Behavioral patterns I

- Chain of responsibility
- Command
- Iterator
- Memento
- Strategy
- State

Chain of responsibility

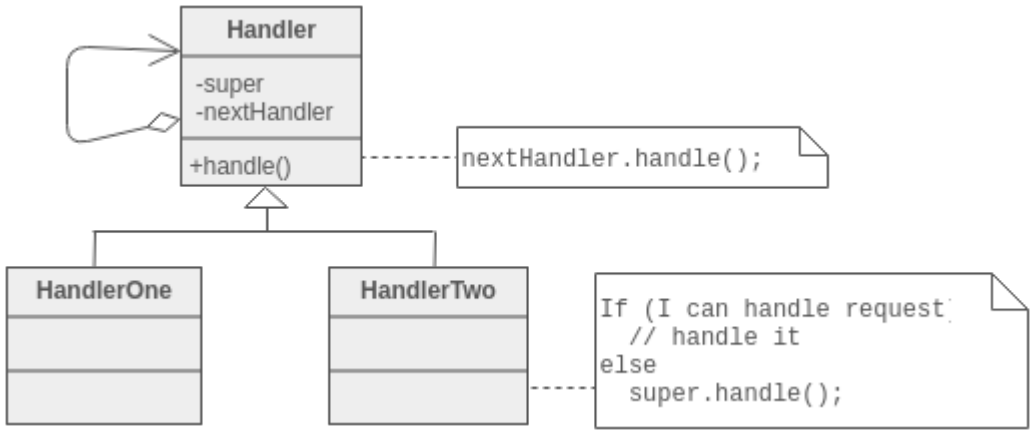
Problem

There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.

Solution

The pattern chains the receiving objects together, and then passes any request messages from object to object until it reaches an object capable of handling the message. The number and type of handler objects isn't known a priori, they can be configured dynamically. The chaining mechanism uses recursive composition to allow an unlimited number of handlers to be linked.

Diagram


Handler
-super
-nextHandler
+handle()
nextHandler.handle();
HandlerOne
HandlerTwo
If (I can handle request
// handle it
else
super.handle();

Command

Problem

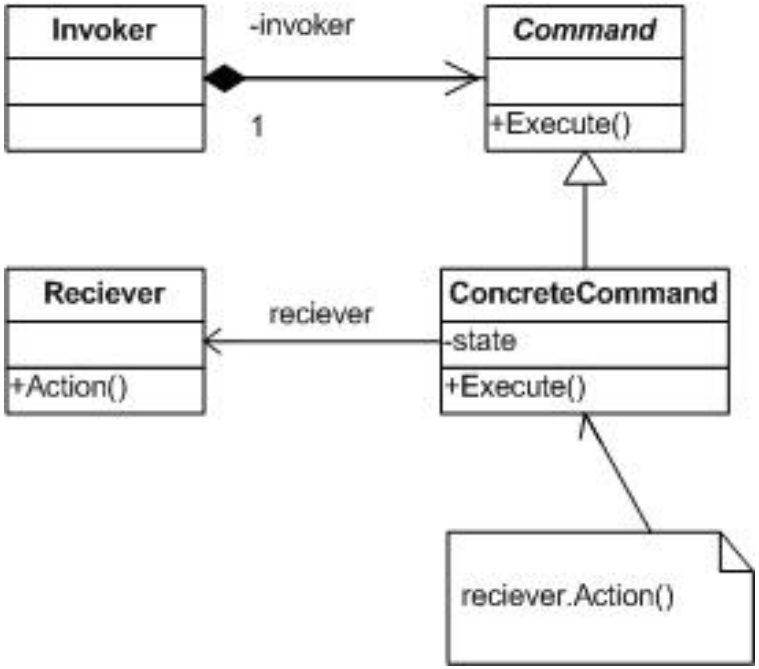
Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Solution

Command decouples the object that invokes the operation from the one that knows how to perform it. To achieve this separation, the designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an execute() method that simply calls the action on the receiver.

All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual execute() method whenever the client requires the object's "service".

Diagram


Invoker
-invoker
1
Command
+Execute()
Reciever
+Action()
reciever
ConcreteCommand
-state
+Execute()
reciever.Action()

Iterator

Problem

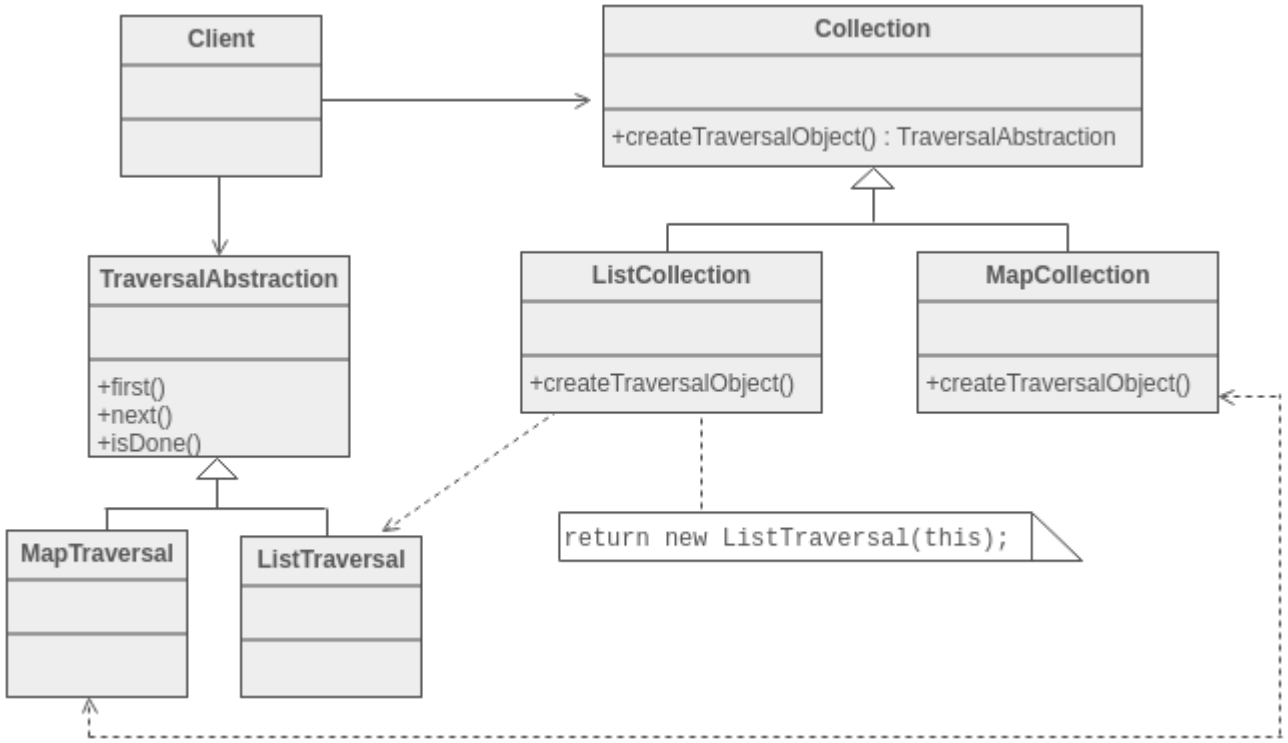
Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.

Solution

The key idea of the Iterator pattern is to take the responsibility for access and traversal out of the aggregate object and put it into an Iterator object that defines a standard traversal protocol.

The Iterator abstraction is fundamental to an emerging technology called "generic programming". This strategy seeks to explicitly separate the notion of "algorithm" from that of "data structure". The motivation is to: promote component-based development, boost productivity, and reduce configuration management.

Diagram


Client
Collection
+createTraversalObject() : TraversalAbstraction
TraversalAbstraction
+first()
+next()
+isDone()
ListCollection
+createTraversalObject()
MapCollection
+createTraversalObject()
MapTraversal
ListTraversal
return new ListTraversal(this);

Memento

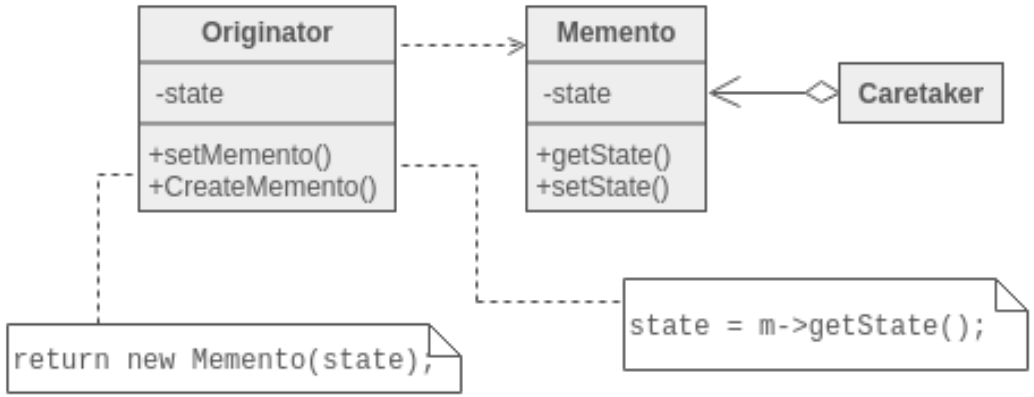
Problem

Need to restore an object back to its previous state (e.g. "undo" or "rollback" operations).

Solution

The client requests a Memento from the source object when it needs to checkpoint the source object's state. The source object initializes the Memento with a characterization of its state. The client is the "care-taker" of the Memento, but only the source object can store and retrieve information from the Memento (the Memento is "opaque" to the client and all other objects). If the client subsequently needs to "rollback" the source object's state, it hands the Memento back to the source object for reinstatement.

Diagram


Originator
-state
+setMemento()
+CreateMemento()
Memento
-state
+getState()
+setState()
Caretaker
return new Memento(state);
state = m->getState();

Strategy

Problem

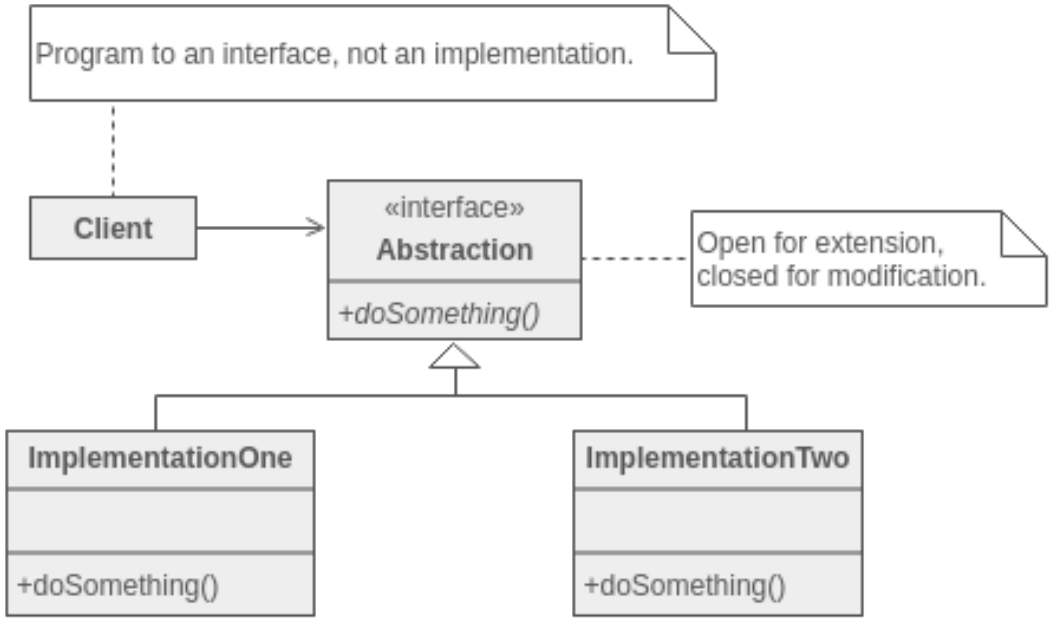
Create interchangeable polymorphic algorithms and change the algorithm dynamically.

Enable an algorithm's behavior selection at runtime

Solution

Create an interface with a method for each algorithm. Implement concrete algorithm in in derived classes. Store reference to current algorithm and change it by an event.

Diagram


Program to an interface, not an implementation.
Client
«interface»
Abstraction
+doSomething()
Open for extension,
closed for modification.
ImplementationOne
+doSomething()
ImplementationTwo
+doSomething()

State

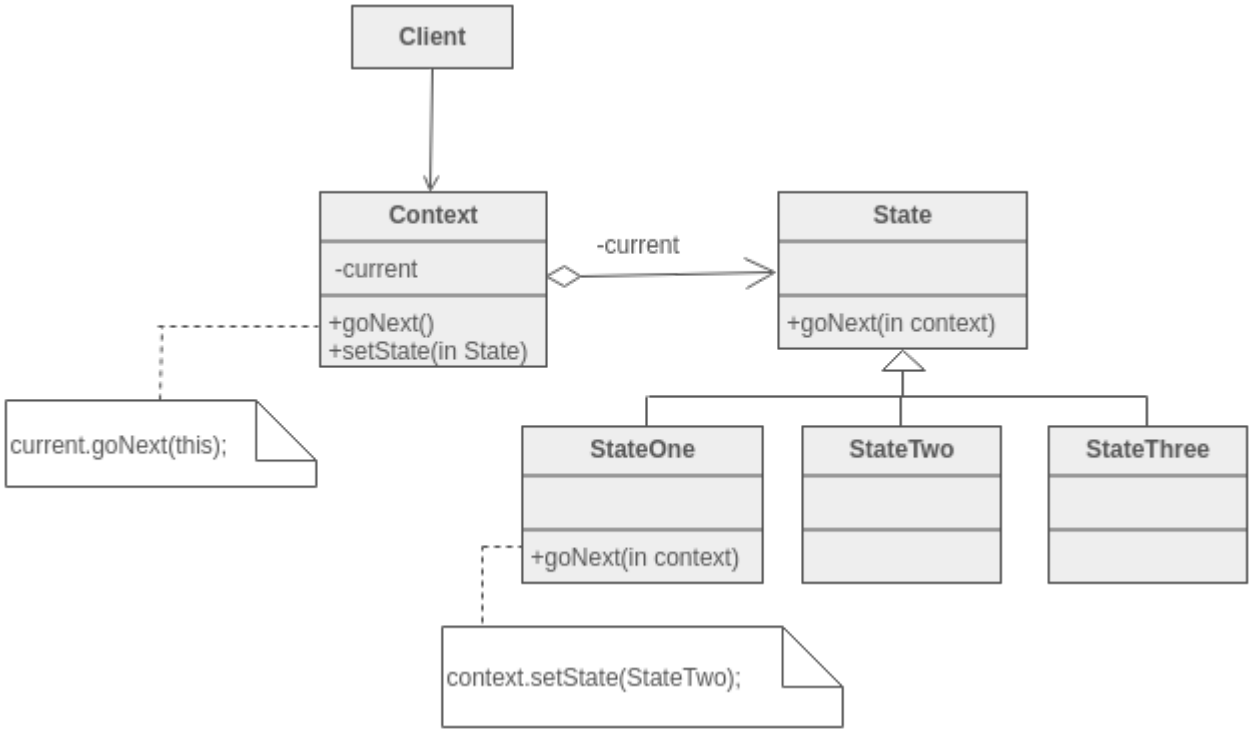
Problem

Be able to set object's behavior as a function of its state. Change its behavior at run-time depending on that state.

Solution

- Define a "context" class to present a single interface to the outside world.
- Define a State abstract base class.
- Represent the different "states" of the state machine as derived classes of the State base class.
- Define state-specific behavior in the appropriate State derived classes.
- Maintain a pointer to the current "state" in the "context" class.
- To change the state of the state machine, change the current "state" pointer.

Diagram


Client
Context
-current
+goNext()
+setState(in State)
-current
State
+goNext(in context)
current.goNext(this);
StateOne
+goNext(in context)
StateTwo
StateThree
context.setState(StateTwo);