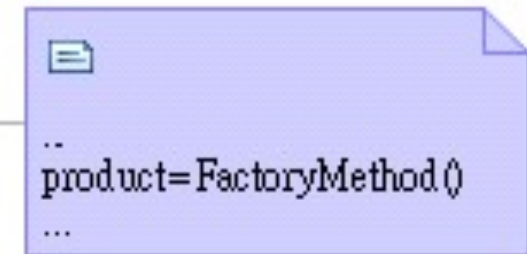
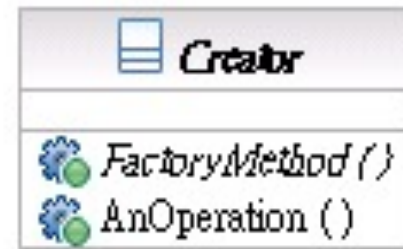


## ConcreteProduct

implements the Product interface

## Creator

declares the factory method returning an object of type Product



## ConcreteCreator

overrides the factory method to return an instance of a ConcreteProduct

«instantiate»



# Transparent Access: Proxy & Decorator

73

- The 2 are similar in structure but for different purposes
- Proxy focuses on **controlling the access** of an object
- Decorator is used to **'decorate'** (adding more functionality) to an object dynamically

# Proxy

74

- What it is
  - ▣ A surrogate or placeholder for another object to control access to it
  - ▣ In a **transparent** way (having the same interface as the proxied object)
- Target problem
  - ▣ Access control between the client and your system, such as
  - ▣ Lazy loading of image or other resources
  - ▣ Transparent access to remote objects

# Without the Proxy Pattern

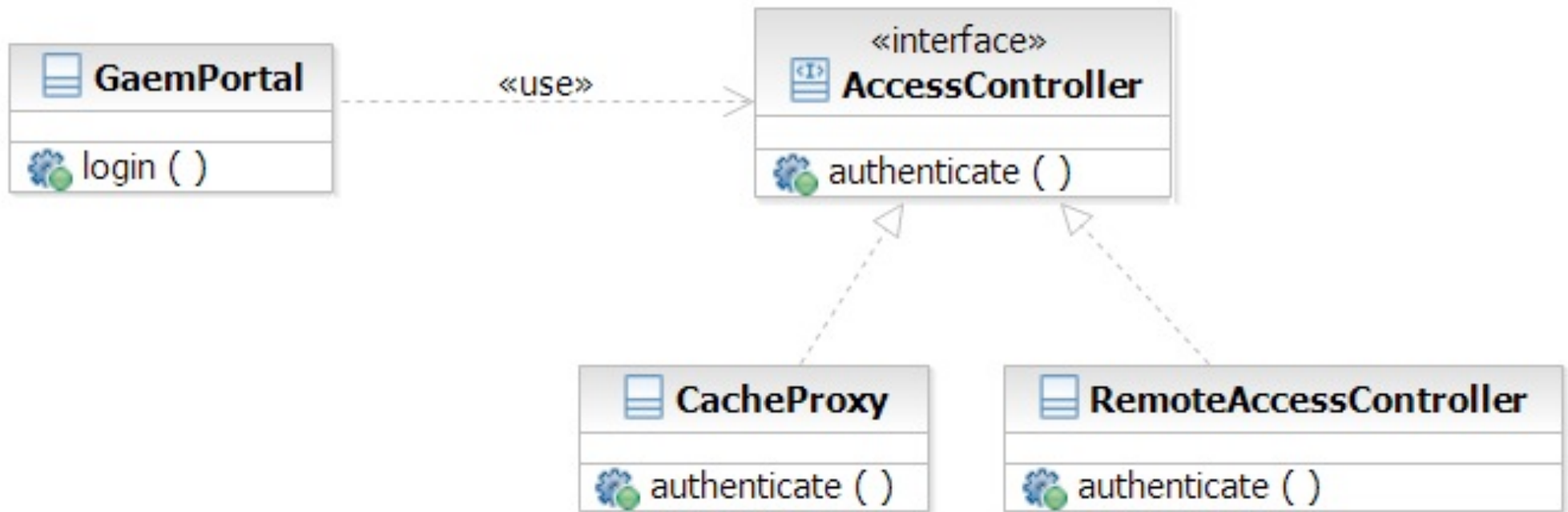
75

- The condition needs to be coded in the proxied class

```
// find cached authentication information
AuthInfo auth = FindCachedAuthInfo();
If (auth != NULL) {
    // already cached. Return authentication info here
}
Else {
    // perform authentication with remote server
}
```

# Applying the Pattern

76



# Decorator

77

- What it is
  - ▣ Attaching additional responsibilities to an object dynamically
  - ▣ An alternative to subclassing
- Target Problem
  - ▣ Enabling/disabling additional features at runtime
    - Caching, logging
  - ▣ Dynamic composition of these features (subclassing is infeasible)

# Without the Decorator Pattern

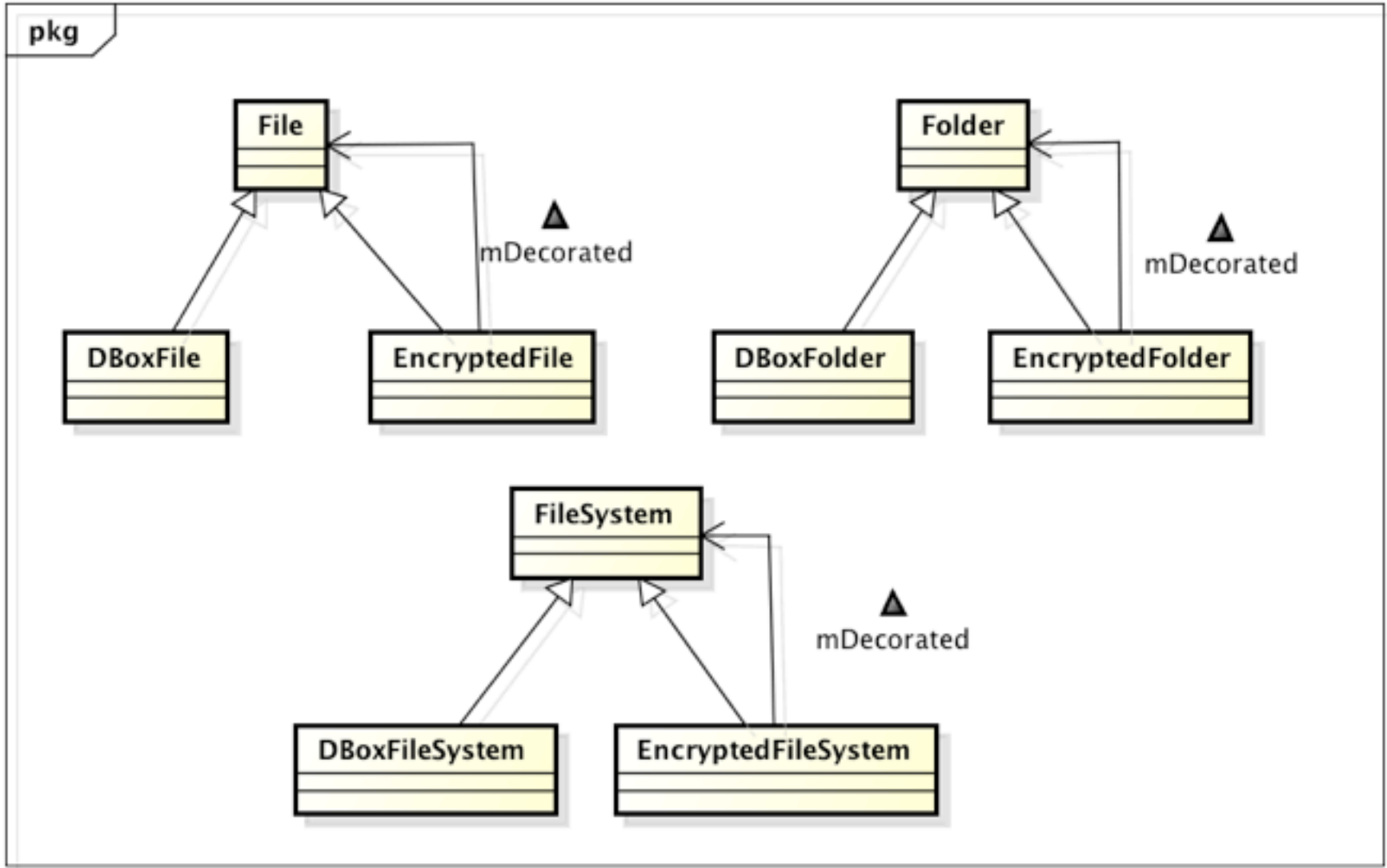
78

- The added functionality needs to be coded in the decorated class:

```
If (decoration1Enabled) {  
    // Perform decoration1 action part 1.  
}  
  
// function body  
  
If (decoration2Enabled) {  
    // Perform decoration2 action.  
}  
  
If (decoration1Enabled) {  
    // Perform decoration1 action part 2.  
}
```

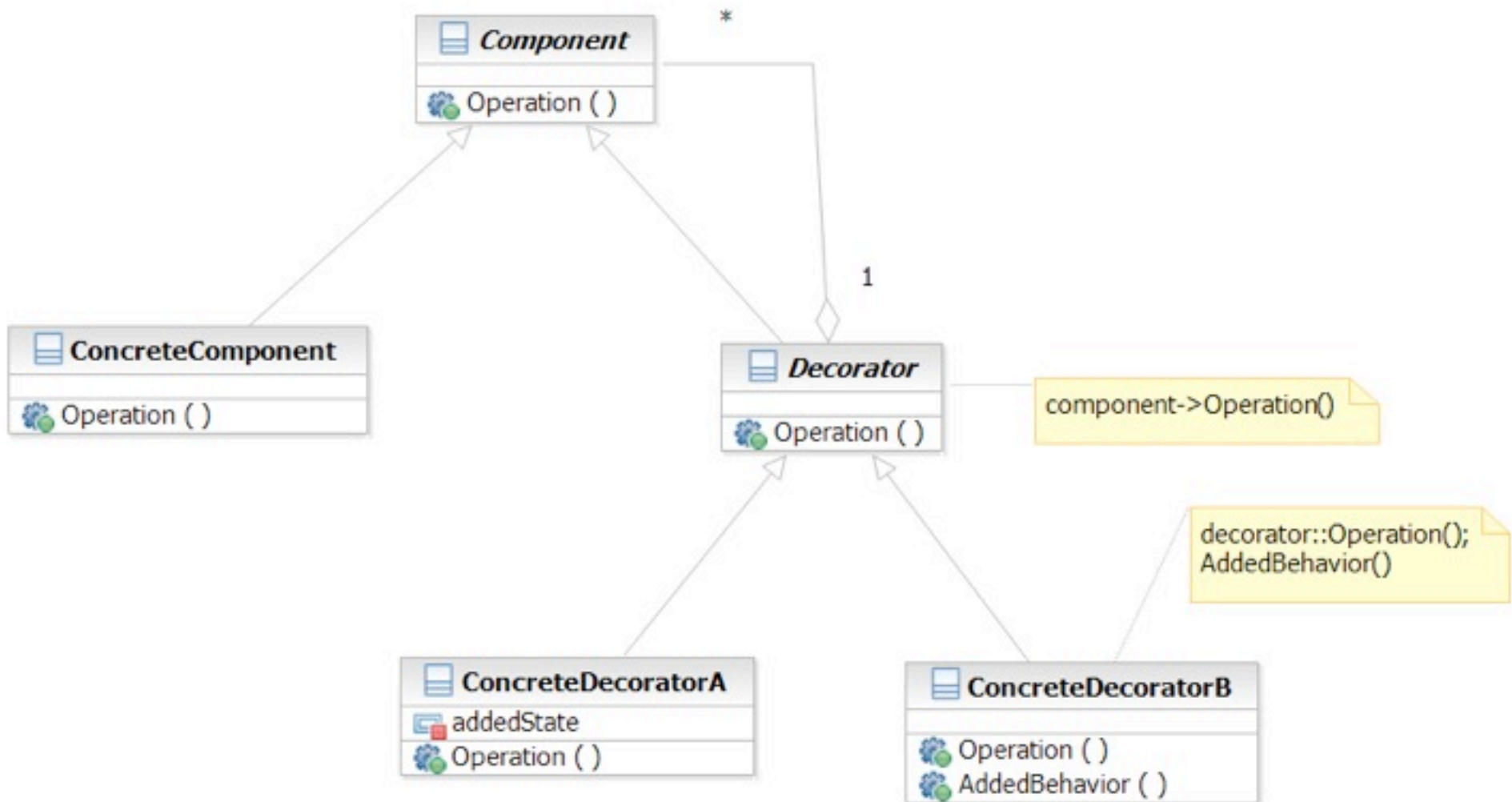
# Applying the Pattern

79



# Structure

80





# State

81

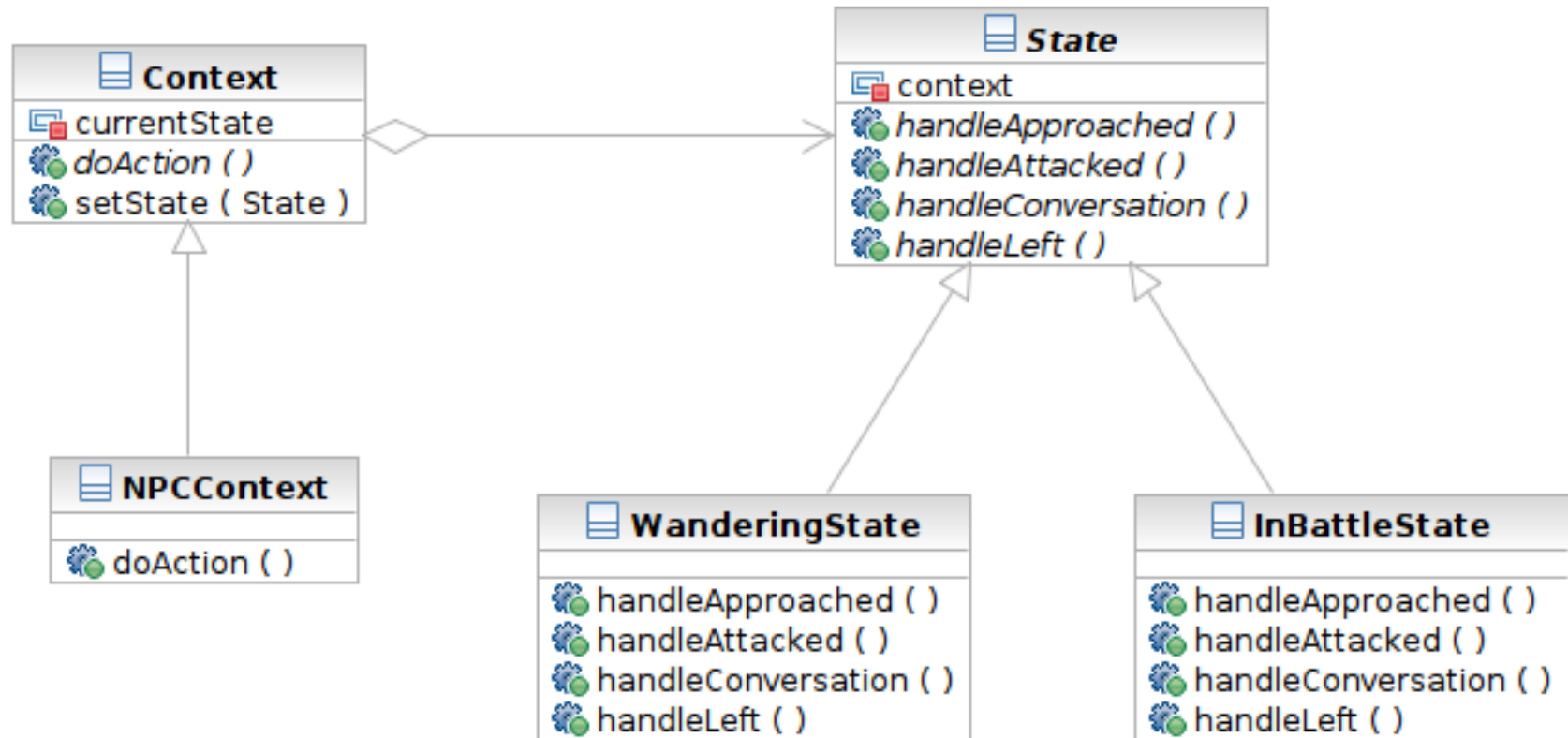
- What it is
  - ▣ Allowing an object to change its behavior when its internal state changes
- Target Problem
  - ▣ State machines
    - Network protocols (e.g. TCP state machine)
    - Drawing tools
    - Document editors
    - Games
    - Complex business rules

# Without the State Pattern

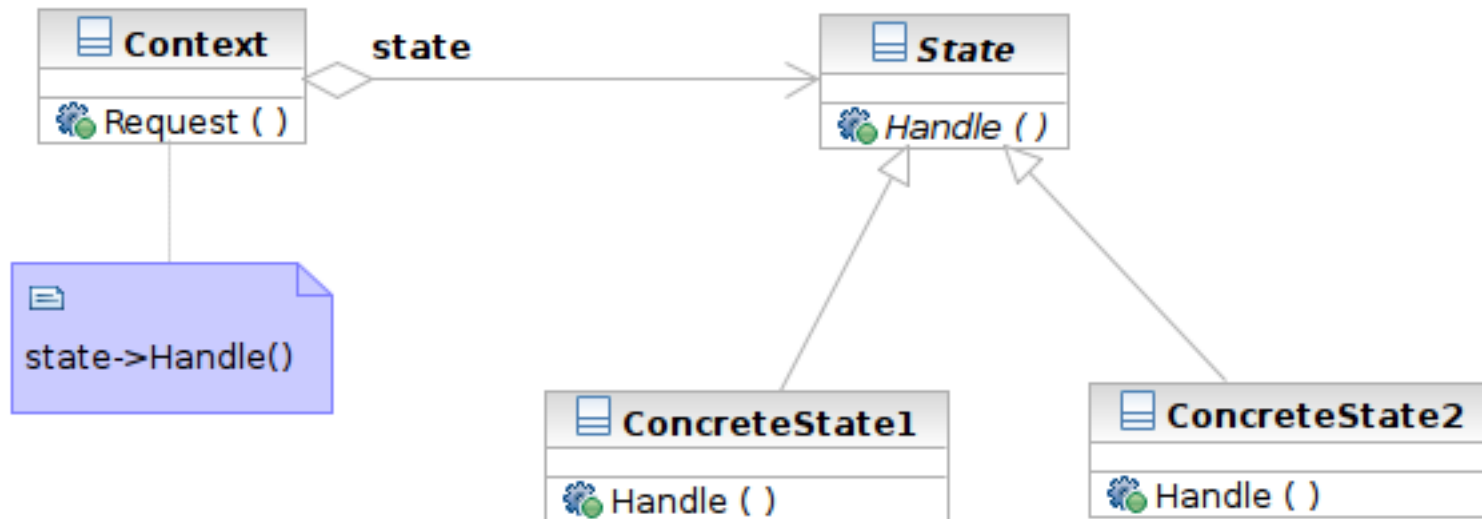
- Use if or switch structure to produce lengthy functions

```
switch (character.getState()) {  
  case wandering:  
    // character is wandering  
    break;  
  case battle:  
    // in battle and behaves aggressively  
    break;  
  default:  
    break;  
}
```

# Applying the Pattern



# Structure



# Participants

85

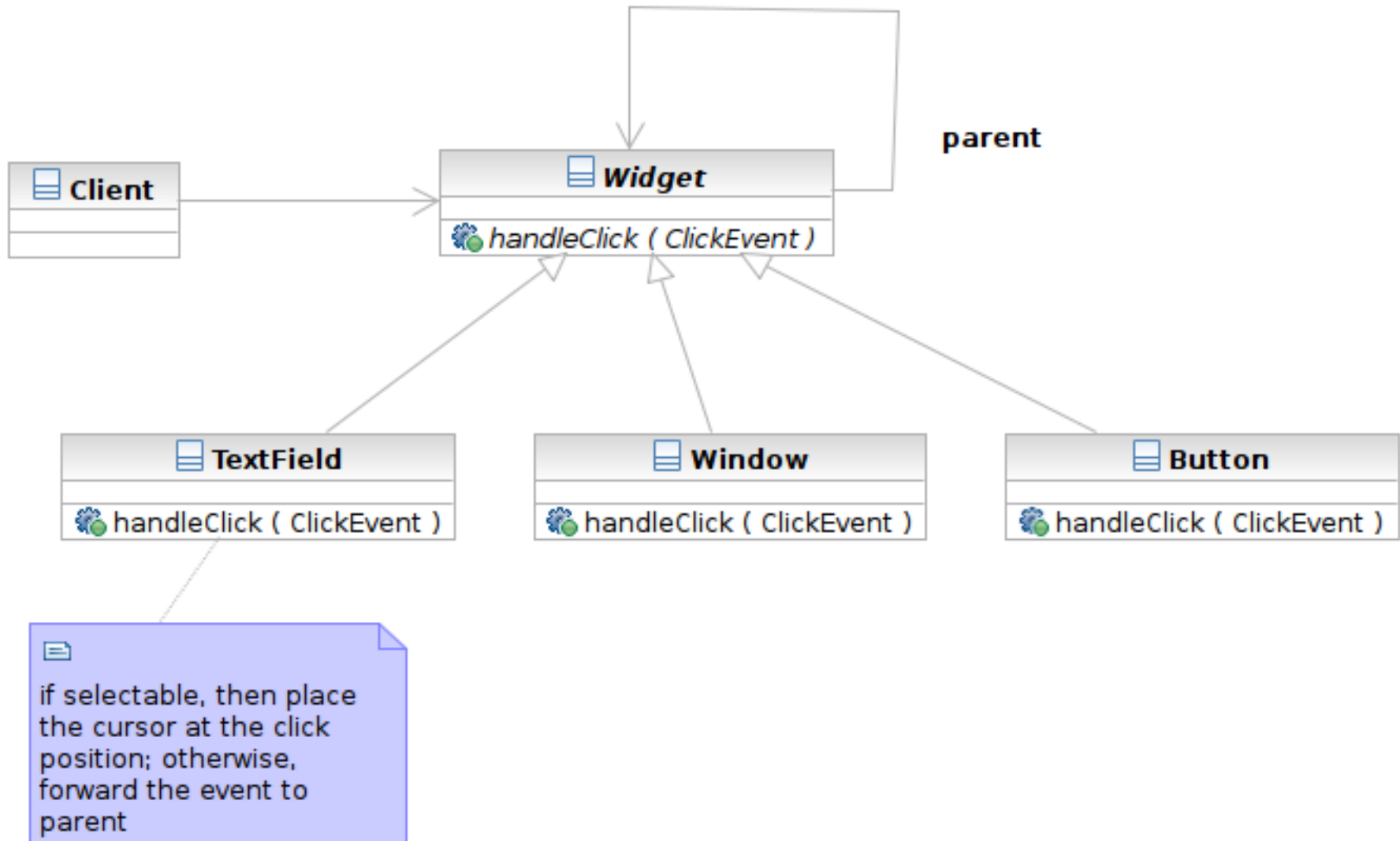
- Class **Context** defines the interface to client and maintains an instance of a **ConcreteState** subclass.
- Class **State** defines an interface for encapsulating the behavior associated with a particular state of the **Context**.
- Class **ConcreteState** subclasses implement a behavior associated with a state of the **Context**.

# Chain of Responsibility

86

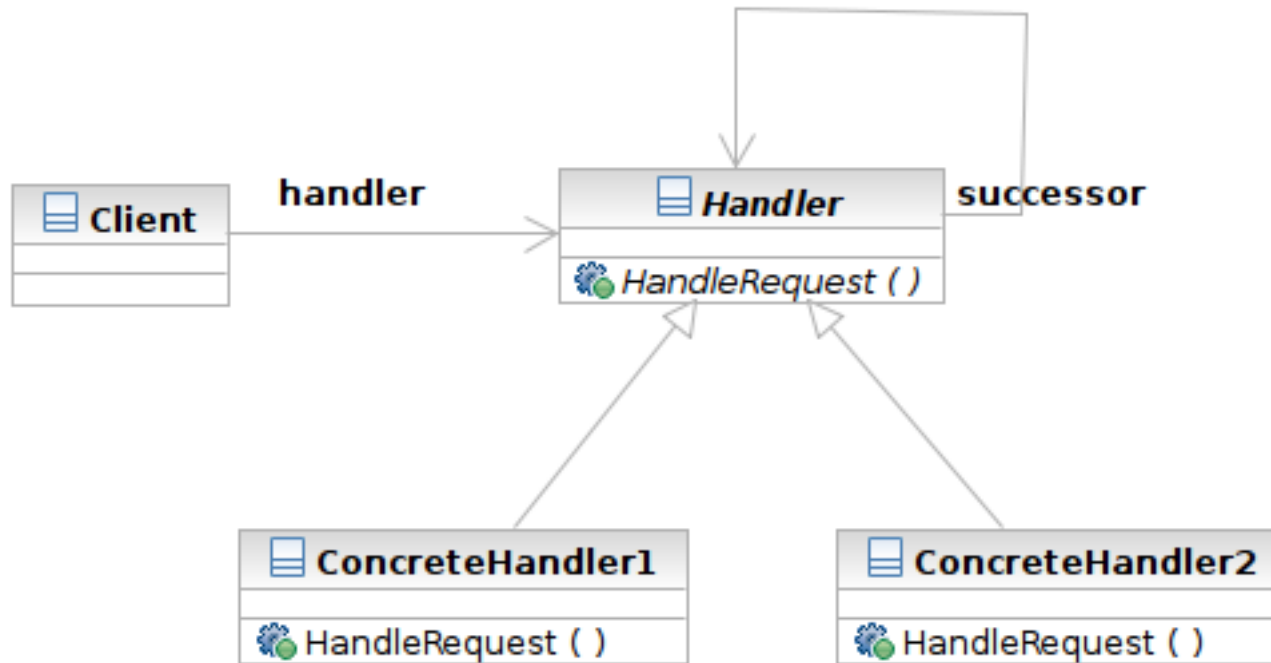
- What it is
  - ▣ Decouple the request sender and handler by chaining the possible handlers and passing the request along the chain until handled
- Target Problem
  - ▣ Handling the request if multiple objects may take responsibility, but without specifying explicitly which one will
  - ▣ Specifying the object that handles the request dynamically

# Applying the Pattern



# Structure

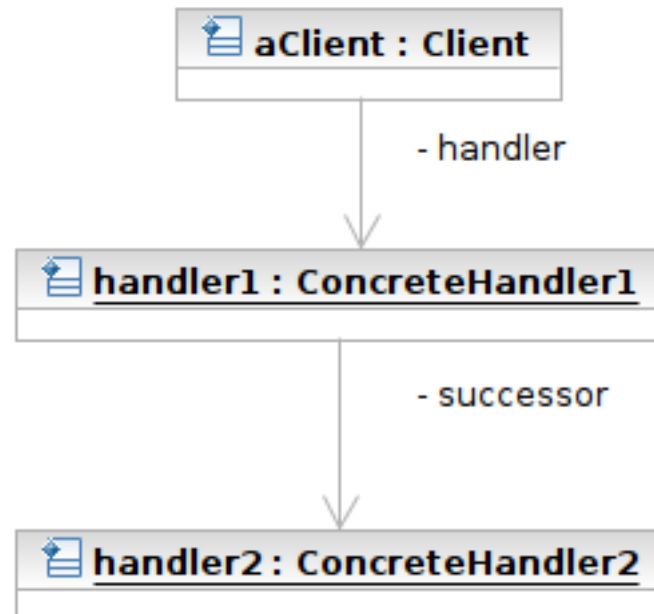
88





# Structure

89



# Participants

90

- 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

# Prototype

91

- What it is
  - ▣ An object that creates other object by ‘cloning’ itself
- Target Problem
  - ▣ Some objects are expensive to instantiate from scratch
  - ▣ Cloning the already instantiated object is cheaper
    - Default user profile stored in database -- no need to retrieve from DB each time when creating a new user.

# Without the Prototype Pattern

(Suppose instantiation of ShoppingCart requires access of remote system, which is expensive)

```
// anonymous user place an item to the shopping cart  
aShoppingCart = new ShoppingCart () // 1000 ms
```

...

# Applying the Pattern

(Suppose instantiation of ShoppingCart requires access of remote system, which is expensive)

```
// anonymous user place an item to the shopping cart  
aShoppingCart = prototype.clone\(\) // 10 ms
```

```
...
```

# Structure

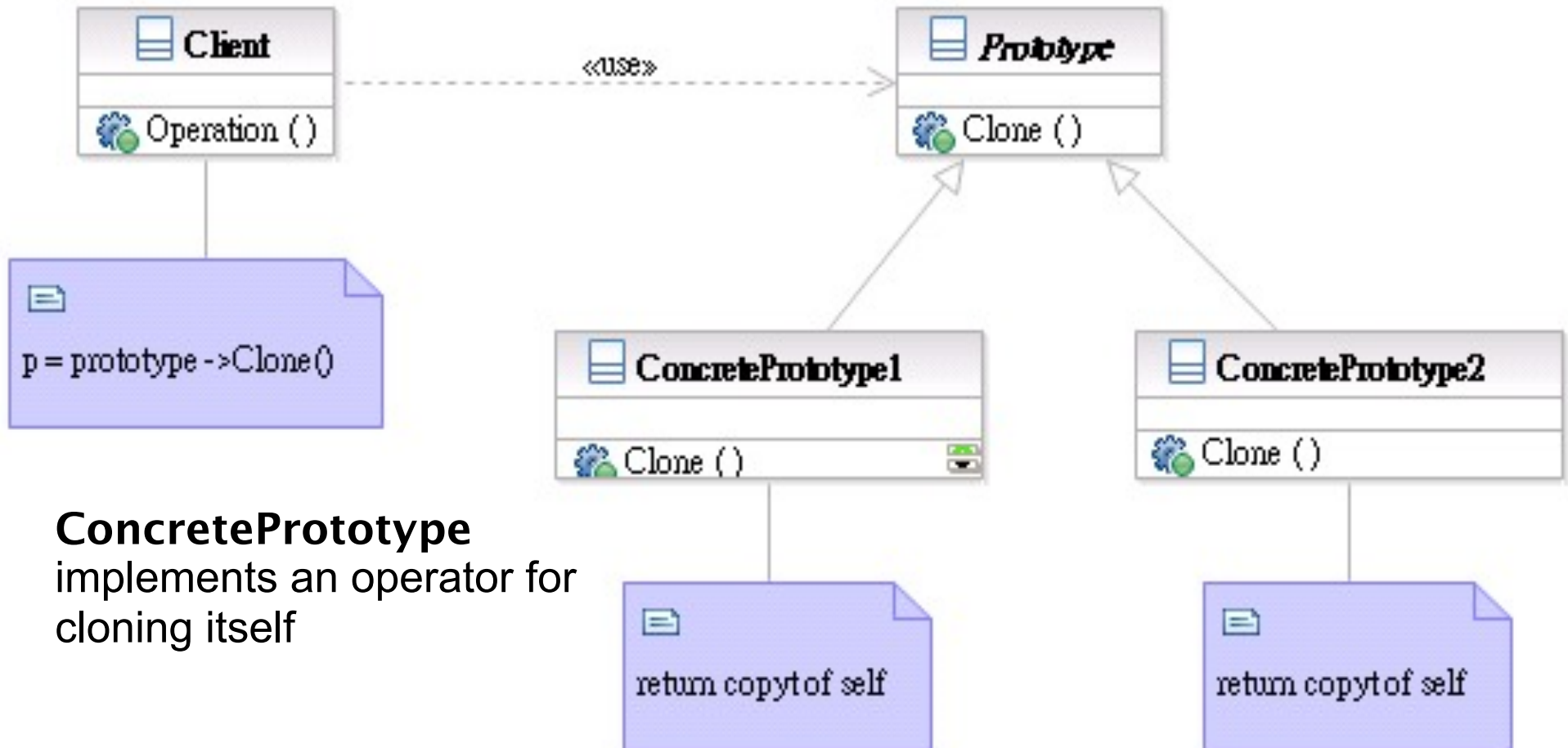
94

## Client

creates a new object by asking a prototype to clone itself

## Prototype

declares an interface for cloning itself



## ConcretePrototype

implements an operator for cloning itself

# Participants

95

- Class **Prototype** declares an interface for cloning itself.
- Class **ConcretePrototype** implements an operator for cloning itself.
- Class **Client** creates a new object by asking a prototype to clone itself.

# Patterns Dealing with Complex Object Hierarchies

96

- Composite: the representation (structure) of the hierarchy
- Builder: to create the representation
- Visitor: to extend the operations that can be applied to the composite structure



# Sample Problem

97

- Cross-platform GUI framework
  - ▣ Widgets have hierarchical structures/representations
  - ▣ Use define the GUI interface with XML
  - ▣ Support native interface (Mac, Linux, Windows) and web interface
  - ▣ Convert the representation to json for AJAX

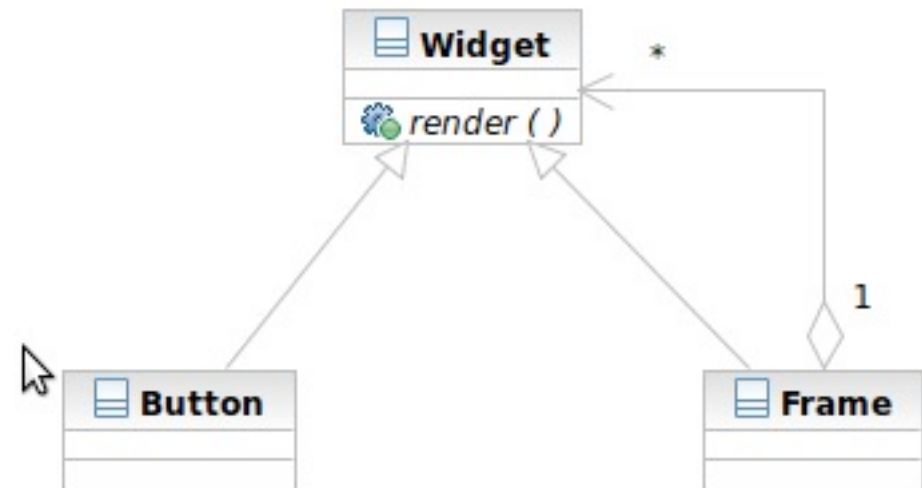
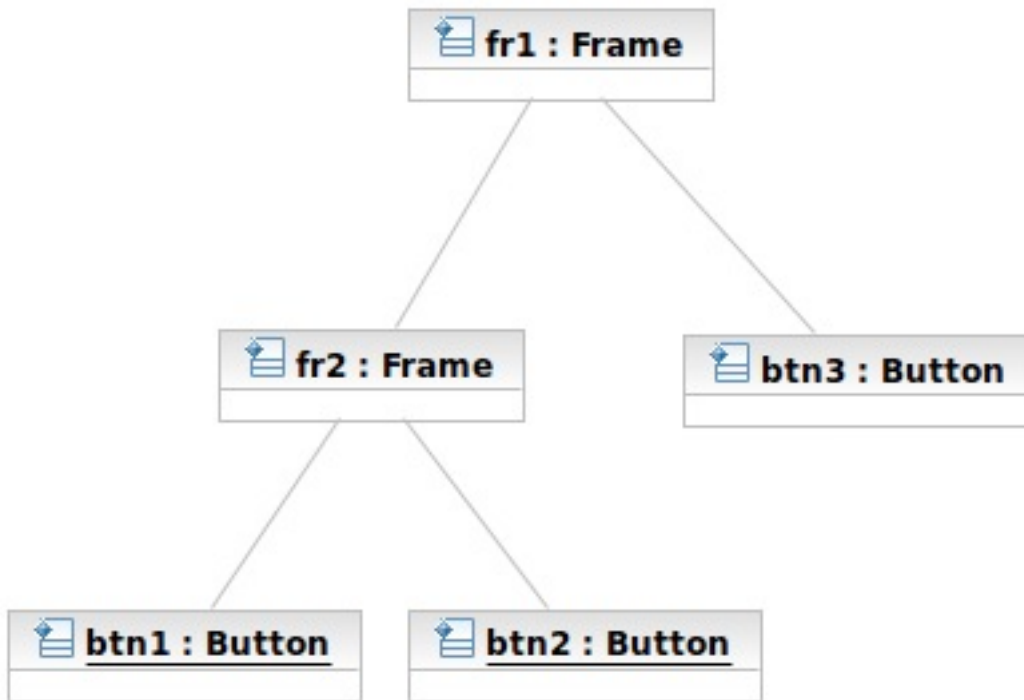
# Composite

98

- What it is?
  - ▣ A structure to compose objects into tree structures to represent part–whole hierarchies
  - ▣ Individual objects and compositions are treated uniformly (with the same interface)
- Target Problem
  - ▣ Parse tree
  - ▣ GUI widget composition
  - ▣ Macro commands

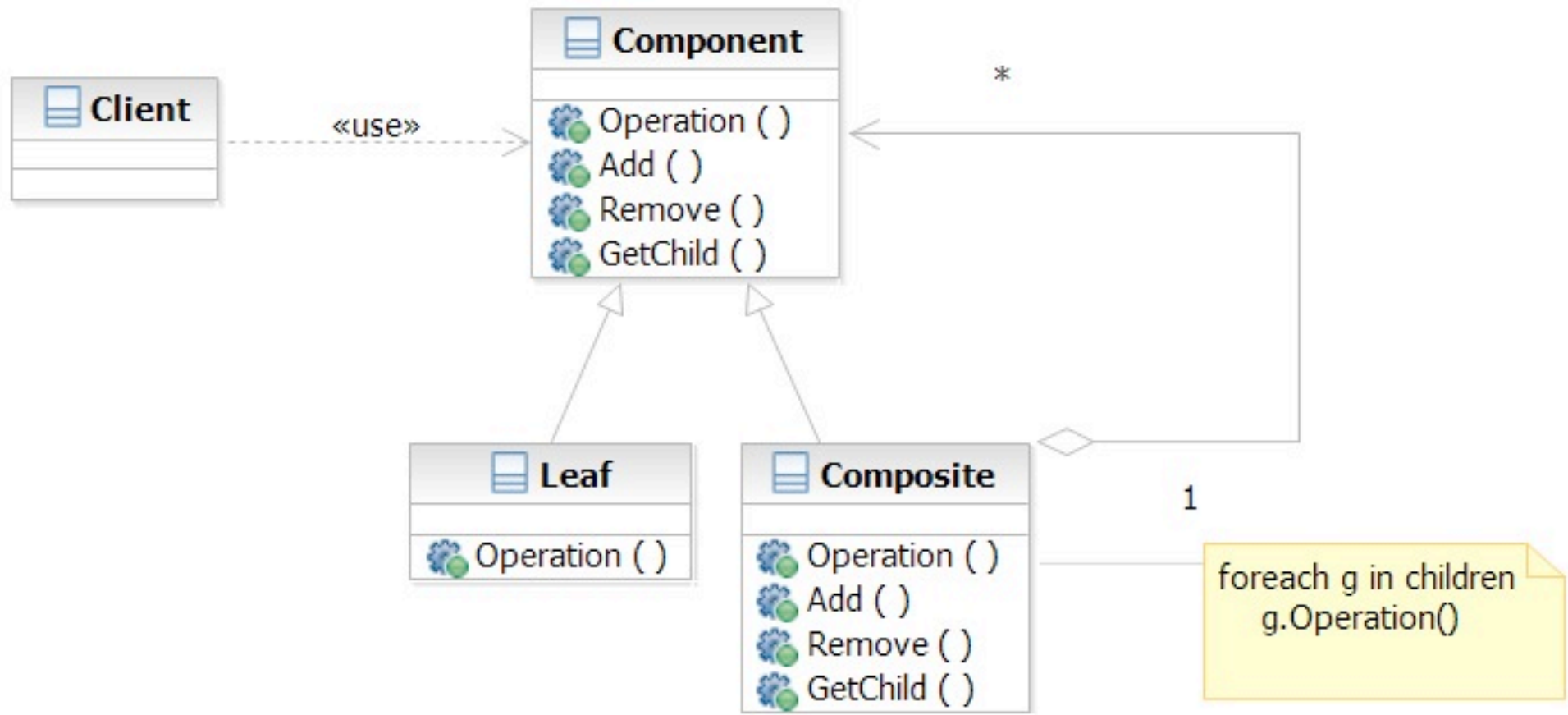
# Apply the Composite Pattern

99



# Structure/Participants

100



# Composite and Builder

101

- The composite structure is often built with the builder
- What Builder is?
  - ▣ Separation of the construction of a complex object from its representation
  - ▣ The construction process can optionally create different representations
- Target Problem
  - ▣ Parser reading source file to represent it as parse tree

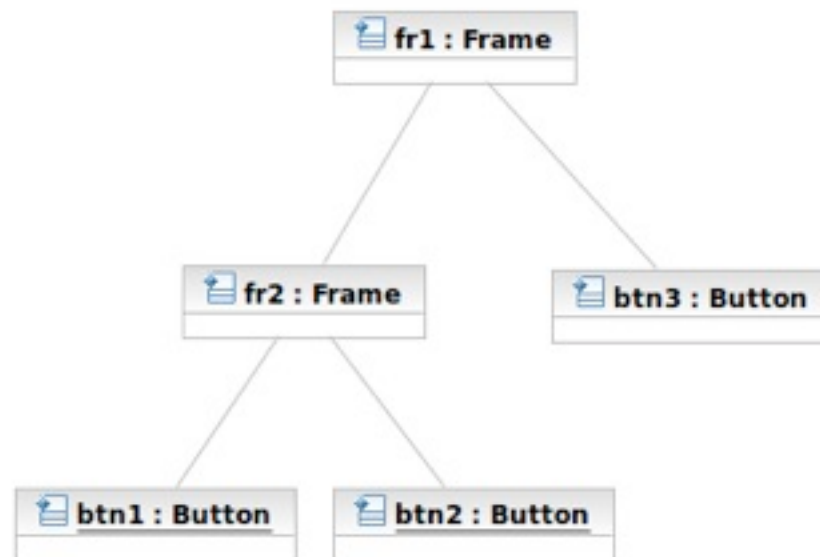
# Apply the Builder Pattern

102

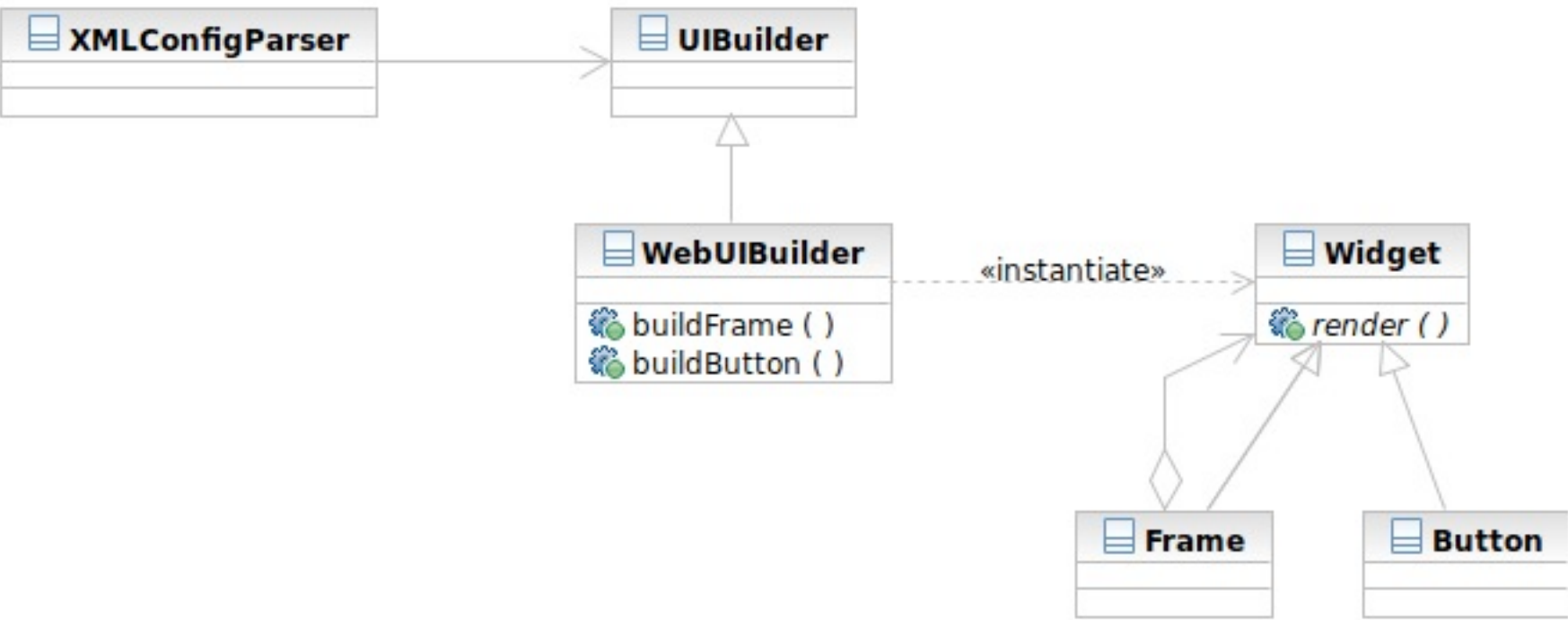
Input Config:

```
<Frame name="fr1">  
  <Frame name="fr2">  
    <Button name="btn1">...</Button>  
    <Button name="btn2">...</Button>  
  </Frame>  
<Button name="btn3">...</Button>  
</Frame>
```

Parsed result:



# Apply the Builder Pattern

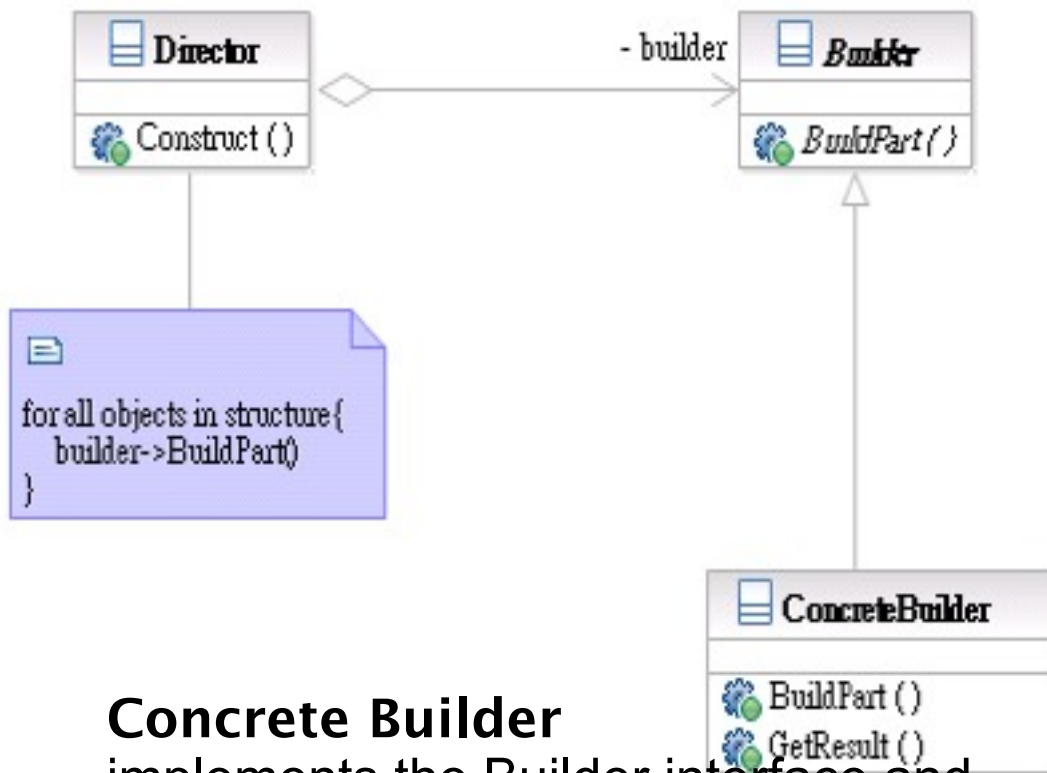


# Structure

104

## Director

constructs an object using the Builder interface



```
for all objects in structure {
    builder->BuildPart()
}
```

## Concrete Builder

implements the Builder interface and keeps track of the product and objects

## Builder

specifies an interface for creating parts of a Product object



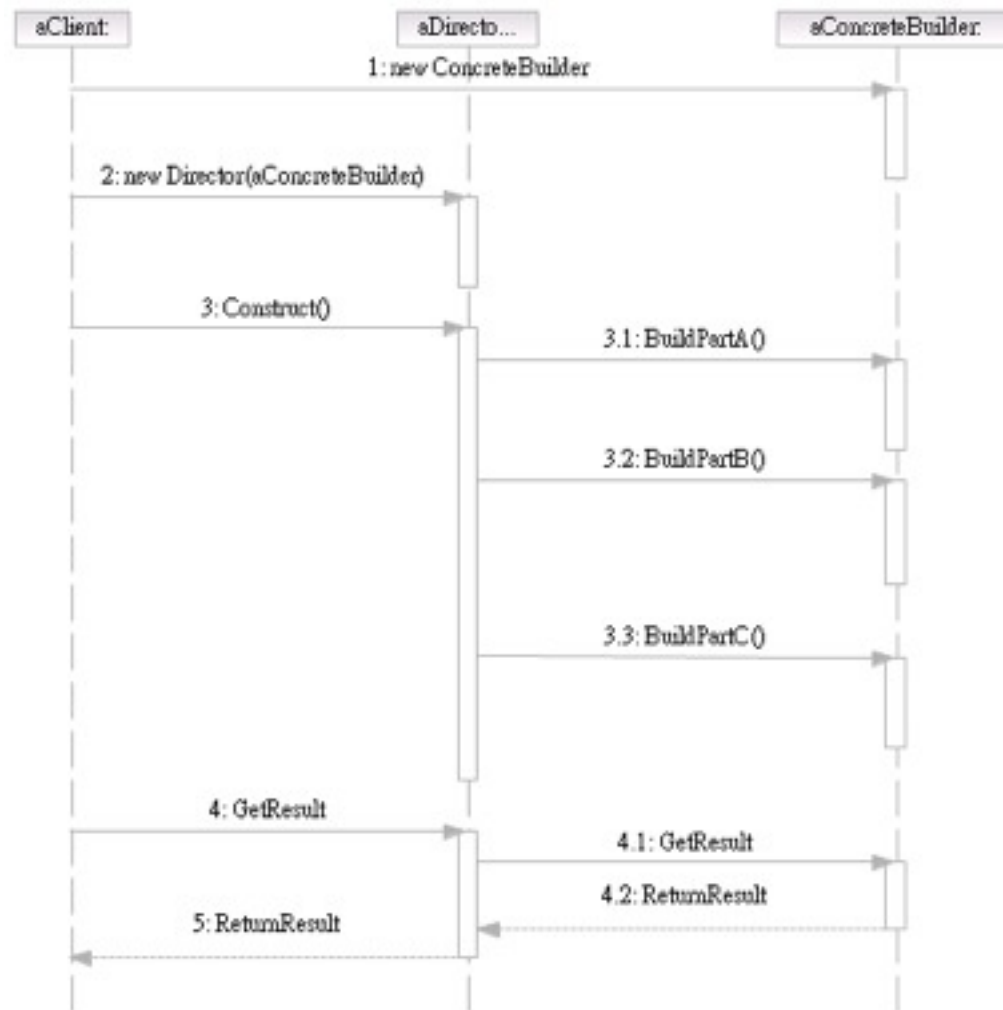
## Product

represents the final product and its constituent parts



# Builder Interaction

105



# Participants

106

- Class **Builder** specifies an interface for creating parts of a Product object.
- Class **ConcreteBuilder** implements the Builder interface and keeps track of the product and objects.
- Class **Director** constructs an object using the Builder interface.
- Class **Product** represents the final product and its constituent parts.

# Visitor and Composite

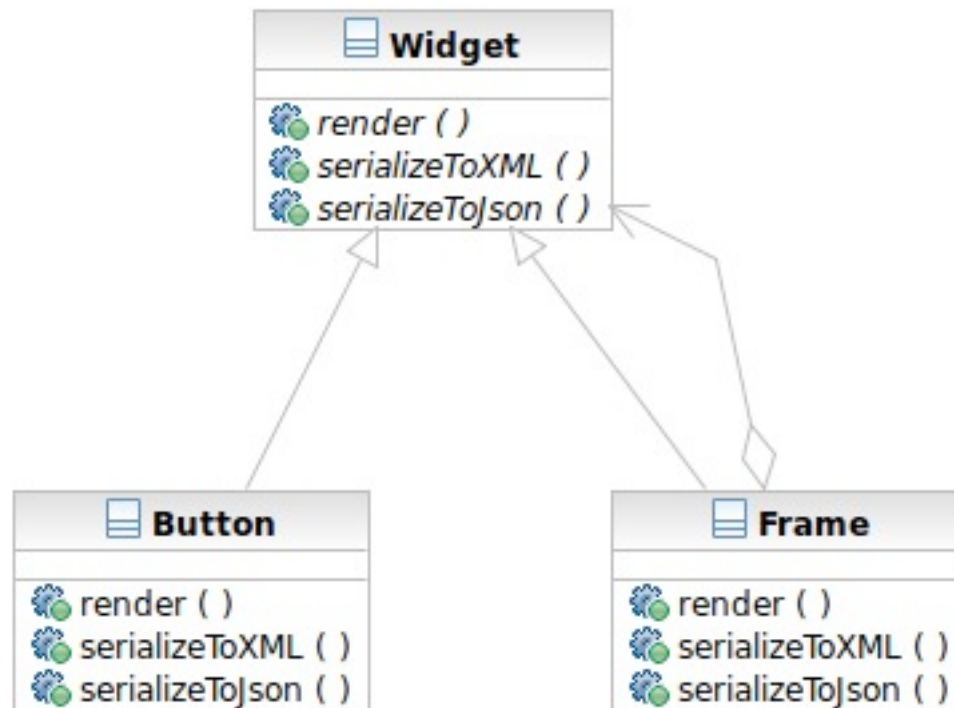
107

- The visitor lets you add new operations to the composite structure without modifying it
- What Visitor is?
  - ▣ The representation of an operation that can be applied to different elements in the composite structure
- Target Problem
  - ▣ Serialization of the parse tree into json, database, etc

# Without the Visitor Pattern

108

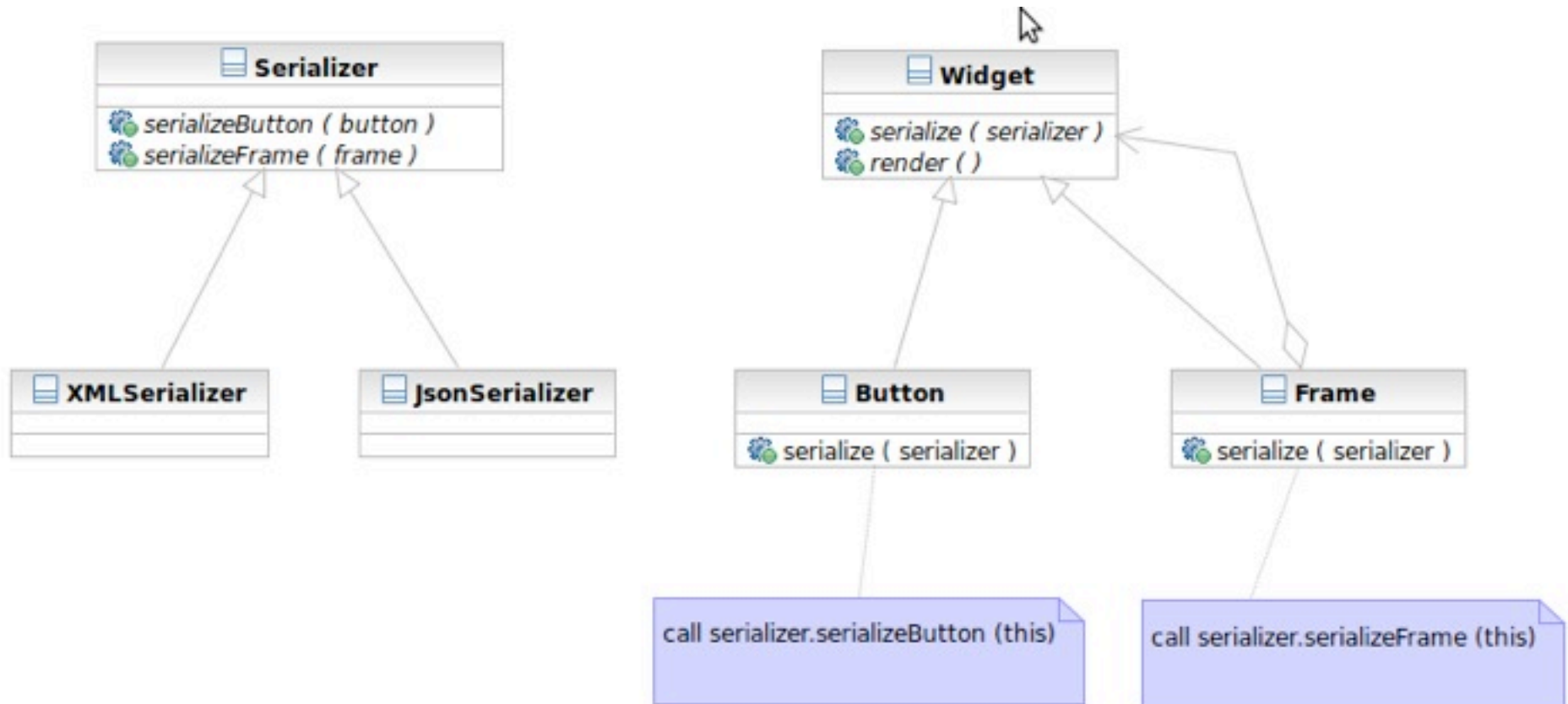
- Adding new operations to the whole class family:



# Applying the Pattern

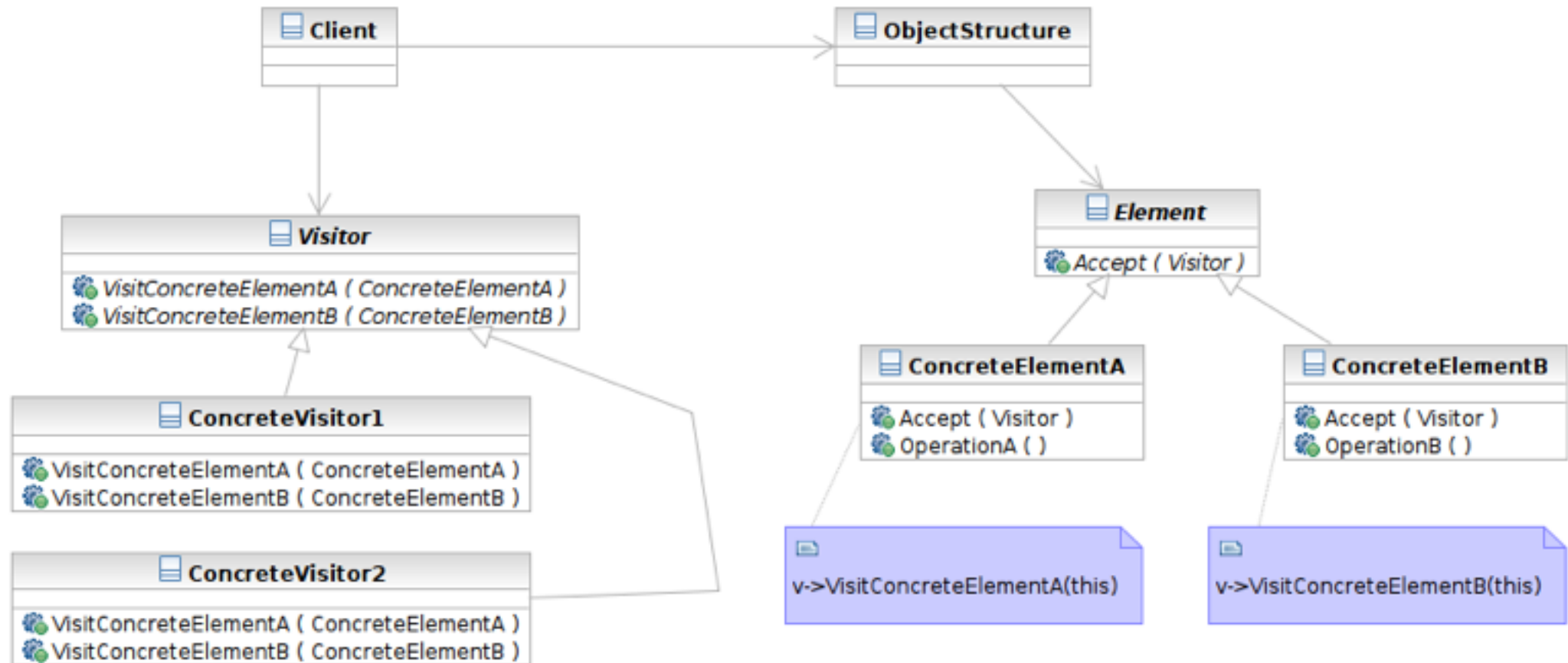
109

- The operations to serialize to Json and XML are extracted into visitors



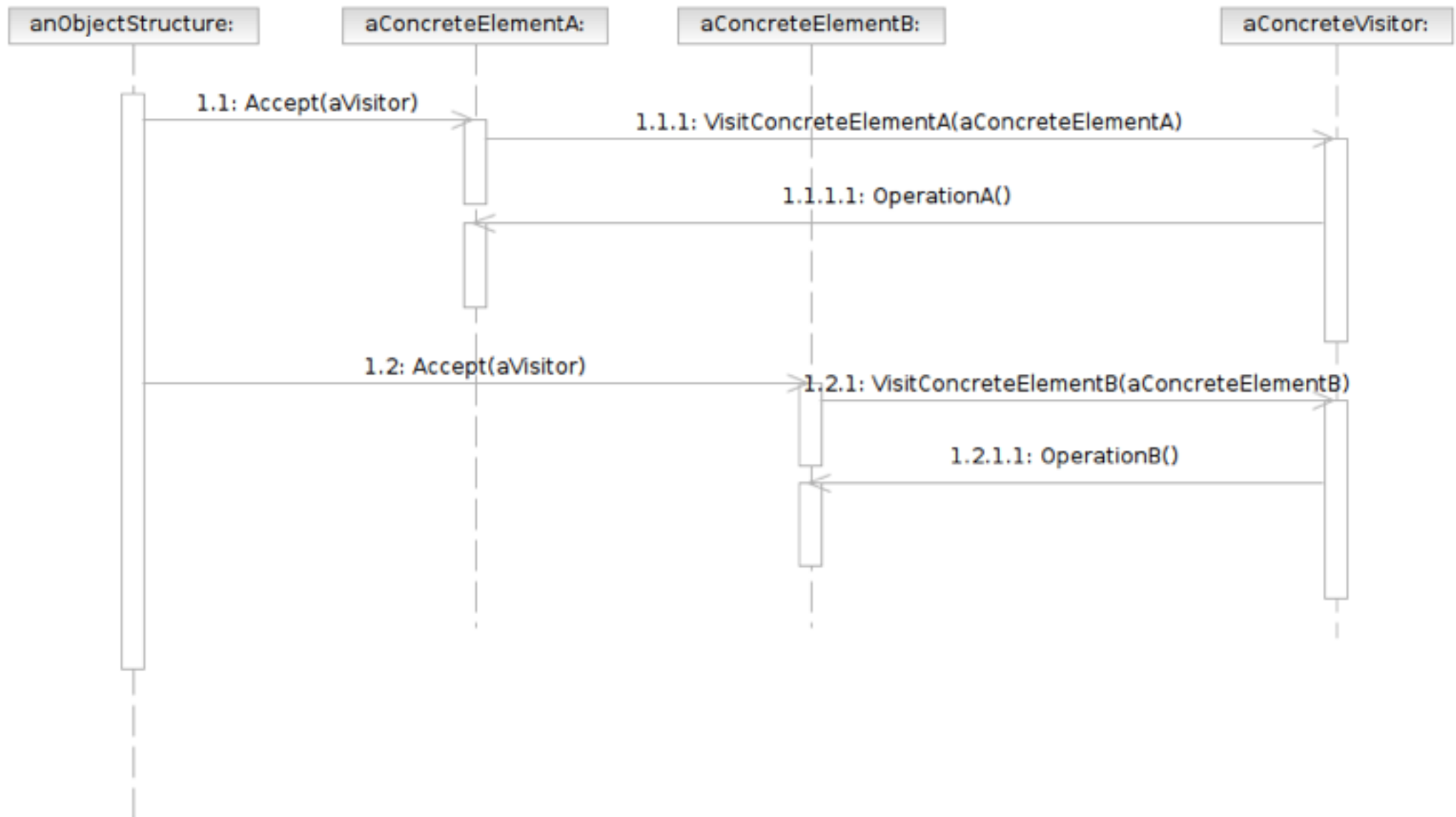
# Structure

110



# Interaction

111



# Participants

112

- Class **Visitor** declares a Visit operation for each class of ConcreteElement in the object structure.
- Class **ConcreteVisitor** implements each operation declared by Visitor.
- Class **Element** defines an Accept operation that takes a visitor as an argument.



# Participants

113

- Class **ConcreteElement** implements an **Accept** operation that takes a visitor as an argument.
- Class **ObjectStructure** enumerates its elements