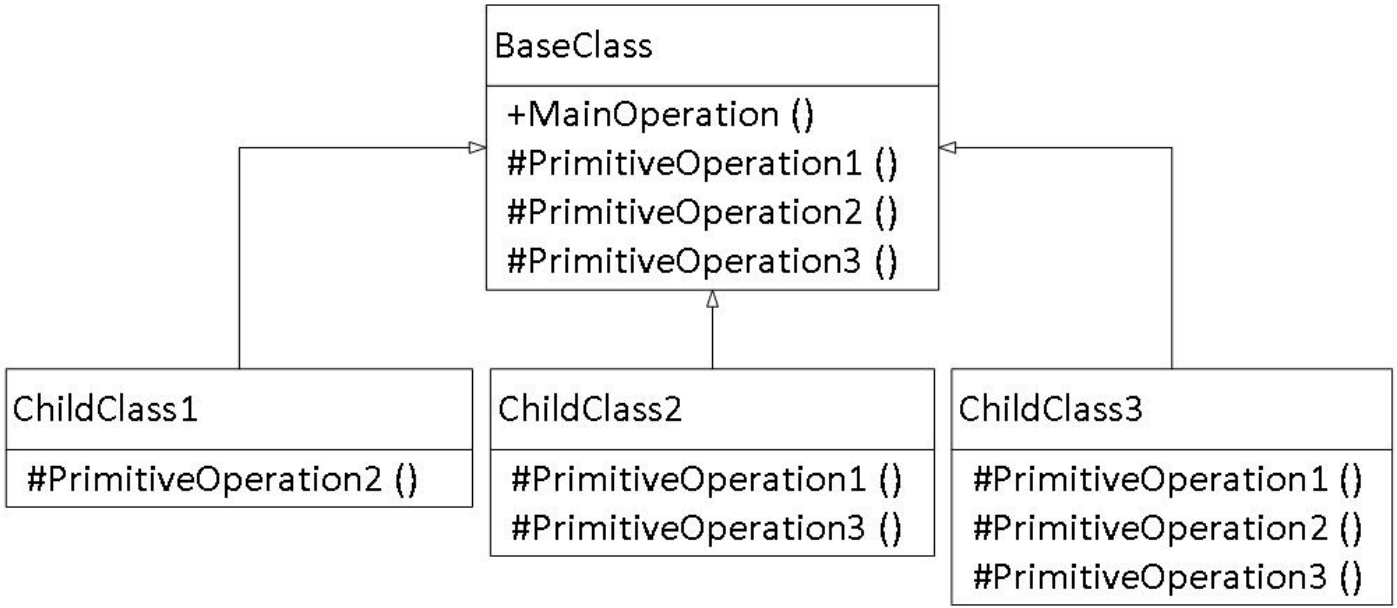
Design Patterns (part 2 – bihevioral patterns)

Template Method Pattern

Template Method Pattern

- Behavioral pattern
- Intent
  - ✓ Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
  - ✓ Base class declares algorithm 'placeholders', and derived classes implement the placeholders

Template Method Pattern Structure



When to Use Template Method Pattern

- When you want to implement invariant parts of an algorithm only once, and want to leave it to subclasses to implement the rest of the behavior
- When you want to control which part of an algorithm subclasses can vary
- When you have a set of algorithms that don’t vary much

Template Method Pattern Consequences

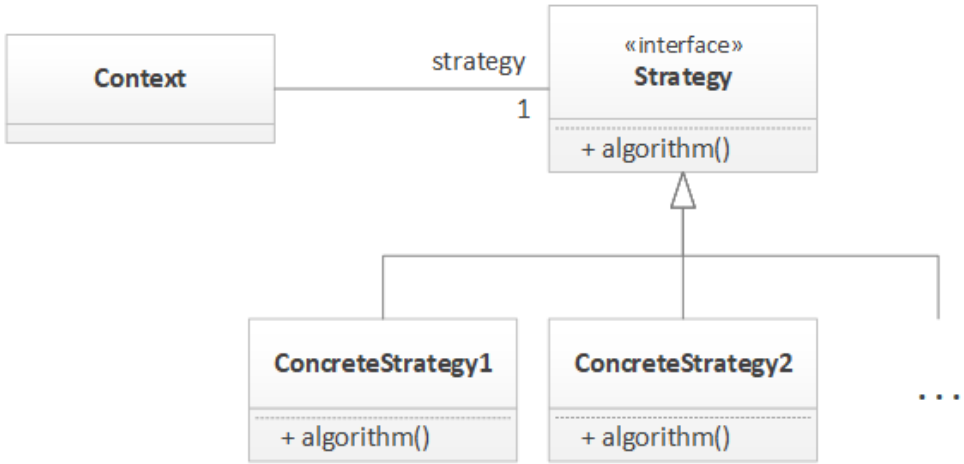
- Template methods are fundamental technique for code reuse
- Template methods cannot be changed: the order of methods they call is fixed

Strategy Pattern

Strategy Pattern

- Behavioral pattern
- Intent
  - ✓ Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
  - ✓ Capture the abstraction in an interface, bury implementation details in derived classes.
- Very useful for maintaining adherence to Open/Closed Principle
- “Program to an interface, not an implementation.”

Strategy Pattern Structure



When to Use Strategy Pattern

- When many related classes differ only in their behavior
- When you need different variants of an algorithm which you want to be able to switch at runtime
- When your algorithm uses data, code or dependencies that the clients shouldn't know about
- When a class defines many different behaviors which appear as a bunch of conditional statements in its method

Strategy Pattern Consequences

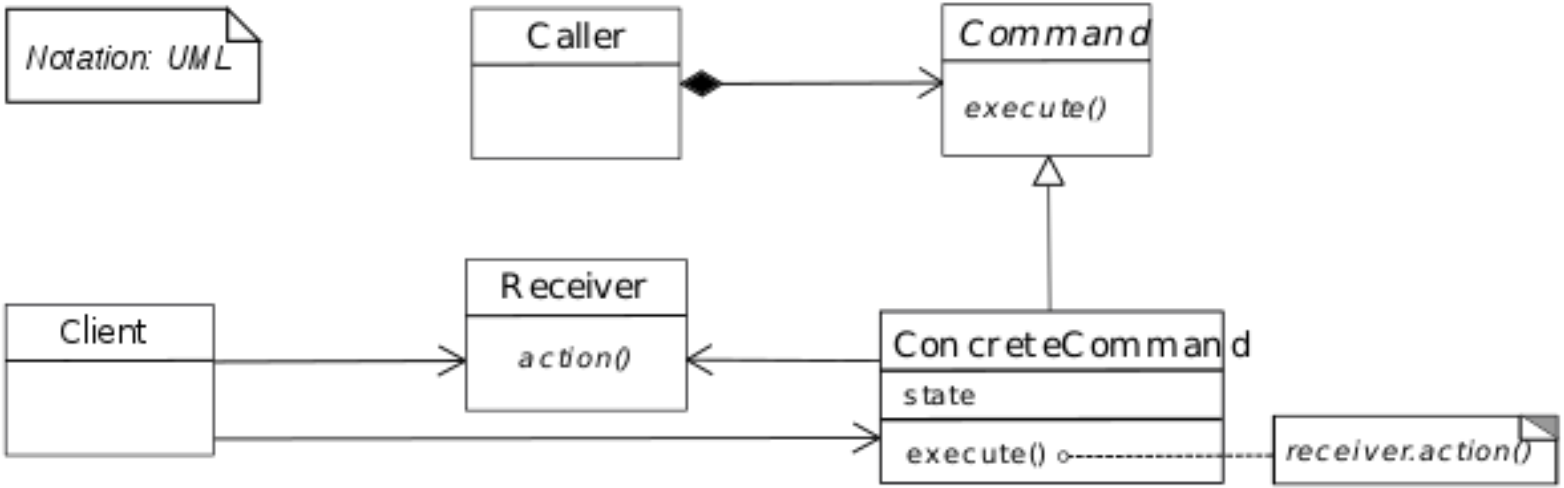
- It offers an alternative to subclassing your context
- New strategies can be introduced without having to change the context: open/closed principle
- It eliminates conditional statements
- It can provide a choice of implementations with the same behavior

Command Pattern

Command Pattern

- Behavioral pattern
- Intent
  - Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
  - Promote "invocation of a method on an object" to full object status
  - An object-oriented callback

Command Pattern Structure



When to Use Command Pattern

- When you want to parameterize objects with an action to perform
- When you want to support undo
- When you want to specify, queue and execute requests at different times
- When you need to store a list of changes to potentially reapply later on

Command Pattern Consequences

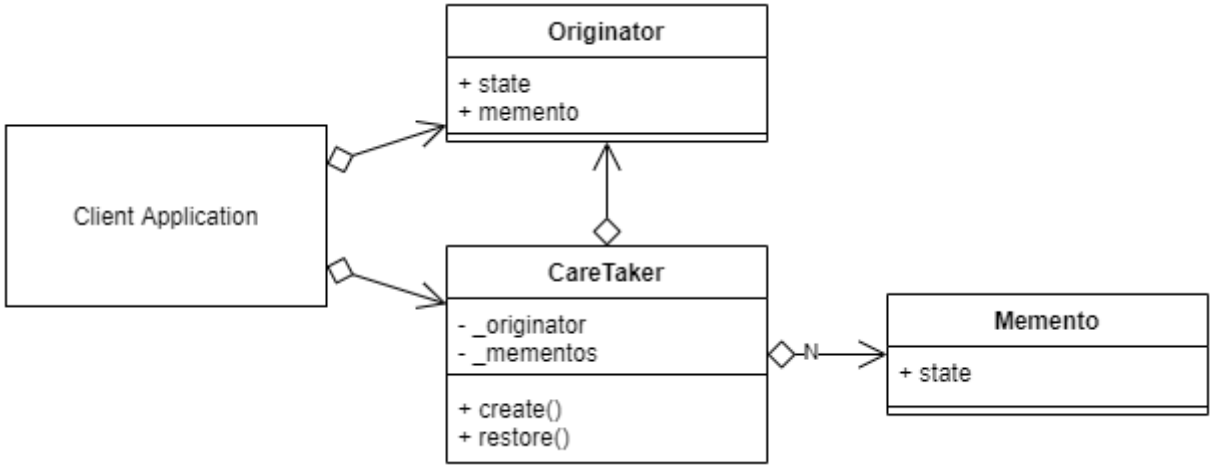
- It decouples the class that invokes the operation from the one that knows how to perform it: single responsibility principle
- Commands can be manipulated and extended
- Commands can be assembled into a composite command
- Existing implementations don’t have to be changed to add new commands: open/closed principle
- Because an additional layer is added, complexity increases

Memento Pattern

Memento Pattern

- Behavioral pattern
- Intent
  - ✓ Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
  - ✓ Promote "invocation of a method on an object" to full object status
  - ✓ An object-oriented callback

Memento Pattern Structure



When to Use Memento Pattern

- When part of an object's state must be saved so it can be restored later on
- AND when a direct interface to obtaining the state would expose implementation details and break encapsulation

Memento Pattern Consequences

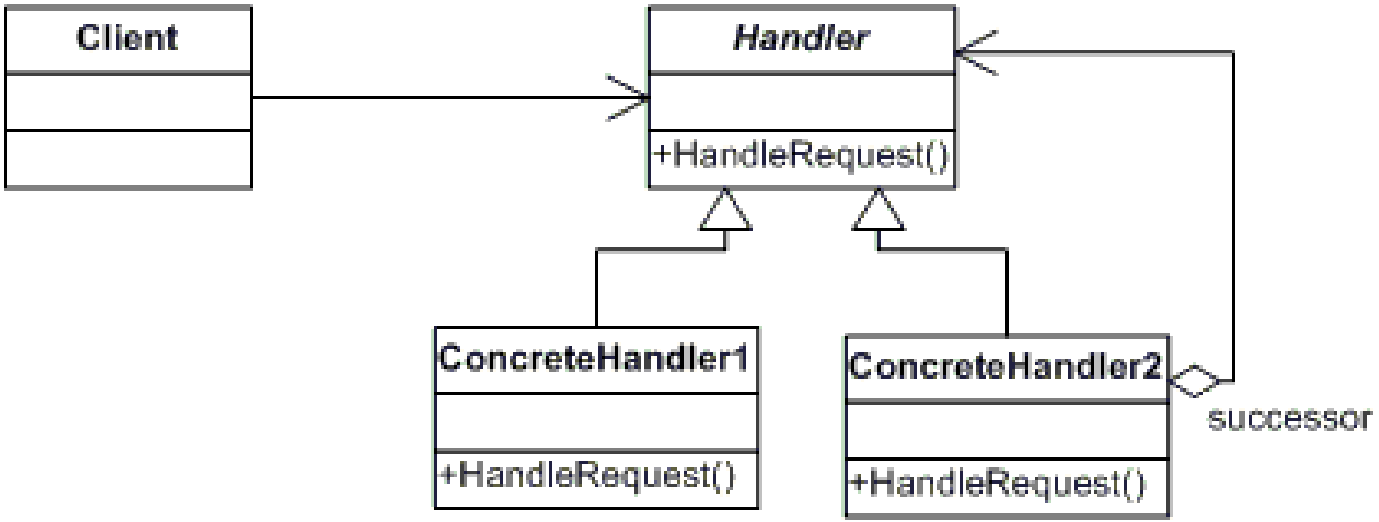
- It preserves encapsulation boundaries
- It simplifies the originator
- Using mementos might be expensive
- It can introduce complexity to your code base

Chain of Responsibility Pattern

Chain of Responsibility Pattern

- Behavioral pattern
- Intent
  - ✓ Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
  - ✓ Launch-and-leave requests with a single processing pipeline that contains many possible handlers.

Chain of Responsibility Pattern Structure



When to Use Chain of Responsibility Pattern

- When more than one object may handle a request and the handler isn't known beforehand
- When you want to issue a request to one of several objects (handlers) without specifying the receiver explicitly
- When the set of objects that handle a request should be specified dynamically

Chain of Responsibility Pattern Consequences

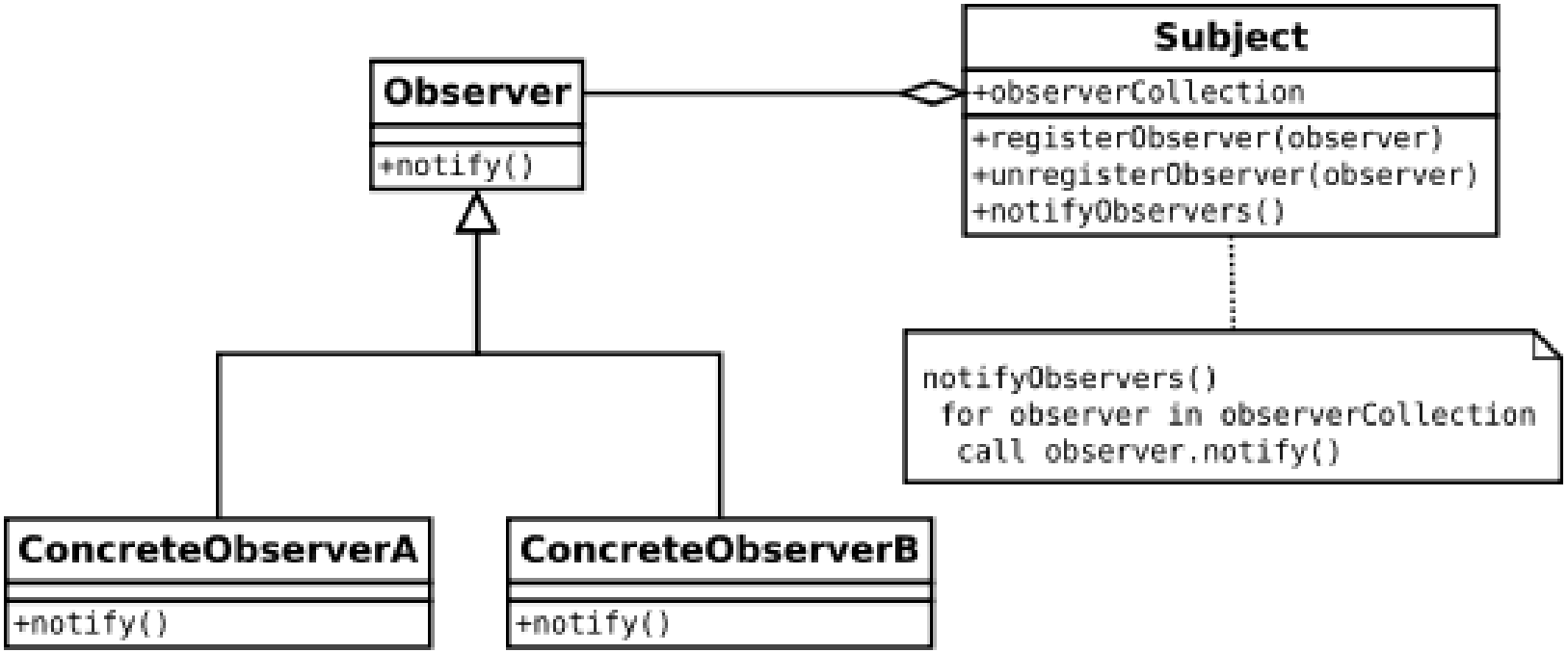
- It enables reduced coupling & works towards a single responsibility per class
- It adds flexibility in regards to assigning responsibilities to objects
- It does not guarantee receipt of the request

Observer Pattern

Observer Pattern

- Behavioral pattern
- Intent
  - ✓ Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  - ✓ Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
- Also known as Publish-Subscribe pattern

Observer Pattern Structure



When to Use Observer Pattern

- When a change to one object requires changing others, and you don’t know in advance how many objects need to be changed
- When objects that observe others are not necessarily doing that for the total amount of time the application runs
- When an object should be able to notify other objects without making assumptions about who those objects are

Observer Pattern Consequences

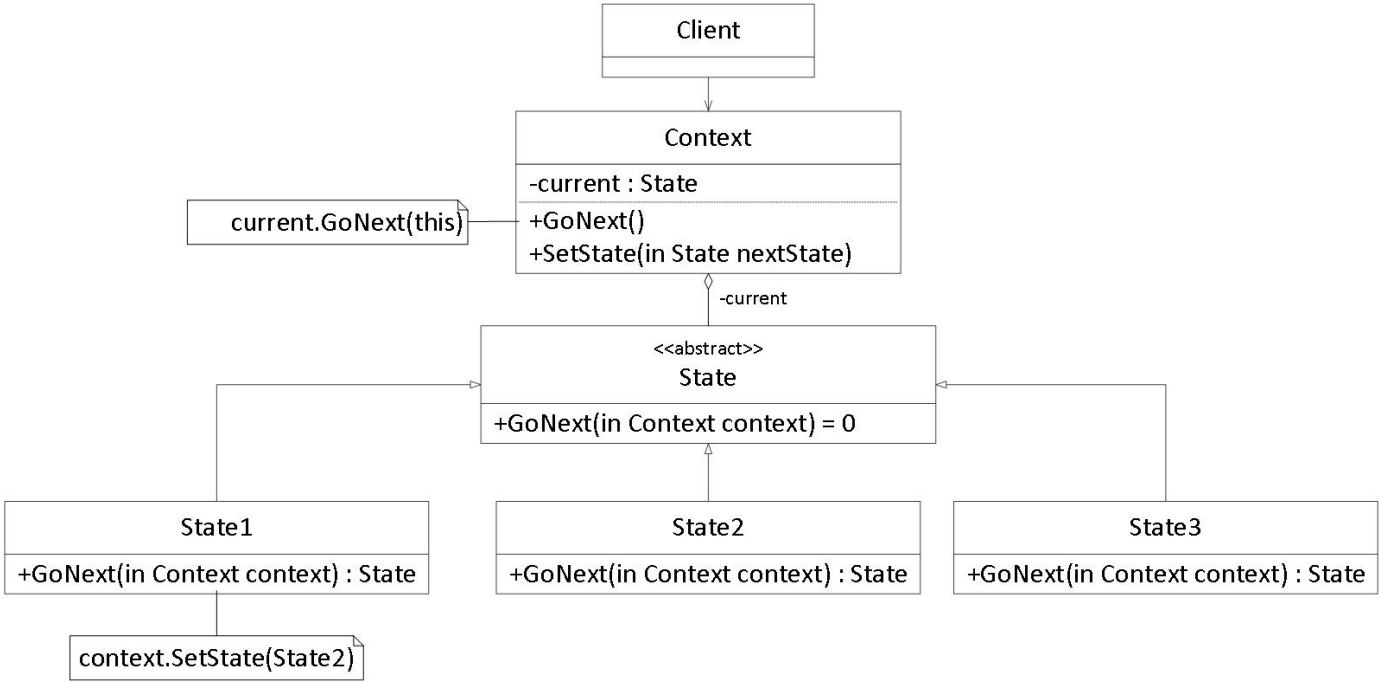
- It allows subjects and observers to vary independently: subclasses can be added and change without having to change others: open/closed principle
- Subject and observer are loosely coupled: open/closed principle
- It can lead to a cascade of unexpected updates

State Pattern

State Pattern

- Behavioral pattern
- Intent
  - ✓ Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
  - ✓ An object-oriented state machine
- State objects are often singletons
- Volatile strategy pattern

State Pattern Structure



When to Use State Pattern

- When an object's behavior depends on its state and it must change it at runtime (depending on that state)
- When your objects are dealing with large conditional statements that depend on the object's state

State Pattern Consequences

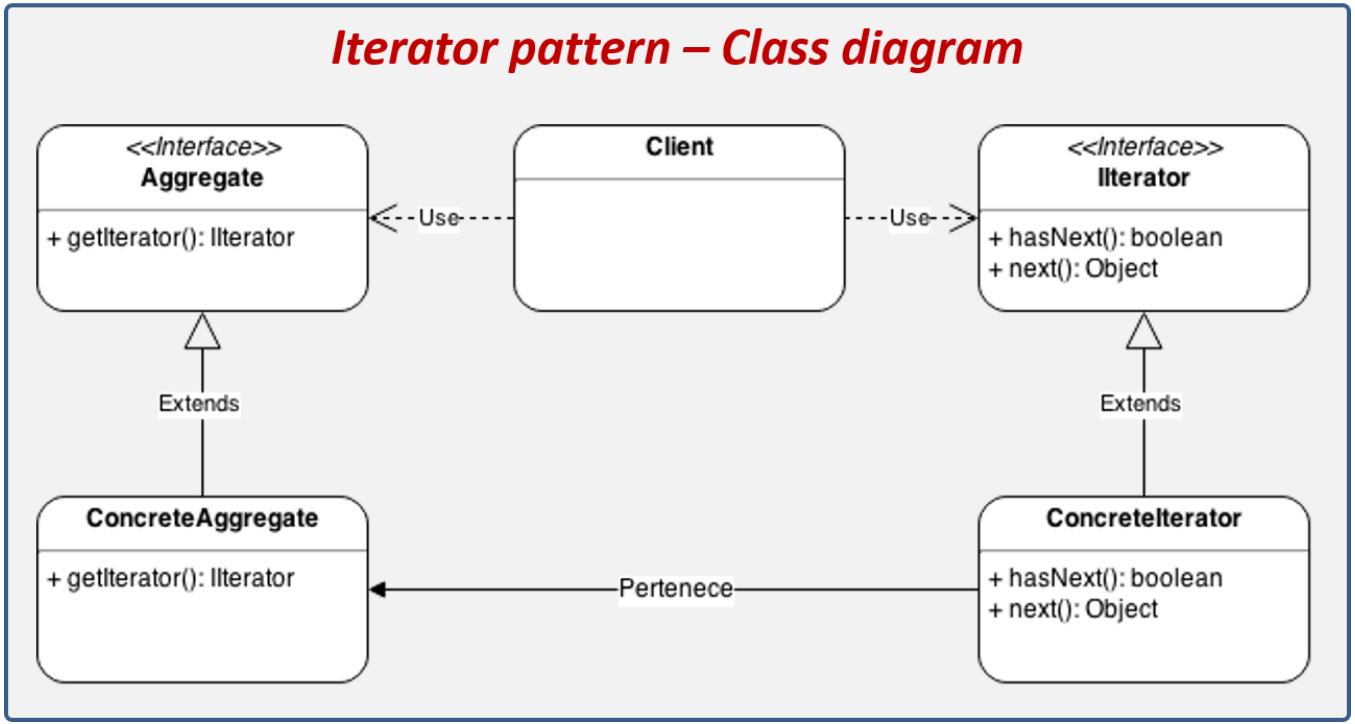
- It localizes state-specific behavior and partitions behavior for different states: single responsibility principle
- New states and transitions can easily be added by defining new subclasses: open/closed principle
- The number of classes is increased, which adds additional complexity

Iterator Pattern

Iterator Pattern

- Behavioral pattern
- Intent
  - ✓ providing a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- Aggregate objects (List, Dictionary, Stack, Queue, …) keep their items in an internal structure

Iterator Pattern Structure



When to Use Iterator Pattern

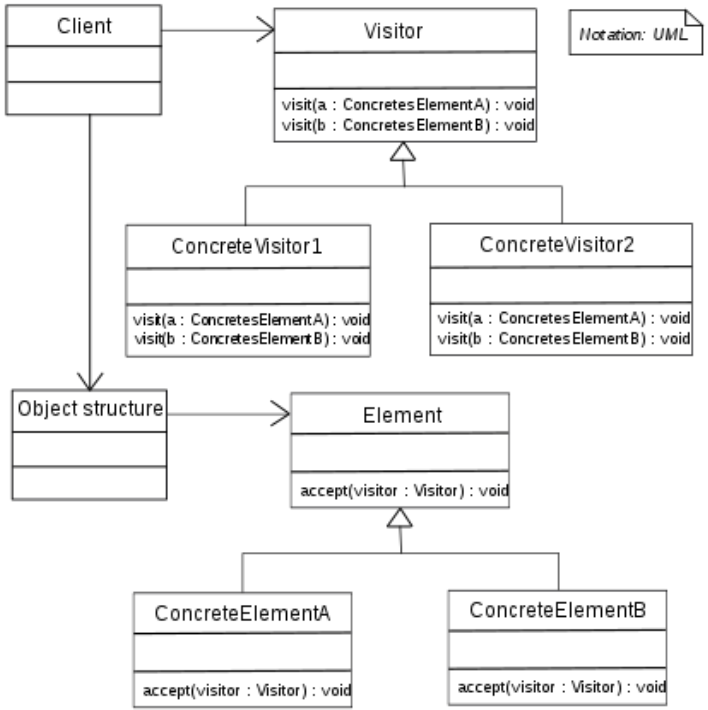
- When you want to access an aggregate object's content without exposing its internal representation
- When you want to support multiple ways of traversal for the same aggregate object
- When you want to avoid code duplication in regards to traversing the aggregate object

Visitor Pattern

Visitor Pattern

- Behavioral pattern
- 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.
  - ✓ The classic technique for recovering lost type information.
  - ✓ Do the right thing based on the type of two objects.
  - ✓ Double dispatch – the operation executed depends on: the name of the request, and the type of TWO receivers (the type of the Visitor and the type of the element it visits).
- The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters.

Visitor Pattern Structure



When to Use Visitor Pattern

- When an object structure contains many classes of objects with differing interfaces, and you want to perform operations on them that depend on their concrete classes
- When the classes defining your object structure don't have to change often, but you do often want to define new operations over the structure
- When you've got potentially many operations that need to be performed on objects in your object structure, but not necessarily on all of them

Visitor Pattern Consequences

- It makes adding new operations easy; you can define a new operation by creating a new visitor: open/closed principle
- A visitor can accumulate info on the objects it visits
- A visitor gathers related operations together (and separates unrelated ones: single responsibility principle
- Adding a new ConcreteElement class can be hard
- It might require you to break encapsulation

Literature

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software

John Dooley:

Software Development and Professional Practice (Chapter 11)

Thank You!