Laboratorio di Progettazione di Sistemi Software
Design Pattern Comportamentali

Riccardo Solmi

Indice degli argomenti

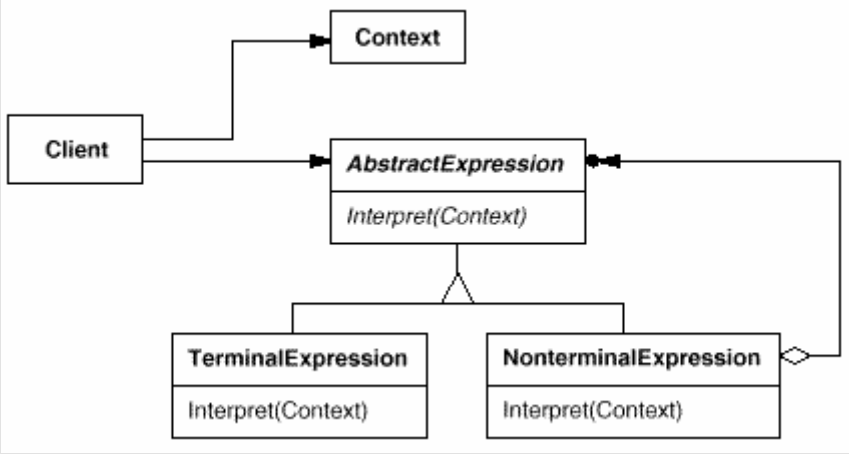
- Catalogo di Design Patterns comportamentali:
  - Interpreter
  - Template Method
  - Visitor
  - Iterator
  - Strategy
  - State
  - Observer
  - Command
  - Null Object

Interpreter

- Intent
  - Represent an operation to be performed on the elements of an object structure. Interpreter lets you define the operation into the classes of the elements on which it operates.
  - NB. The *behavior* of an interpreter *operation* is restricted only by the constraint that must include a traversal of an [object] structure (What? Any. How? Traversal)
  - NB. Popular definitions for use cases with an implied context:
    - The "interpreter [operation] of a language" is an interpreter implementing the execution semantics (traversing…)
    - The "interpreter [*program*]" is an interpreter operation together with a parser for the input sources and an unparser for the output .
  - *Was*: Given a language, define a represention *for its grammar* along with an interpreter that uses the representation to interpret sentences in the language.
- Applicability
  - An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their classes
  - The operations over the structure rarely change, but you often want to extend the object structure with new classes

Interpreter /2

- **Structure**



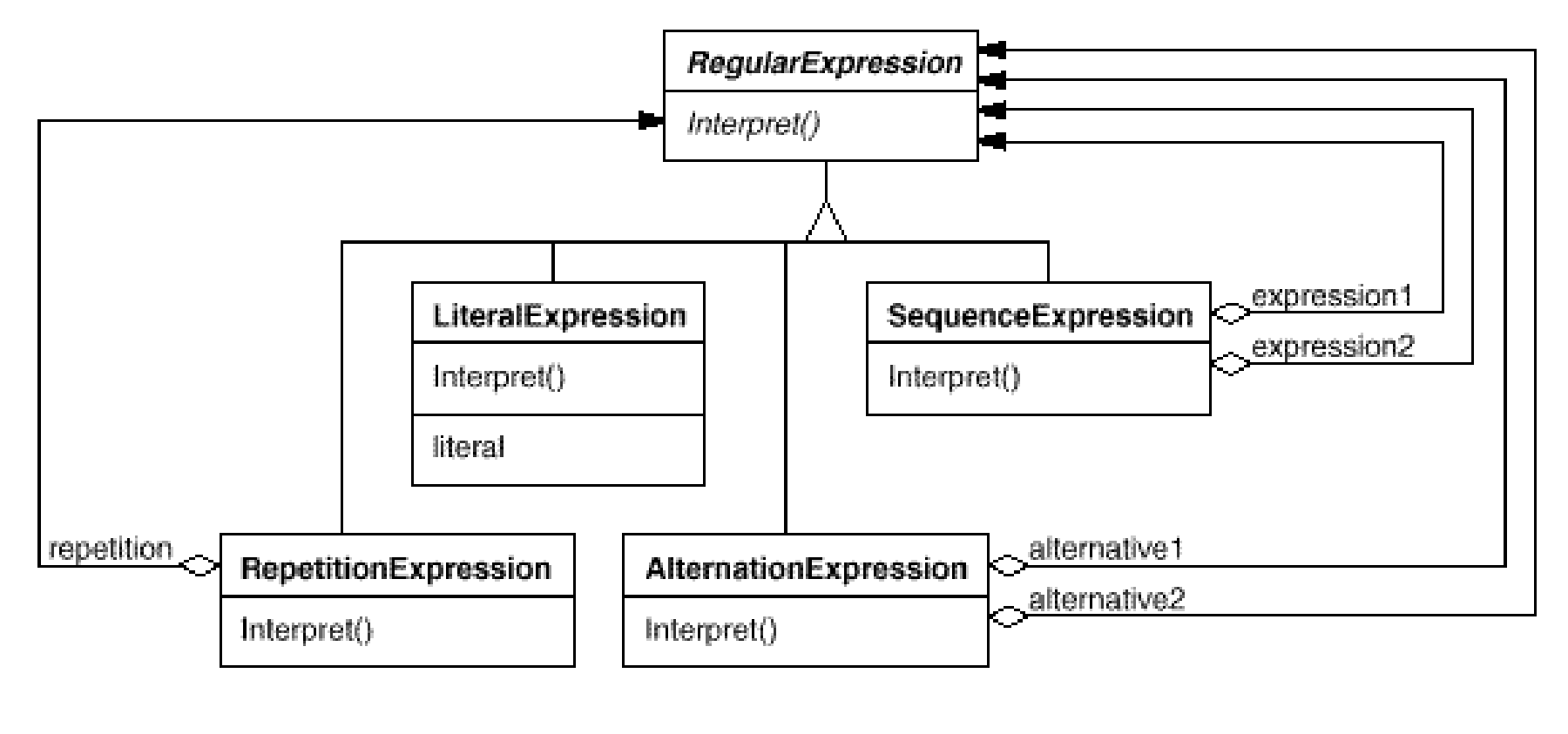
- **Participants**
- **AbstractExpression**
  - declares an abstract Interpret operation that is common to all nodes in the object structure.
- **TerminalExpression**
  - implements an Interpret operation associated with the *leaves* of the object structure.
  - an instance is required for every entity without associations to the object structure.
- **NonterminalExpression**
  - implements an Interpret operation associated with the *internal nodes* of the object structure.
  - one such class is required for every entity having one or more associations to the object structure..
  - Interpret typically calls itself recursively on the variables representing the associations.
- **Context**
  - contains information that's global to the interpreter.
- **Client**
  - builds (or is given) an object structure representing a particular sentence in the language that the grammar defines.
  - invokes the Interpret operation

Interpreter /3

- Collaborations
  - The client builds (or is given) the sentence as an object structure. Then the client initializes the context and invokes the Interpret operation.
  - Each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression. The Interpret operation of each TerminalExpression defines the base case in the recursion.
  - The Interpret operations at each node use the context to store and access the state of the interpreter
- Consequences
  - It's easy to change and extend the object structure (classes).
  - Implementing the object structure is easy, too.
  - Complex object structures are hard to maintain.
  - If you keep creating new ways of interpreting an expression, then consider using other patterns to avoid changing the object structure.

Interpreter example 1

- Interprete di espressioni regolari

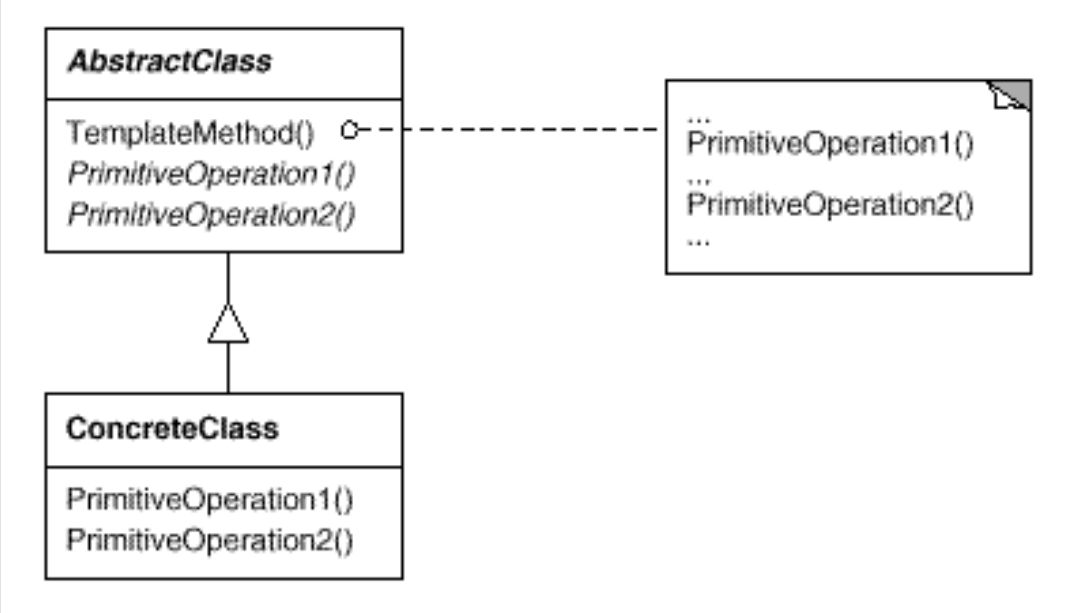


Template Method

- Intent
  - 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
- Applicability
  - 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

Template Method /2

- Structure



Participants

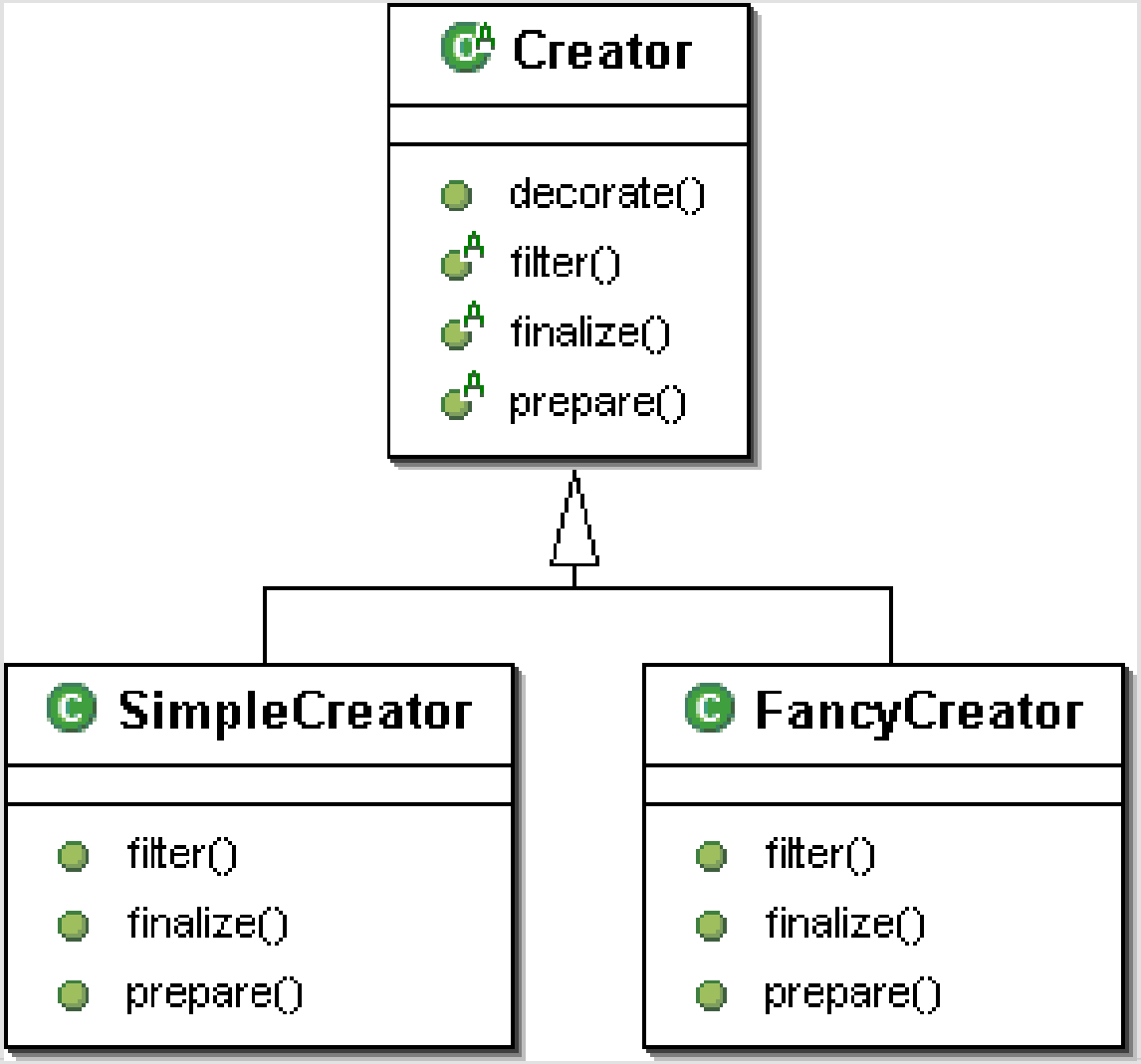
AbstractClass
defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.

- **ConcreteClass**
- implements the primitive operations to carry out subclass-specific steps of the algorithm.

Template Method /3

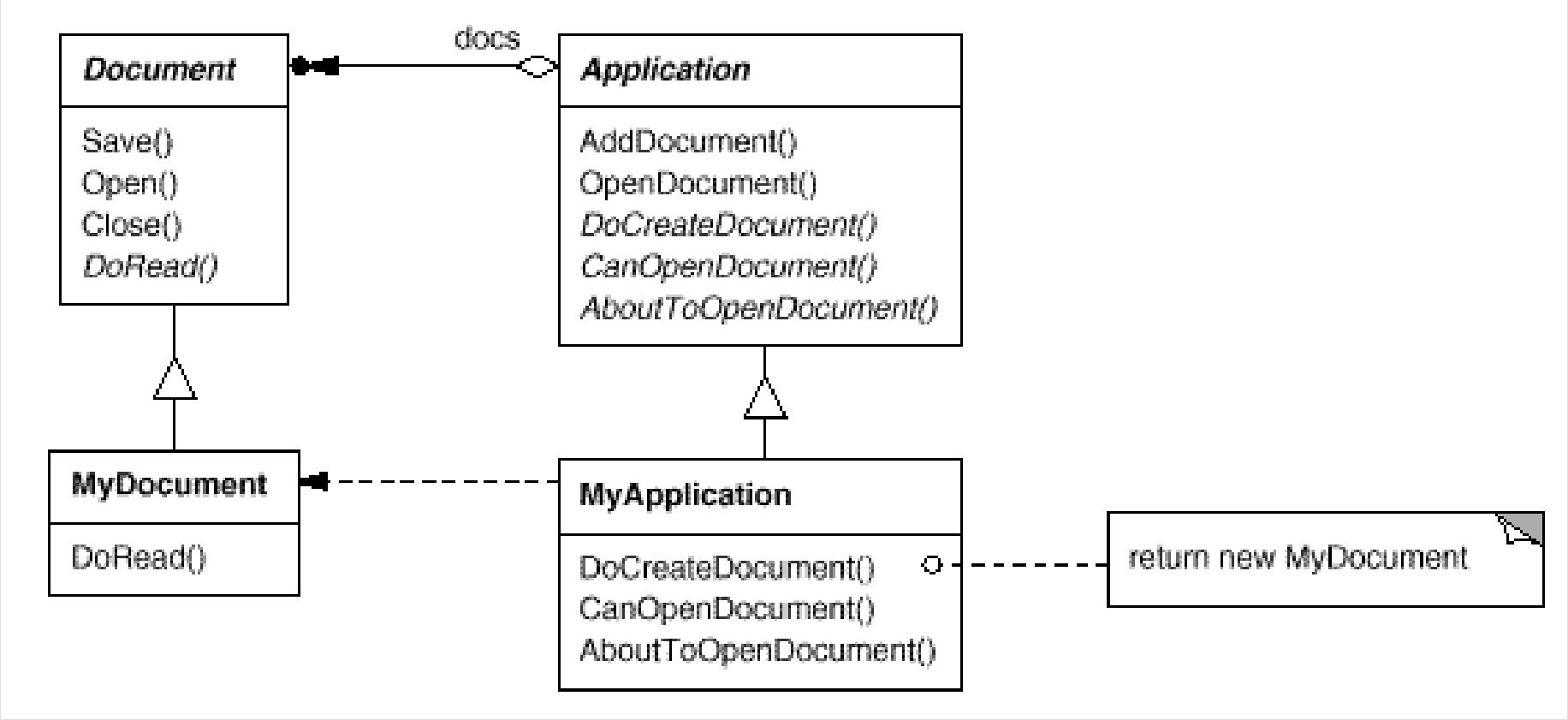
- Collaborations
  - ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm
- Consequences
  - Template methods lead to an inverted control structure that's sometimes referred to as "the Hollywood principle," that is, "Don't call us, we'll call you"
  - It's important for template methods to specify which operations are hooks (*may* be overridden) and which are abstract operations (*must* be overridden). To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding

Template Method example 1



Template Method example 2

- Application framework that provides Application and Document classes



Template Method questions

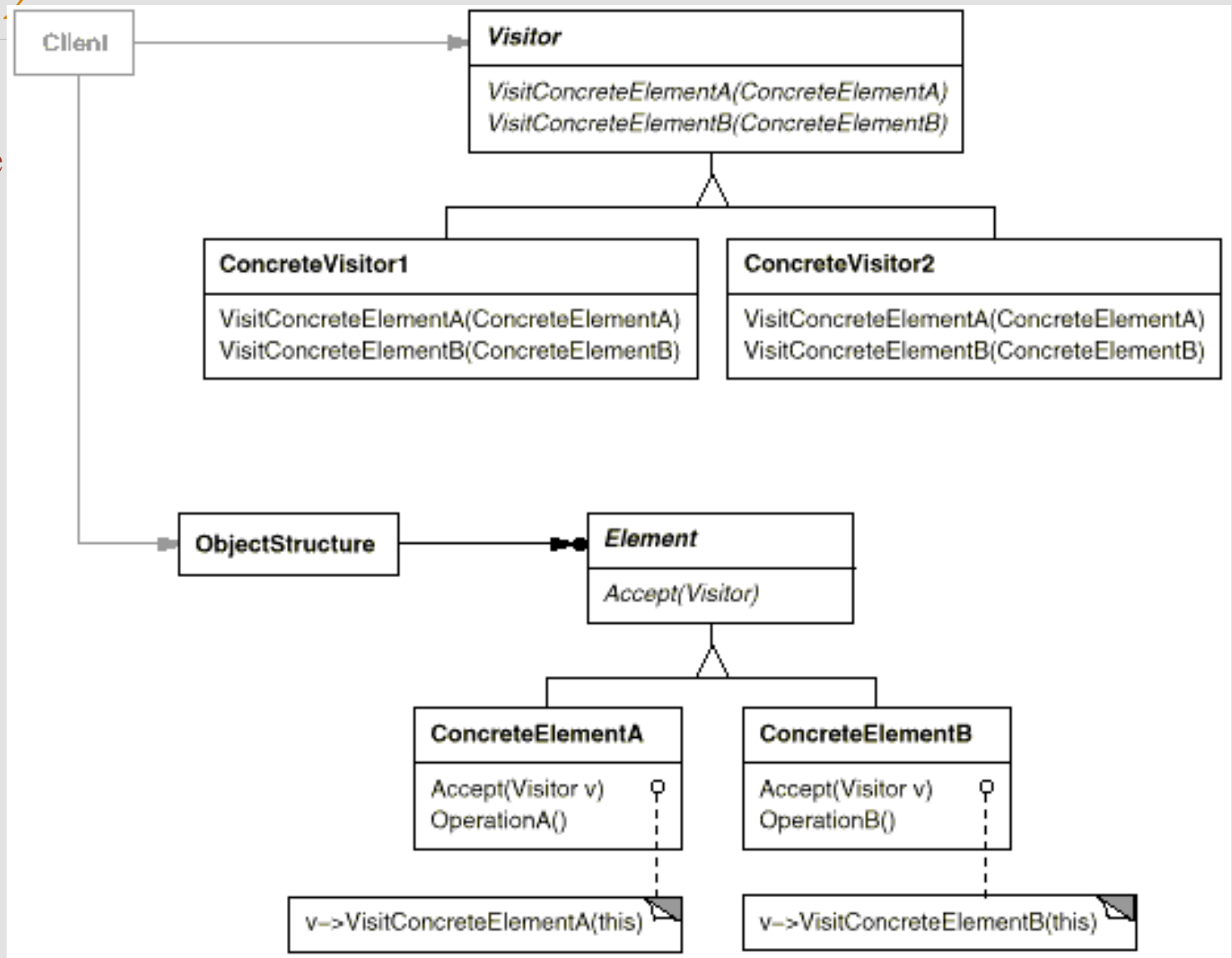
- The Template Method relies on inheritance. Would it be possible to get the same functionality of a Template Method, using object composition? What would some of the tradeoffs be?

Visitor

- Intent
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates
- Applicability
  - an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes
  - many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations
  - the classes defining the object structure rarely change, but you often want to define new operations over the structure

Visitor /2

- Structure

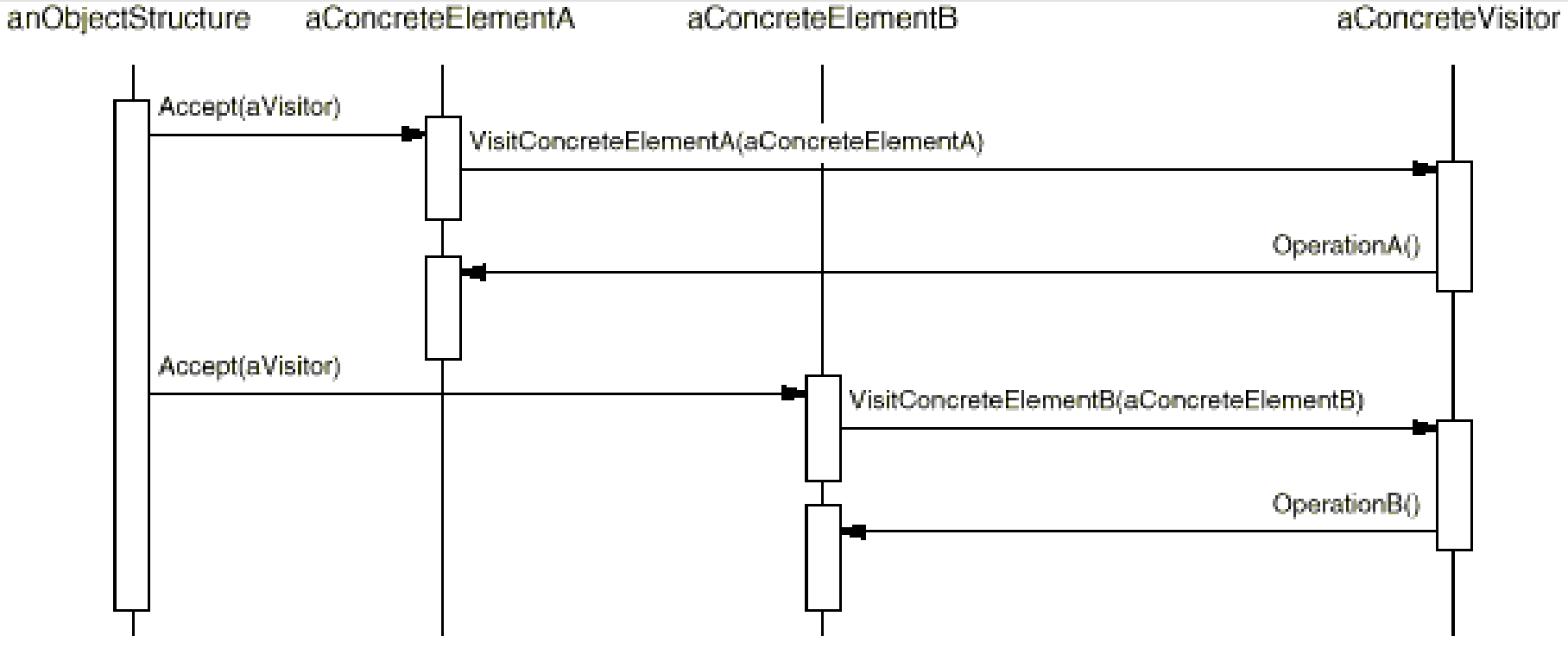


Visitor /3

- Collaborations
  - A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor
  - When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary

Visitor /4

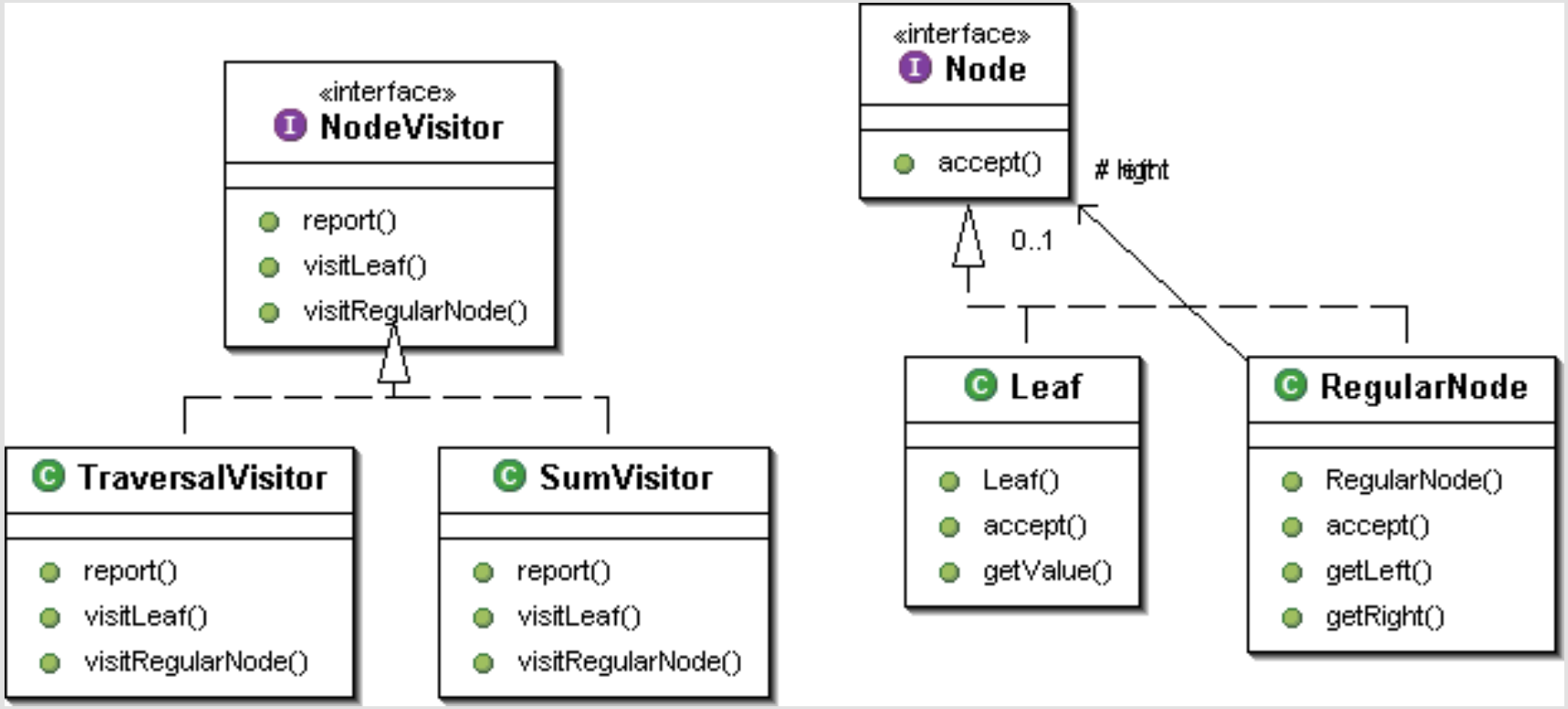
- The following interaction diagram illustrates the collaborations between an object structure, a visitor, and two elements



Visitor /4

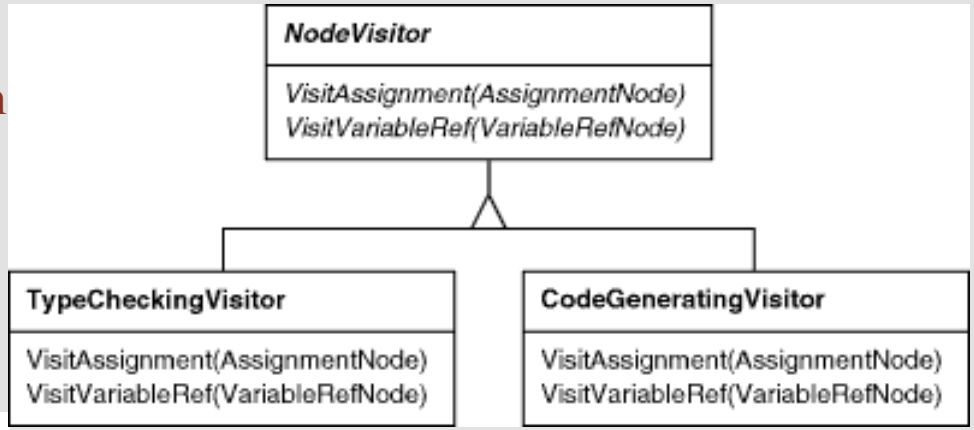
- Consequences
  - Visitor makes adding new operations easy
  - A visitor gathers related operations and separates unrelated ones
  - Adding new ConcreteElement classes is hard
  - Visiting across class hierarchies
  - Accumulating state
  - Breaking encapsulation

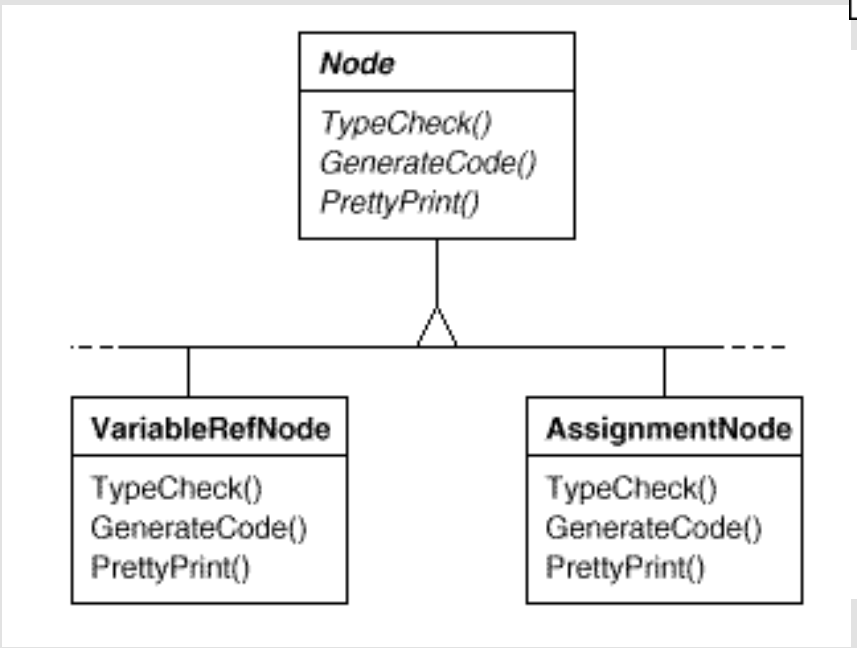
Visitor example 1

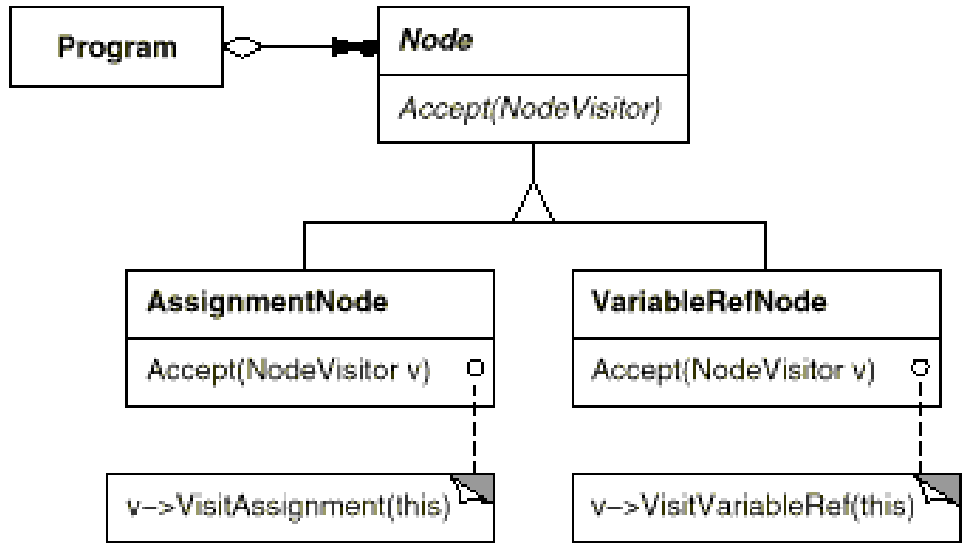


Visitor example 2

- Compiler that represents program as abstract syntax trees







Visitor Questions

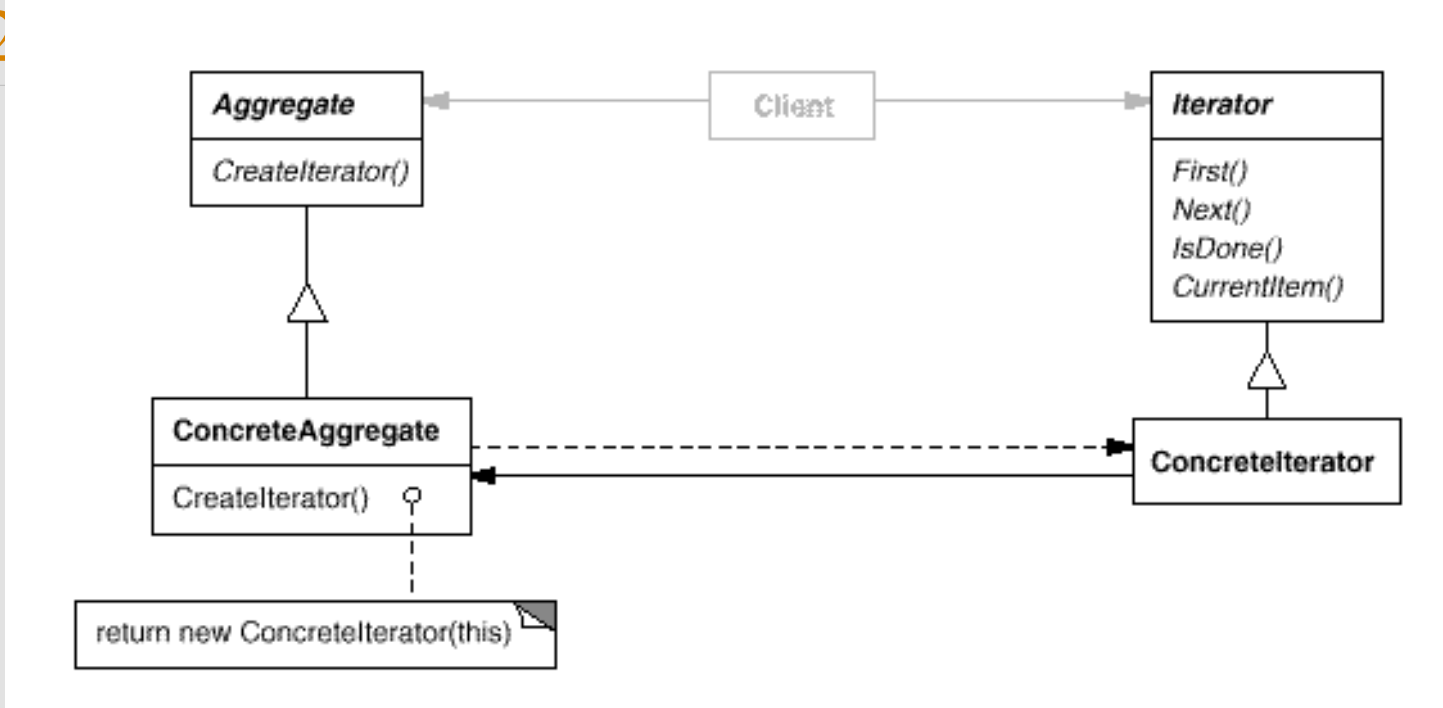
- One issue with the Visitor pattern involves cyclicality. When you add a new Visitor, you must make changes to existing code. How would you work around this possible problem?

Iterator

- Intent
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- Applicability
  - 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 /2

- Structure



Participants

Iterator

defines an interface for accessing and traversing elements.

ConcreteIterator

implements the Iterator interface.
keeps track of the current position in the traversal of the aggregate.

Aggregate

defines an interface for creating an Iterator object.

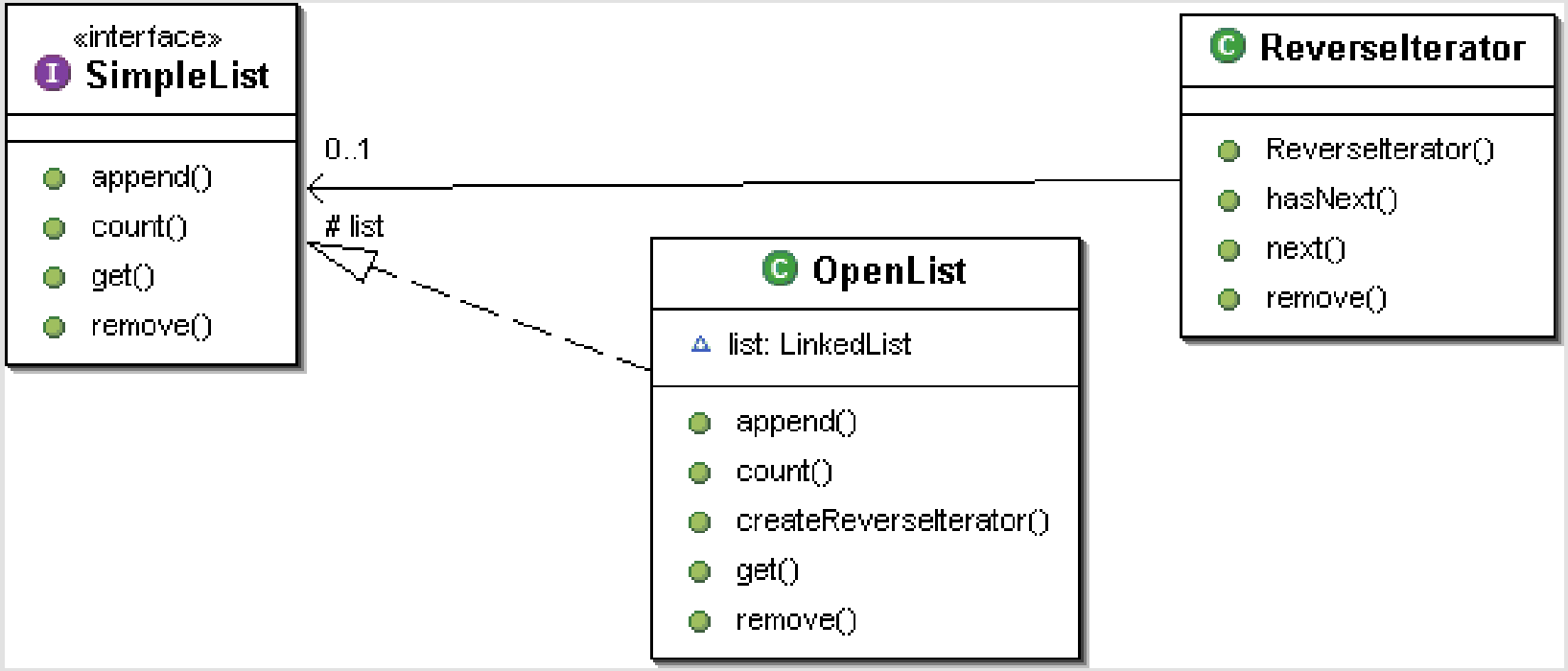
ConcreteAggregate

implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Iterator /3

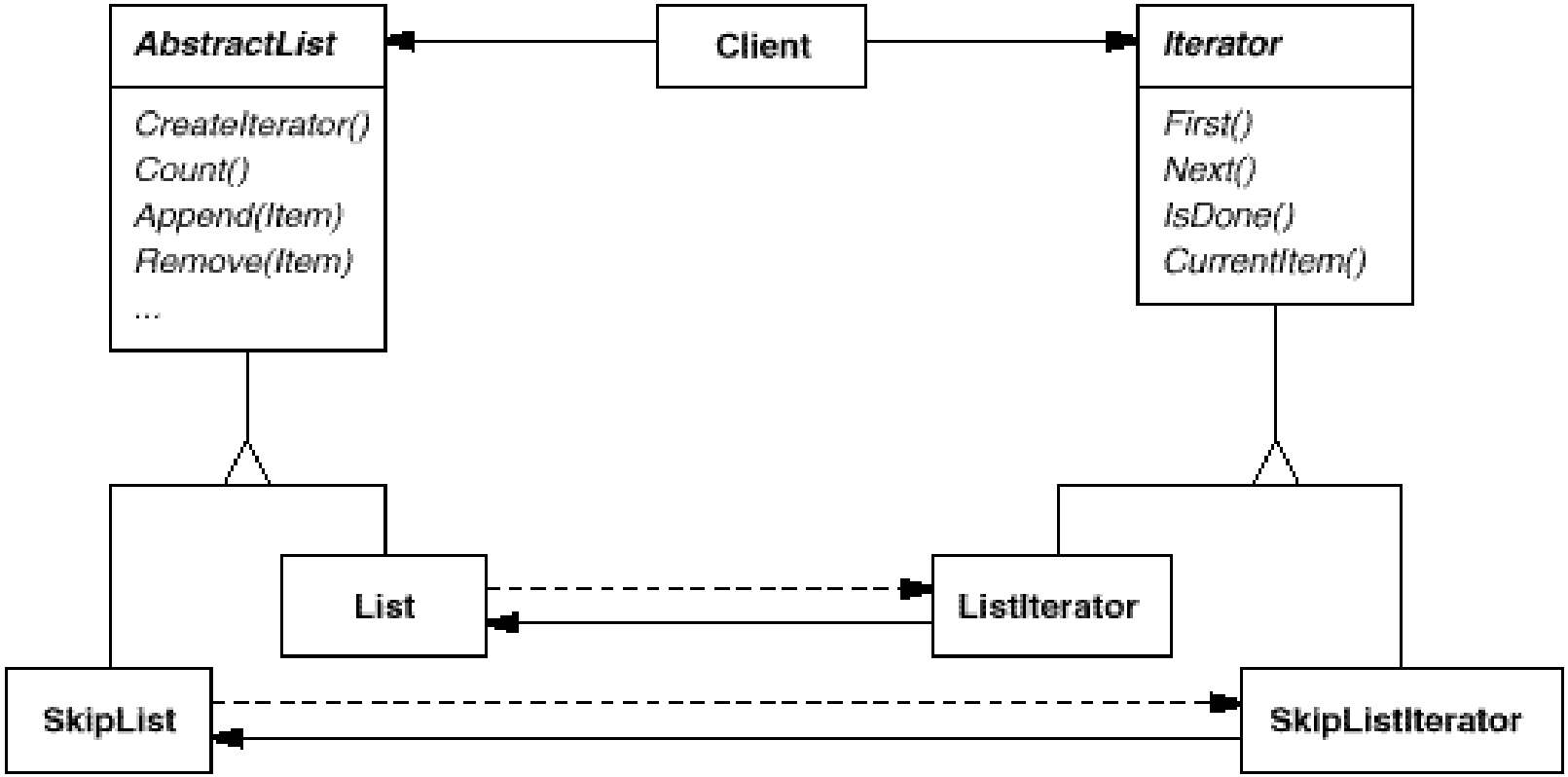
- Collaborations
  - A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal
- Consequences
  - It supports variations in the traversal of an aggregate
  - Iterators simplify the Aggregate interface
  - More than one traversal can be pending on an aggregate

Iterator example 1



Iterator example 2

- An aggregate object such as a list should give you a way to access its elements



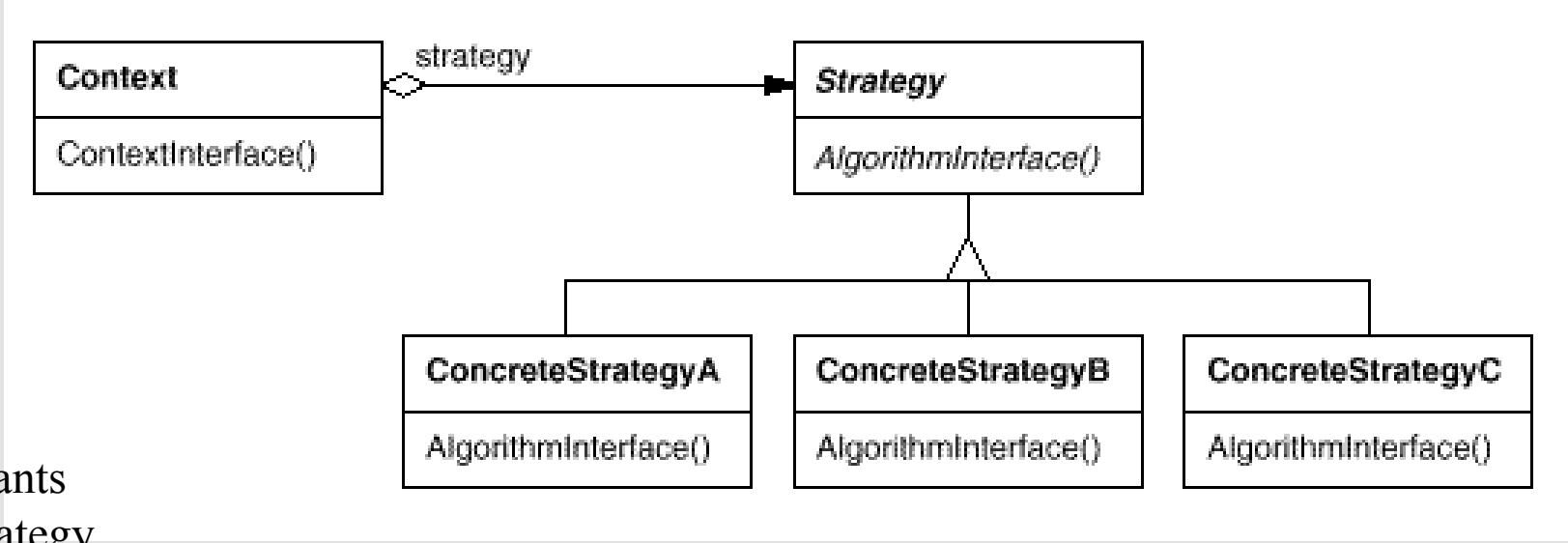
Iterator questions

- Consider a composite that contains loan objects. The loan object interface contains a method called "AmountOfLoan()", which returns the current market value of a loan. Given a requirement to extract all loans above, below or in between a certain amount, would you write or use an Iterator to do this?

Strategy

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Applicability
  - Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - You need different variants of an algorithm.
  - An algorithm uses data that clients shouldn't know about.
  - Instead of many conditionals.

Strategy /2

- 

Participants

Strategy

declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy

implements the algorithm using the Strategy interface

Context

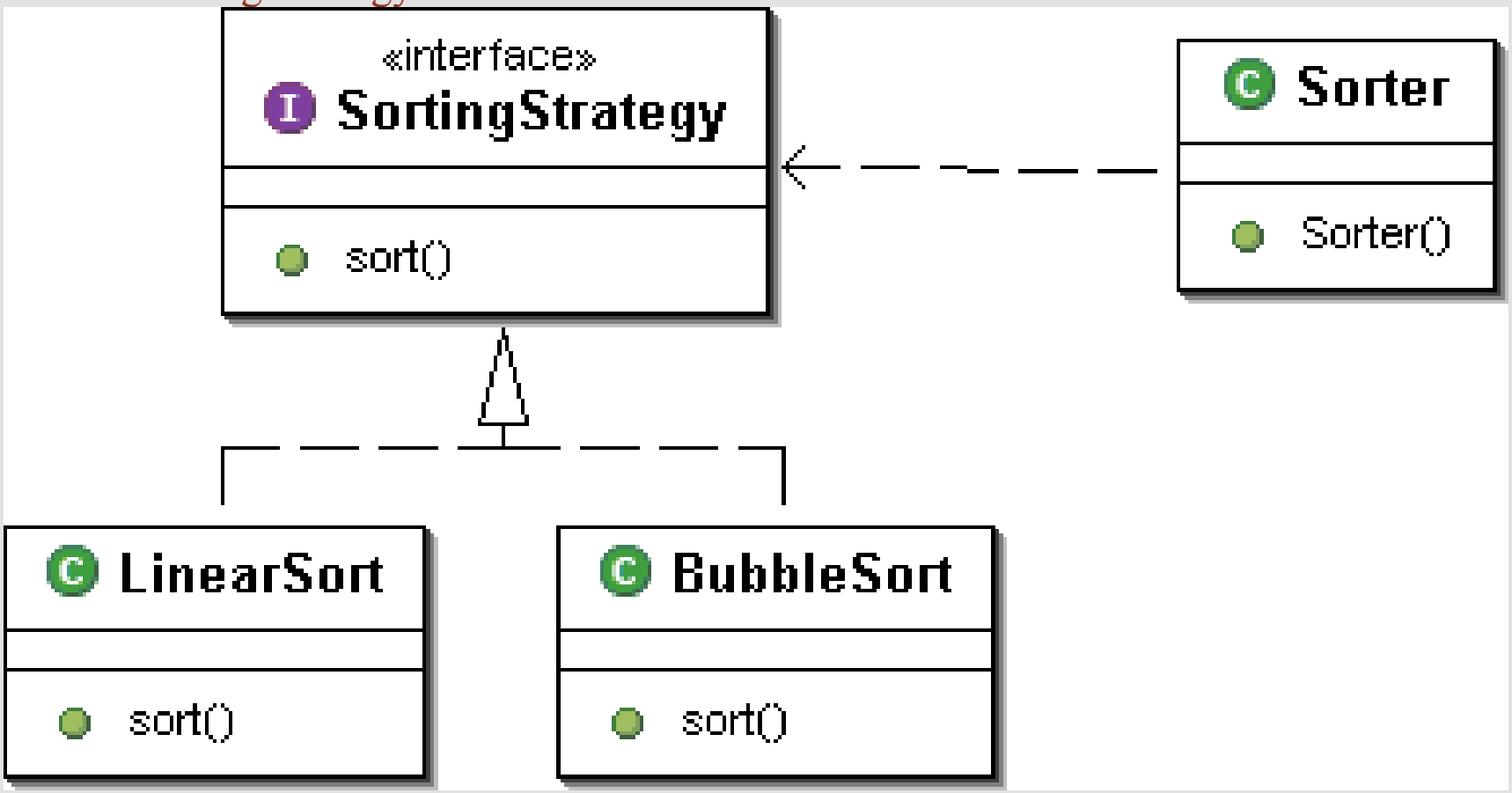
is configured with a ConcreteStrategy object.
maintains a reference to a Strategy object.
may define an interface that lets Strategy access its data

Strategy /3

- Collaborations
  - Strategy and Context interact to implement the chosen algorithm. A context may pass data or itself to the Strategy.
  - A context forwards requests from its clients to its strategy.
  - Clients usually create and pass a ConcreteStrategy to the context; thereafter, they interact with the context exclusively.
- Consequences
  - Families of related algorithms.
  - An alternative to subclassing. Vary the algorithm independently of its context even *dynamically*.
  - Strategies eliminate conditional statements.
  - A choice of implementations (space/time trade-offs).
  - Clients must be aware of different Strategies.
  - Communication overhead between Strategy and Context.
  - Increased number of objects (*stateless* option).

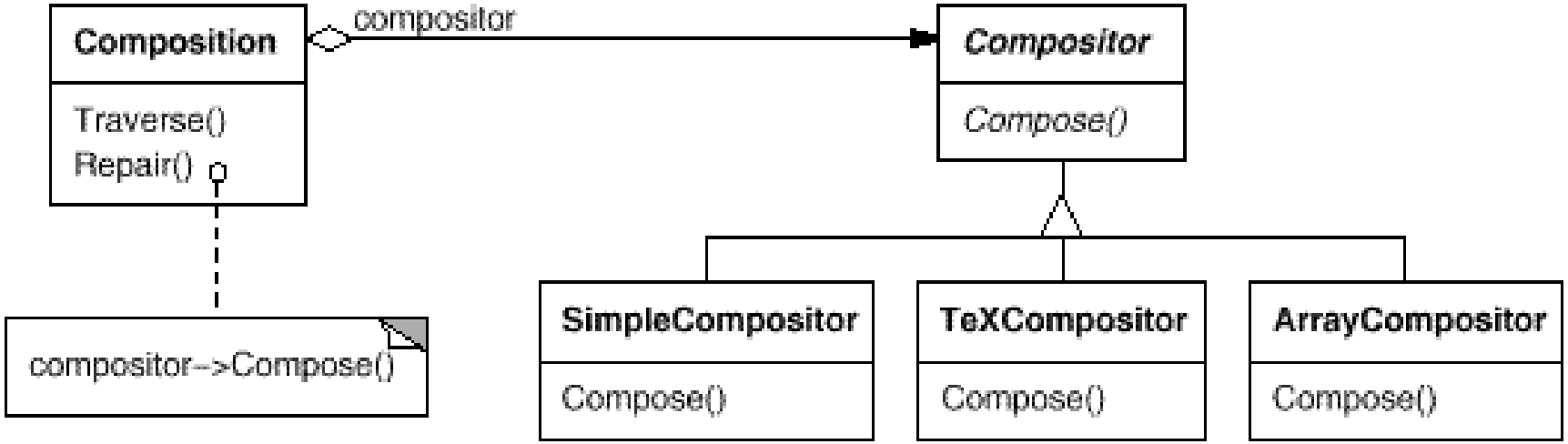
Strategy example 1

- Sorting Strategy



Strategy example 2

- To define different algorithms



Strategy questions

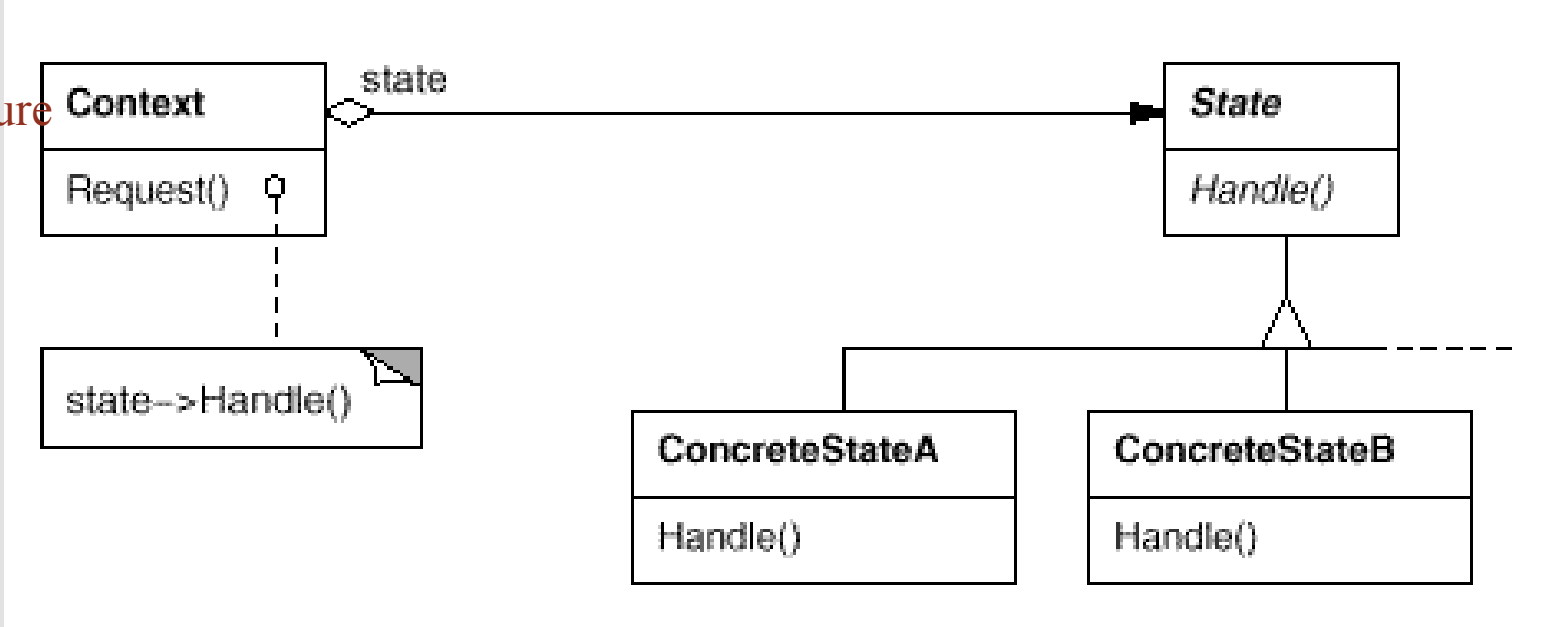
- What happens when a system has an explosion of Strategy objects? Is there some way to better manage these strategies?
- Is it possible that the data required by the strategy will not be available from the context's interface? How could you remedy this potential problem?

State

- Intent
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Applicability
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
  - Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure.

State /2

- Structure



Participants

Context

defines the interface of interest to clients.

maintains an instance of a ConcreteState subclass that defines the current state.

State

defines an interface for encapsulating the behavior associated with a particular state of the Context.

ConcreteState subclasses

each subclass implements a behavior associated with a state of the Context.

State /3

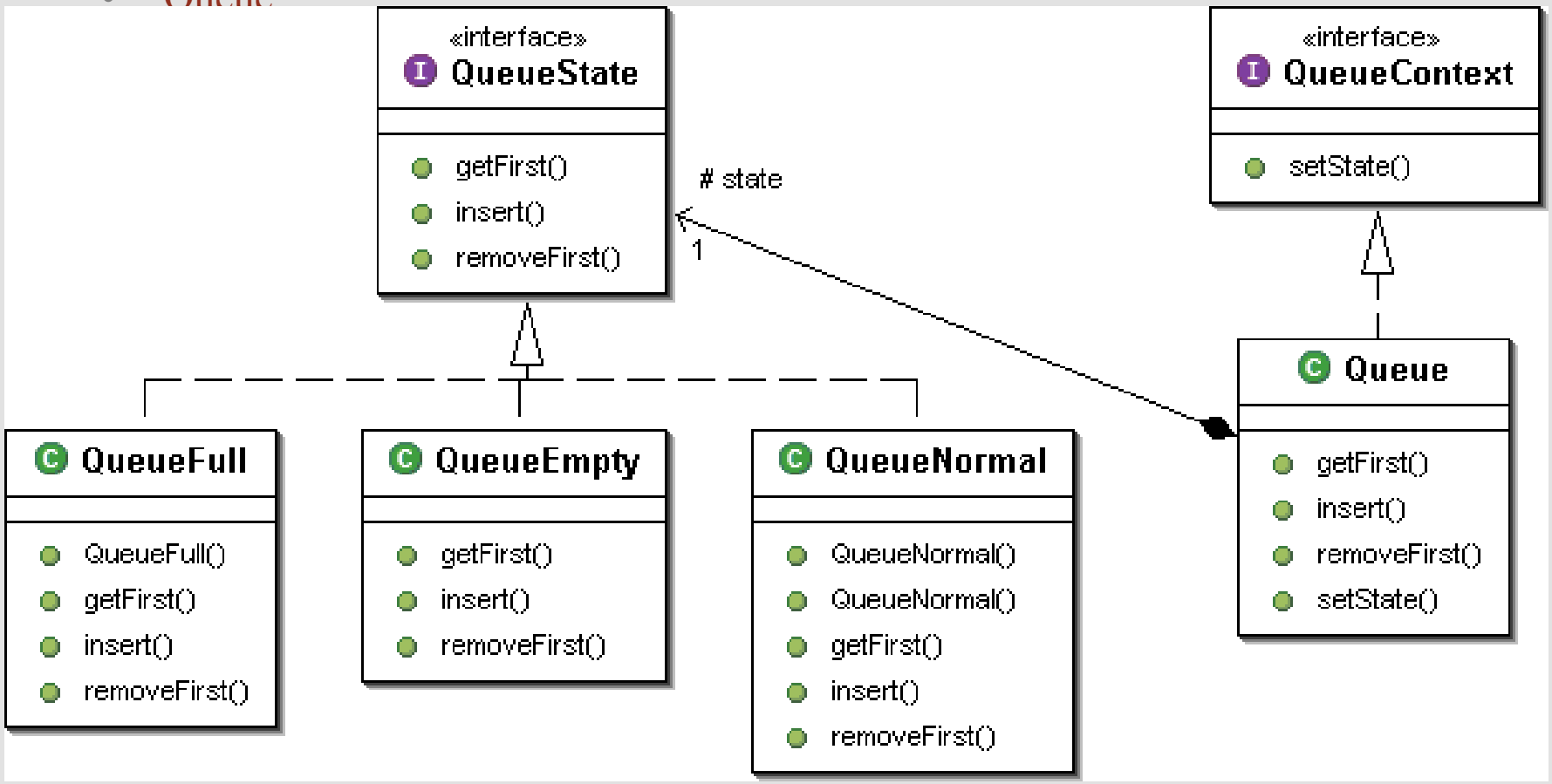
- Collaborations
  - Context delegates state-specific requests to the current ConcreteState object.
  - A context may pass itself as an argument to the State object handling the request.
  - Context is the primary interface for clients.
  - Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

State /4

- Consequences
  - It localizes state-specific behavior and partitions behavior for different states.
  - It makes state transitions explicit.
  - State objects can be shared.
- Implementation
  - Who defines the state transitions?
  - Creating and destroying State objects
  - A table-based alternative

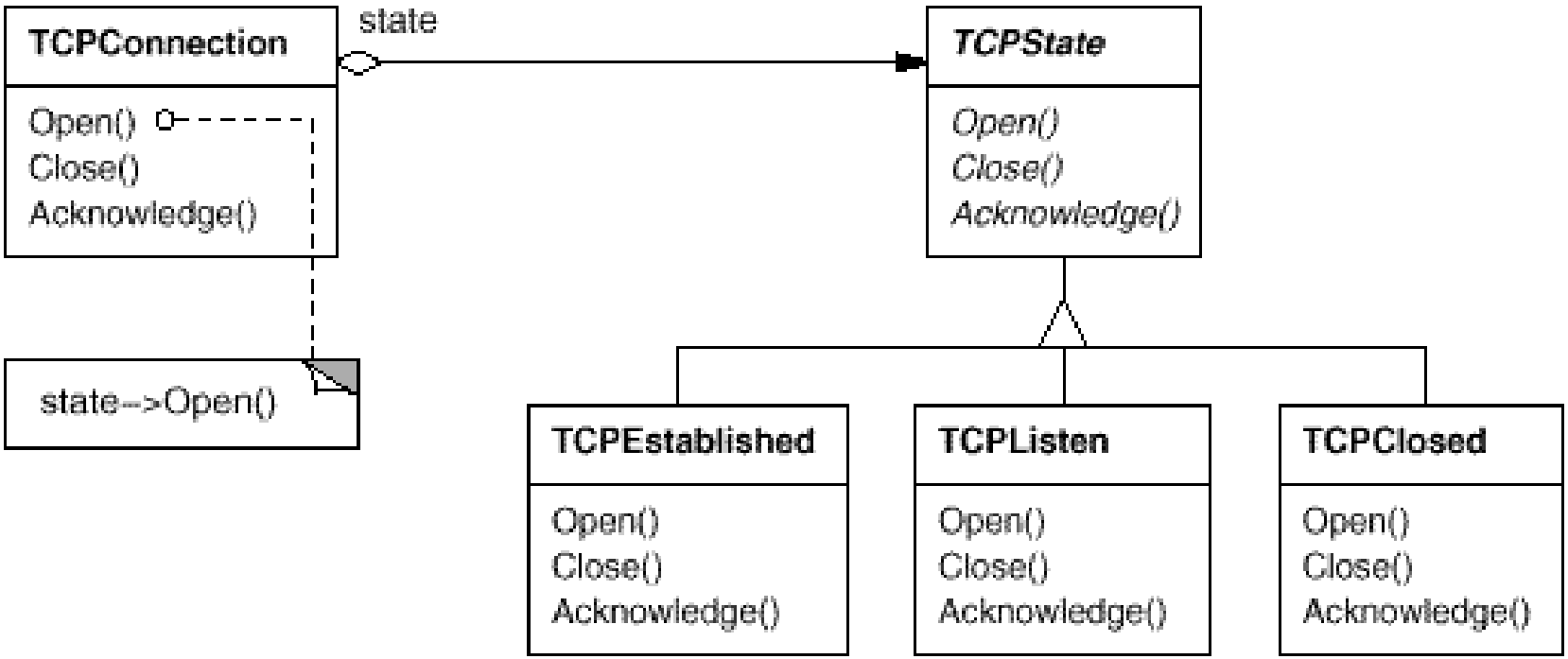
State example 1

- Queue



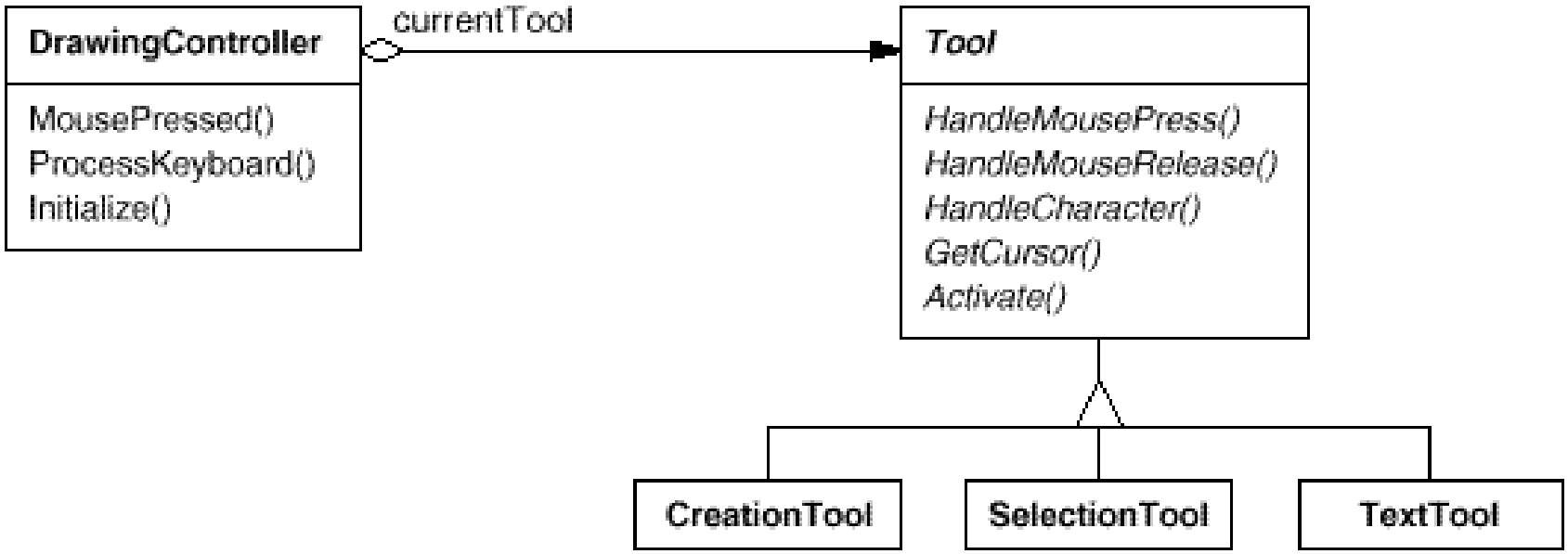
State example 2

- Network connection



State example 3

- Drawing tool



State Questions

- If something has only two to three states, is it overkill to use the State pattern?

Observer

Intent

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

Applicability

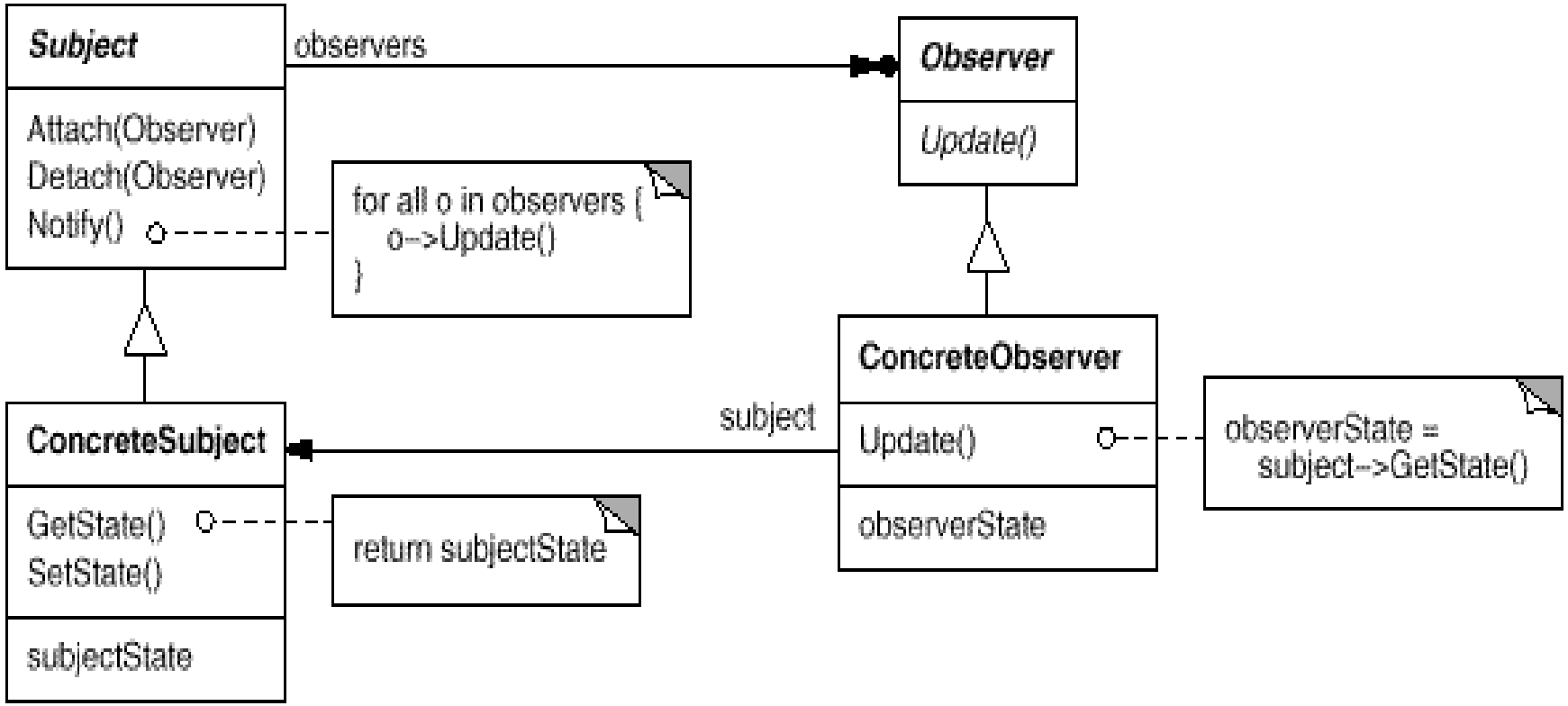
When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.

When a change to one object requires changing others, and you don't know how many objects need to be changed.

When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Observer/2

- Structure



Observer/3

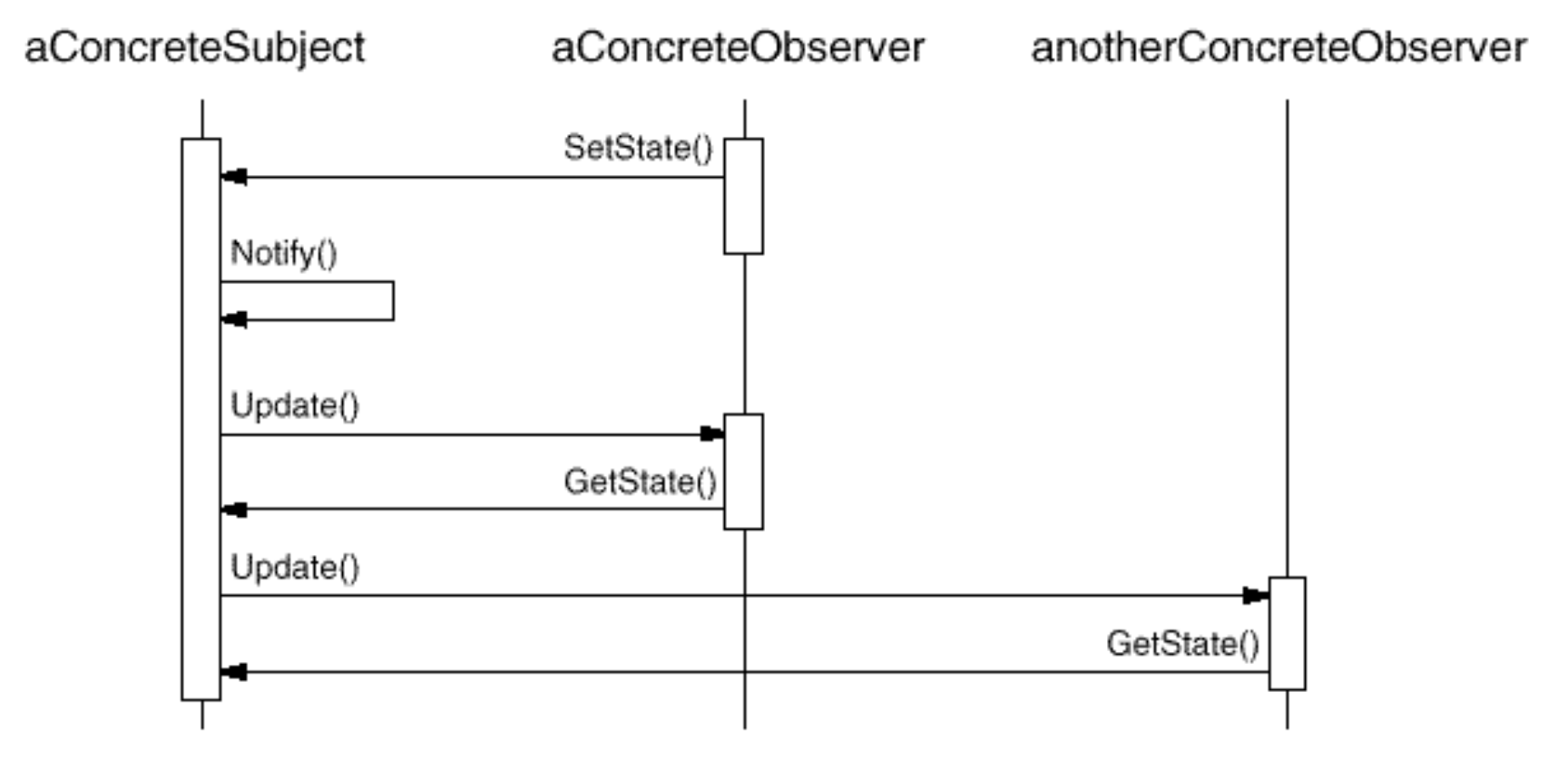
- Participants
  - **Subject**: knows its observers. Any number of Observer objects may observe a subject.◦provides an interface for attaching and detaching Observer objects.
  - **Observer**: defines an updating interface for objects that should be notified of changes in a subject.
  - **ConcreteSubject:** stores state of interest to ConcreteObserver objects and sends a notification to its observers when its state changes.
  - **ConcreteObserver:** maintains a reference to a ConcreteSubject object, stores state that should stay consistent with the subject's and implements the Observer updating interface to keep its state consistent with the subject's.

Observer/4

- Collaborations
  - ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
  - After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

Observer/5

- The following interaction diagram illustrates the collaborations between a subject and two observers:



Observer/6

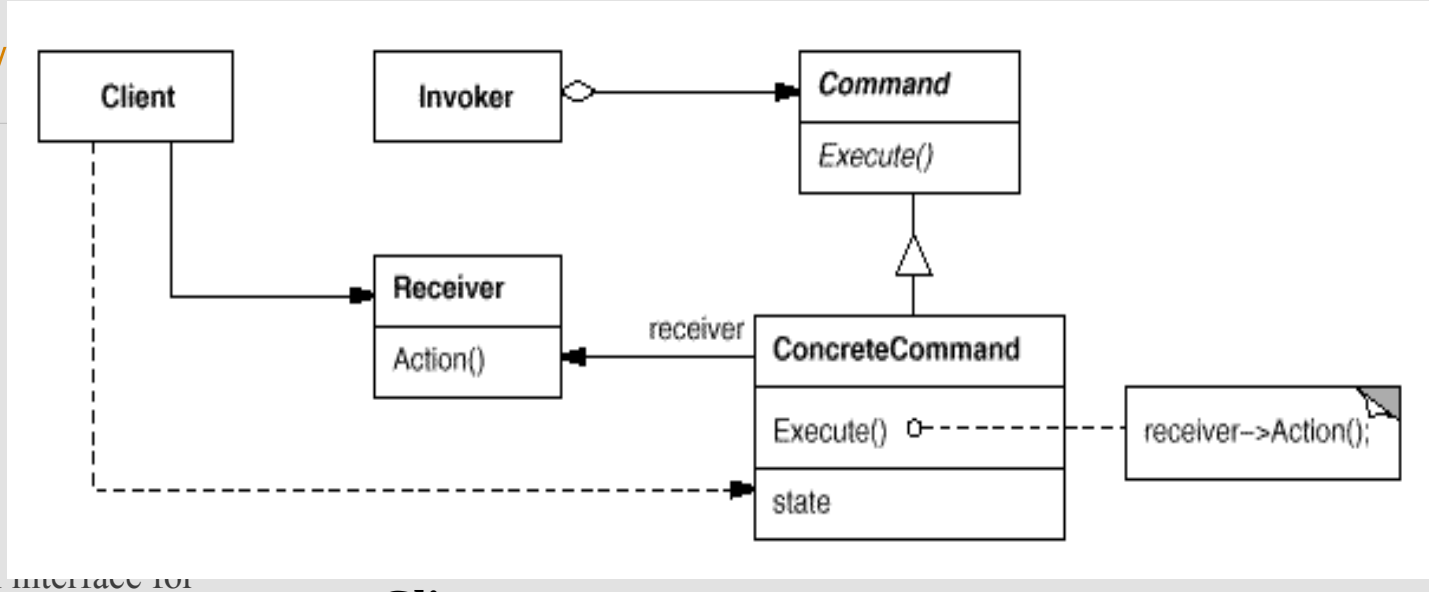
- Consequences
  - The Observer pattern lets you vary subjects and observers independently.
  - You can reuse subjects without reusing their observers, and vice versa.
  - It lets you add observers without modifying the subject or other observers.
  - Abstract coupling between Subject and Observer.
  - Support for broadcast communication.
  - Unexpected updates.

Command

- Intent
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Applicability
  - parameterize objects by an action to perform. Commands are an object-oriented replacement for callbacks.
  - specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request.
  - support undo. The Command's Execute operation can store state for reversing its effects in the command itself.

Command/

- Structure



- Participants
  - **Command:**
    - declares an interface for executing an operation
  - **ConcreteCommand:**
    - defines a binding between a Receiver object and an action
    - Implements Execute by invoking the corresponding operation on Receiver

Client:

Creates a ConcreteCommand object and sets its receiver

Invoker:

Asks the command to carry out the request

Receiver:

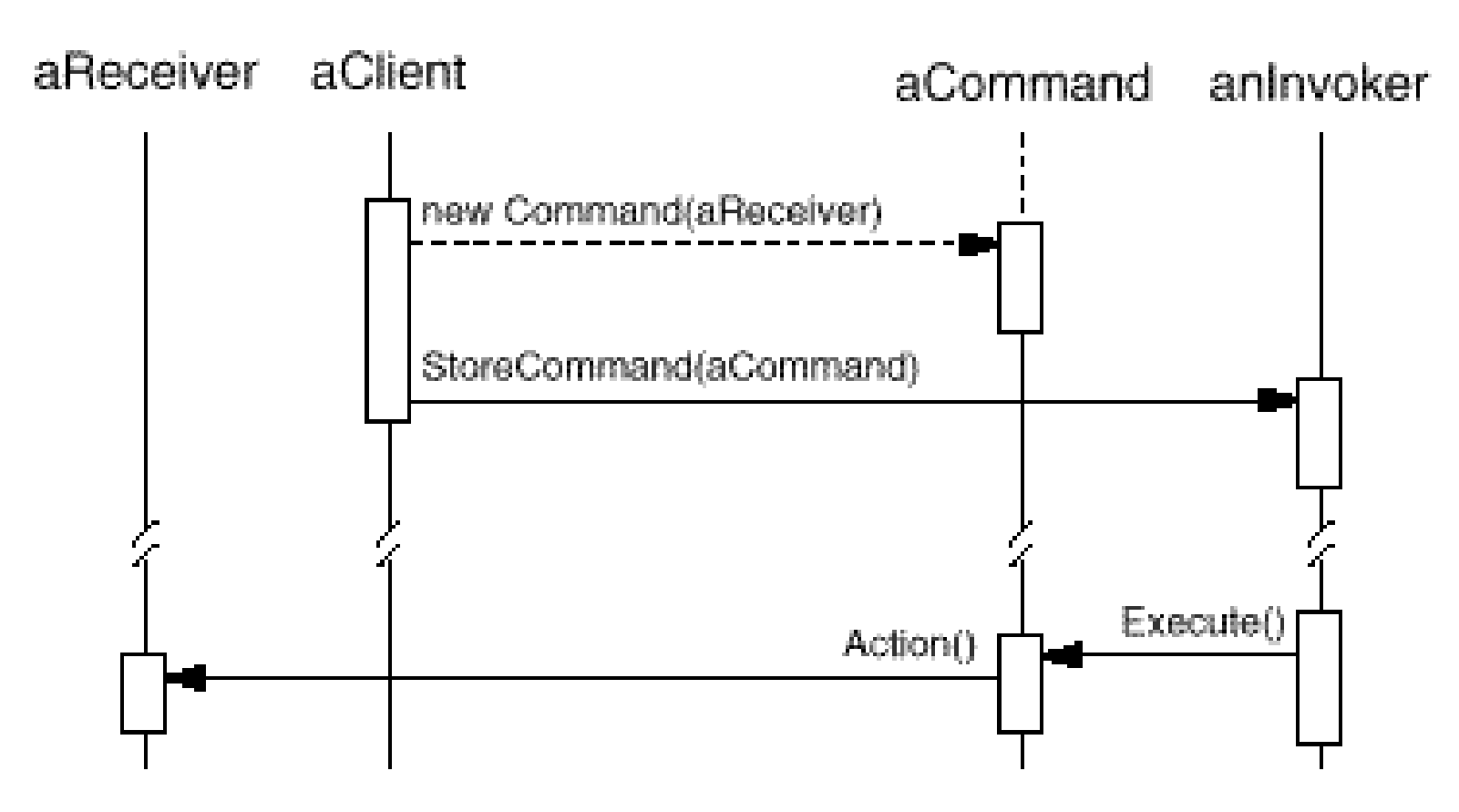
Knows how to perform the operations associated with carrying out a request. Any class may serve as a receiver

Command/3

- Collaborations
  - The client creates a ConcreteCommand object and specifies its receiver.
  - An Invoker object stores the ConcreteCommand object.
  - The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
  - The ConcreteCommand object invokes operations on its receiver to carry out the request.

Command/4

- Collaborations (continue)
- The following diagram shows the interactions between these objects



Command/5

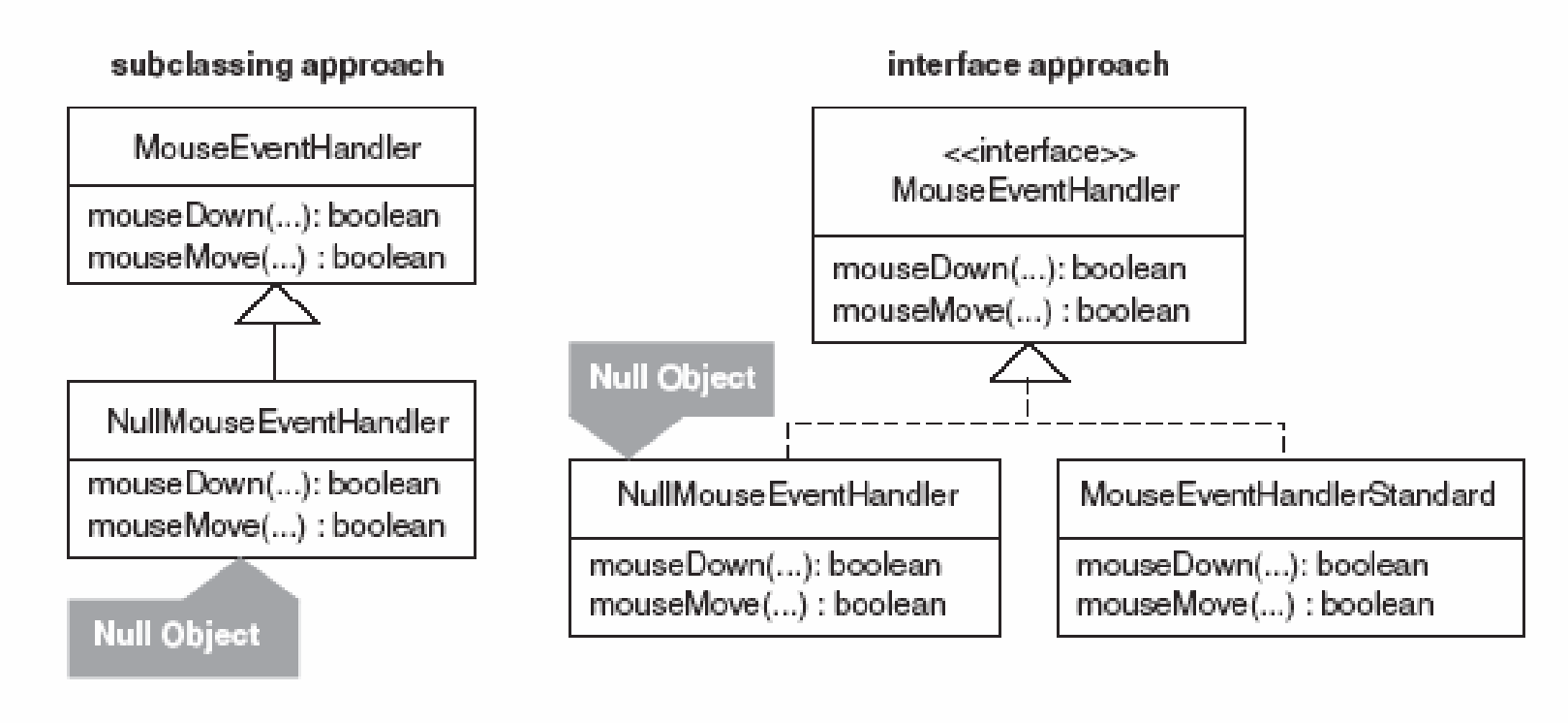
- Consequences
  - Command decouples the object that invokes the operation from the one that knows how to perform it.
  - Commands are first-class objects. They can be manipulated and extended like any other object.
  - You can assemble commands into a composite command. In general, composite commands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

Null Object (1)

- Intent
  - It removes the need to check whether a field or variable is null by making it possible to always call the field or variable safely.
- Motivation
  - If a client calls a method on a field or variable that is null, an exception may be raised, a system may crash, or similar problems may occur.
  - To protect our systems from such unwanted behavior, we write checks to prevent null fields or variables from being called.
  - Repeating this “null logic” in multiple places bloats the system with unnecessary code.
  - Compared with code that is free of null logic, code that is full of it takes longer to comprehend and requires more thinking about how to extend.
  - If new code is written and programmers forget to include null logic for it, null errors can begin to occur.

Null Object (2)

- Structure
  - Replace the null logic with a Null Object that provides the appropriate null behavior.



Null Object (3)

- Consequences
  - Prevents null error without duplicating null logic.
  - Simplifies code by minimizing null tests.
  - Complicates a design when a system needs few null tests.
  - Complicates maintainance. Null objects that have a superclass nust override all newly inherited public methods.