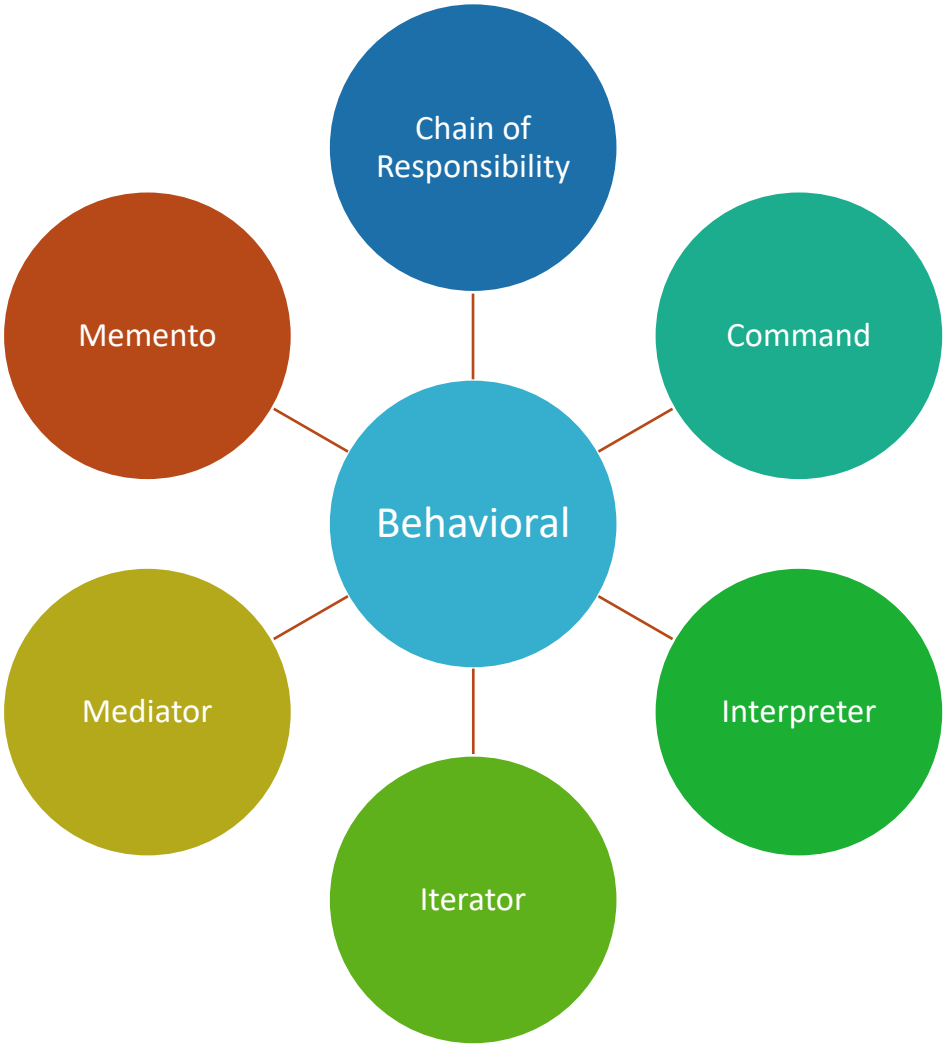
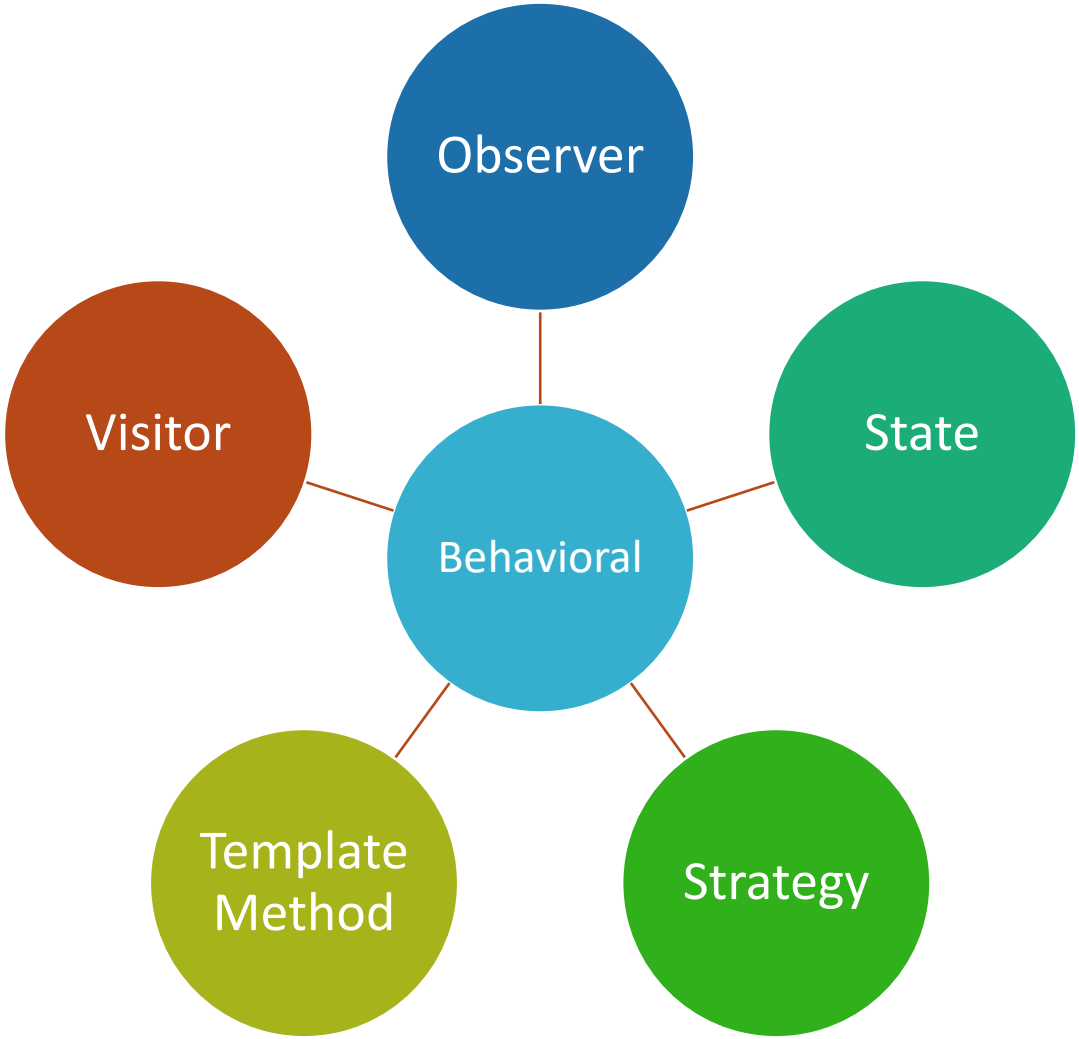
Design Patterns

Behavioral Patterns

Behavioral Design Patterns



Behavioral Design Patterns



Chain Of Responsibility

Chain Of Responsibility

Intent

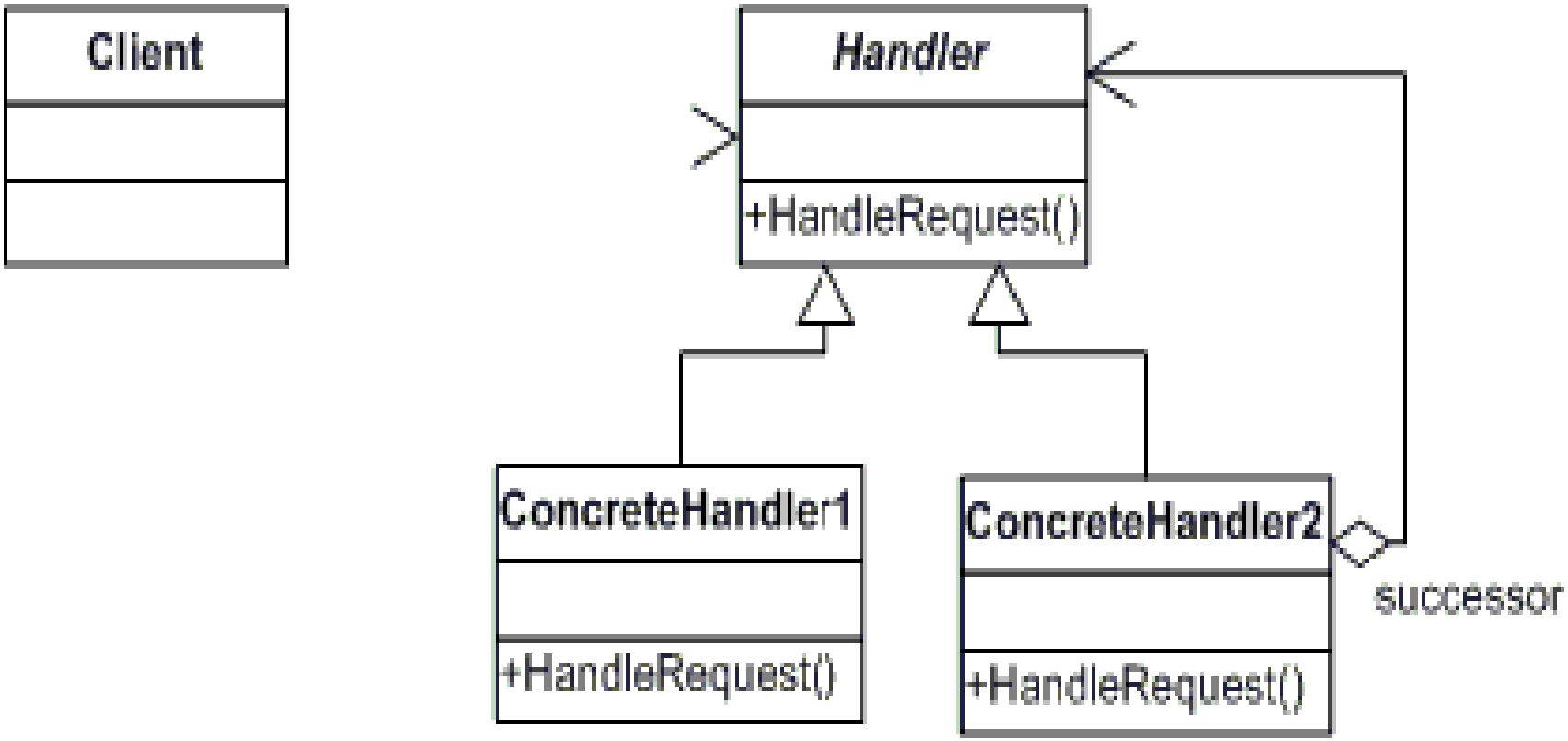
- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
- Chain the receiving objects and pass the request along the chain until an object handles it.

Applicability

- More than one object may handle a request, and the handler isn't known *a priori*.
- The set of objects that can handle a request should be specified dynamically.

Chain Of Responsibility - Structure

- Structure



Chain Of Responsibility - Participants

Handler	• defines an interface for handling requests.
ConcreteHandler	• Handles requests it is responsible for. • Can access its successor. • If the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.
Client	• Initiates the request to a ConcreteHandler object on the chain.

Chain Of Responsibility

Consequences

- Reduced Coupling
- Added flexibility in assigning responsibilities
- Receipt isn't guaranteed.

Implementation

- Implementing the successor chain
- Connecting successors
- Representing requests.

Demo

Command Pattern

Command Pattern

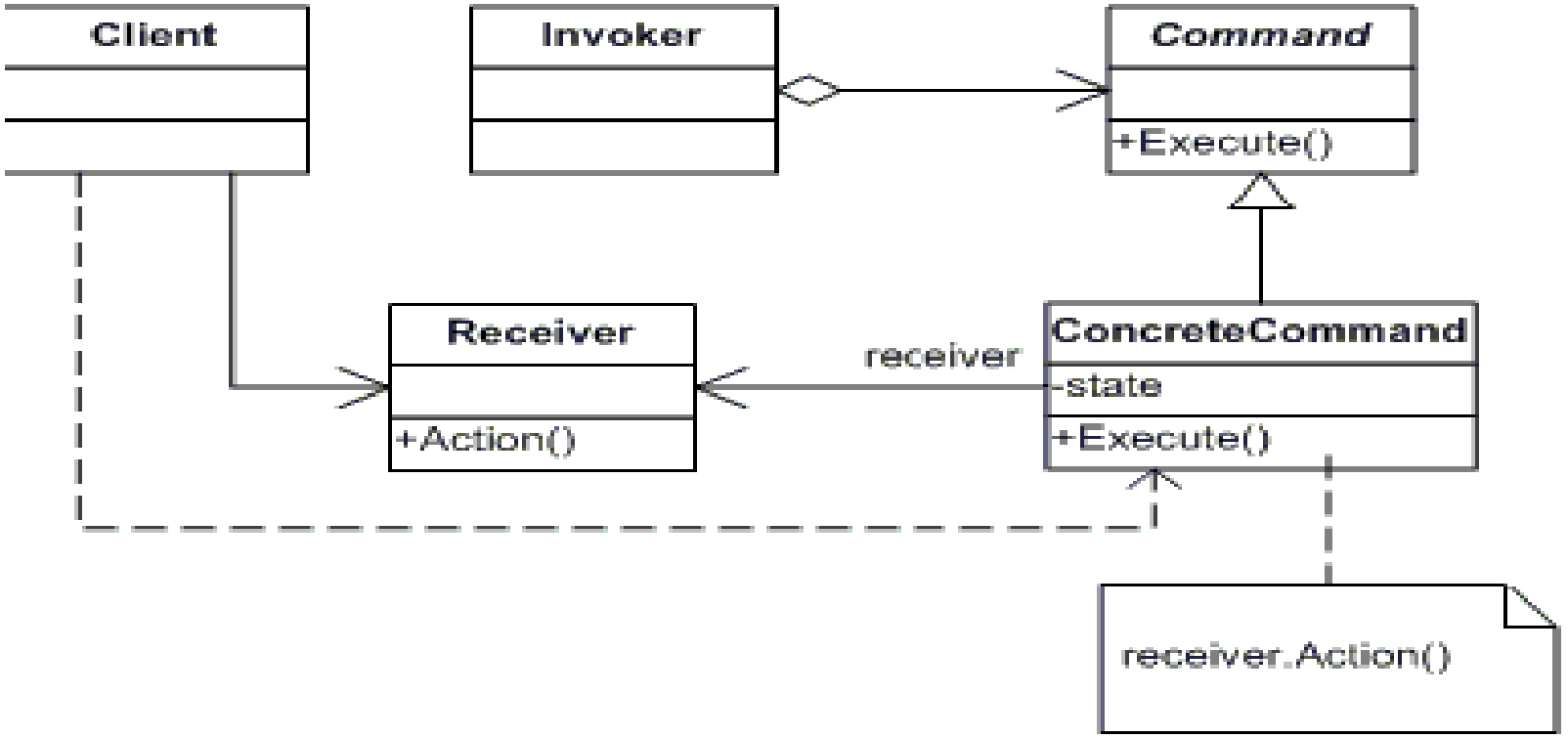
Intent

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Also known as **“Action, Transaction”**

Applicability

- parameterize objects by an action to perform
- specify, queue, and execute requests at different times
- support undo
- support logging changes so that they can be reapplied in case of a system crash

Command Pattern - Structure



Command Pattern - Participants

Command	• Declares an interface for executing an operation.
ConcreteCommand	• Defines a binding between a Receiver and an action. • Implements Execute by invoking the corresponding operation(s) 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 Pattern

Consequences

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands are first-class objects.
- You can assemble commands into a composite command
- It's easy to add new Commands

Implementation

- Commands can have wide range of abilities
- How many levels of undo should be supported
- Undo can cause multiple errors

Demo

Observer Pattern

Observer 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
- Also known as **“Dependents, Publish-Subscribe”**

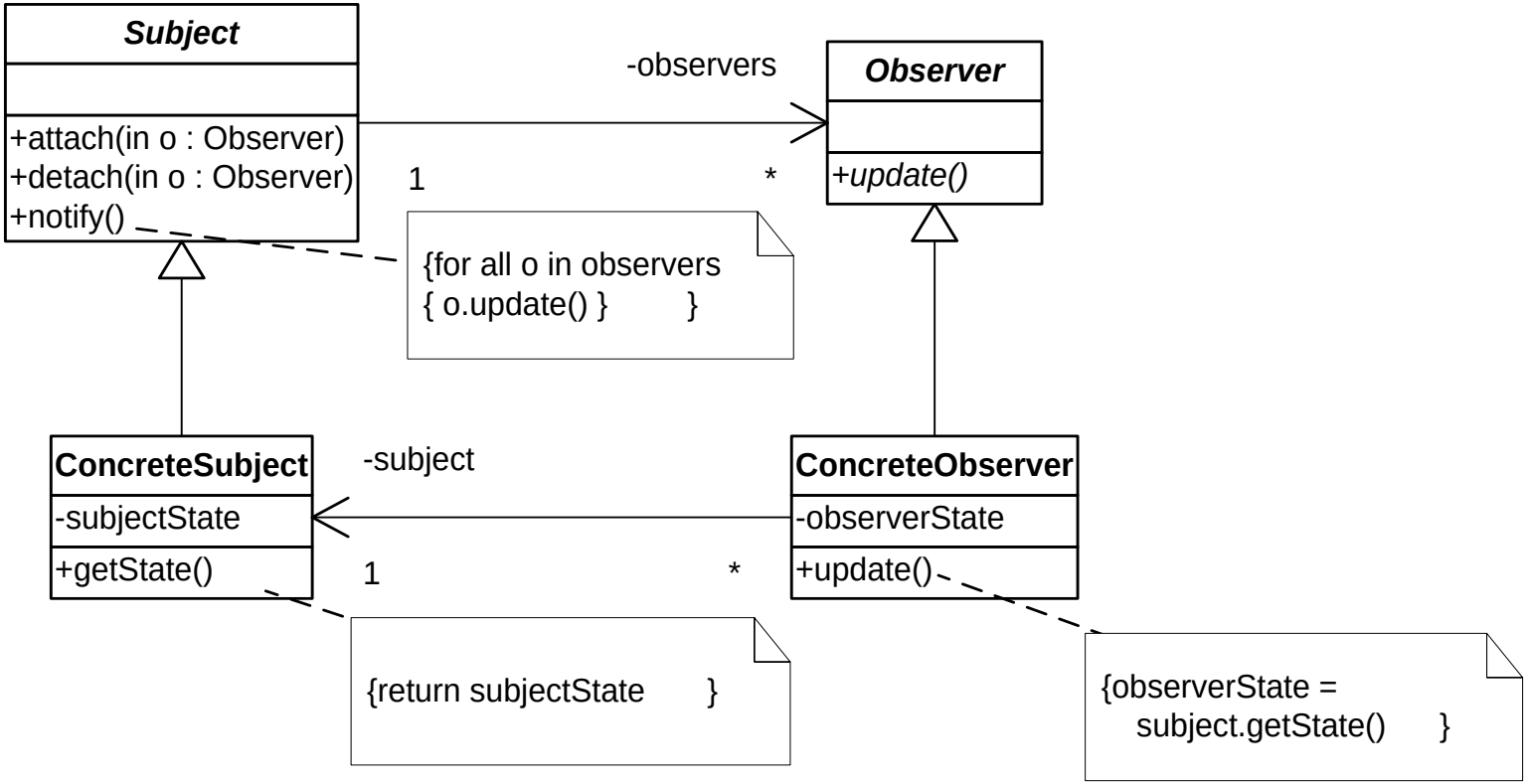
Applicability

- When an abstraction has two aspects, one dependent on the other
- 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 notify other objects without making assumptions about who these objects are

Observer Pattern

- Define a one-to-many relationship between objects so that when one object changes state, all its dependants are notified and updated automatically.
- Pattern usage very high
- Emphasizes and promotes loose coupling.

Observer Pattern - Structure



Observer Pattern - Participants

Subject	• knows its observers • 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 and send notifications
ConcreteObserver	• maintains Subject reference, stores state

Observer Pattern

Consequences

- the coupling between subjects and observers is abstract and minimal.
- observer to handle or ignore a broadcast notifications.
- Unexpected updates can happen as observers do not know of each other

Implementation

- Subject-observer mapping
- Dangling references
- Avoiding observer-specific update protocols: the push and push/pull models
- Registering modifications of interest explicitly

Demo

Iterator Pattern

Iterator Pattern

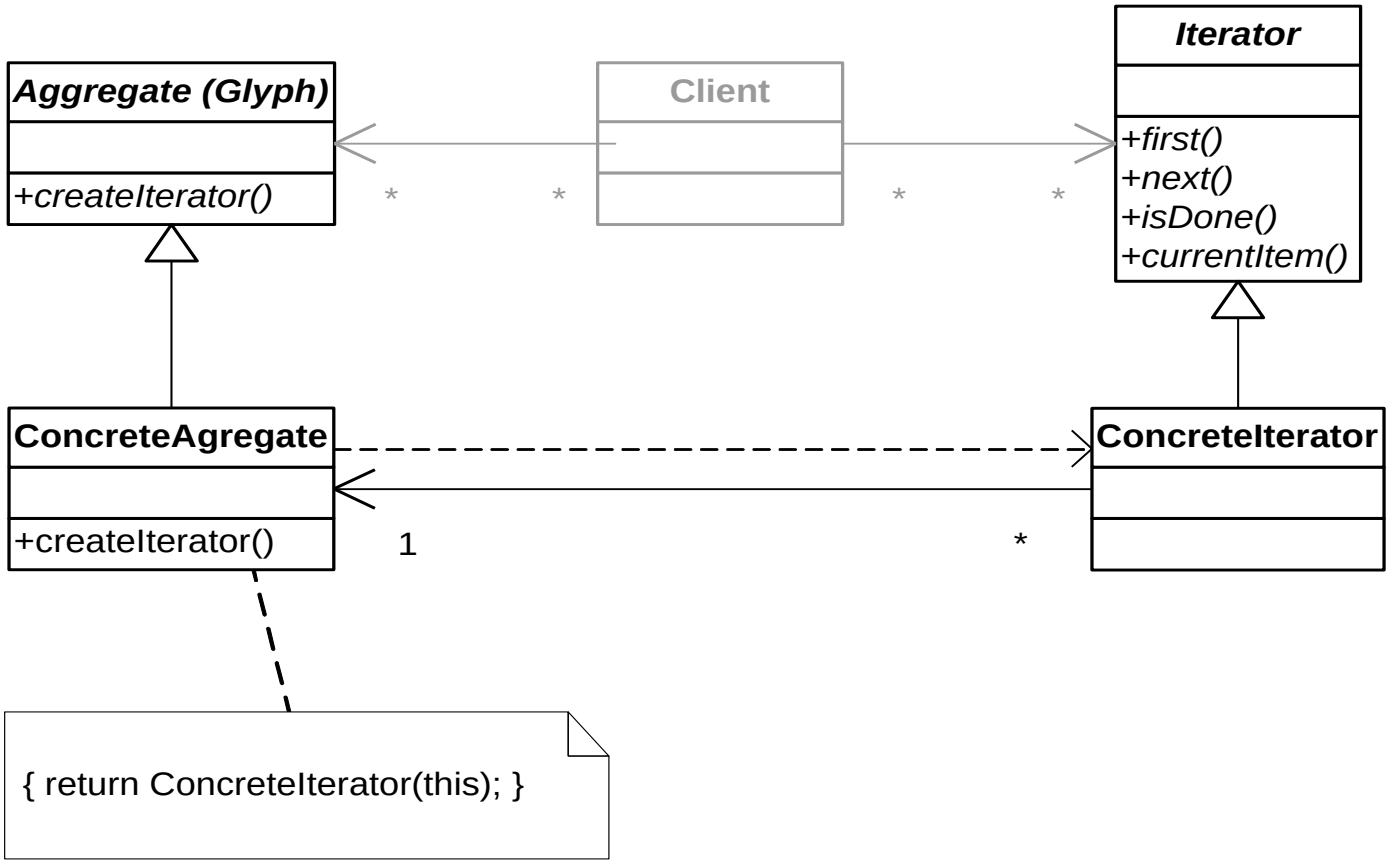
Intent

- Access elements of an aggregate sequentially without exposing its underlying representation
- Also known as **“Cursor”**

Applicability

- Require multiple traversal algorithms over an aggregate
- Require a uniform traversal interface over different aggregates
- When aggregate classes and traversal algorithm must vary independently

Iterator Pattern – Structure



Iterator Pattern - 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 Pattern

Consequences

- Supports variations in the traversal of aggregates
- Simplifies the aggregate interface
- Multiple traversals can happen at a point in time and we should be aware of it

Implementation

- Deciding the controller of iteration
- Decides where the traversal algorithm should be (internal or external)
- Ensure the iterator algorithm is robust

Demo

Memento Pattern

Memento Pattern

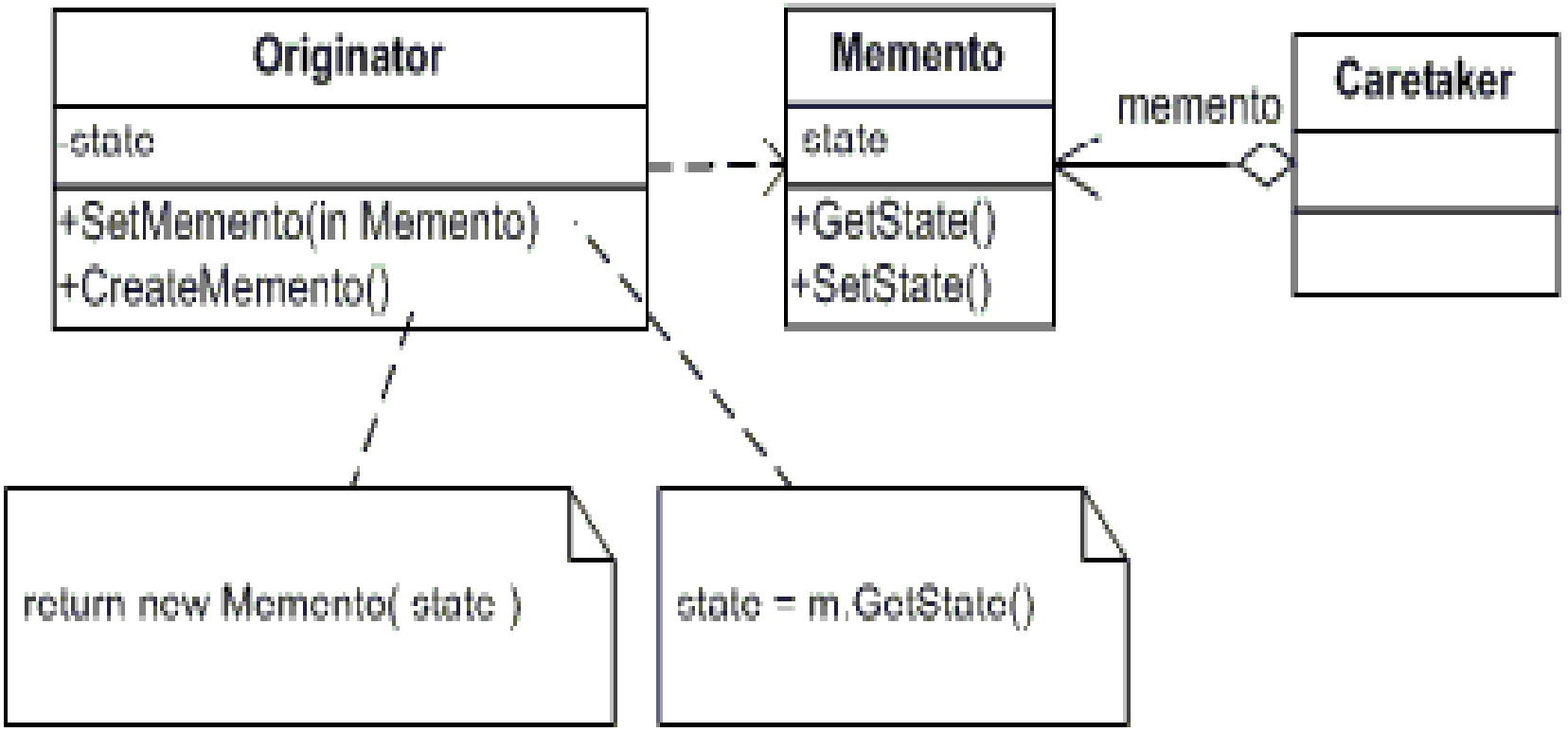
Intent

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later
- Also known as **“Token”**

Applicability

- a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later
- a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Memento Pattern - Structure



Memento Pattern - Participants

Memento

- stores internal state of the Originator object
- protects against access by objects other than the originator.

Originator

- creates a memento containing a snapshot of its current internal state
- uses the memento to restore its internal state

Caretaker

- is responsible for the memento's safekeeping
- never operates on or examines the contents of a memento

Memento Pattern

Consequences

- Perserving Encapsulation boundaries
- Simplifies the Originator
- Using mementoes might be expensive
- Defining too narrow or too wide interfaces
- Hidden costs in caring for the Memento

Implementation

- Need to consider Language support
- Storing incremental changes

Demo

Mediator Pattern

Mediator Pattern

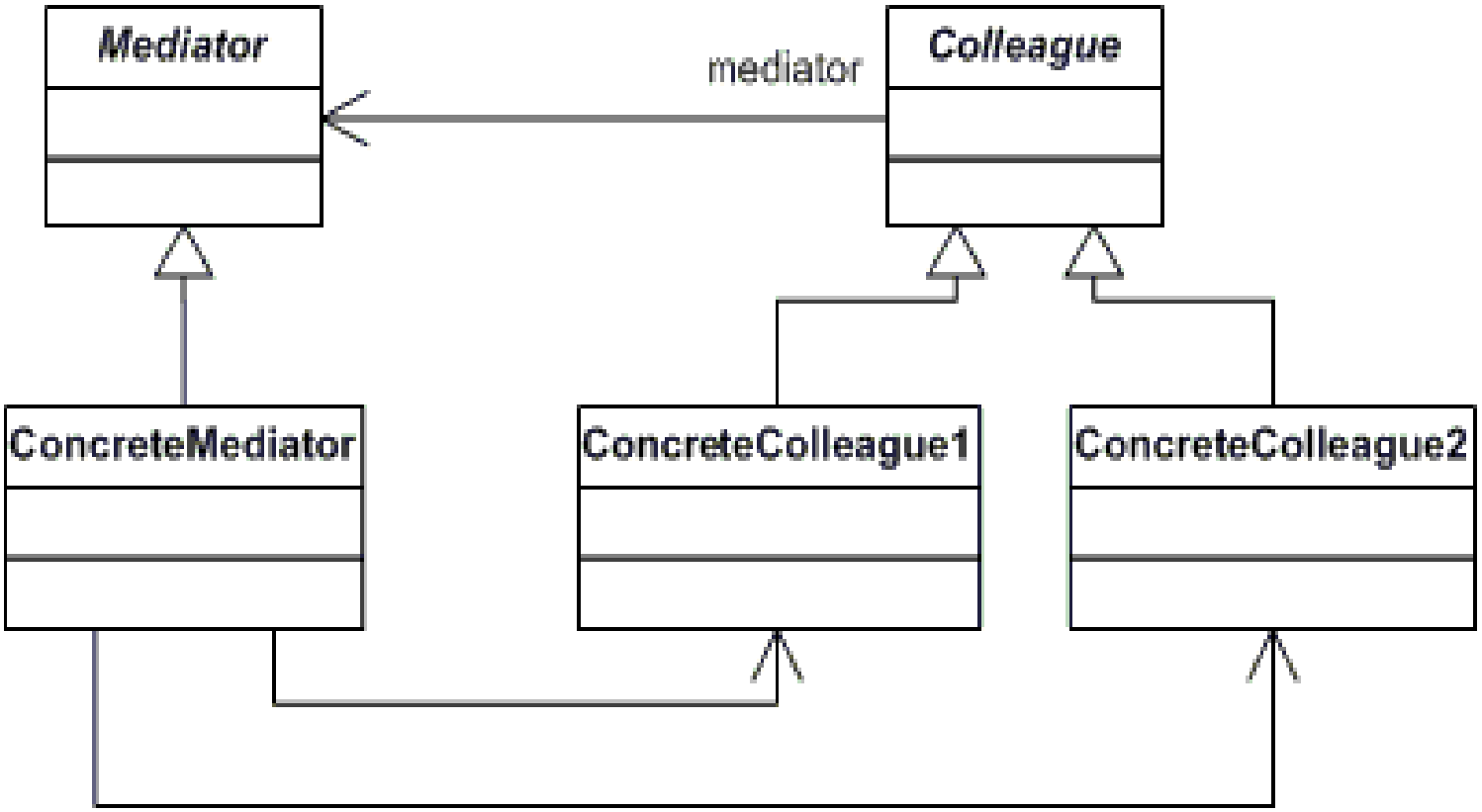
Intent

- Define an object that encapsulates how a set of objects interact.
- Mediator promotes loose coupling

Applicability

- a set of objects communicate in well-defined but complex ways
- reusing an object is difficult because it refers to and communicates with many other objects
- a behavior that's distributed between several classes should be customizable without a lot of subclassing

Mediator Pattern - Structure



Mediator Pattern - Participants

Mediator	• defines an interface for communicating with Colleague objects.
ConcreteMediator	• implements cooperative behavior by coordinating Colleague objects. • knows and maintains its colleagues
Colleague classes	• each Colleague class knows its Mediator object and communicates with it

Mediator Pattern

Consequences

- Limits Subclassing
- Decouples Colleagues
- Simplifies Object protocols
- Abstracts how objects cooperate
- Centralizes control

Implementation

- The Mediator abstract class can be omitted
- Colleague-Mediator communication can be implemented as Observer-Subject pattern

Demo

Interpreter Pattern

Interpreter Pattern

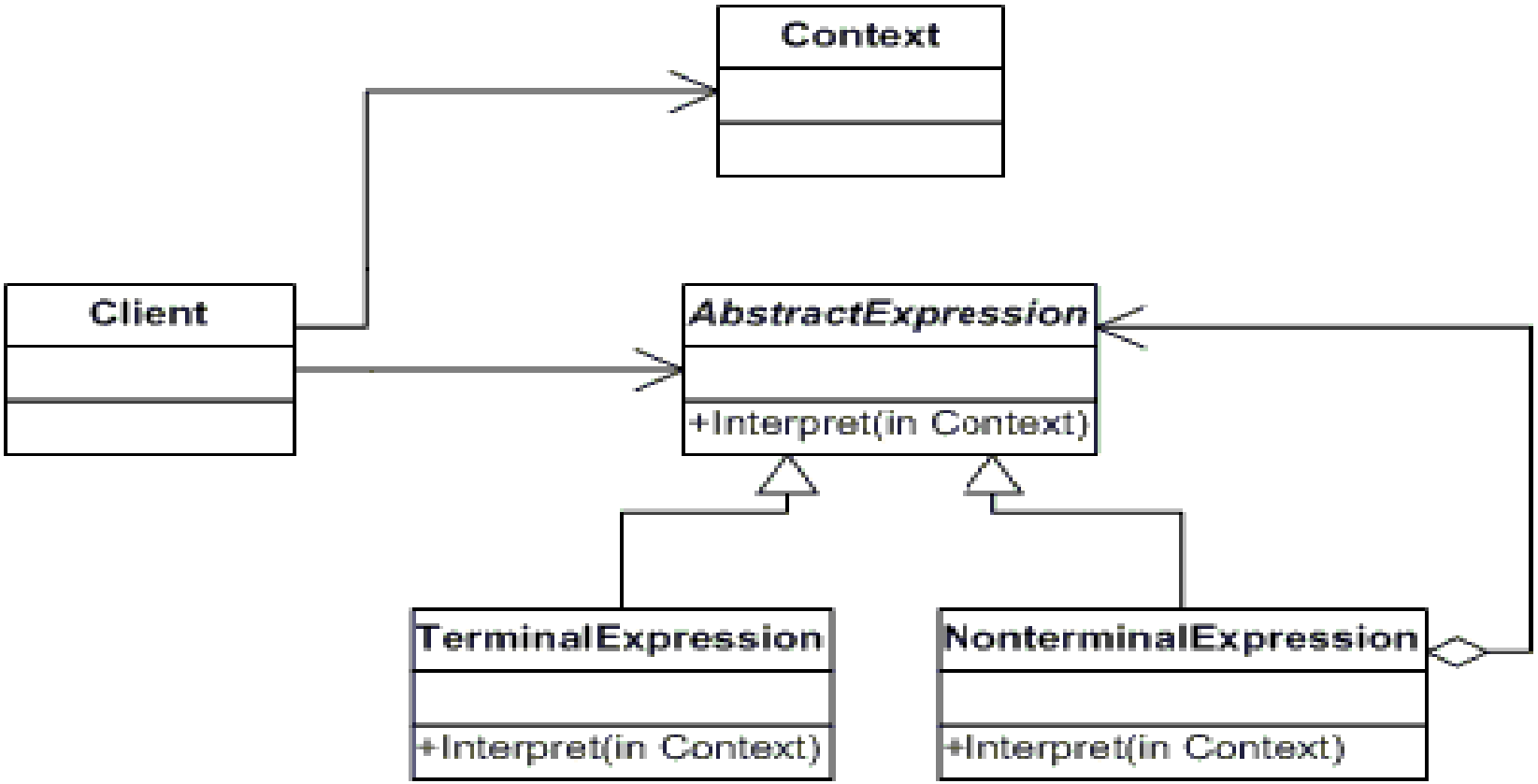
Intent

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language

Applicability

- Works best when the grammar is simple
- When efficiency is not a critical concern

Interpreter Pattern - Structure



Interpreter Pattern - Participants

AbstractExpression	• Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
TerminalExpression	• implements an Interpret operation associated with terminal symbols. • an instance required for every terminal symbol in a sentence.
NonterminalExpression	• Implements an Interpret operation for nonterminal symbols in the grammar
Context	• Contains information that's global to the interpreter.
Client	• Builds the expressions and invokes the Interpret operation

Interpreter Pattern

Consequences

- easy to change and extend the grammar
- Implementing the grammar is easy
- Complex grammars are hard to maintain
- Add new ways to interpret expressions.

Implementation

- Creating abstract syntax tree
- Defining the Interpret operation
- Sharing terminal symbols with a Flyweight pattern

Demo

State Pattern

State Pattern

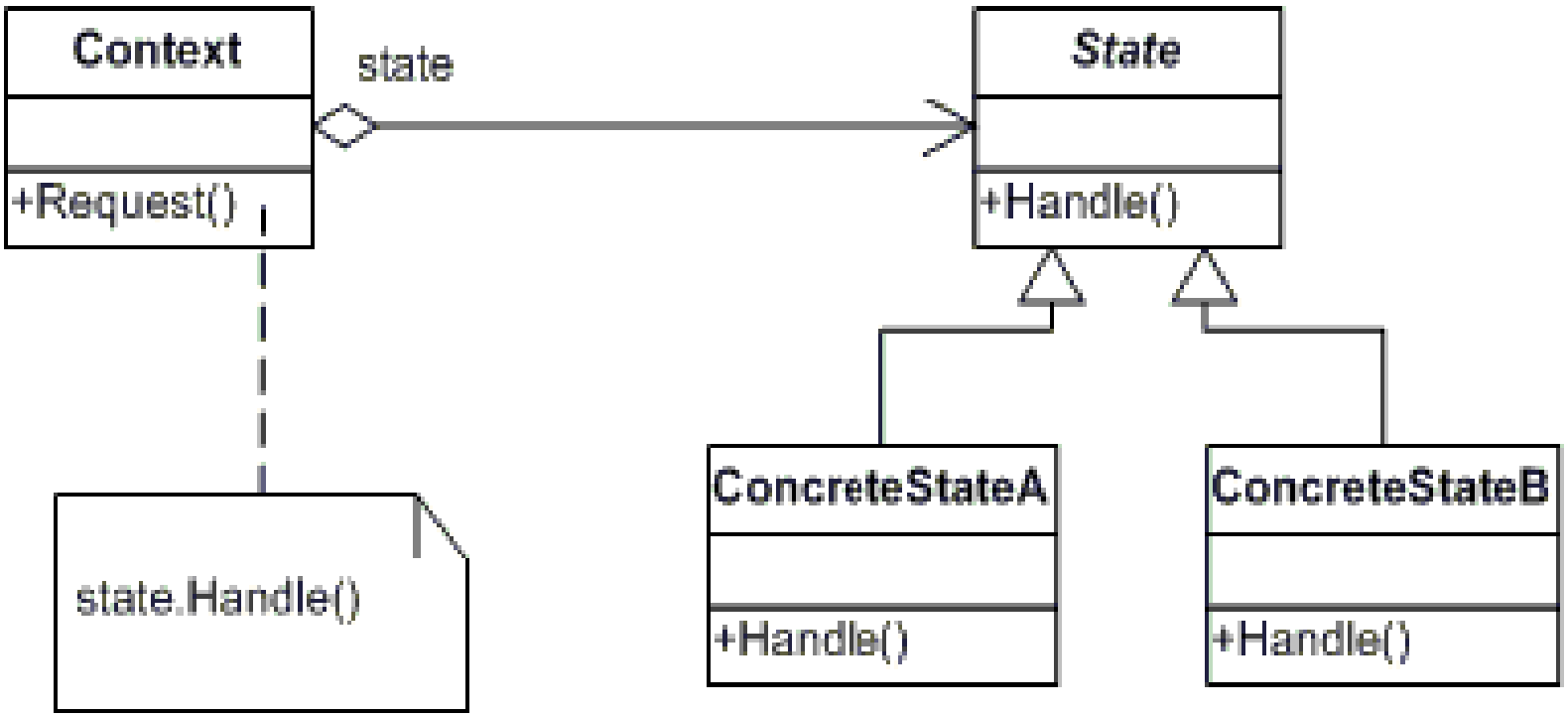
Intent

- Allow an object to alter its behavior when its internal state changes.
- Also known as **“Objects for States”**

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

State Pattern - Structure



State Pattern - 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 Pattern

Consequences

- Localizes state-specific behavior
- Make state transition explicit
- State objects can be shared

Implementation

- Control the object which maintains state
- Provide an implementation to create and destroy state objects

Demo

Strategy Pattern

Strategy Pattern

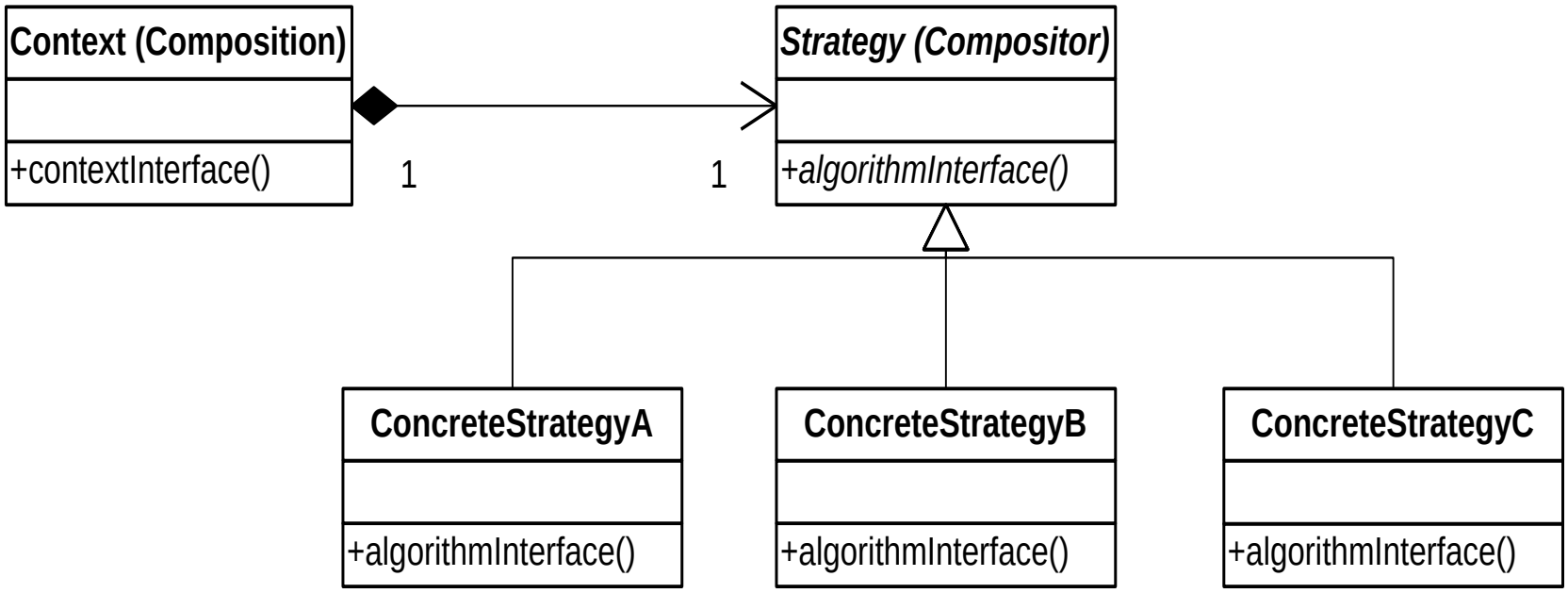
Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable to let clients and algorithms vary independently
- Also known as **“Policy”**

Applicability

- many related classes differ only in their behavior
- you need different variants of an algorithm
- an algorithm uses data that clients
- a class defines many behaviors, and these appear as multiple conditional statements in its operations

Strategy Pattern - Structure



Strategy Pattern - Participants

Strategy	• declares an interface common to all supported algorithms.
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 Pattern

Consequences

- Greater flexibility, reuse
- Is an alternative to subclassing
- Strategy can eliminate conditional statements
- Increased number of objects

Implementation

- Exchanging information between a strategy and its context
- Static strategy selection via templates

Demo

Template Method Pattern

Template Method Pattern

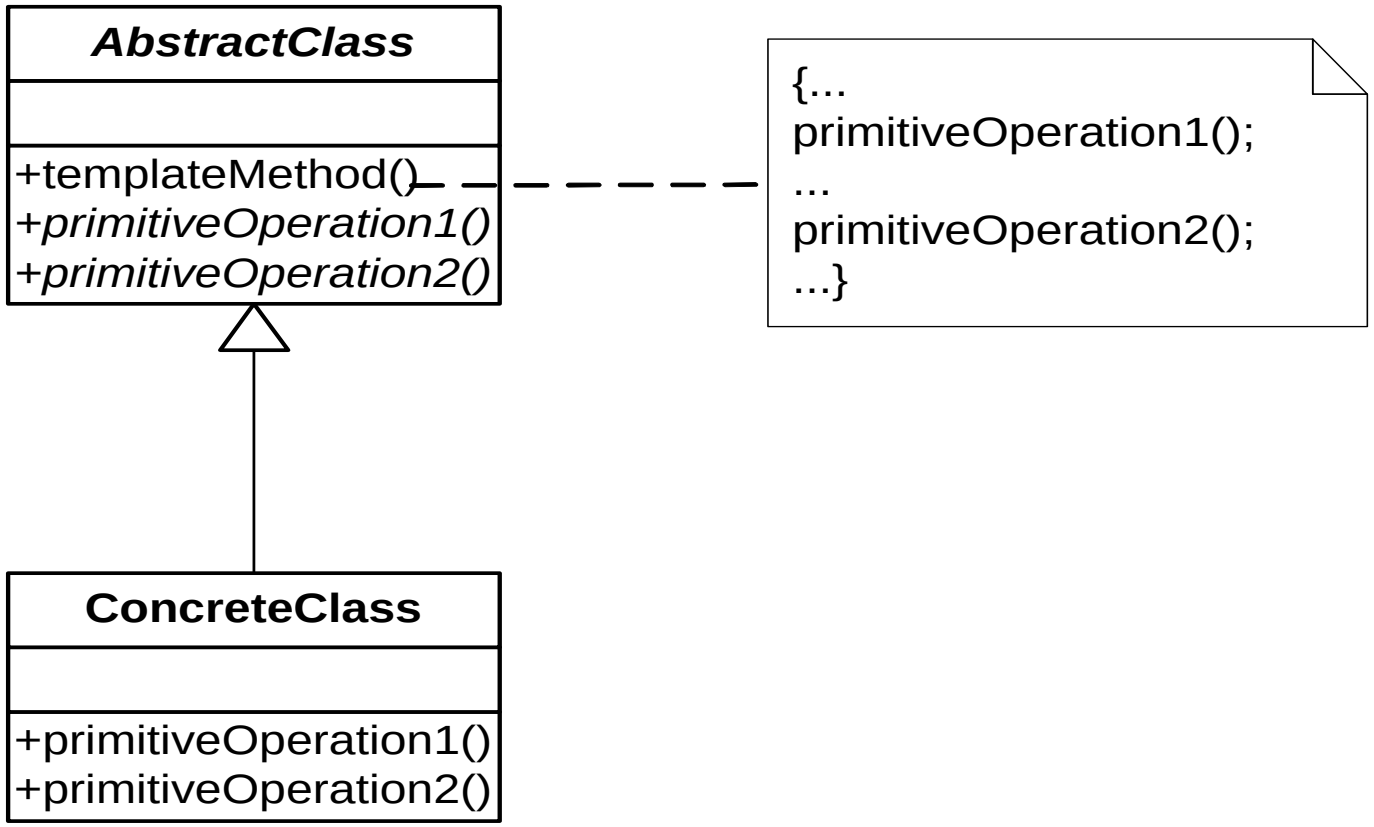
Intent

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses

Applicability

- To implement invariant aspects of an algorithm once and let subclasses define variant parts
- To localize common behavior in a class to increase code reuse
- To control subclass extensions

Template Method - Structure



Template Method - 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.

ConcreteClass

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

Template Method Pattern

Consequences

- Leads to inversion of control (“Hollywood principle”: don't call us – we'll call you)
- Promotes code reuse
- Lets you enforce overriding rules
- Must subclass to specialize behavior

Implementation

- Virtual vs. non-virtual template method
- Few vs. lots of primitive operations
- Naming conventions (do- prefix in Mac)

Demo

Visitor Pattern

Visitor Pattern

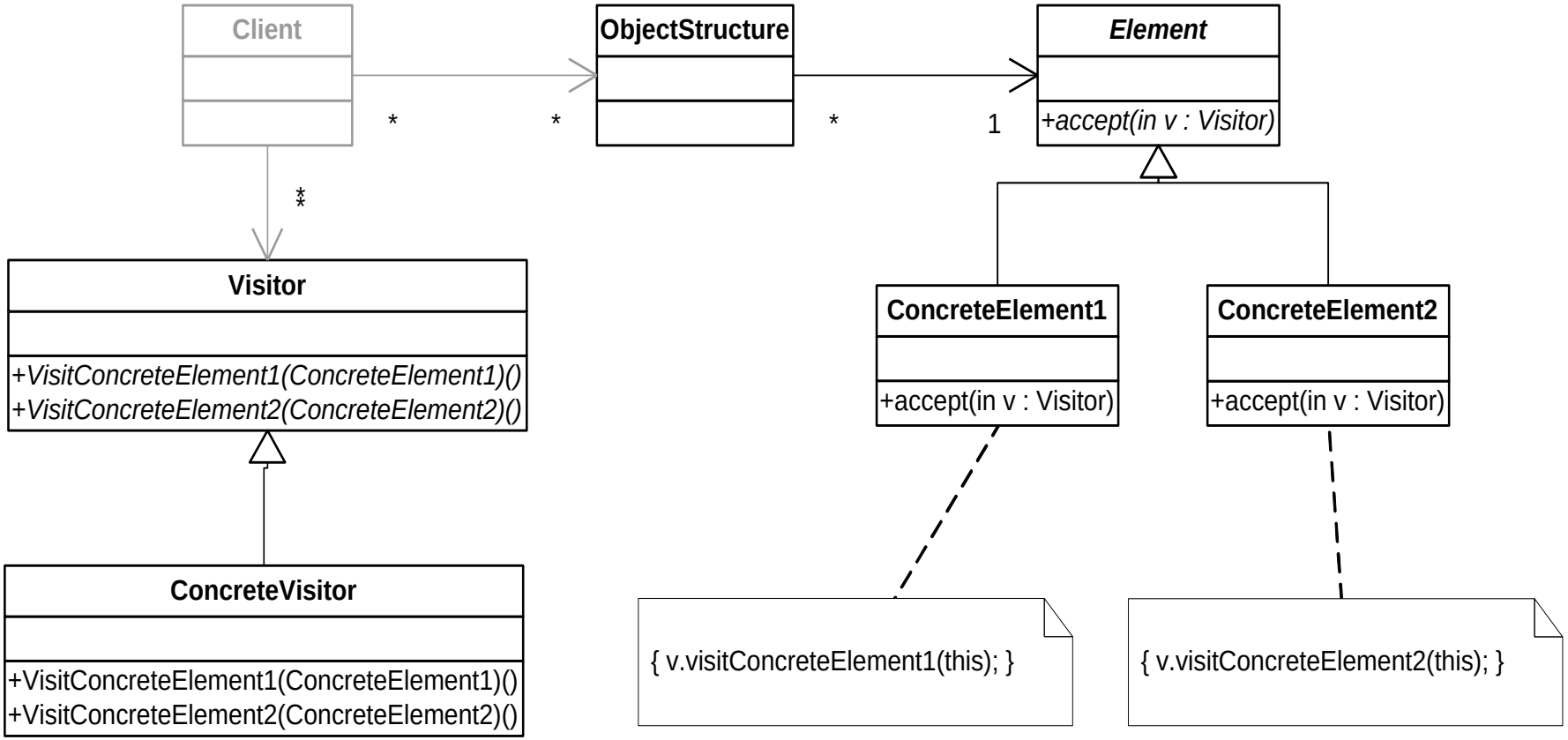
Intent

- Represent an operation to be performed on the elements of an object structure.

Applicability

- an object structure contains many classes of objects with differing interfaces,
- many distinct and unrelated operations need to be performed on objects in an object structure
- the classes defining the object structure rarely change, but you often want to define new operations over the structure

Visitor Pattern - Structure



Visitor Pattern - Participants

Visitor	• declares a Visit operation for each class of ConcreteElement in the object structure.
ConcreteVisitor	• implements each operation declared by Visitor
Element	• defines an Accept operation that takes a visitor as an argument.
ConcreteElement	• implements an Accept operation that takes a visitor as an argument.
ObjectStructure	• can enumerate its elements and may provide high-level interface

Visitor Pattern

Consequences

- Visitor makes adding new operations easy
- visitor gathers related operations and separates unrelated ones
- Adding new ConcreteElement classes is hard
- Visiting across class hierarchies

Implementation

- Double dispatch

Demo