Design Patterns

Ching-Lin Yu

Mozilla Taiwan

Contents

- Why Design Patterns
- Creational, Structural and Behavioral Patterns
- GoF Design Patterns
- Introductions to Enterprise Systems
- Enterprise/Cloud Computing Patterns
- Concluding Remarks

Why Design Patterns

- 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

- 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

- 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

- 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

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

What to Expect from Design Patterns

- 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

- 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

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

Creational Patterns

- 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

- 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

- 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

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

GoF Design Patterns

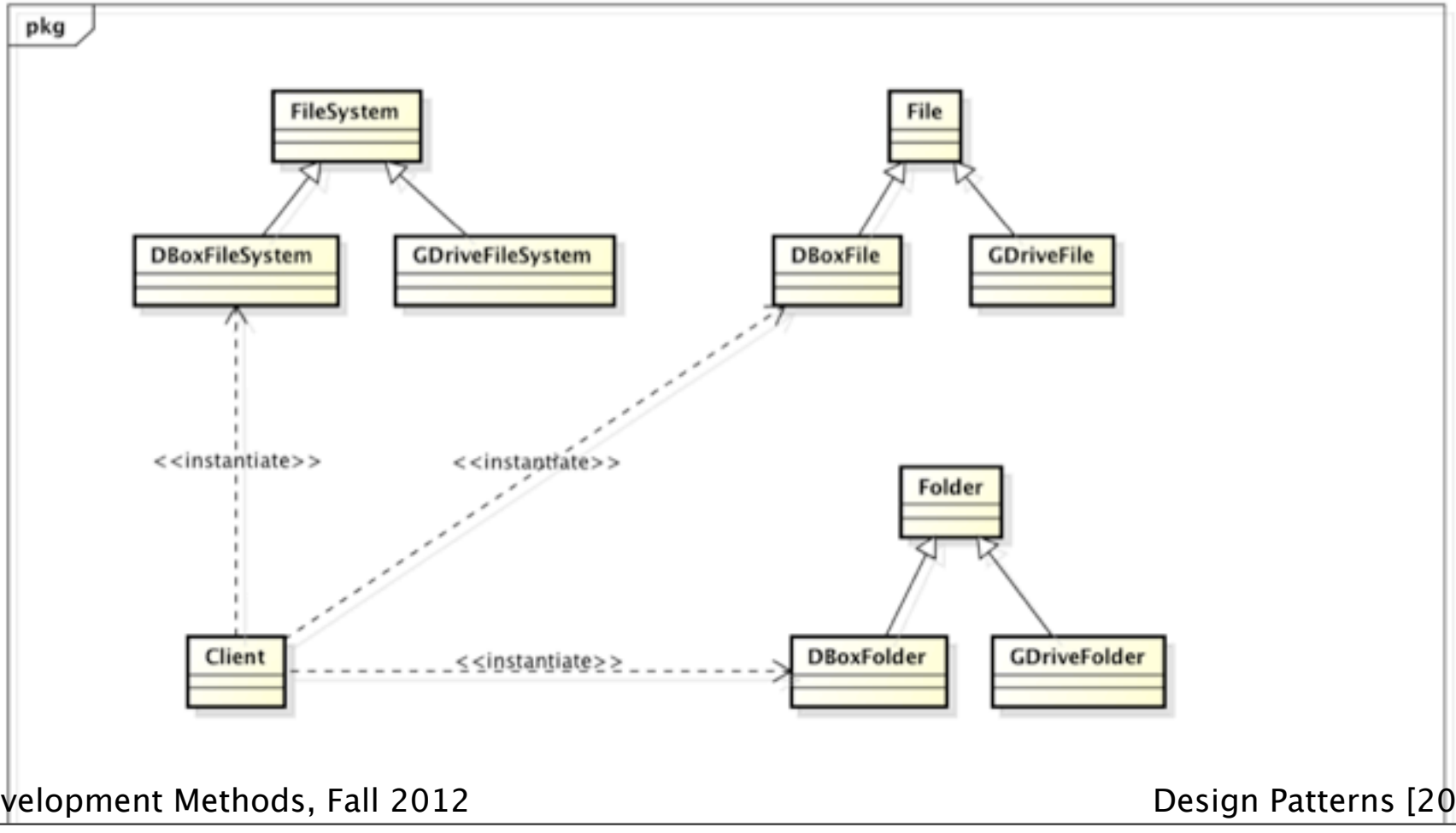
- Proxy & Decorator
- State
- Chain of Responsibility
- Prototype
- Builder & Composite & Visitor

Abstract Factory

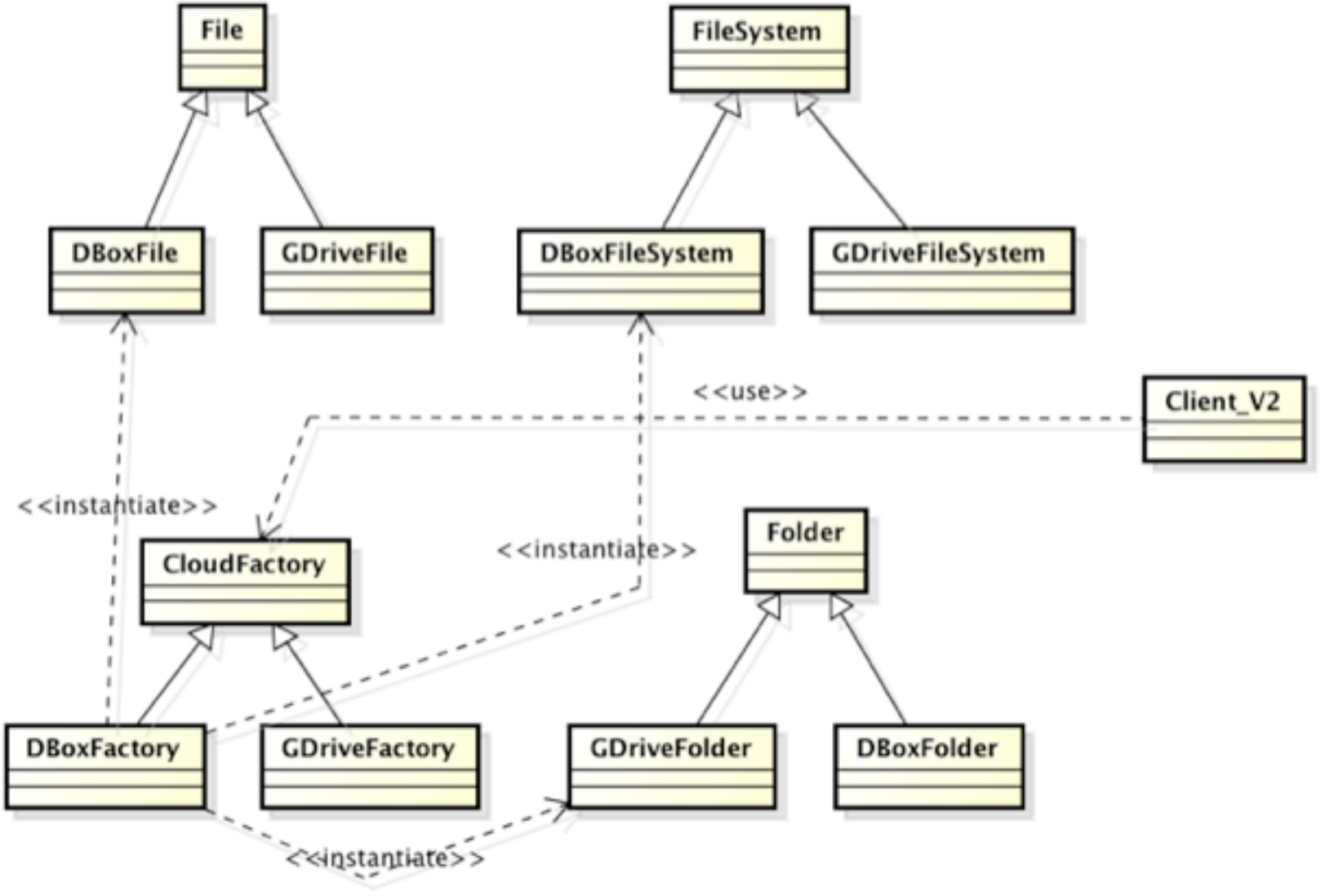
- What it is
  - An interface for creating families of related or dependent objects
    - Without specifying their concrete classes
- Target Problem
  - Cloud drive client needs to instantiate different FileSystem, File and Folder objects
    - Without needing to know the concrete classes for different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

Without the Abstract Factory Pattern

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



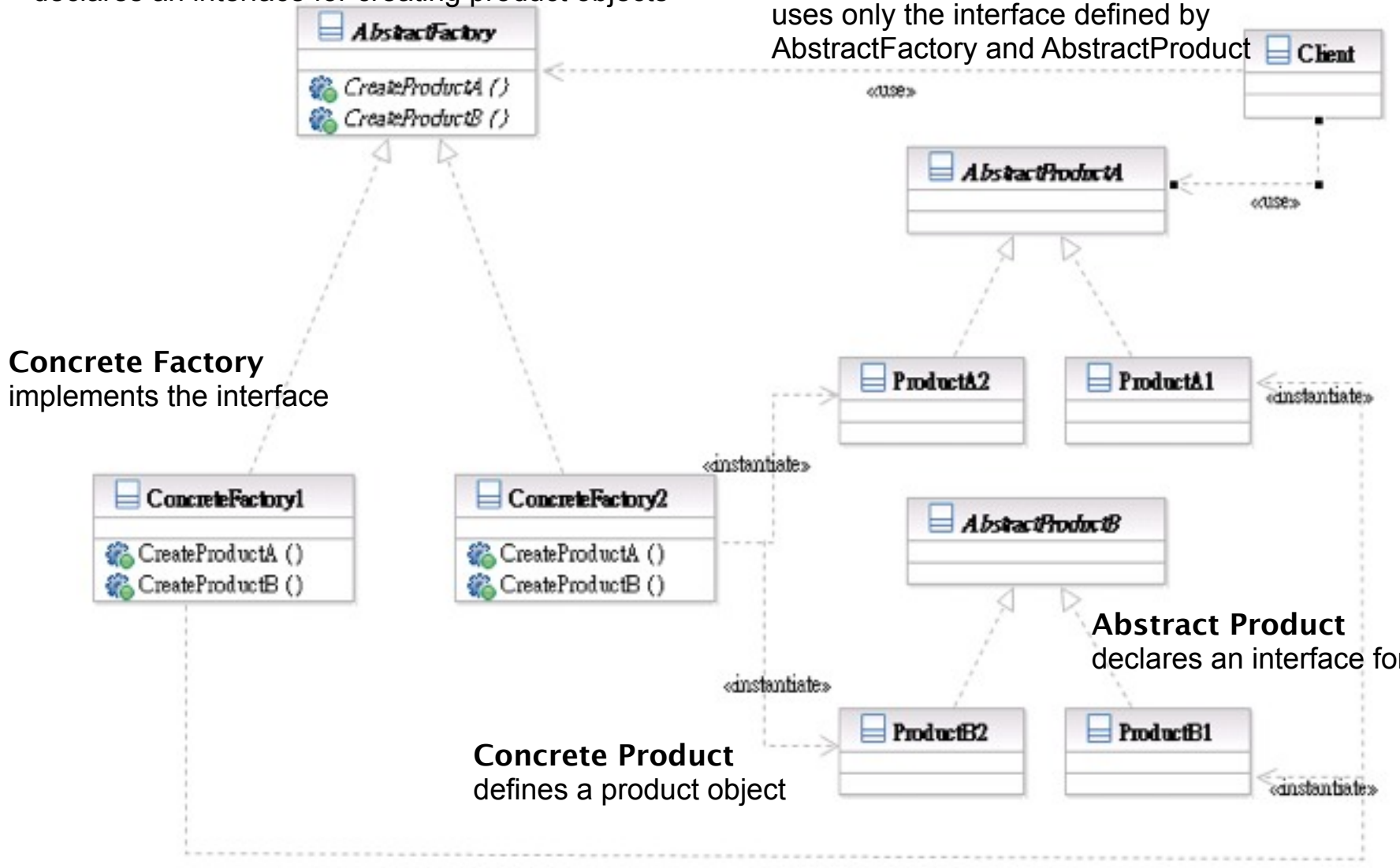
Applying the Pattern



Structure

Abstract Factory
declares an interface for creating product objects

Client
uses only the interface defined by AbstractFactory and AbstractProduct



Concrete Factory
implements the interface

Abstract Product
declares an interface for product obje

Concrete Product
defines a product object

Participants

- 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

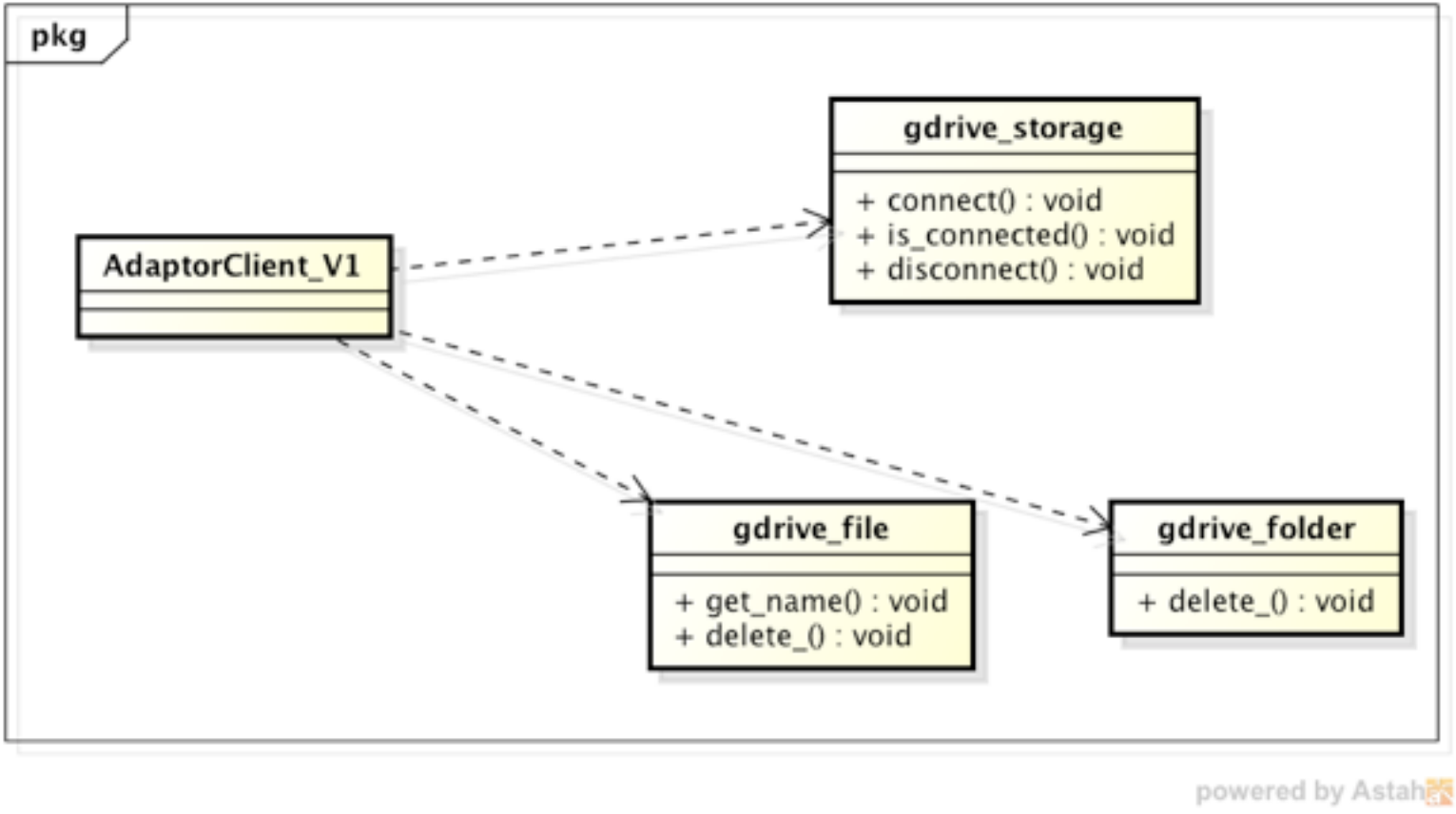
- They both change the interface seen by the using class
- Adapter converts an interface
- Facade simplifies an interface

Adapter

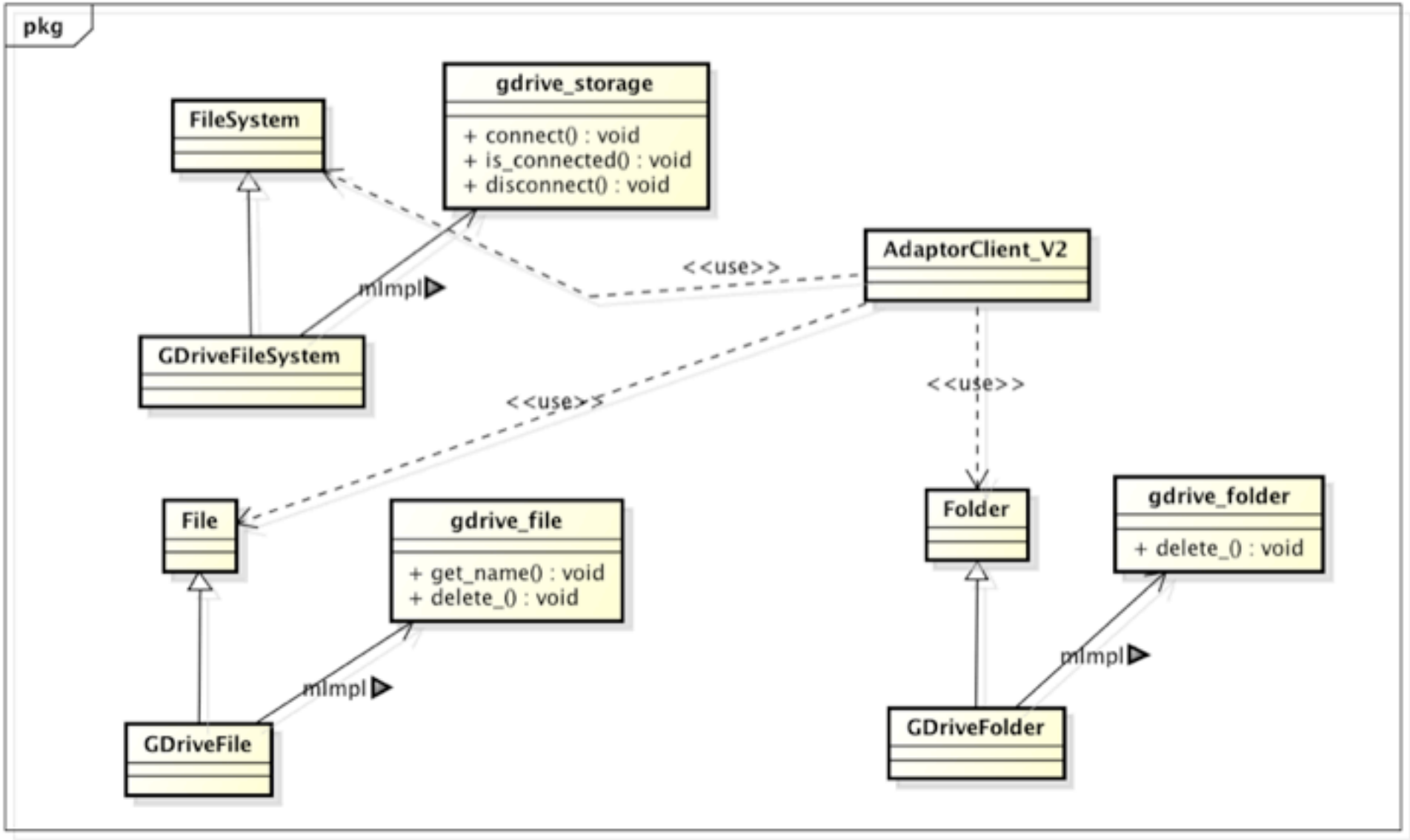
- What it is
  - Conversion of the interface of one class into another the client expects
- Target Problem
  - Integrate a library into your system but the interface is incompatible
  - The interface of the library may change in subsequent versions
  - Replace existing library with another one without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

Without the Adapter Pattern

- Client is bound to the interface of the library

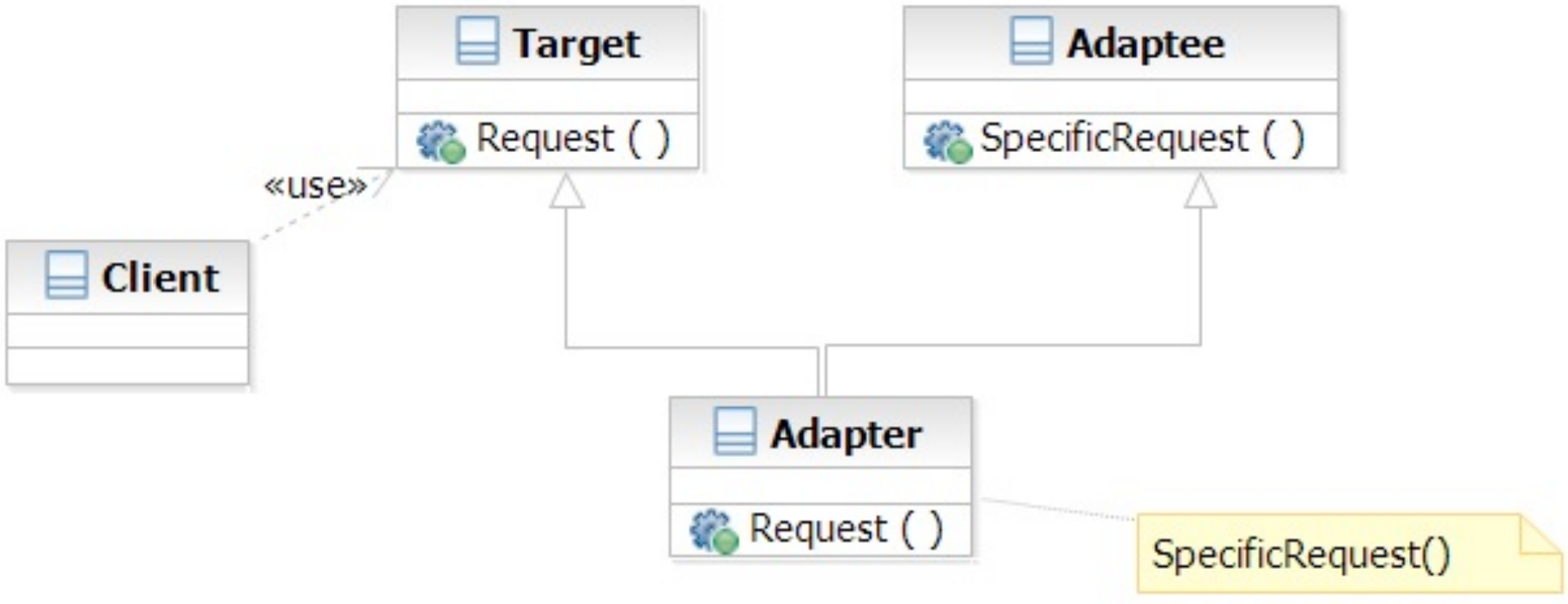


Applying the Pattern



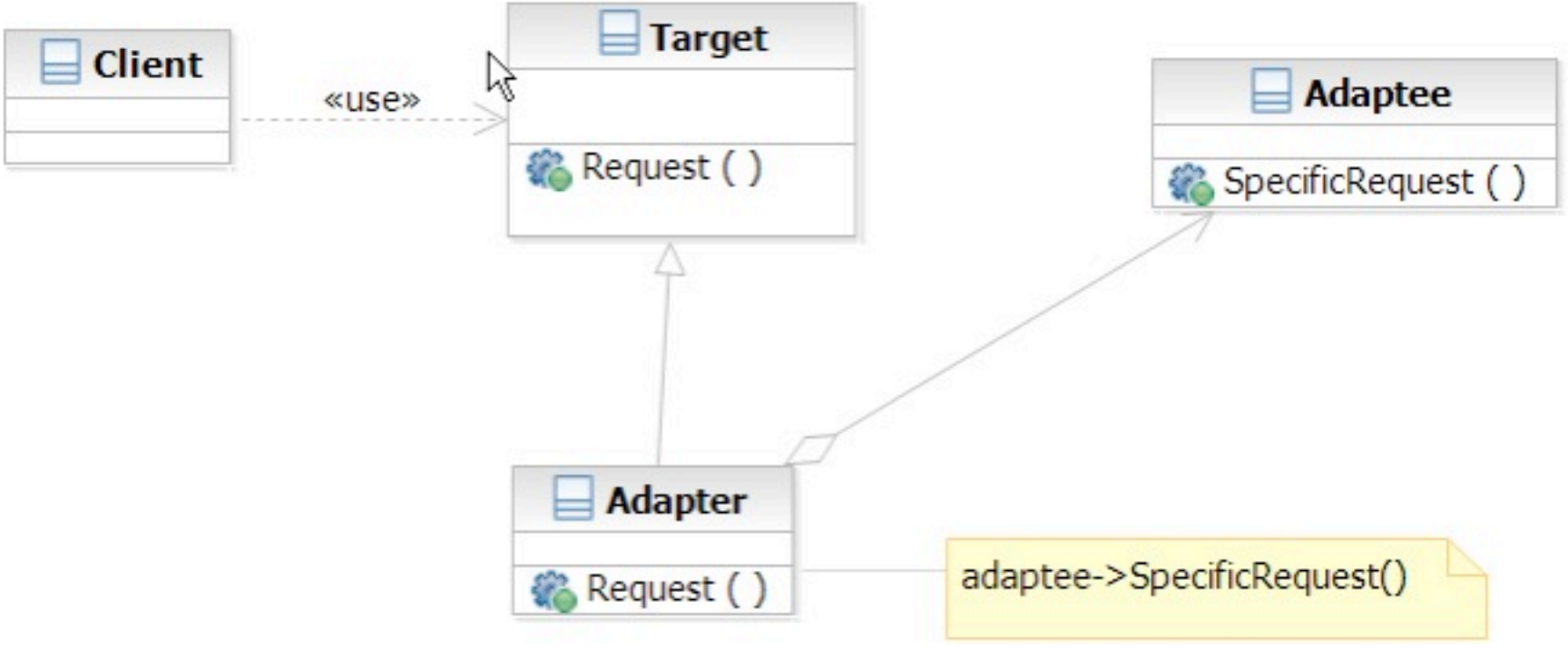
Structure

Class Adapter



Structure

Object Adapter

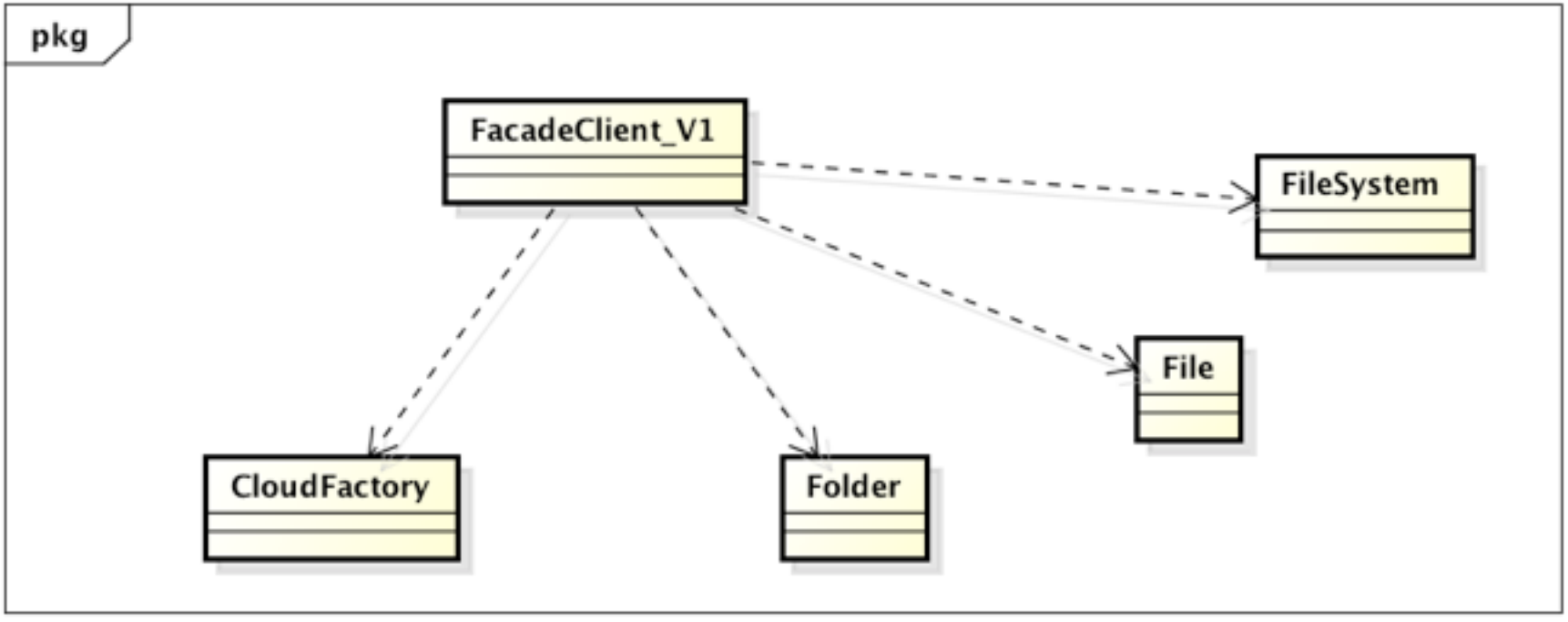


Facade

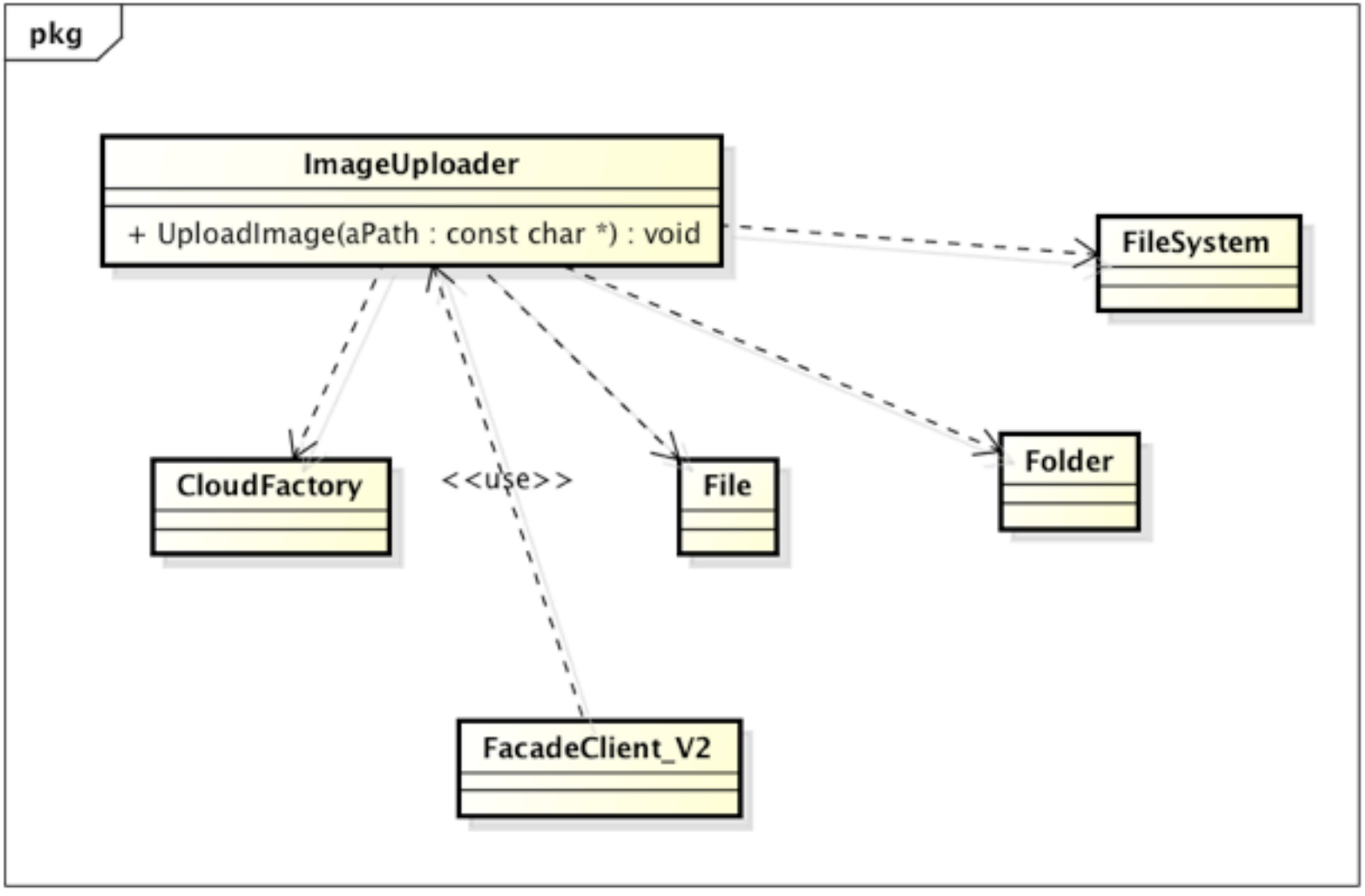
- What it is
  - A high level interface to a set of interfaces in a subsystem
- Target Problem
  - Providing a simplified interface to the low-level, fine-grained subsystems
    - GCC -> scanner, parser, optimizer, code gen, linker
  - Unify the access to subsystems
    - e.g. account manager -> database,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

Without the Facade Pattern

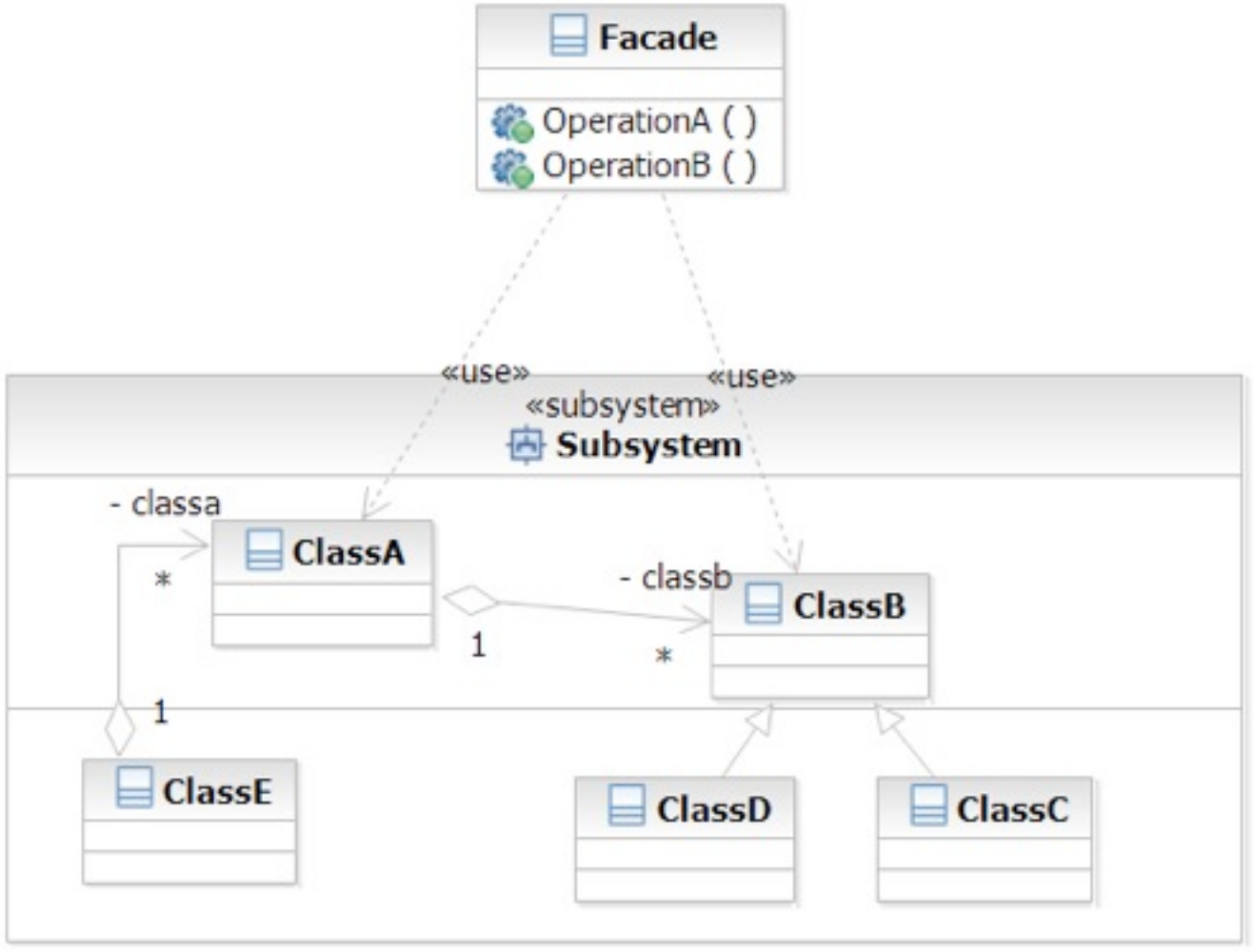
- Client directly uses the interface of the lower-level, fine-grained classes



Apply the Pattern



Structure

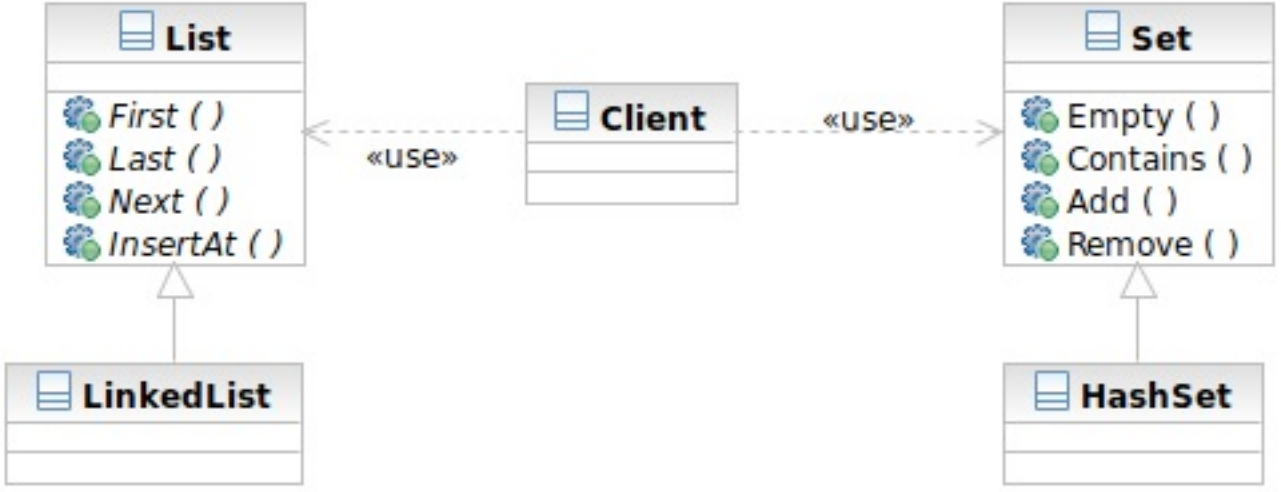

Facade
OperationA ( )
OperationB ( )
«use»
«use»
«subsystem»
Subsystem
- classa
ClassA
*
1
- classb
ClassB
*
1
ClassE
ClassD
ClassC

Iterator

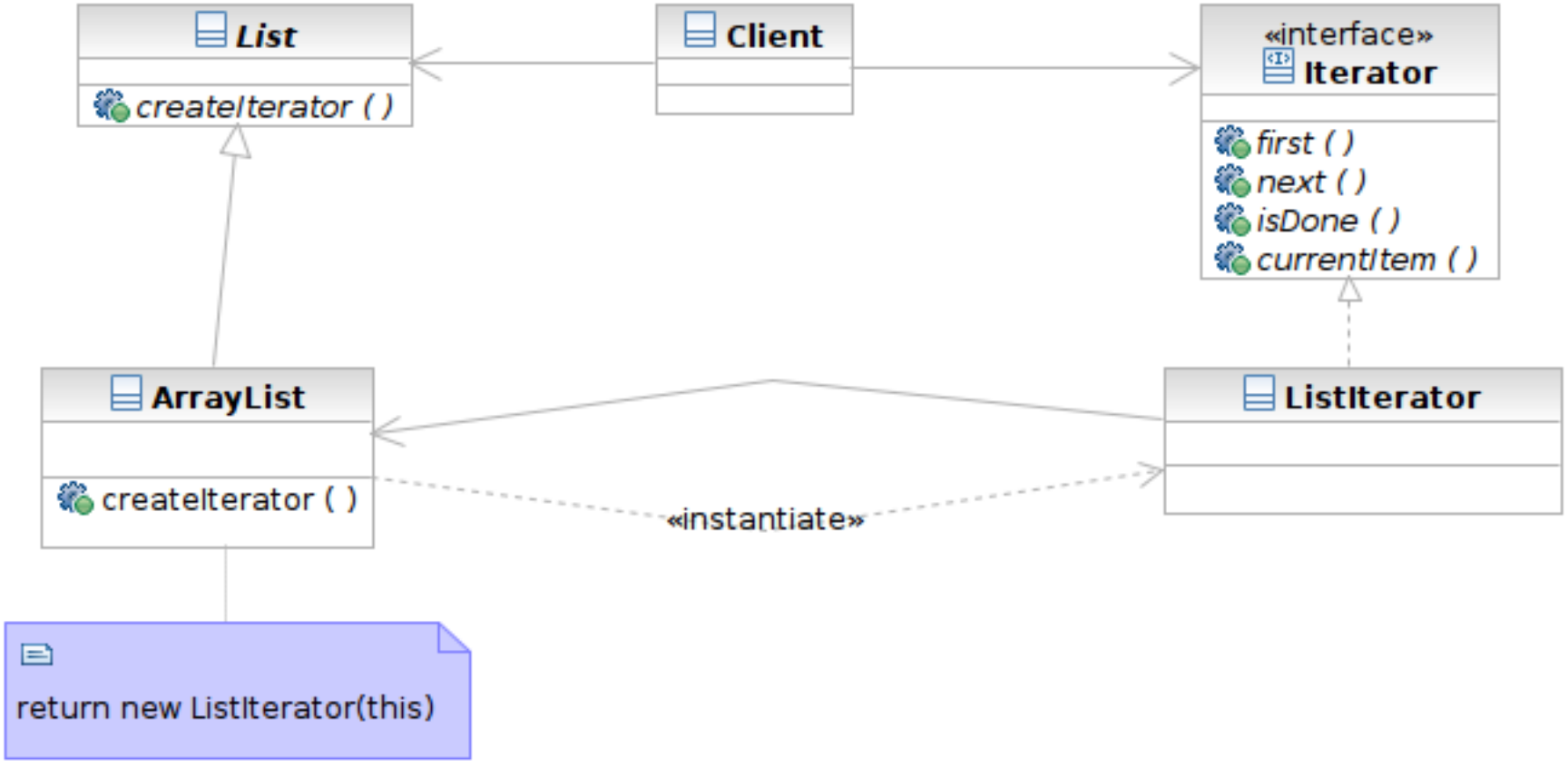
- What it is
  - A way to access the elements of an aggregate objets sequentially
  - Without exposing its internal details
- Target Problem
  - Accessing 'collection classes'
    - List, Vector, Tree, Sets, etc.
  - You don't want your code heavily impacted just because you want to replace a list with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

Without the Iterator Pattern

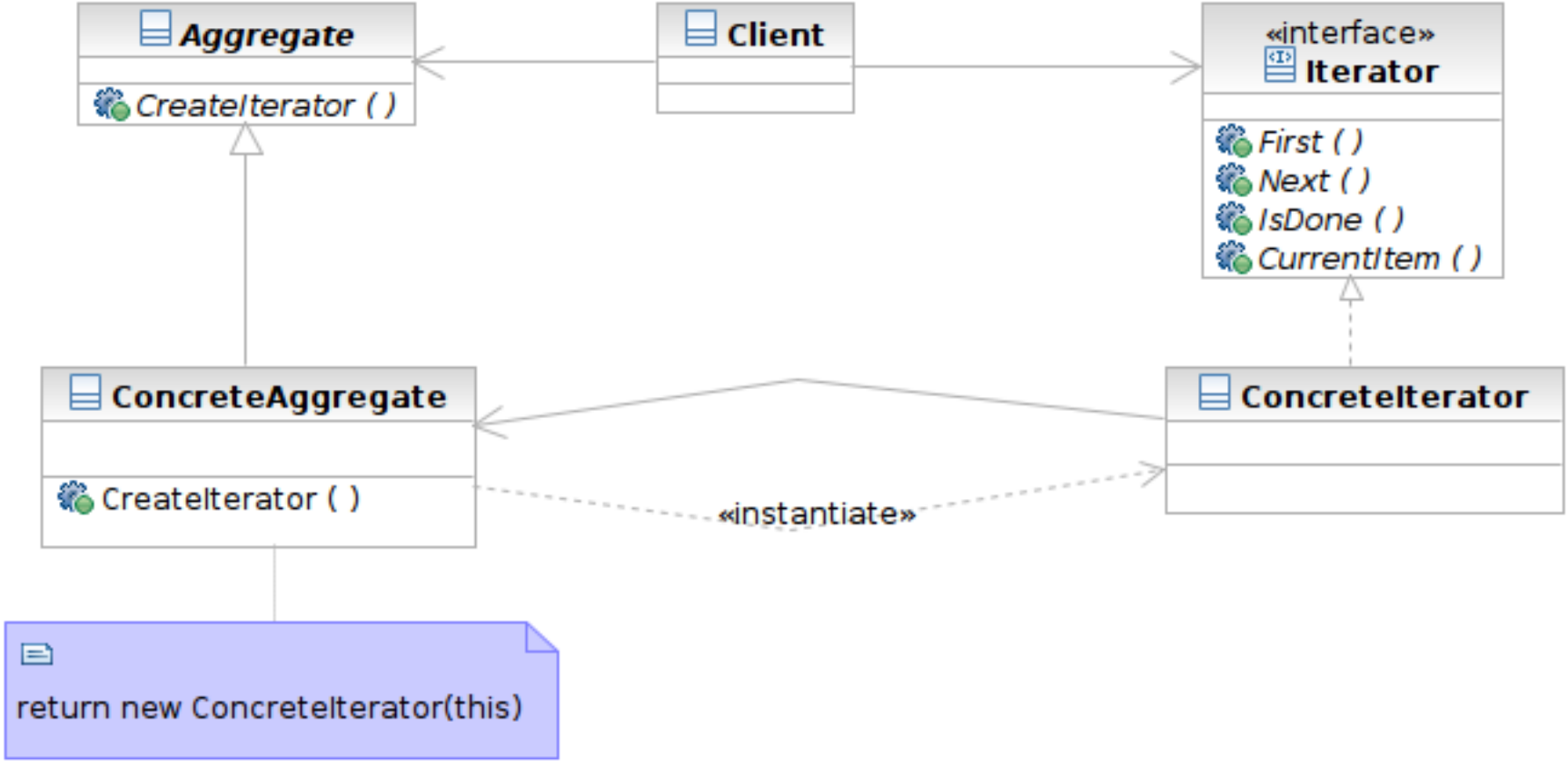
- Client is dependent on the interface of the aggregate classes



Applying the Pattern



Structure


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

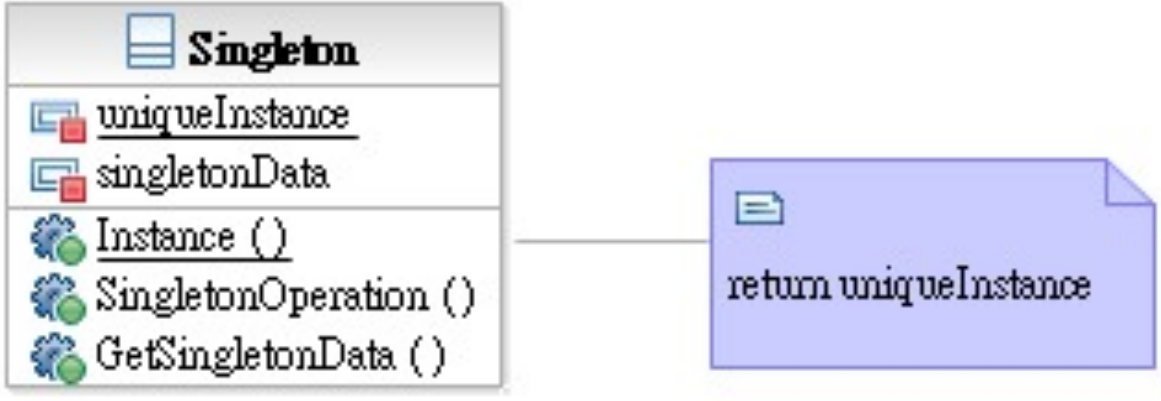
Beyond Iterator

- 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

- 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

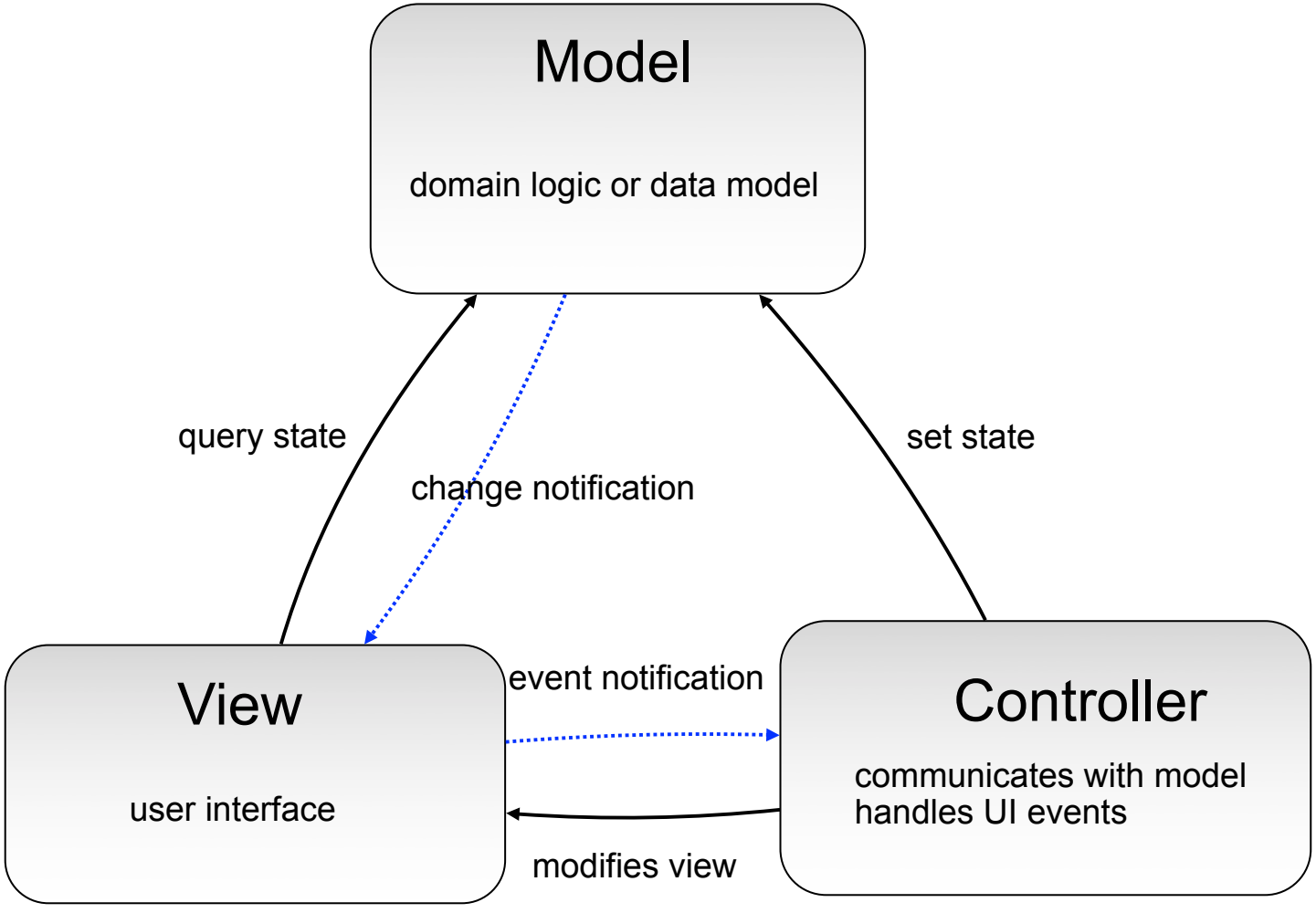


Singleton
defines a static member function that lets clients access its unique instance

Participants

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

Model-View-Controller (MVC)



Patterns Used in MVC

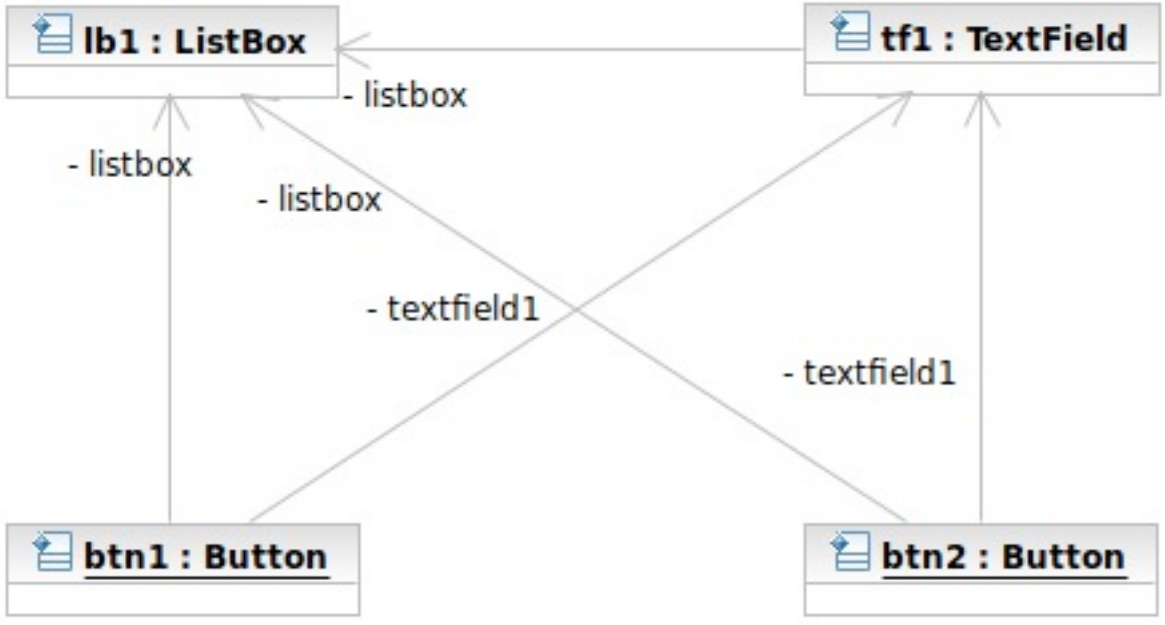
- Mediator: to mediate the communications of widgets
  - The controller
- Observer: to receive event notifications
  - Model to View, View to Controller
  - Async in nature
- Command: to encapsulate the action as objects
  - Action taken on event notifications

Mediator

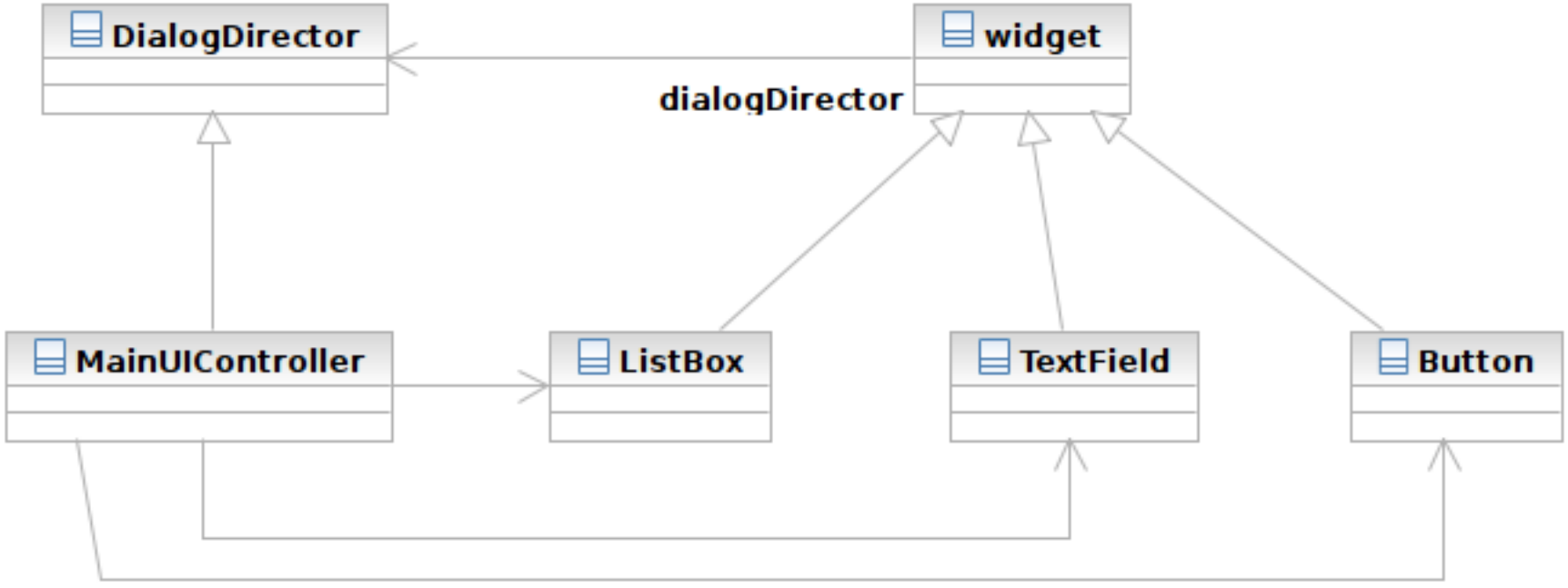
- What it is
  - An object acting as a “hub”
  - Defines how a set of objects (colleagues) interacts
  - So colleagues don’t have to refer to each other
- Target problem
  - Different widgets have to act in response to each other
  - Storing references in widgets is inflexible

Without the Mediator Pattern

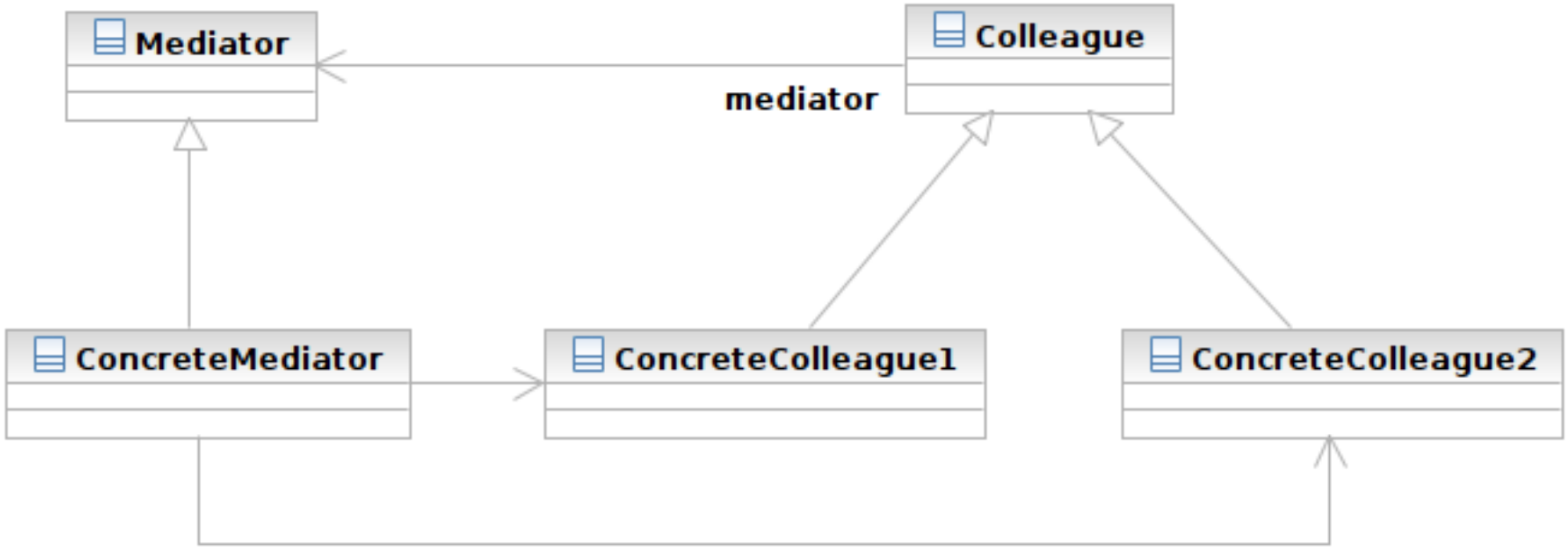
- Each concrete widget refers to other widgets to interact with



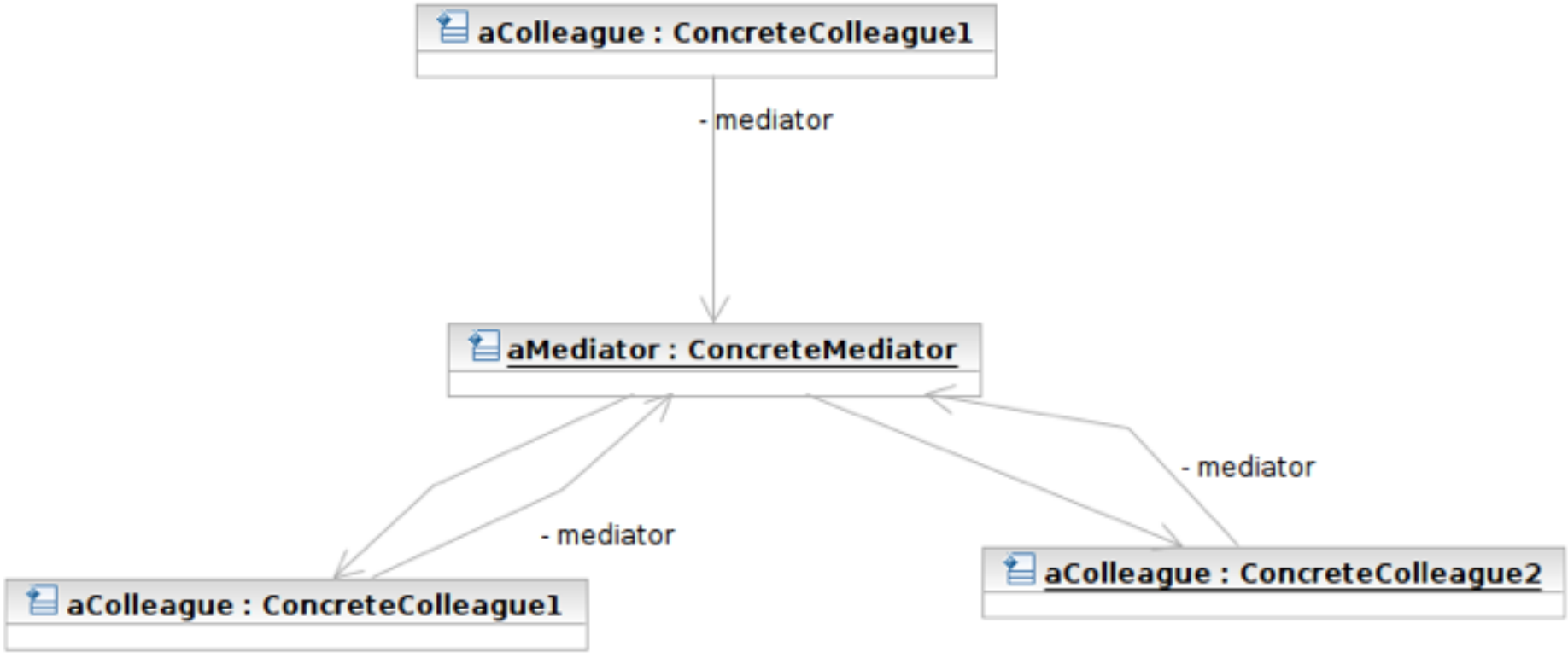
Applying the Pattern



Structure


Mediator
Colleague
mediator
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Structure


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

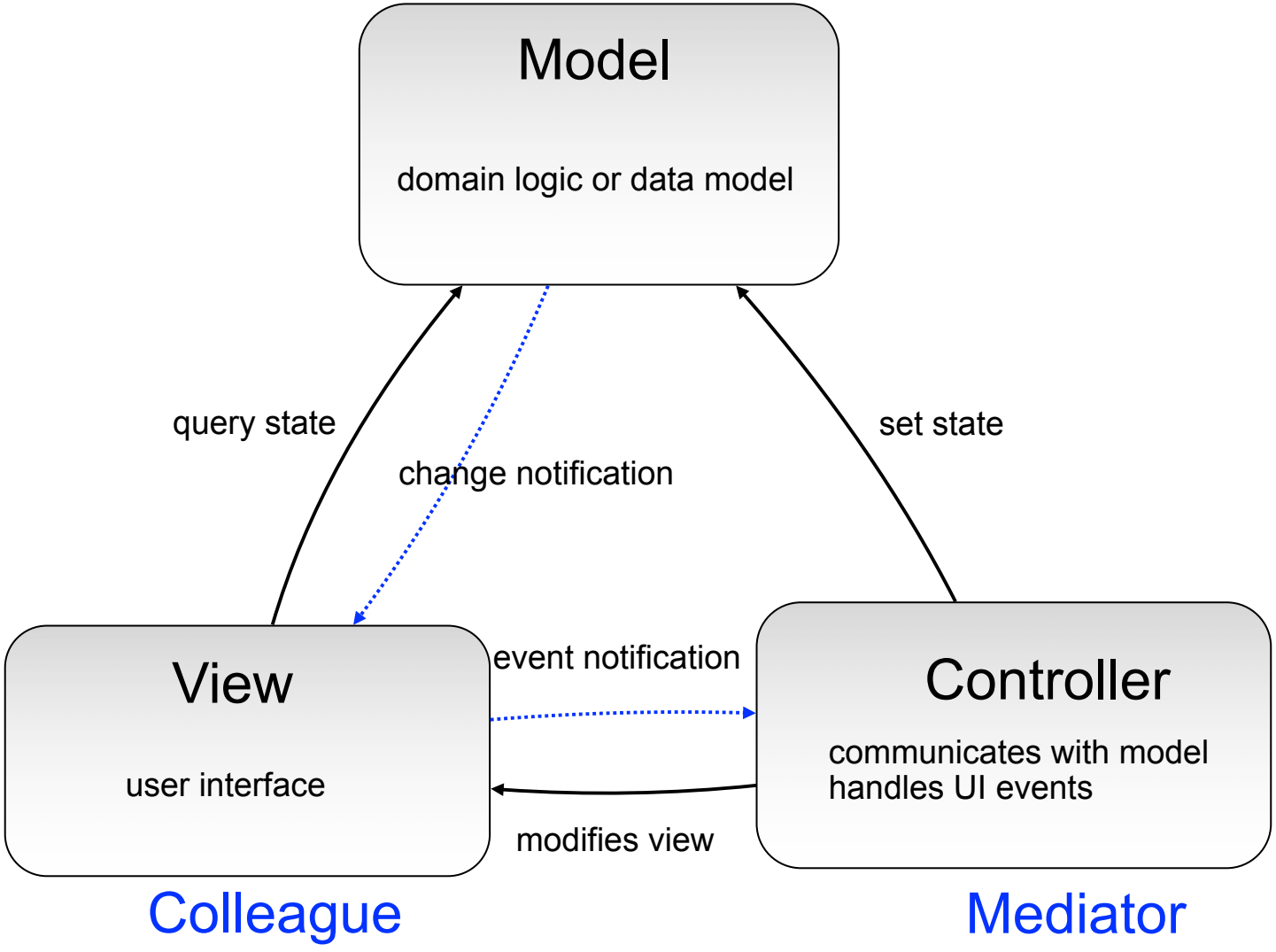
Participants

- 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

MVC and Mediator Pattern



Observer

- 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

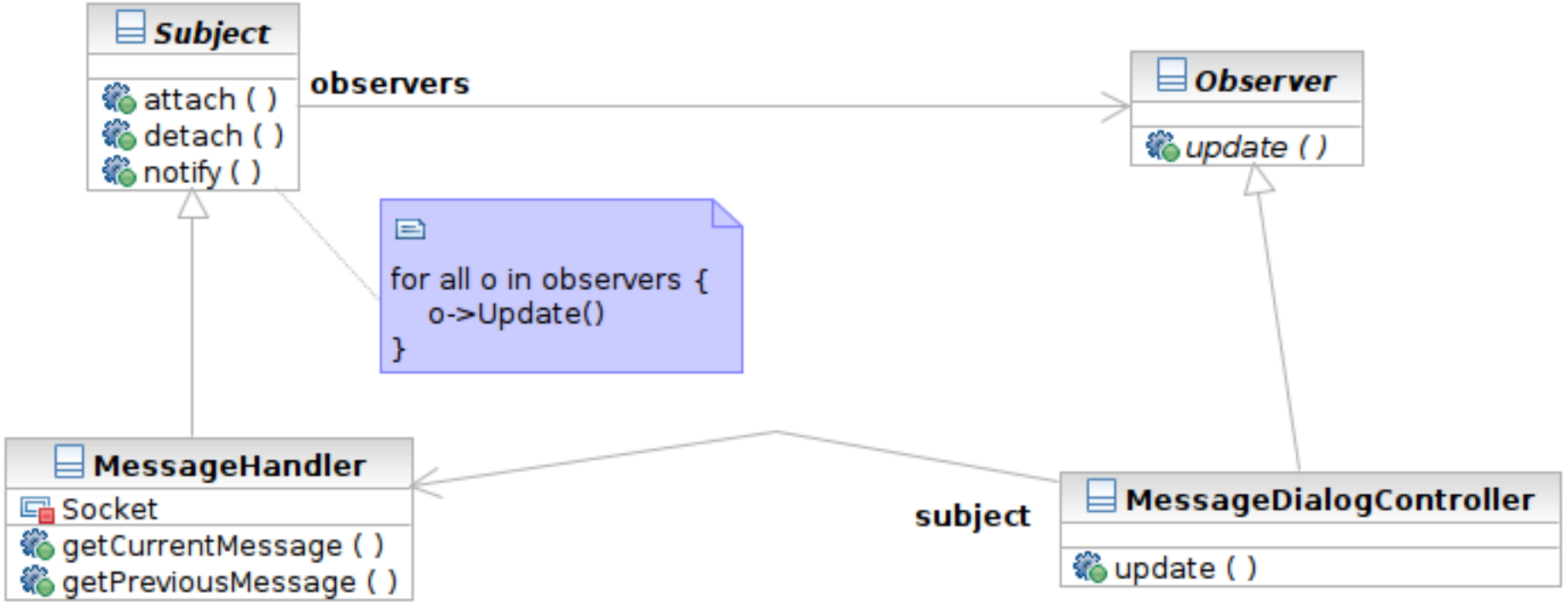
Without the Observer Pattern

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

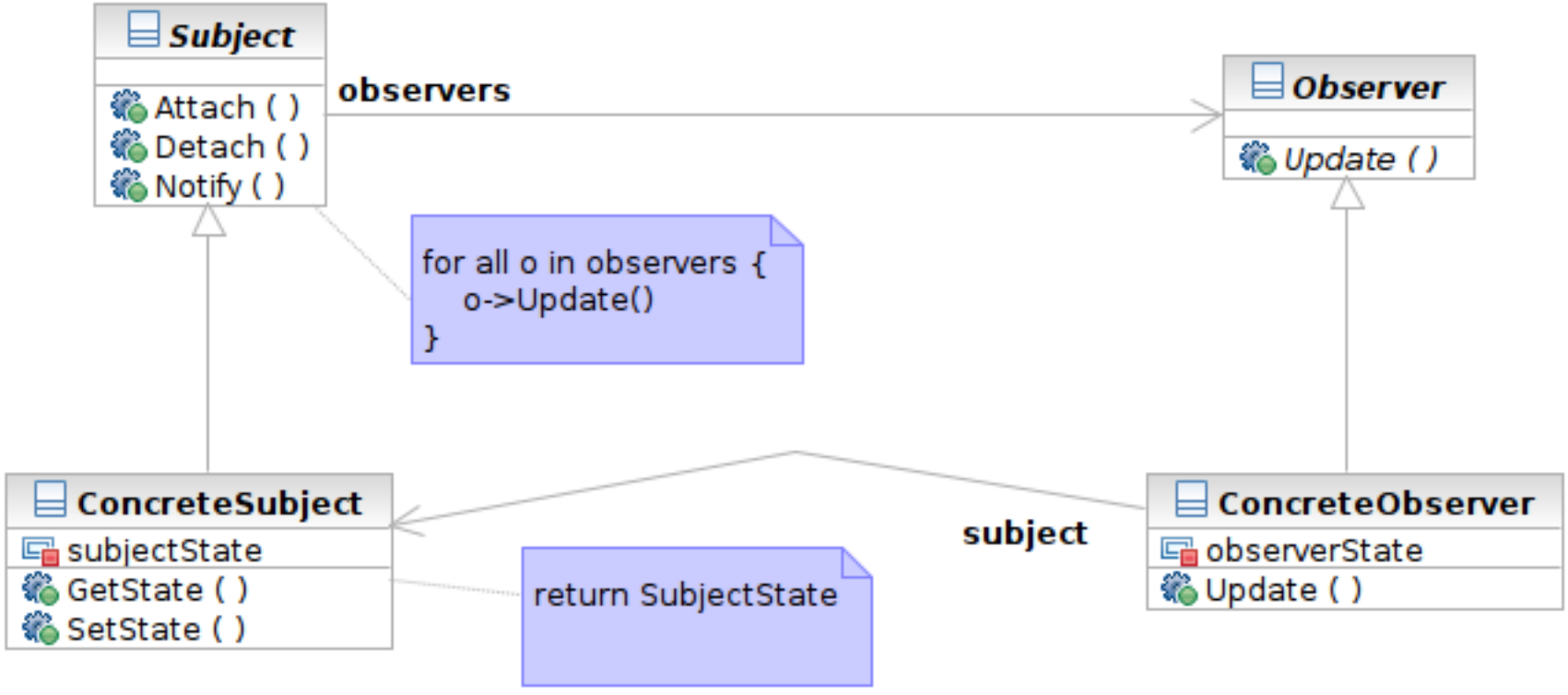
```
While (! aSubject.hasChangedState()) {

}
// now aSubject has changed its state
```

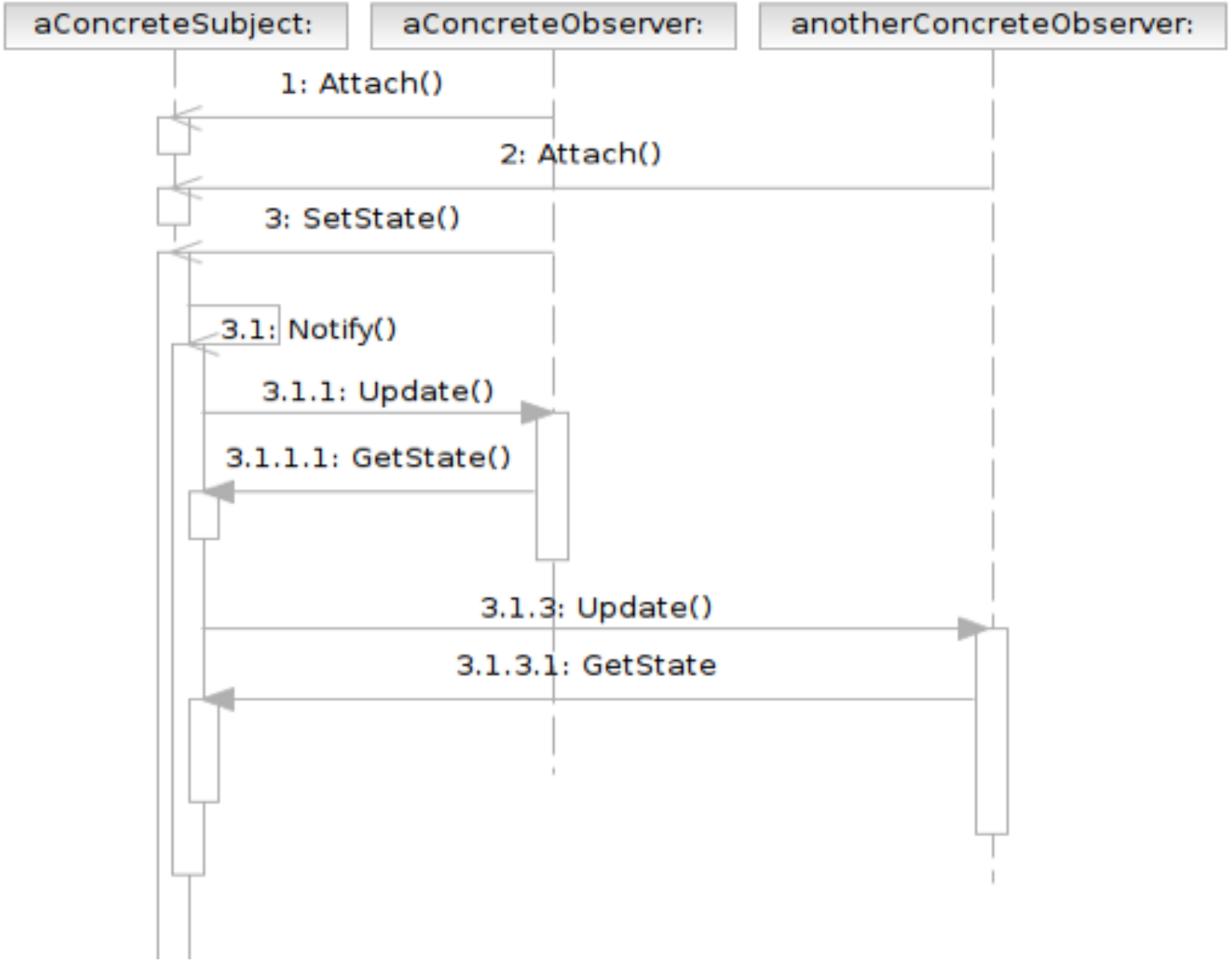
Applying the Pattern



Structure


Subject
Attach ( )
Detach ( )
Notify ( )
observers
Observer
Update ( )
for all o in observers {
o->Update()
}
ConcreteSubject
subjectState
GetState ( )
SetState ( )
return SubjectState
subject
ConcreteObserver
observerState
Update ( )

Interaction



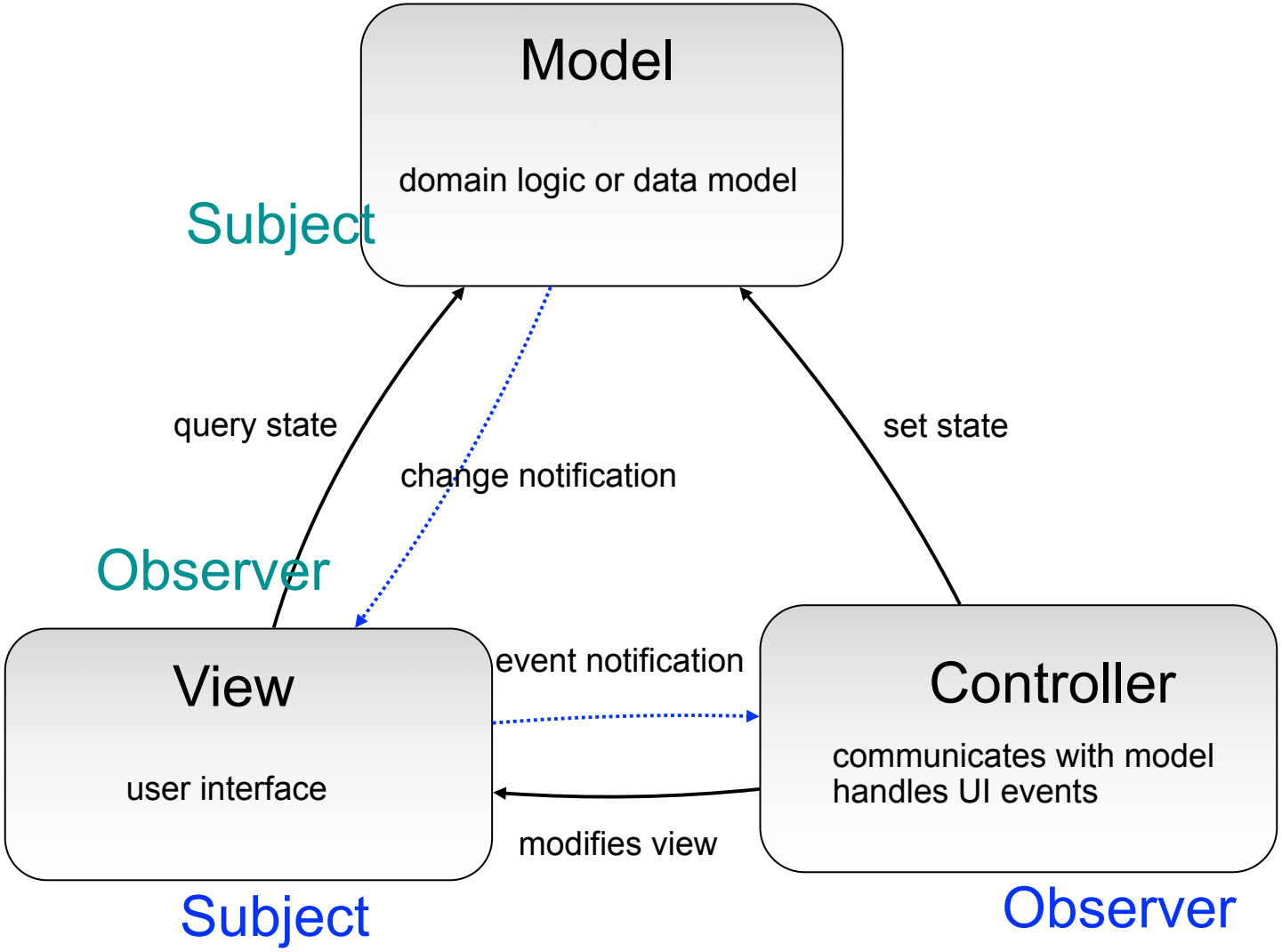
Participants

- 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

- 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

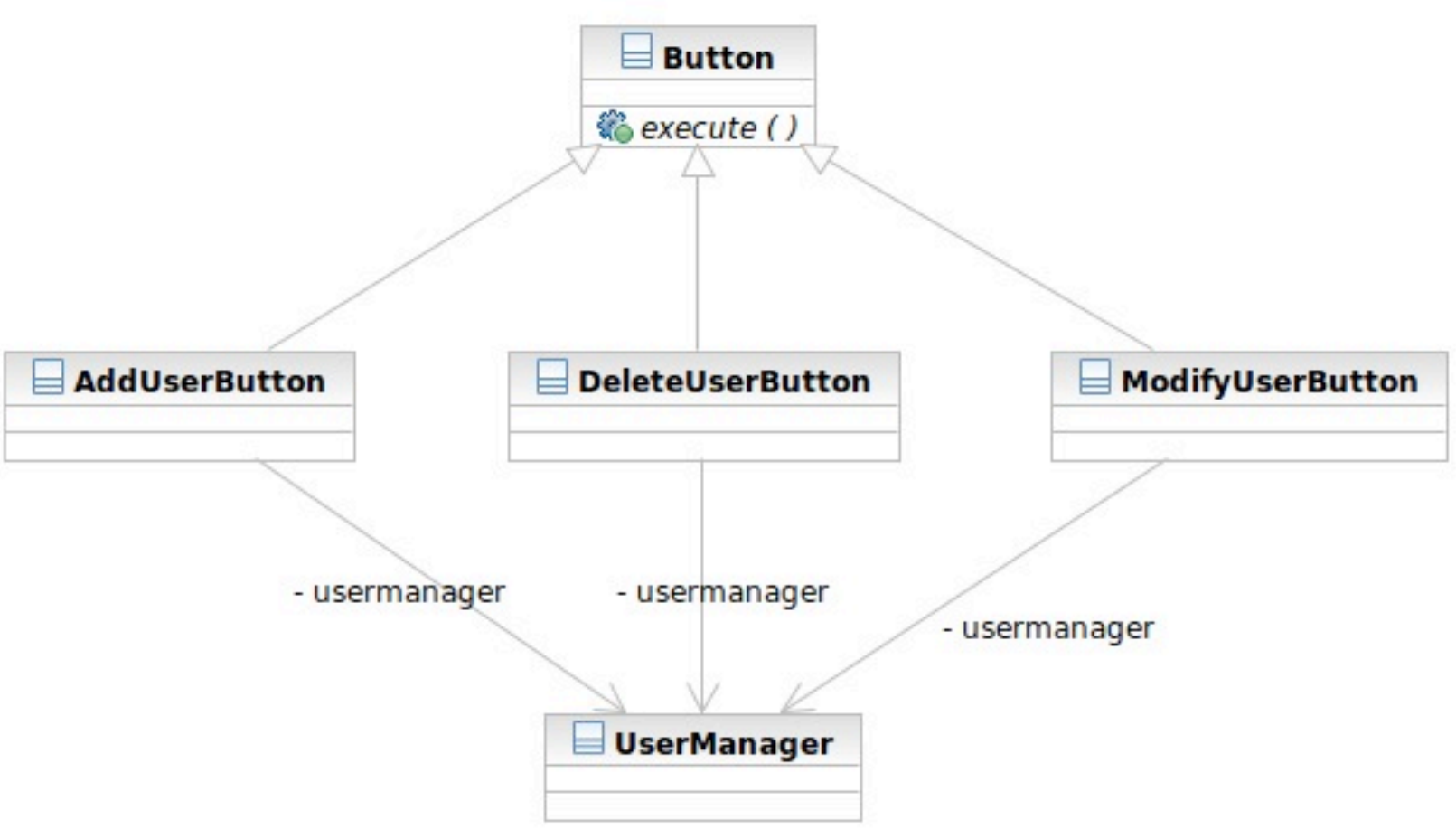


Command

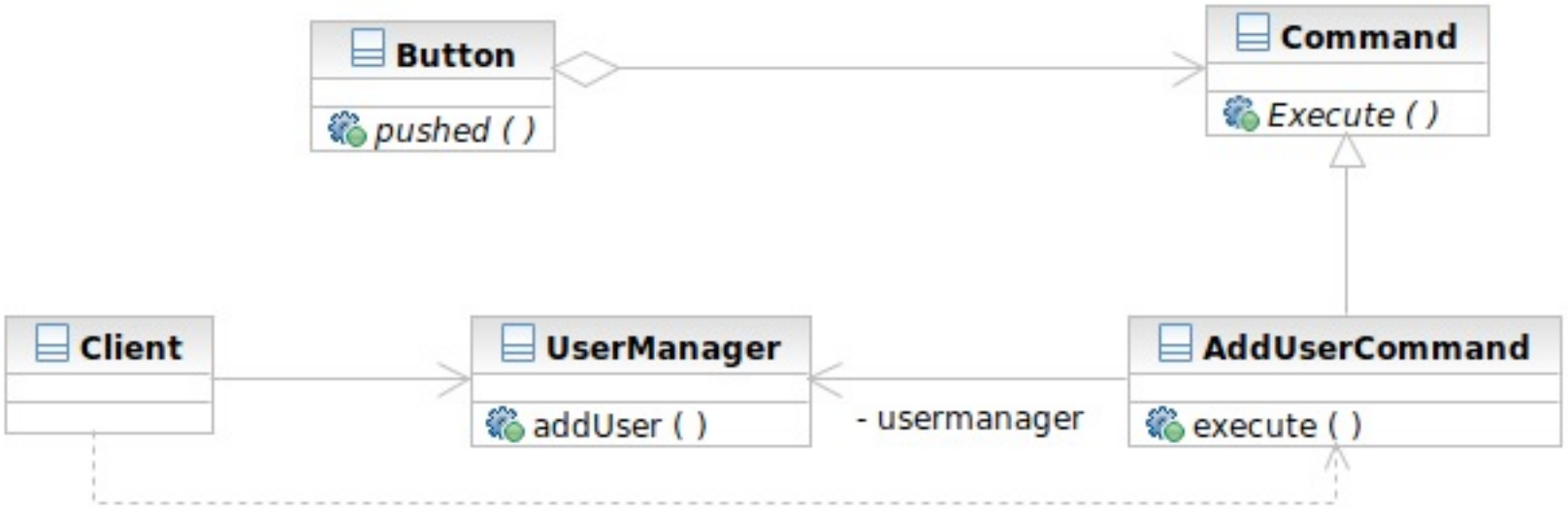
- What it is
  - An action encapsulated as an object
  - To be executed later by another client
  - Can be queued or composed
- Target problem
  - Customize the behavior of reusable widgets
  - Subclassing is not a good solution
    - You will have many derived class only to define custom behavior
 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

Without the Command Pattern

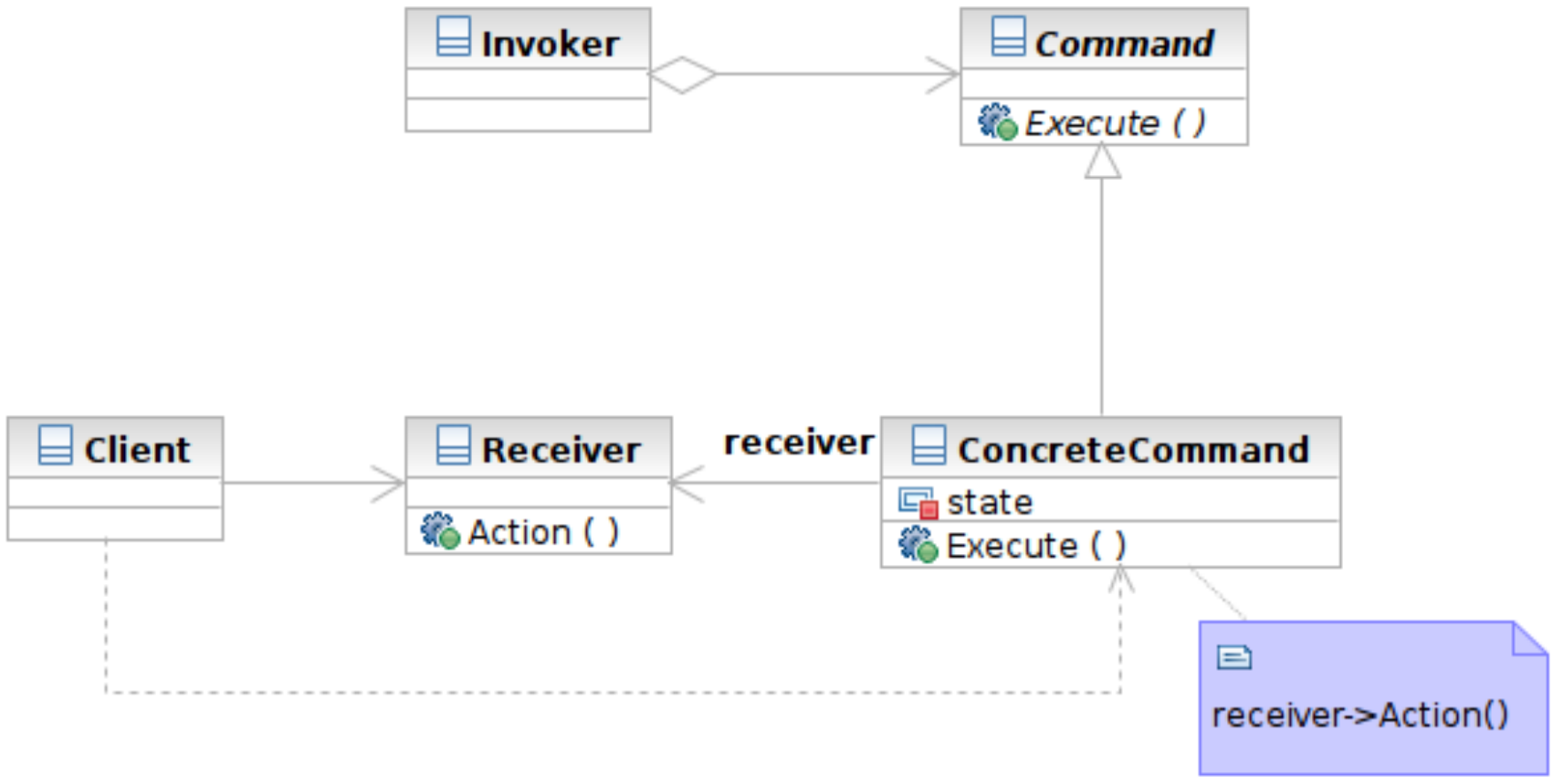
- A subclass for each widget instance



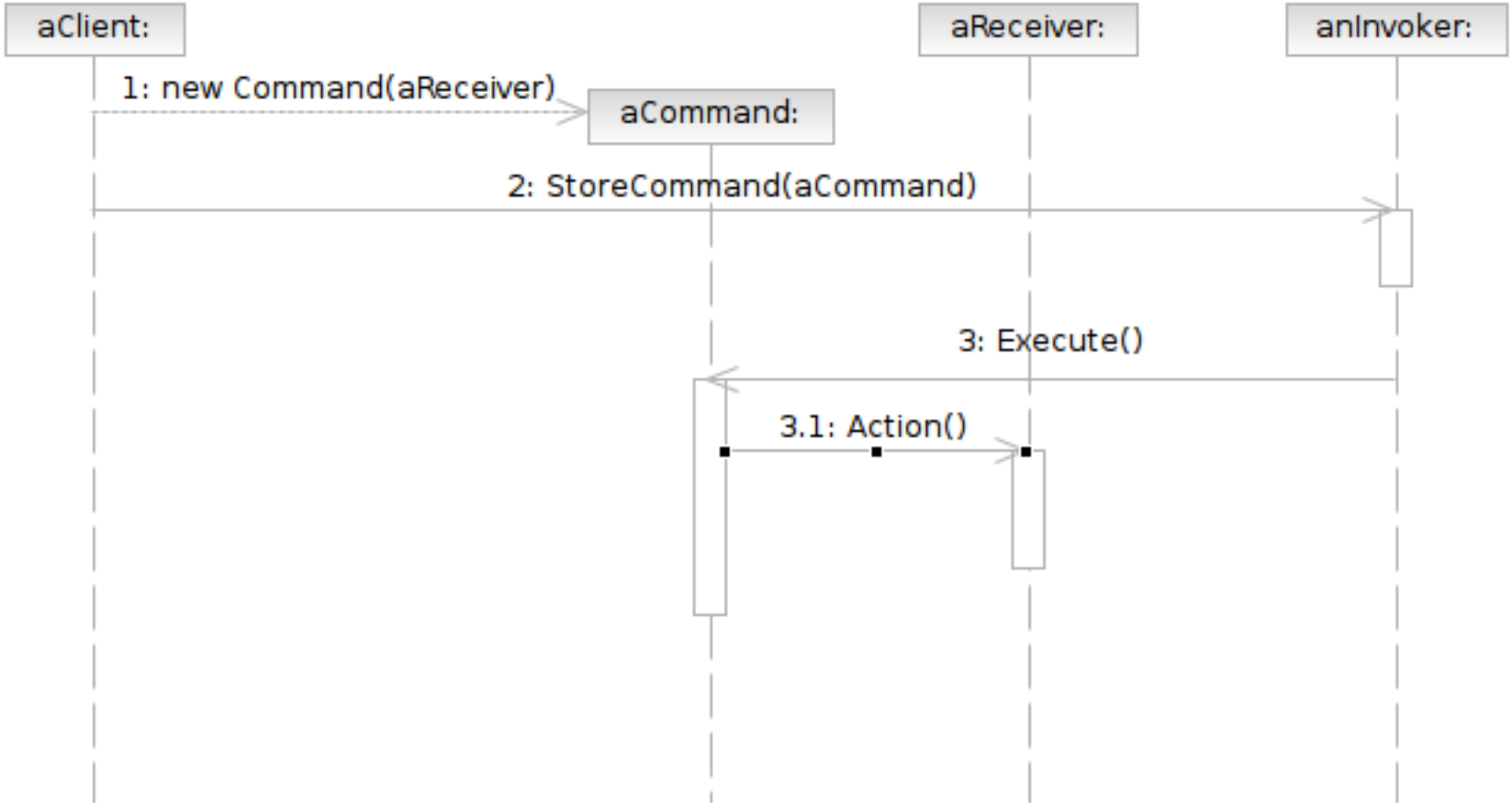
Applying the Pattern



Structure


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

Interaction



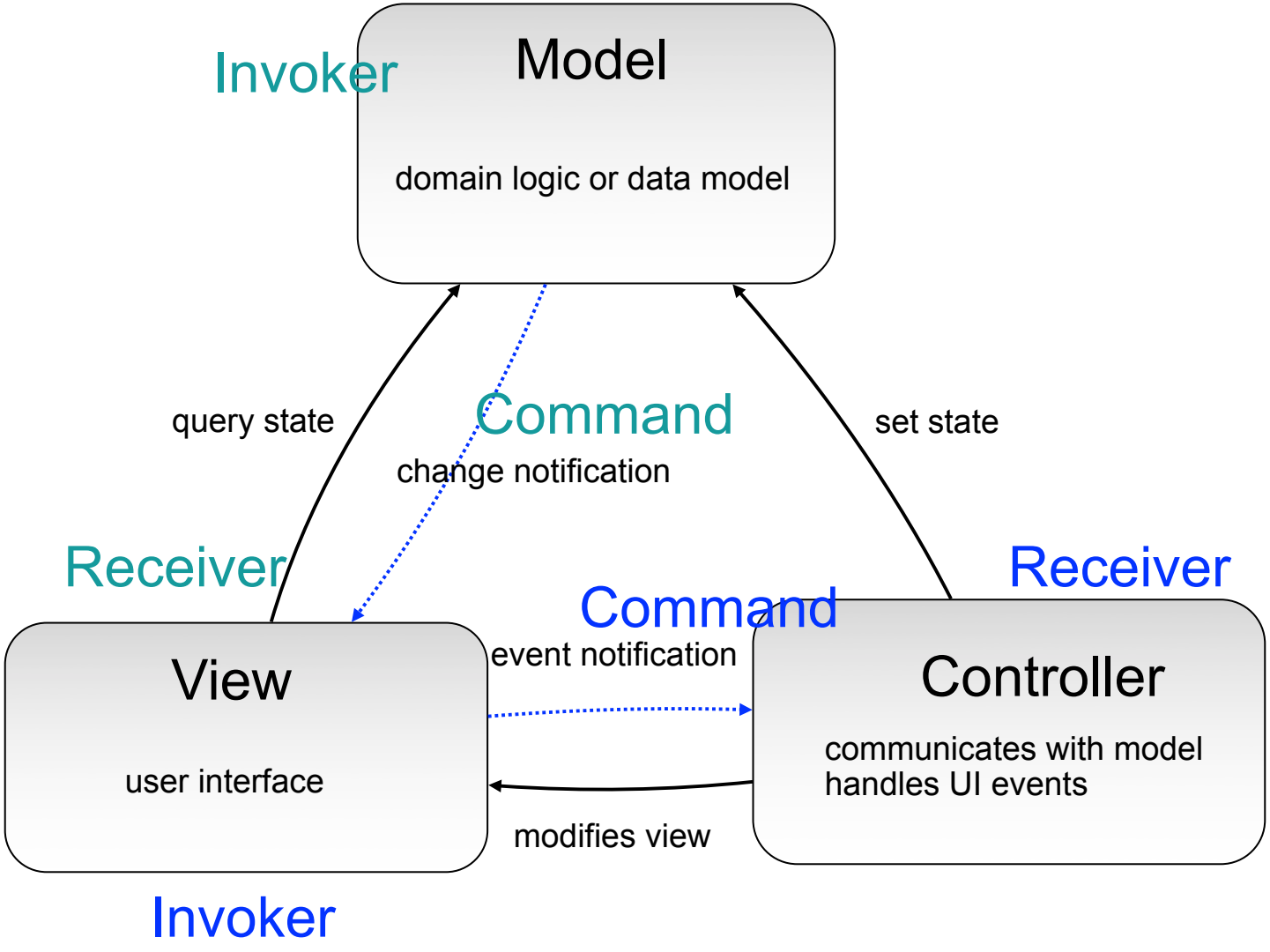
Participants

- 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

- 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



Template Method & Factory Method

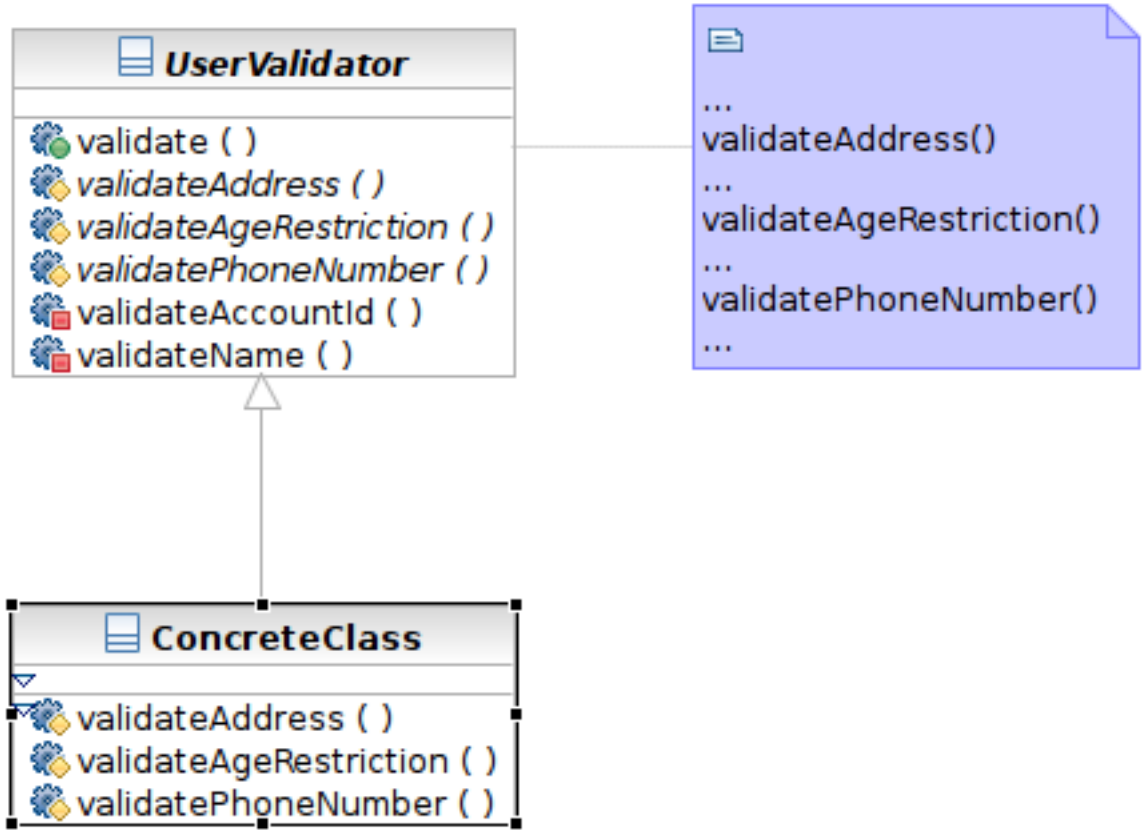
- What Template Method is
  - A method that serves as the 'skeleton' or structure of an algorithm
  - Abstract methods called by the template method is implemented in derived classes
- Target problems
  - Client profile validators for different countries
  - The generic quick sort algorithms for user-defined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

Without the Template Method Pattern

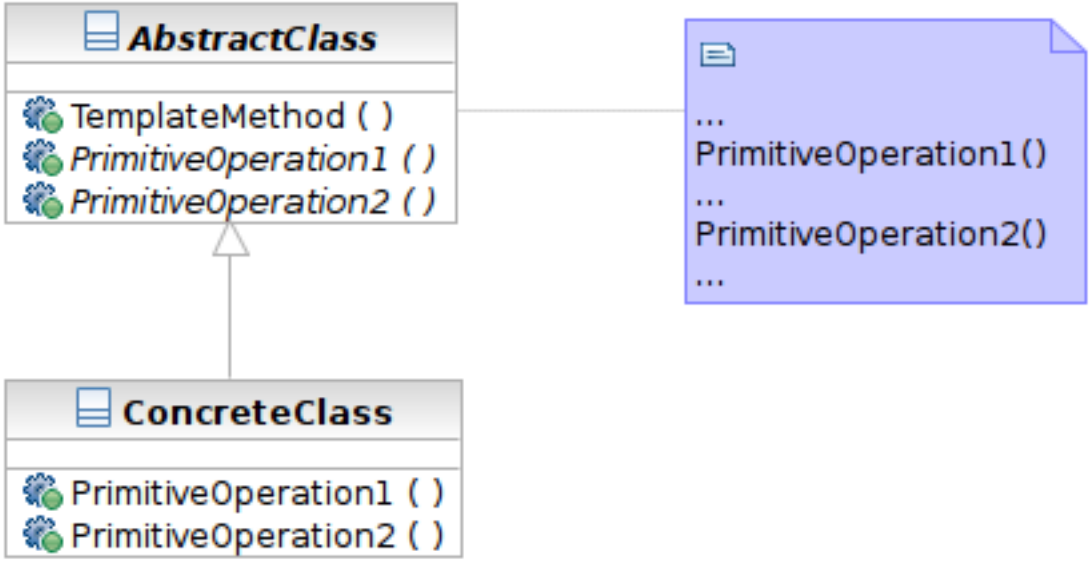

```
ValidateUSUser () {
   // validate account id
   // validate name
   // validate age restriction (US)
   // validate phone number (US)
   // validate address (US)
}
```

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

Applying the Pattern



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.

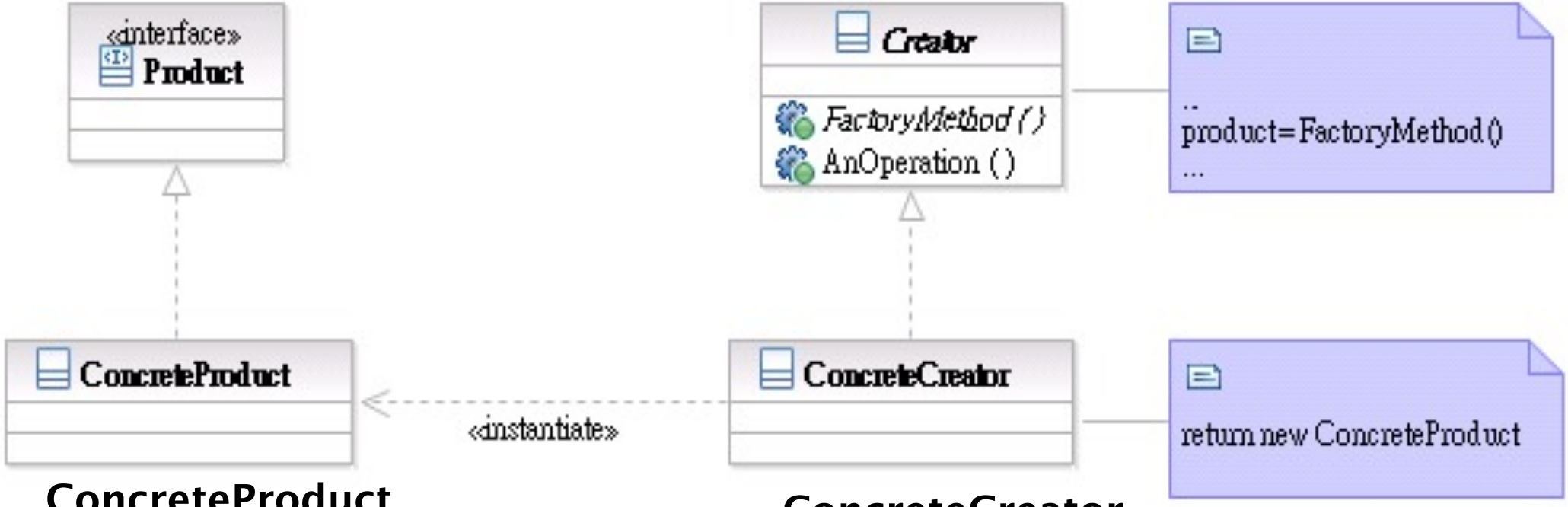
Factory Method

- 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

Creator
declares the factory method returning an object of type Product



ConcreteProduct
implements the Product interface

ConcreteCreator
overrides the factory method to return an instance of a ConcreteProduct

Transparent Access: Proxy & Decorator

- 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

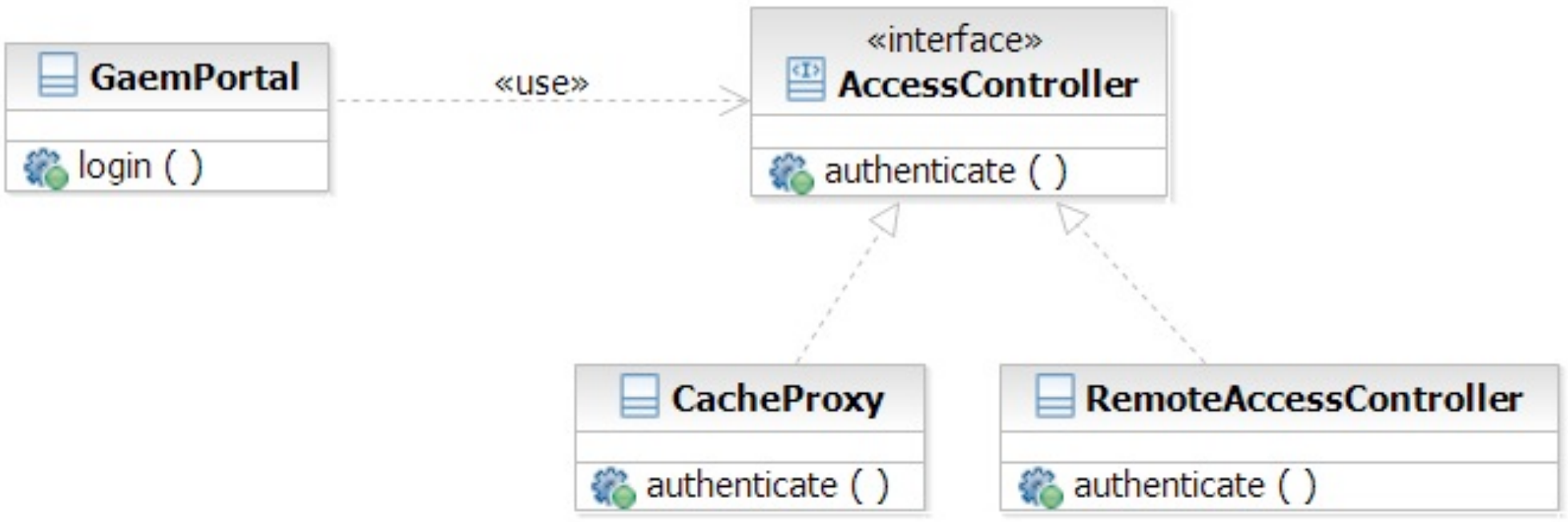
- What it is
  - A surrogate or placeholder for another object to control access to it
  - In a transparent way (having the same interface as the proxied object)
- Target problem
  - Access control between the client and your system, such as
  - Lazy loading of image or other resources
  - Transparent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

Without the Proxy Pattern

- 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



Decorator

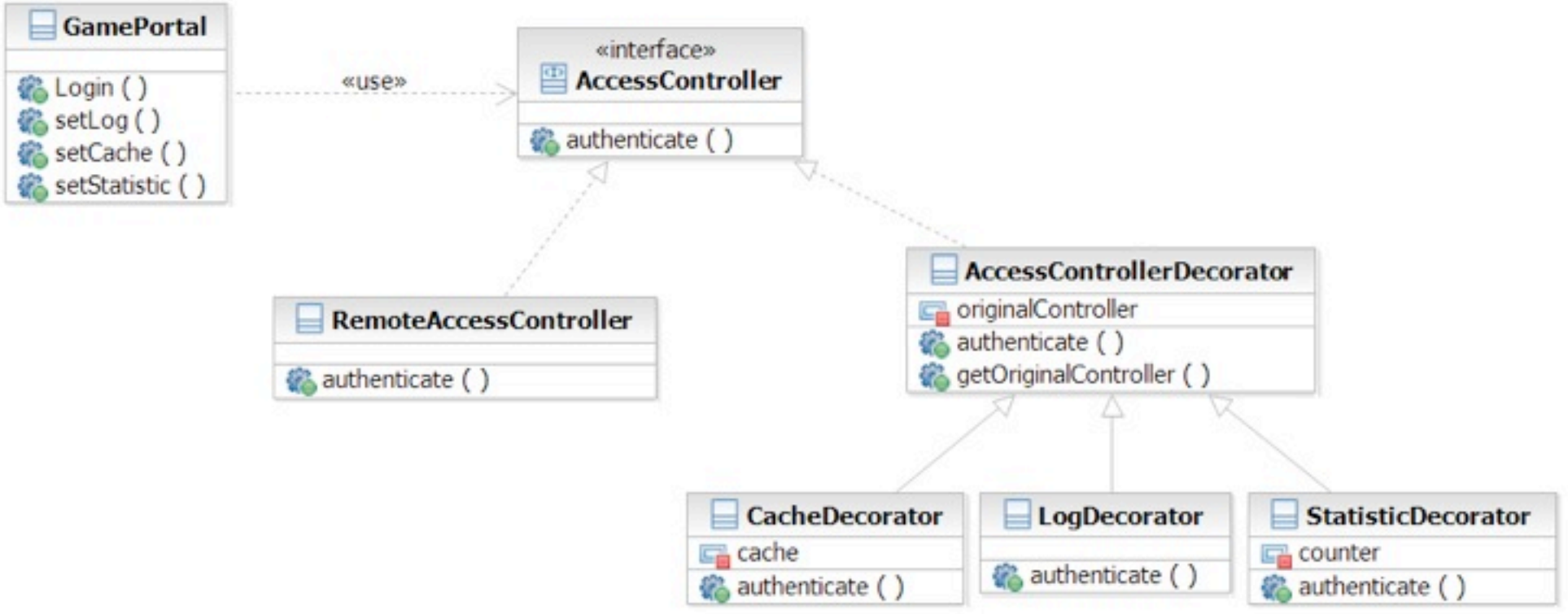
- What it is
  - Attaching additional responsibilities to an object dynamically
  - An alternative to subclassing
- Target Problem
  - Enabling/disabling additional features at runtime
    - Caching, logging
  - Dynamic composition of these features (subclassing is infeasible)

Without the Decorator Pattern

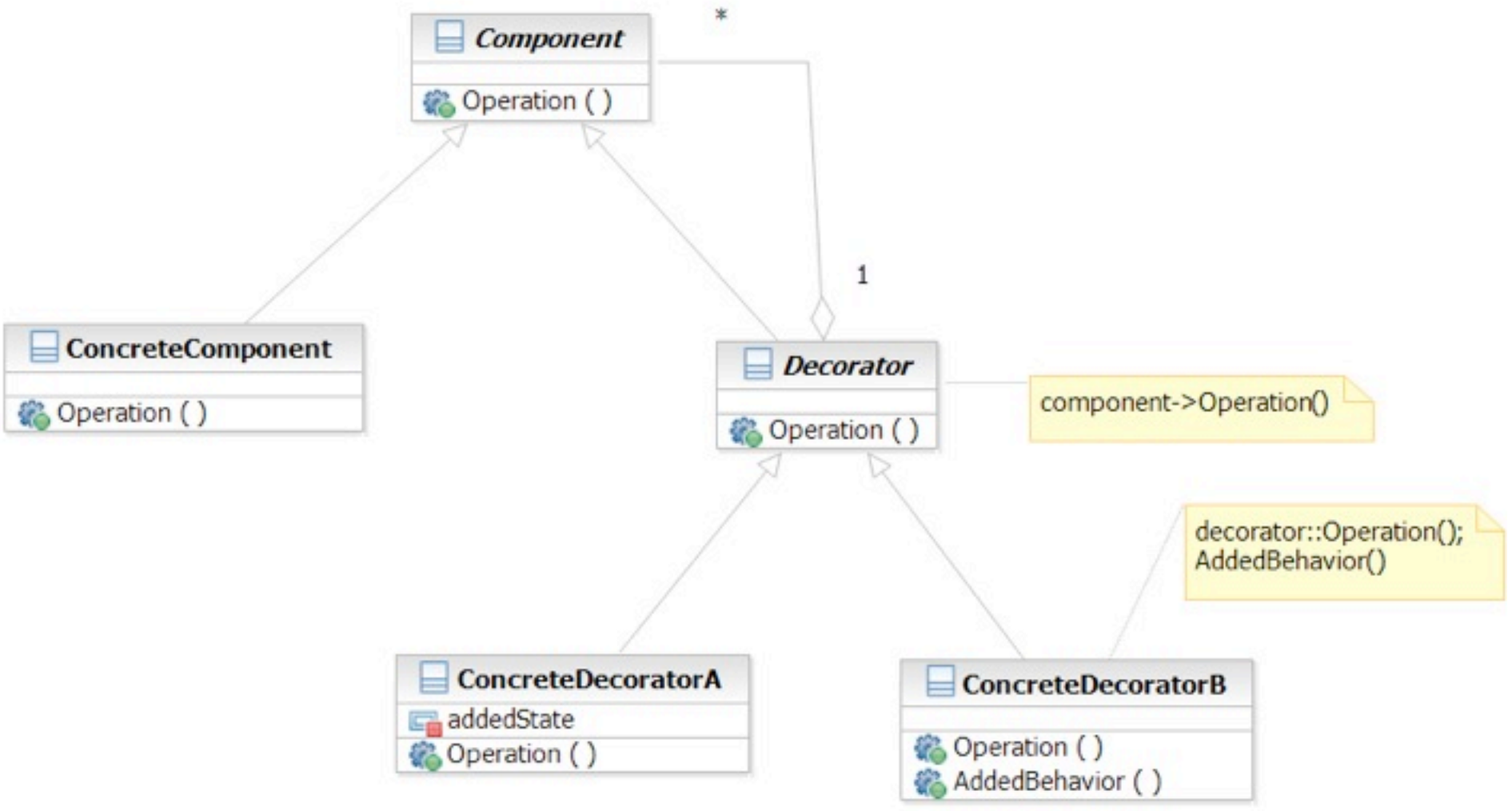
- The added functionality needs to be coded in the decorated class:

```
If (logIsEnabled) {
    // log function entry
}
// function body
If (statisticsIsEnabled) {
  // update statistics
}
If (logIsEnabled) {
  // log function exit
}
```

Applying the Pattern



Structure


Component
Operation ( )
*
1
ConcreteComponent
Operation ( )
Decorator
Operation ( )
component->Operation()
decorator::Operation();
AddedBehavior()
ConcreteDecoratorA
addedState
Operation ( )
ConcreteDecoratorB
Operation ( )
AddedBehavior ( )

State

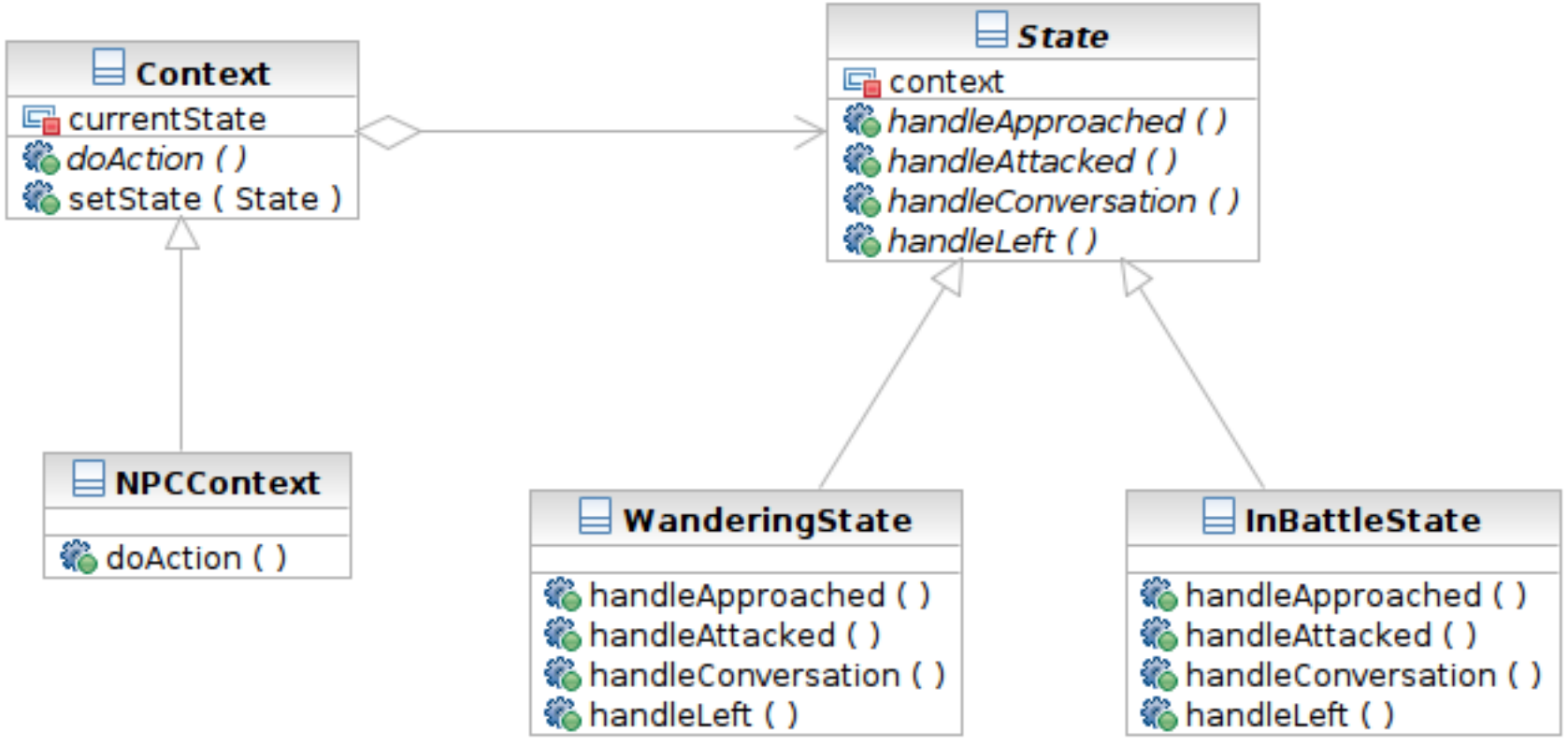
- What it is
  - Allowing an object to change its behavior when its internal state changes
- Target Problem
  - State machines
    - Network protocols (e.g. TCP state machine)
    - Drawing tools
    - Document editors
    - Games
    - Complex business rules

Without the State Pattern

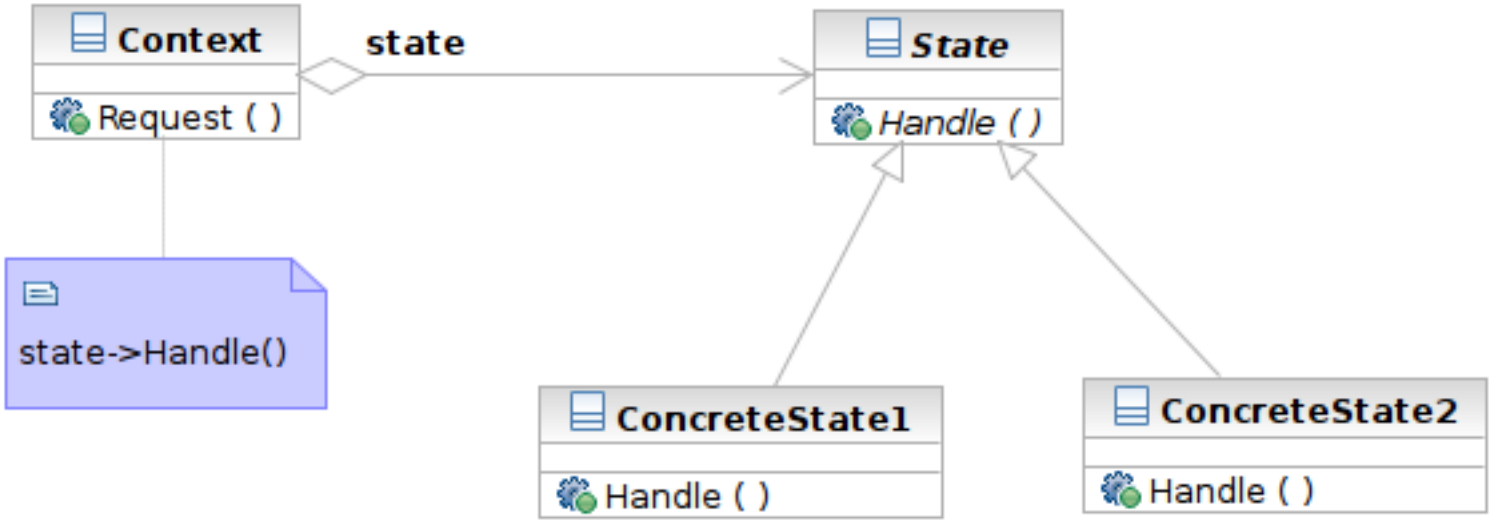
- Use if or switch structure to produce lengthy functions

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

Applying the Pattern



Structure


Context
Request ( )
state
State
Handle ( )
state->Handle()
ConcreteState1
Handle ( )
ConcreteState2
Handle ( )

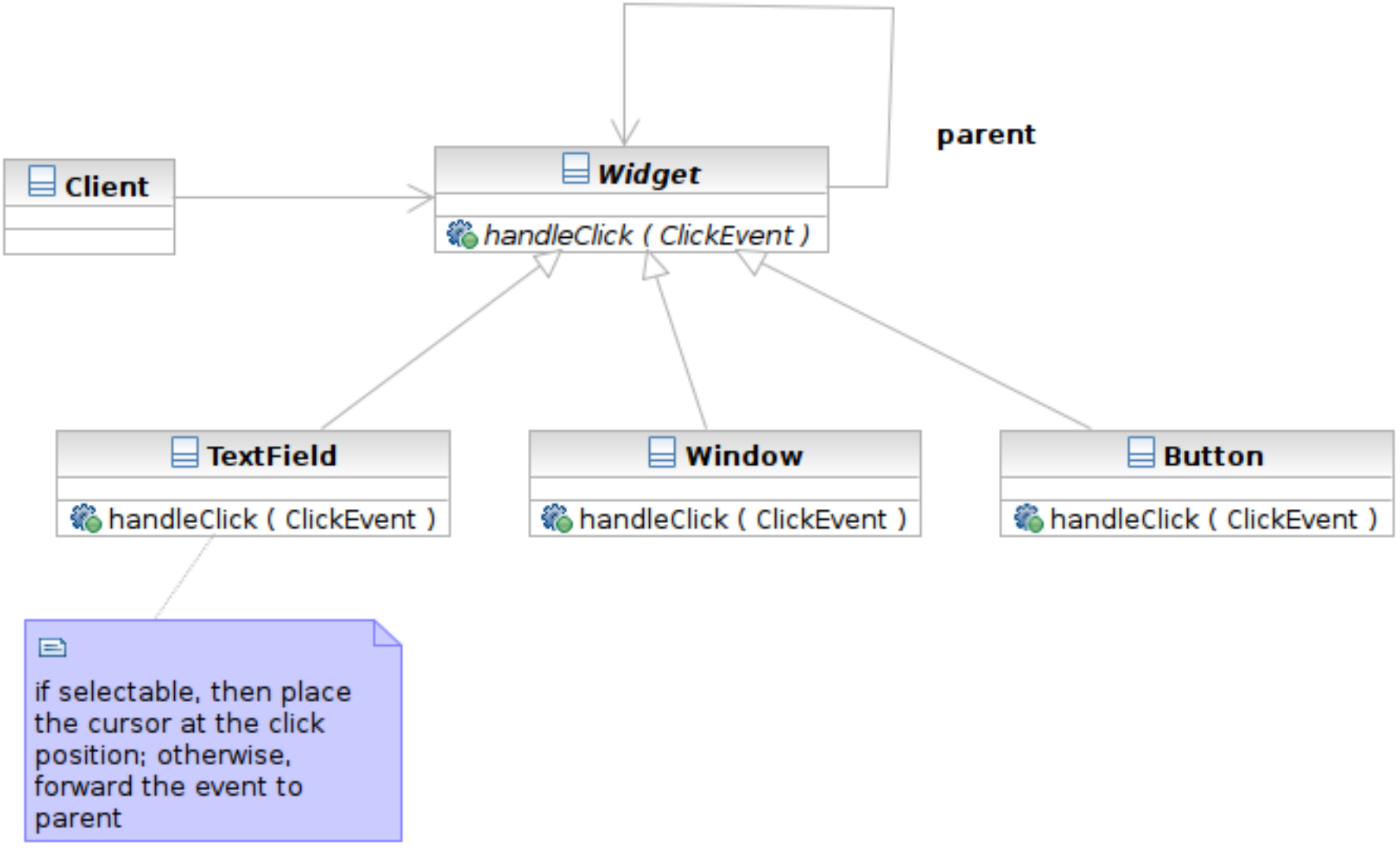
Participants

- 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.

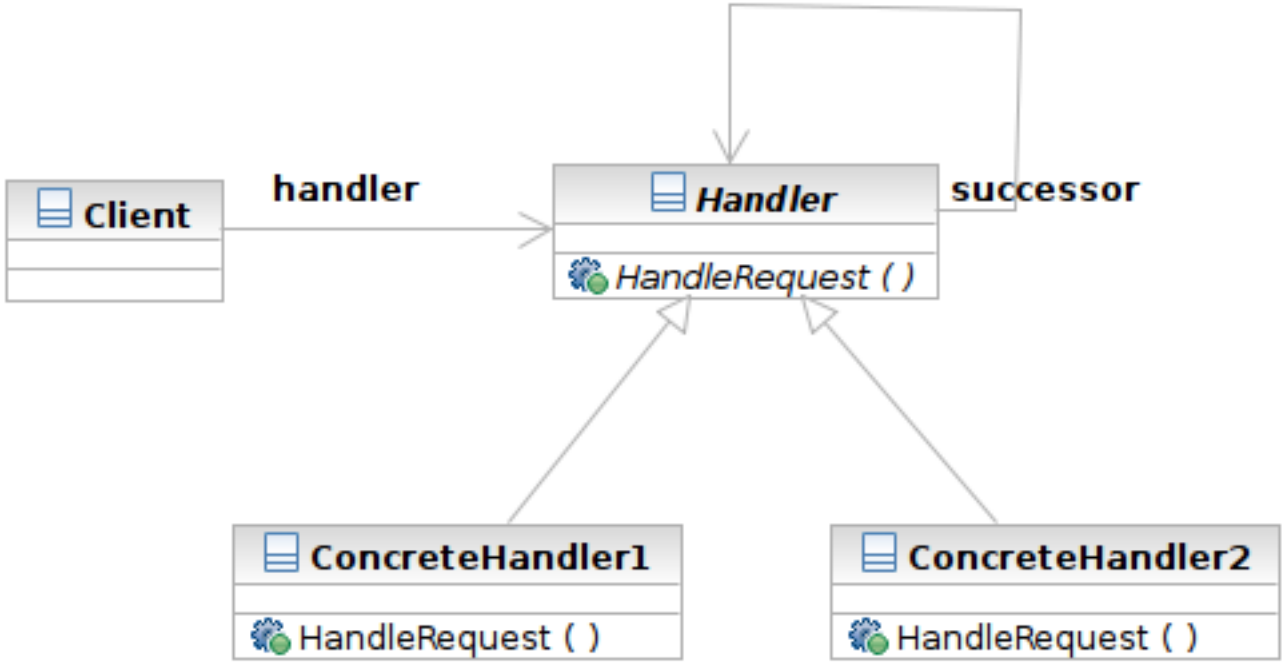
Chain of Responsibility

- 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

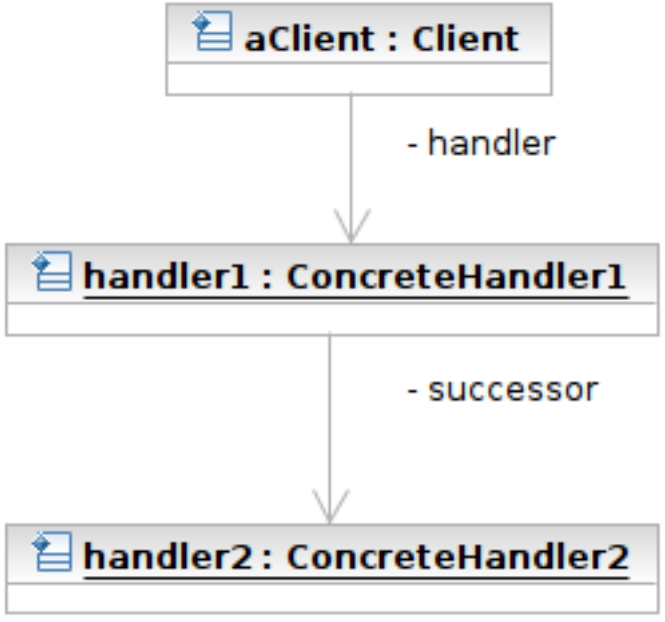
Applying the Pattern



Structure


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

Prototype

- 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

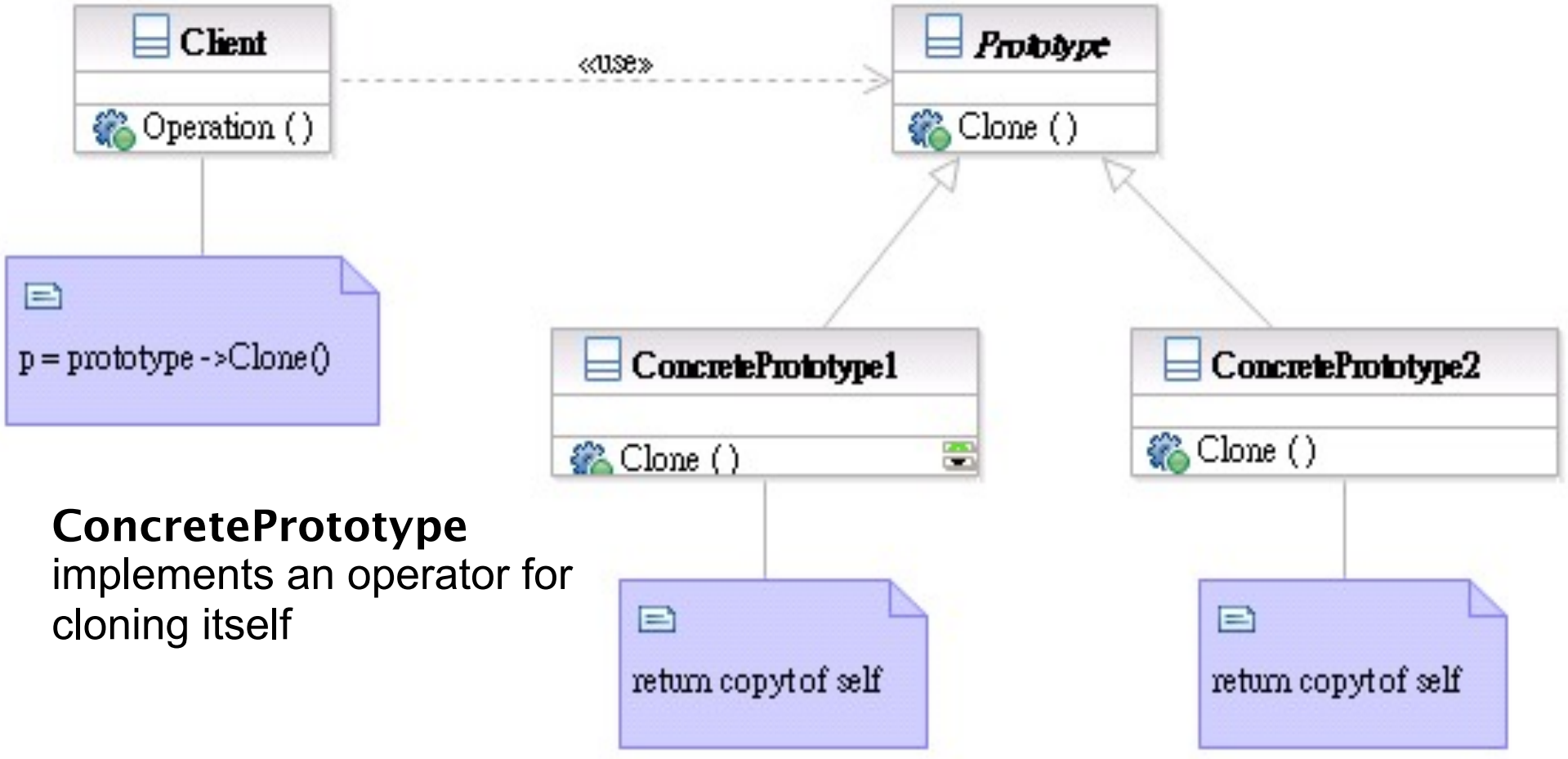
(Suppose instantiation of ShoppingCart requires access of remote system, which is expensive)

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

Structure

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

- 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

- 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

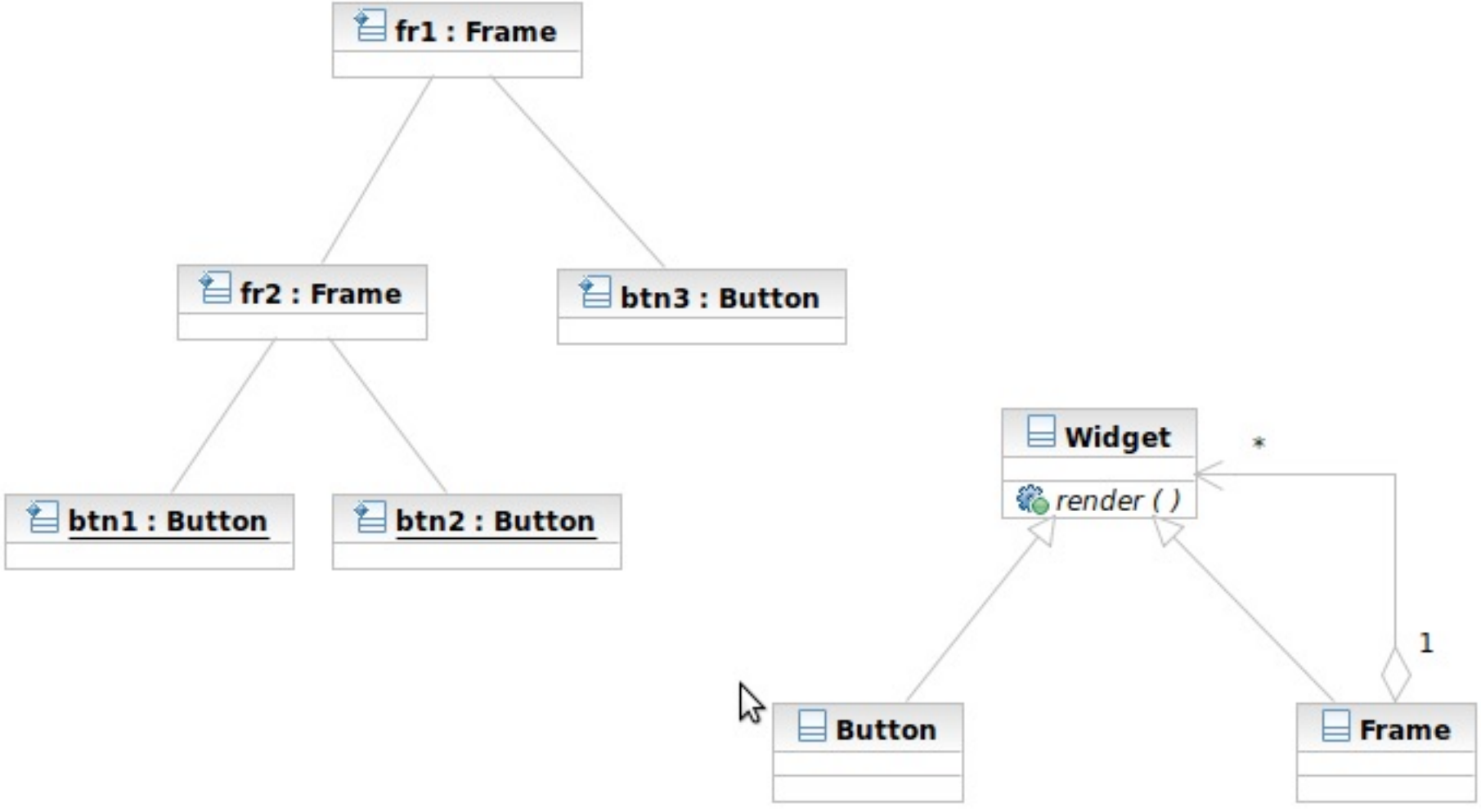
Sample Problem

- 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

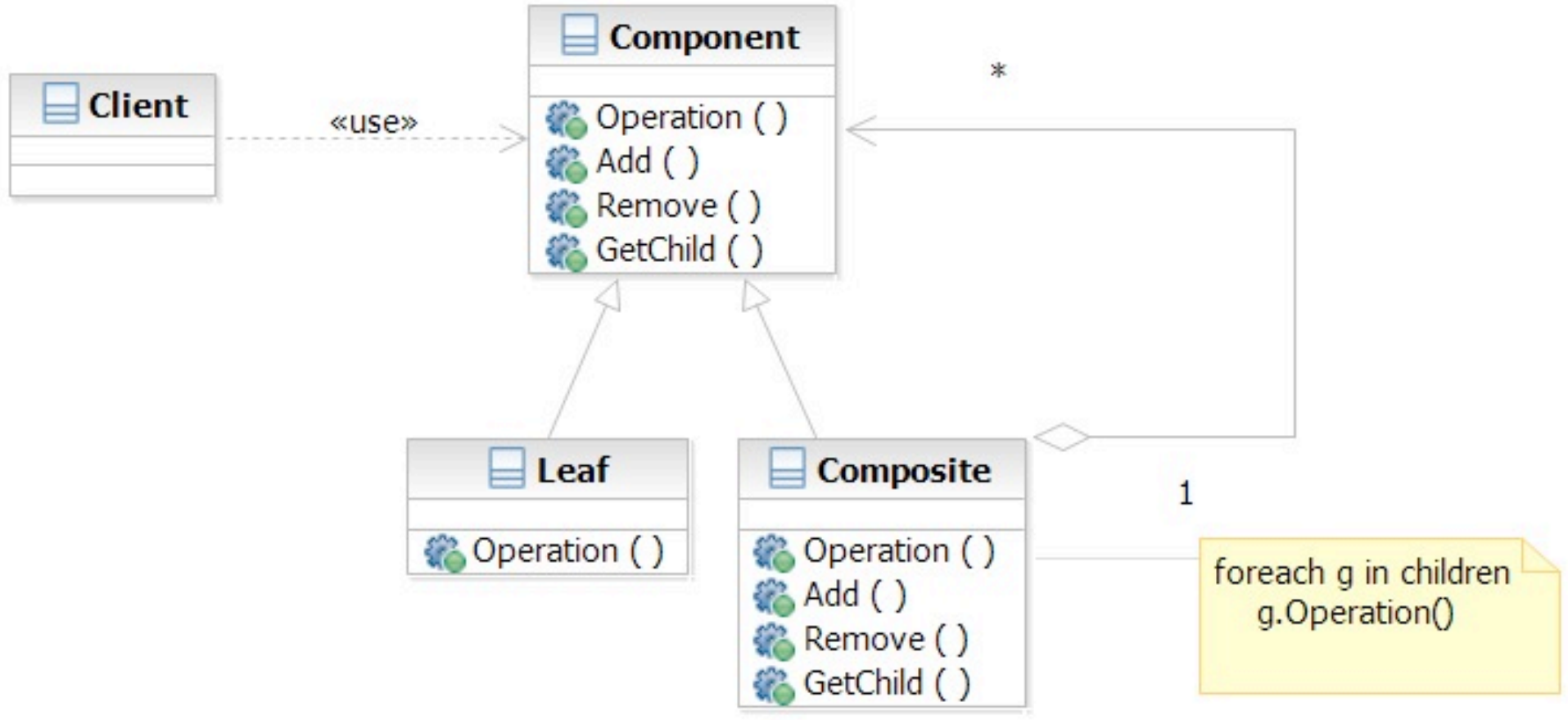
Composite

- 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



Structure/Participants



Composite and Builder

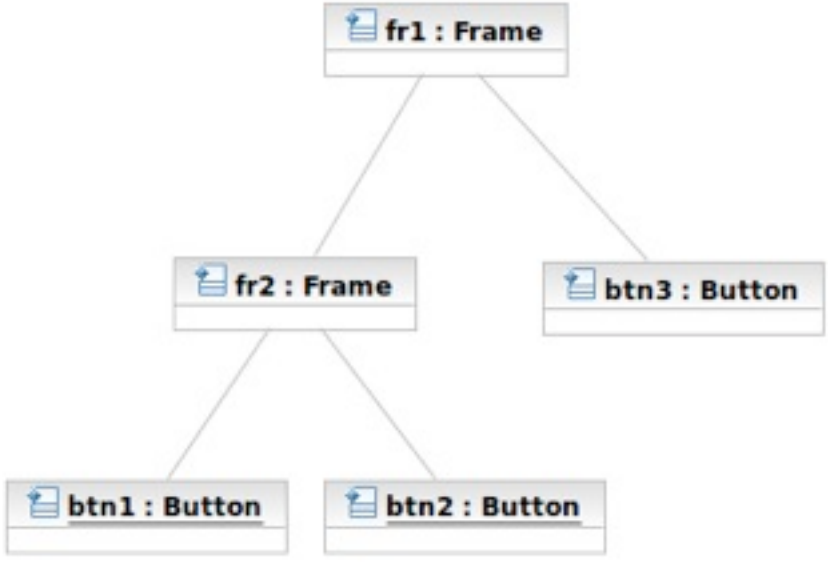
- The composite structure is often built with the builder
- What Builder is?
  - Separation of the construction of a complex object from its representation
  - The construction process can optionally create different representations
- Target Problem
  - Parser reading source file to represent it as parse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

Apply the Builder Pattern

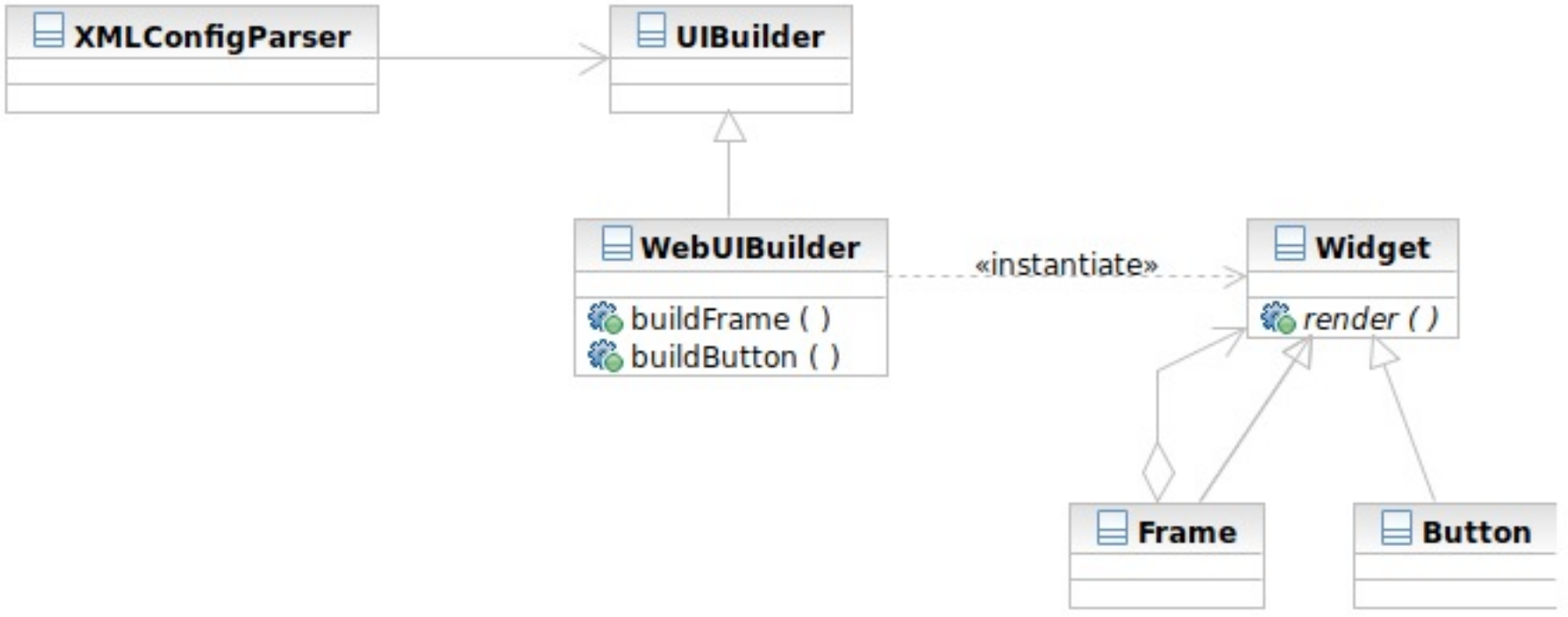
Input Config:

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

Parsed result:

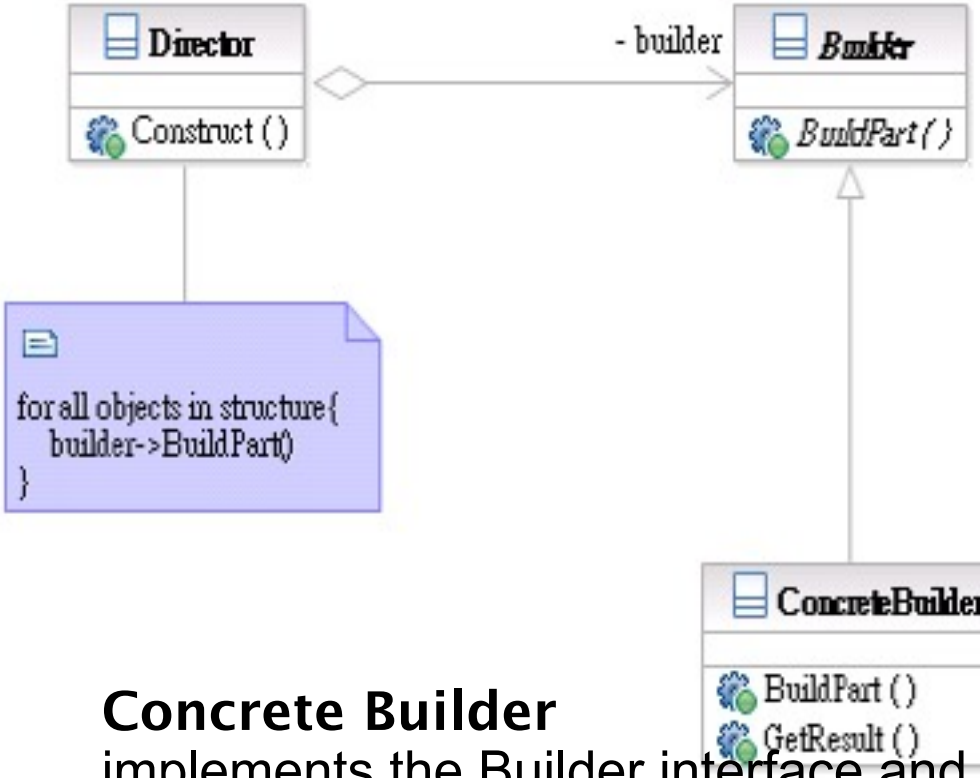


Apply the Builder Pattern



Structure

Director
constructs an object using
the Builder interface

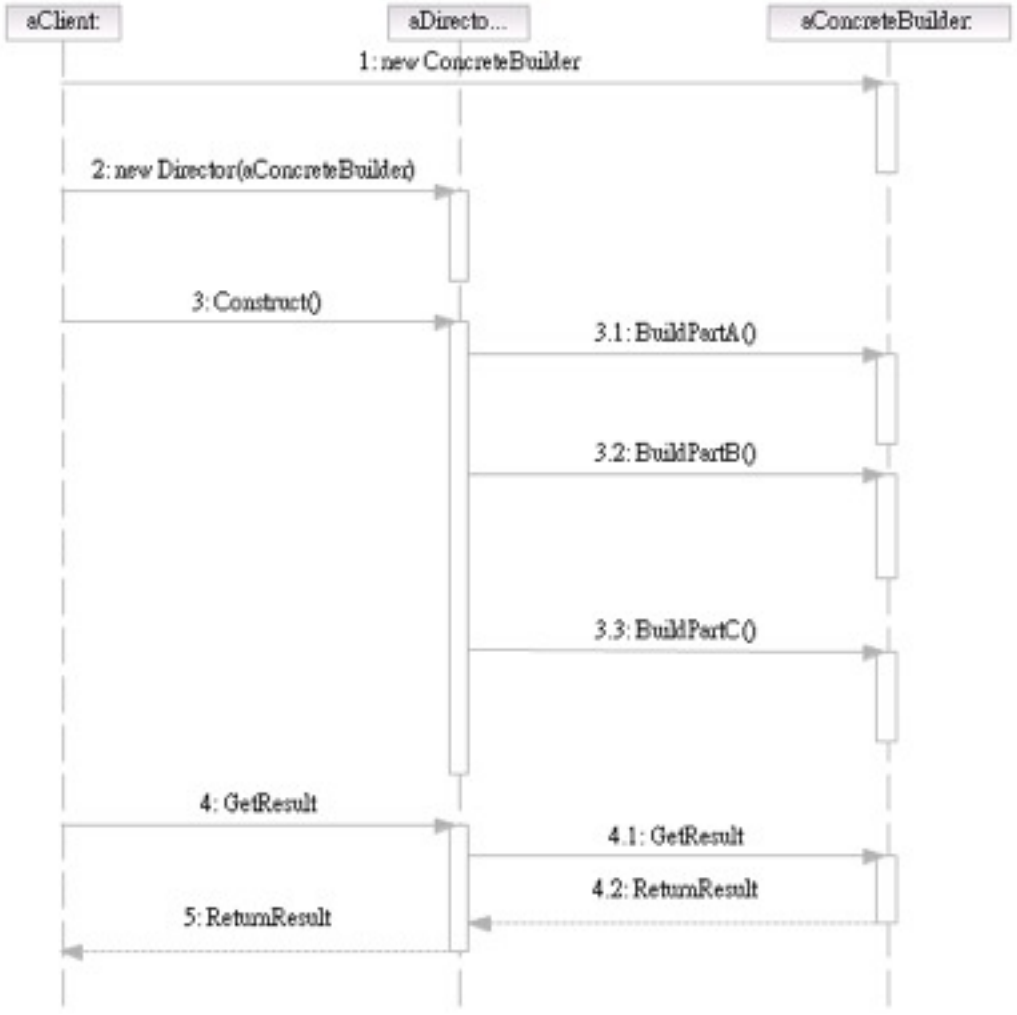


Builder
specifies an interface for
creating parts of a Product object

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

Product
represents the final product
and
its constituent parts

Builder Interaction



Participants

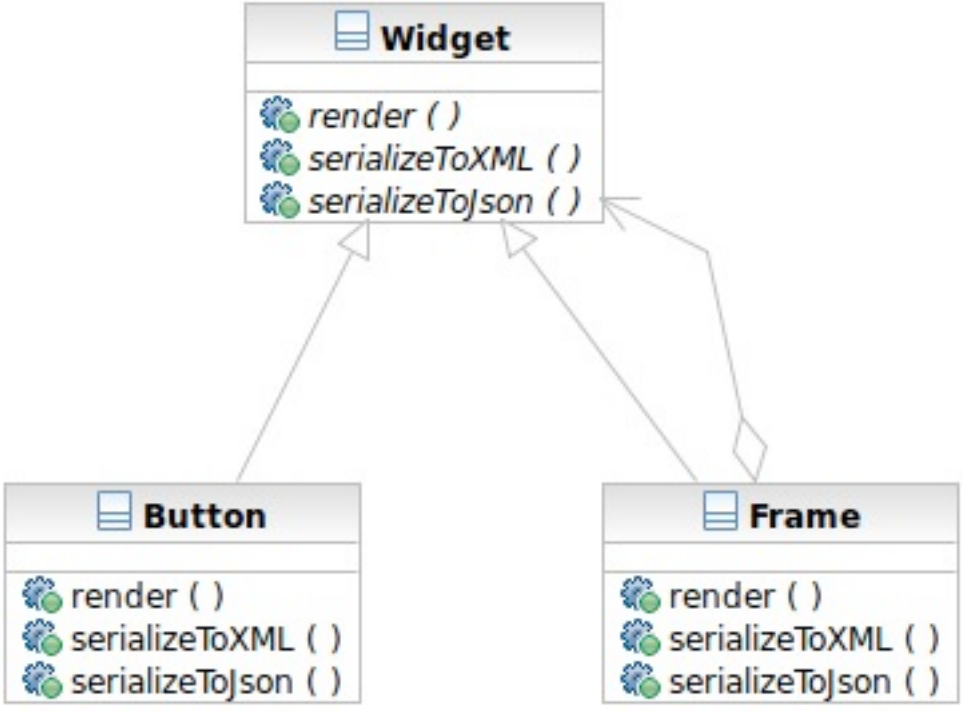
- Class **Builder** specifies an interface for creating parts of a Product object.
- Class **ConcreteBuilder** implements the Builder interface and keeps track of the product and objects.
- Class **Director** constructs an object using the Builder interface.
- Class **Product** represents the final product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

Visitor and Composite

- 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

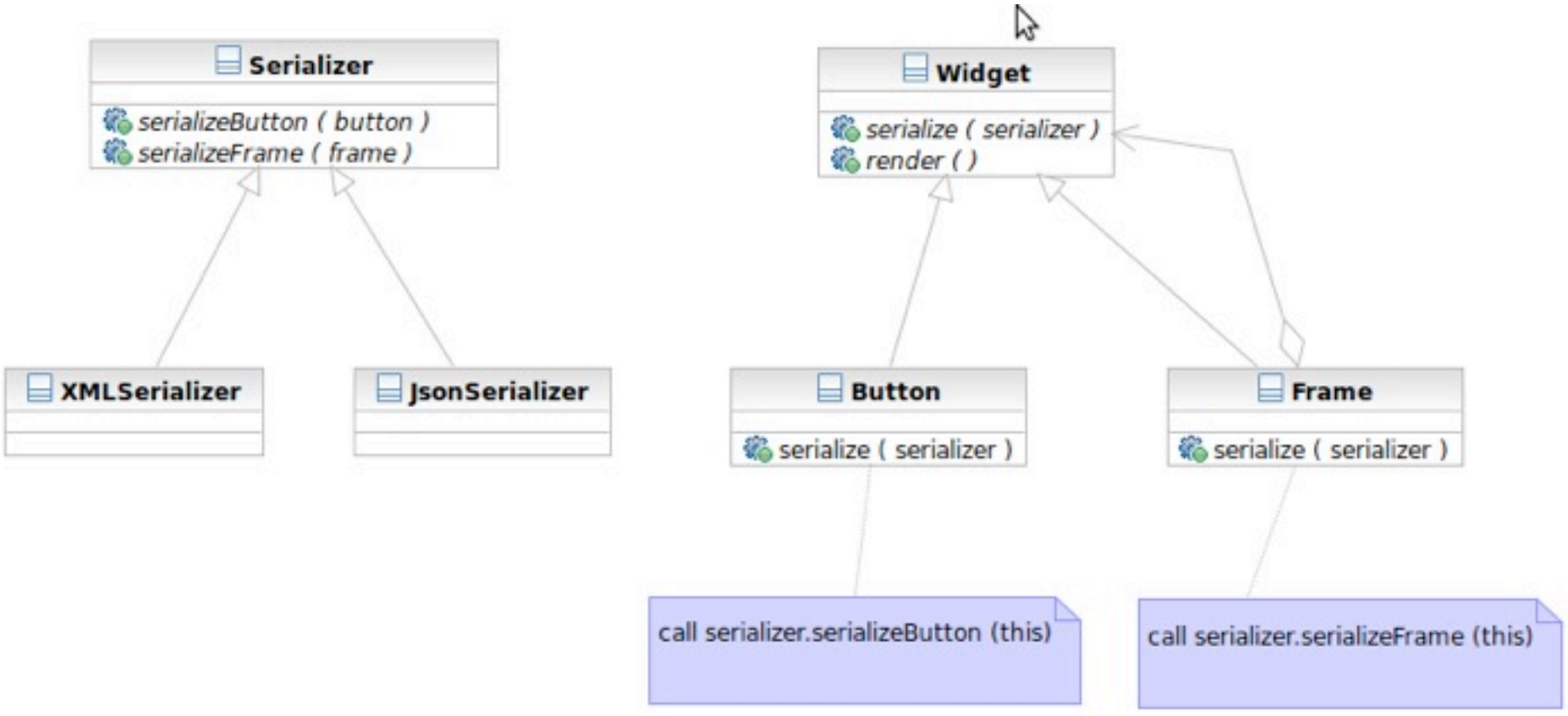
Without the Visitor Pattern

- Adding new operations to the whole class family:

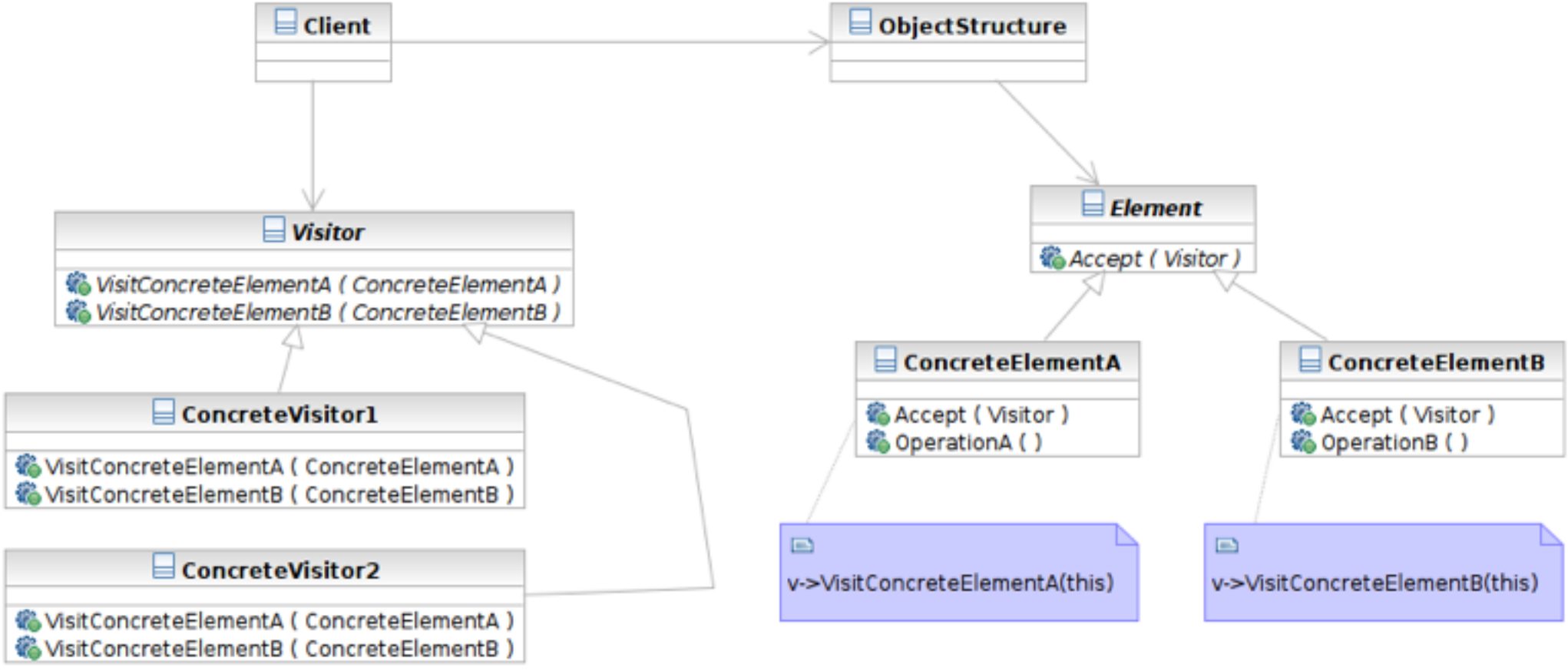


Applying the Pattern

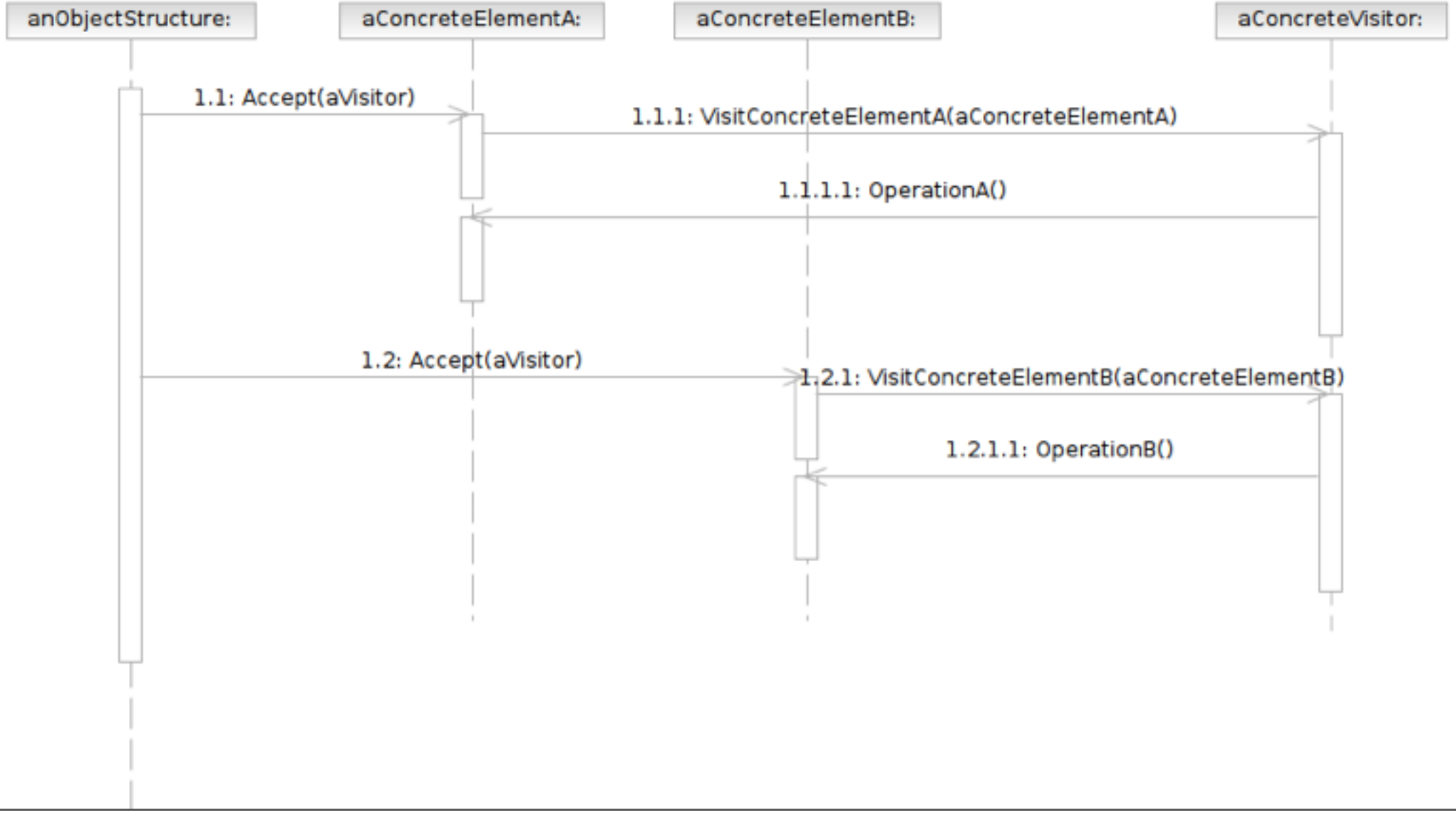
- The operations to serialize to Json and XML are extracted into visitors



Structure


Client
ObjectStructure
Visitor
VisitConcreteElementA (ConcreteElementA)
VisitConcreteElementB (ConcreteElementB)
Element
Accept (Visitor)
ConcreteVisitor1
VisitConcreteElementA (ConcreteElementA)
VisitConcreteElementB (ConcreteElementB)
ConcreteElementA
Accept (Visitor)
OperationA ( )
ConcreteElementB
Accept (Visitor)
OperationB ( )
ConcreteVisitor2
VisitConcreteElementA (ConcreteElementA)
VisitConcreteElementB (ConcreteElementB)
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Interaction



Participants

- 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

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