COMP 3700: Software Modeling and Design

(Behavioral Design Patterns)

Agenda

- Chain of Responsibility (COF)
- Mediator
- Iterator
- Command
- Momento

Chain of Responsibility - Intent

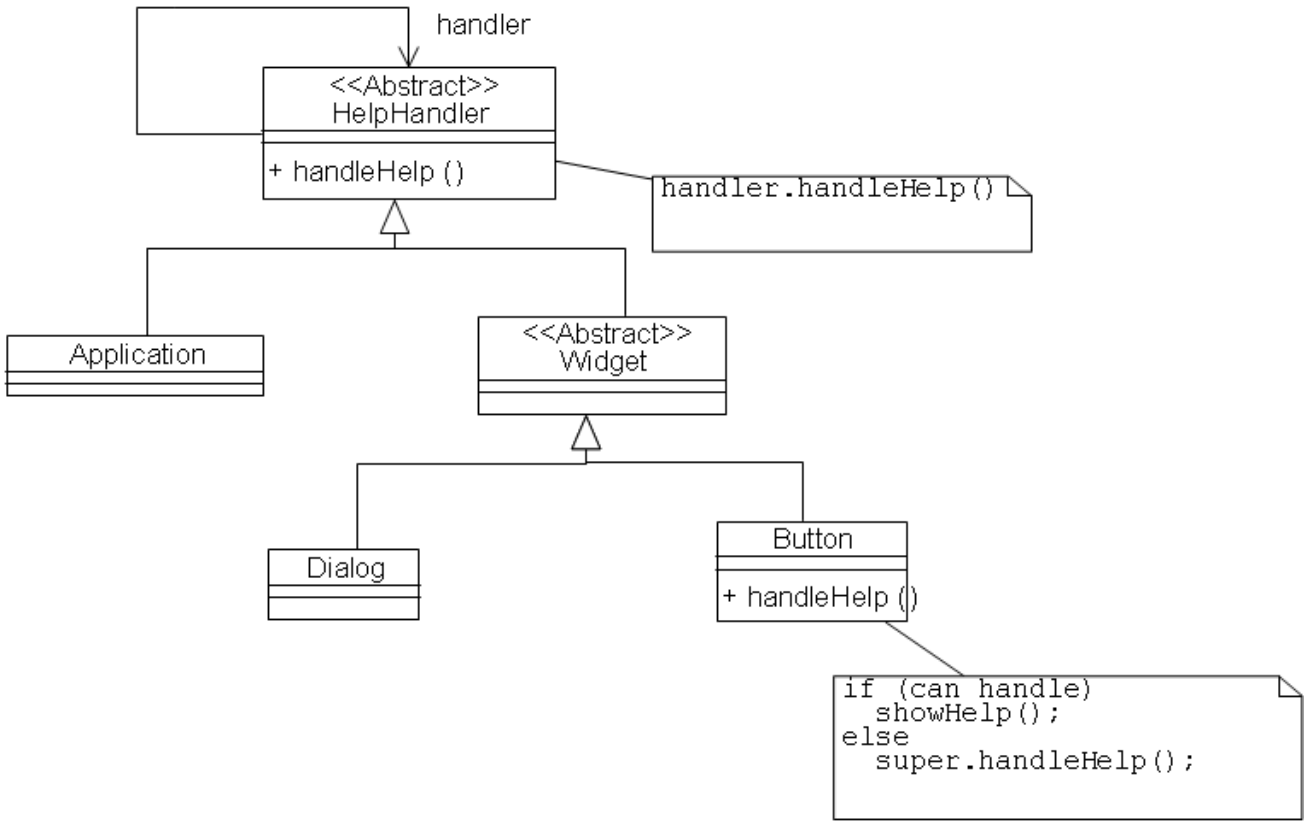
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Chain of Responsibility - Motivation

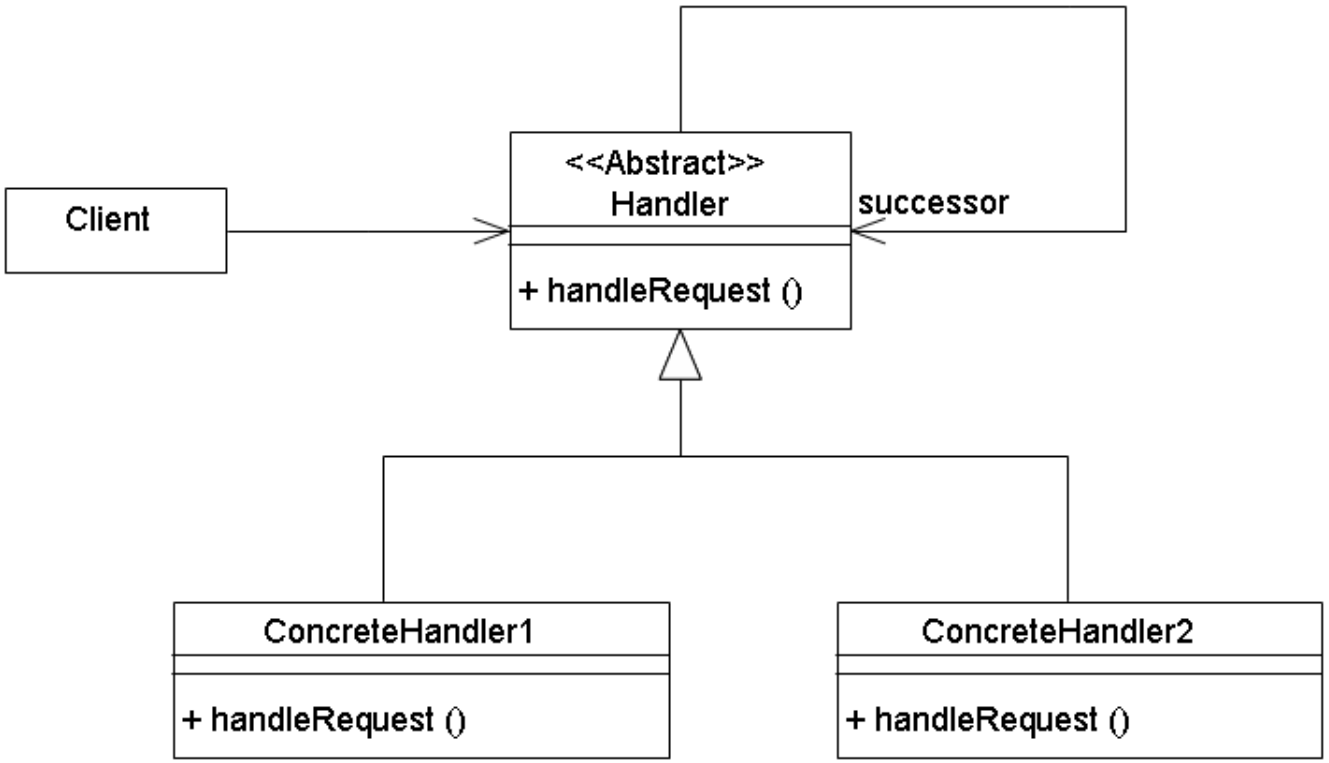
Context-Sensitive Help System

A button on a dialog box may have some specific help, or can pass it to its immediate successor, and so on, until the request gets handled.

Chain of Responsibility - Example



Chain of Responsibility - Structure



Chain of Responsibility - Applicability

- To let more than one object handle the request, and the handler is not known prior to it.
- To issue a request to one of the several objects without specifying the receiver explicitly.
- To allow dynamic selection of the receiver.

Chain of Responsibility - Consequences

- ✓ Reduced coupling between the sender and the receiver.
- ✓ Responsibilities can be added dynamically by configuring the chain at runtime.
- ✘ Receipt is not guaranteed. If the chain is not configured properly, requests can go unhandled .

Chain of Responsibility - Related Patterns

- Composite can be used in defining the successor chain.

Mediator - Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

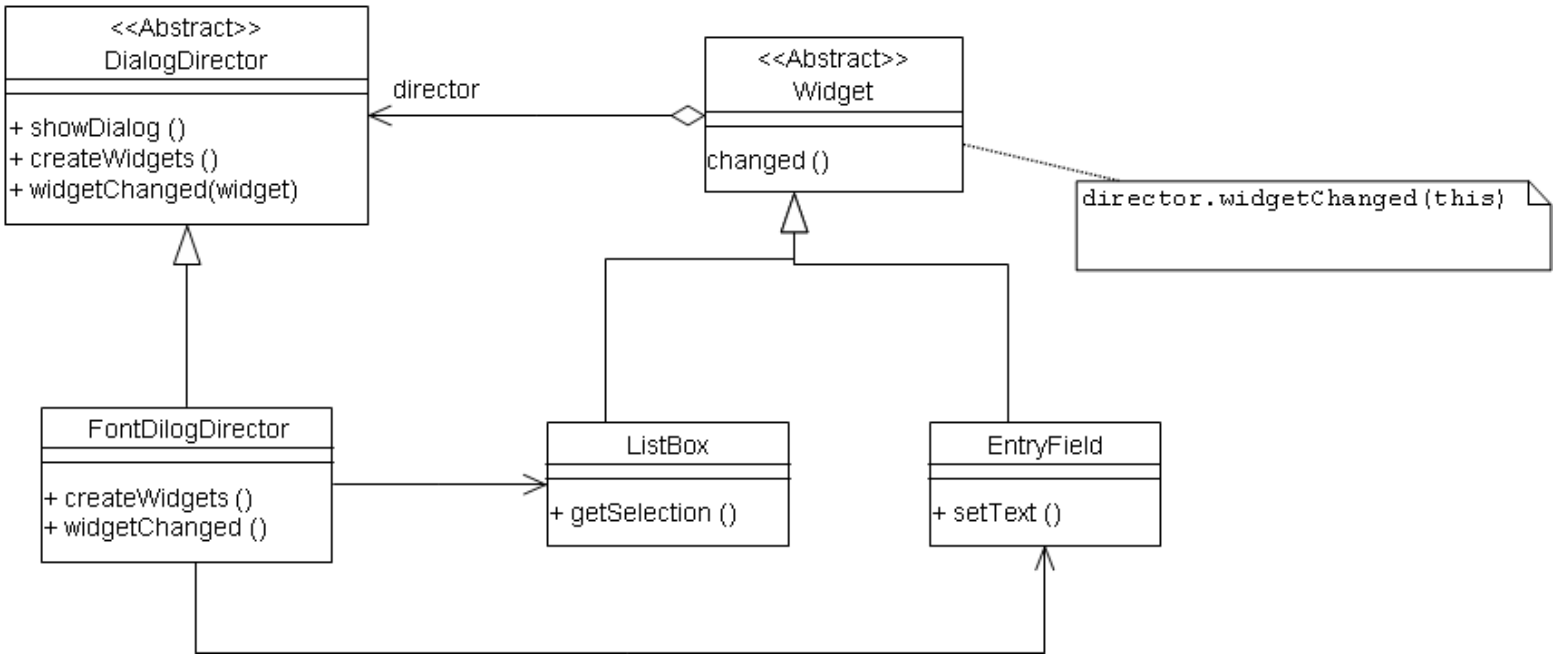
Mediator - Motivation

In designing dialog box, there can be dependencies between the controls (widgets) of the dialog box as:

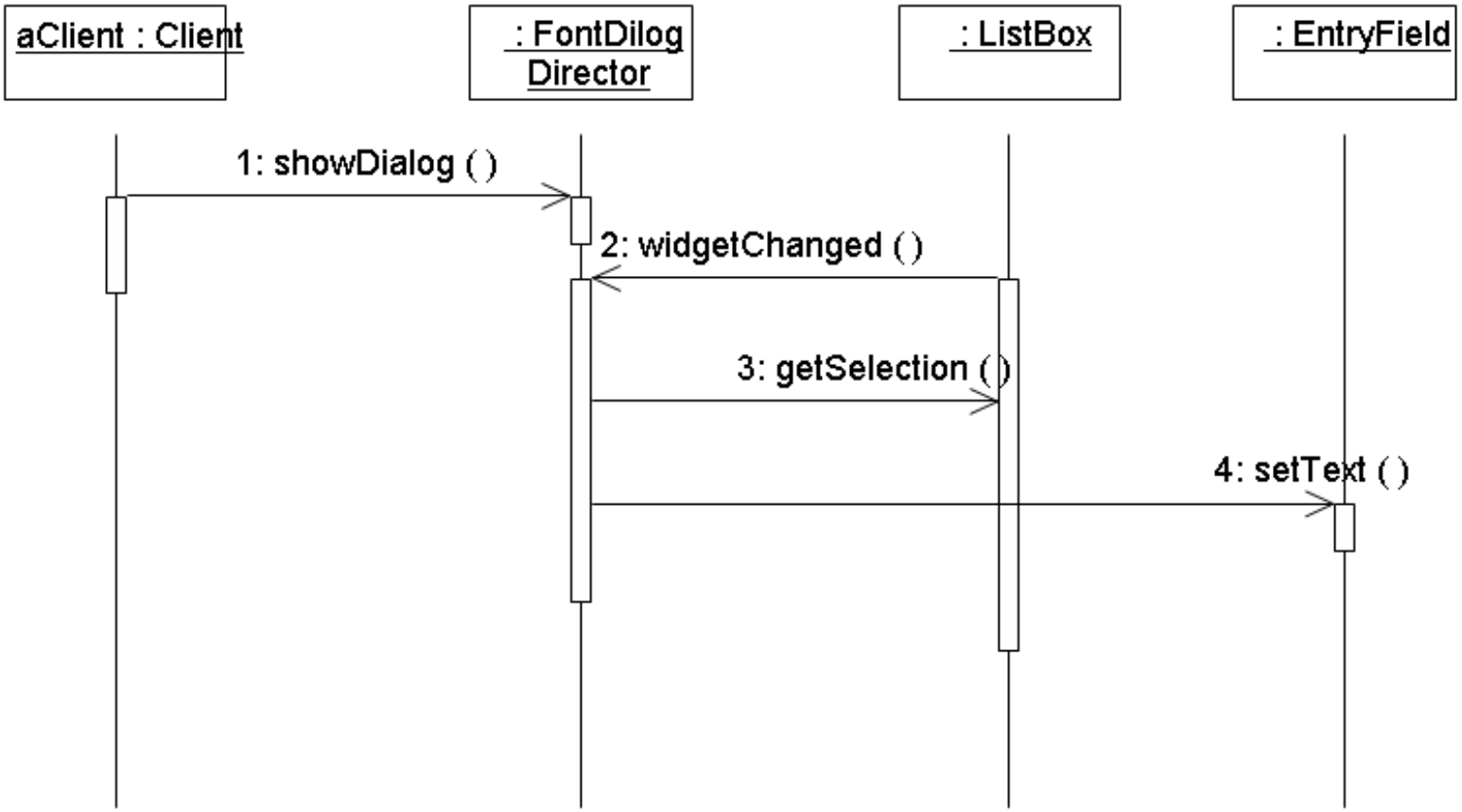
- empty text field disables a button.
- typing text selects an entry in the list-box.

These problems can be avoided by putting the collective behavior in the Mediator object.

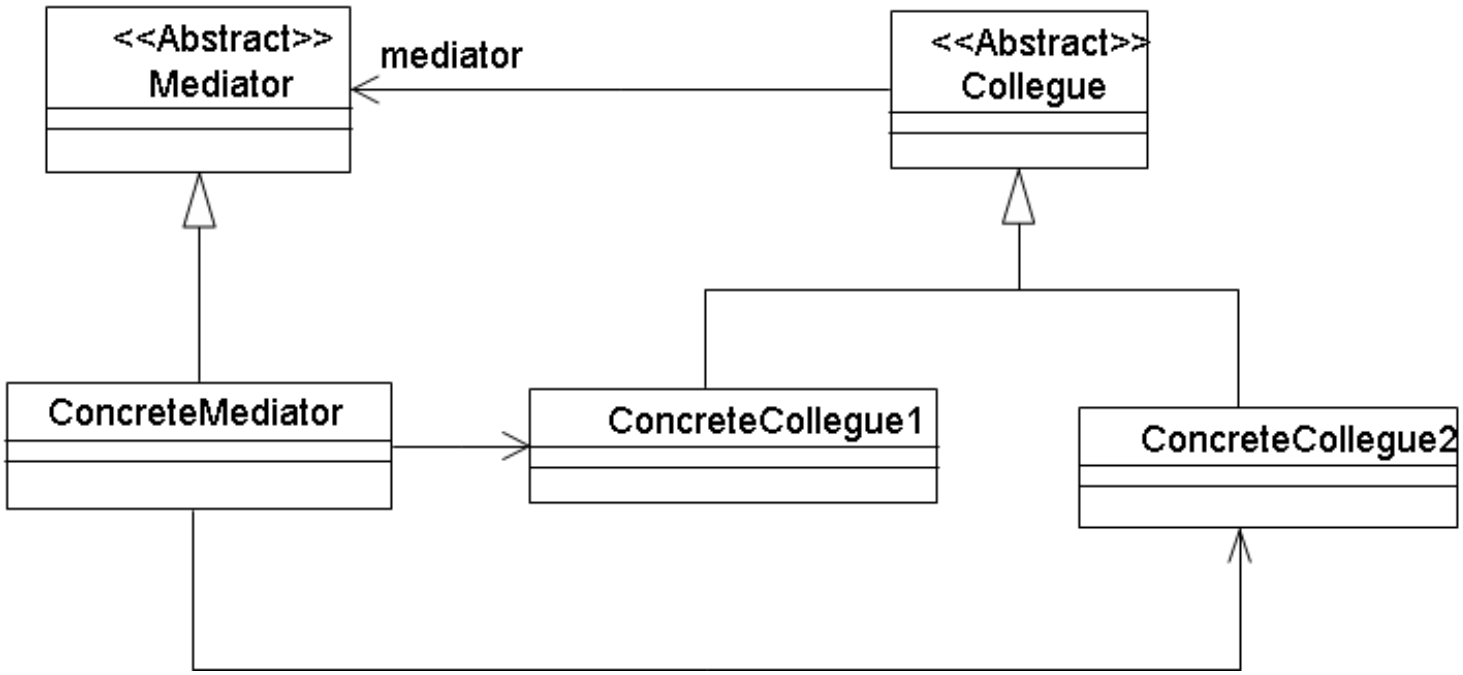
Mediator - Example



Mediator - Collaborations



Mediator - Structure



Mediator - Applicability

- To a set of objects that communicate in a well-defined but complex ways.
- To allow for greater reuse of an object, an object which refers to an communicates with many other objects.
- To easily customize the behavior which is distributed between several classes, without a lot of subclassing.

Mediator - Consequences

- ✓ It limits subclassing.
- ✓ It decouples collegues.
- ✓ It simplifies object protocol - replaces many-to-many interactions with one-to-many.
- ✓ It abstracts the interaction of objects.
- ✗ Mediator class can become a monolith which is hard to maintain.

Mediator - Related Patterns

Differences with Facade

- Facade abstracts a subsystem of objects to provide a more convenient interface.
- Its protocol is unidirectional.

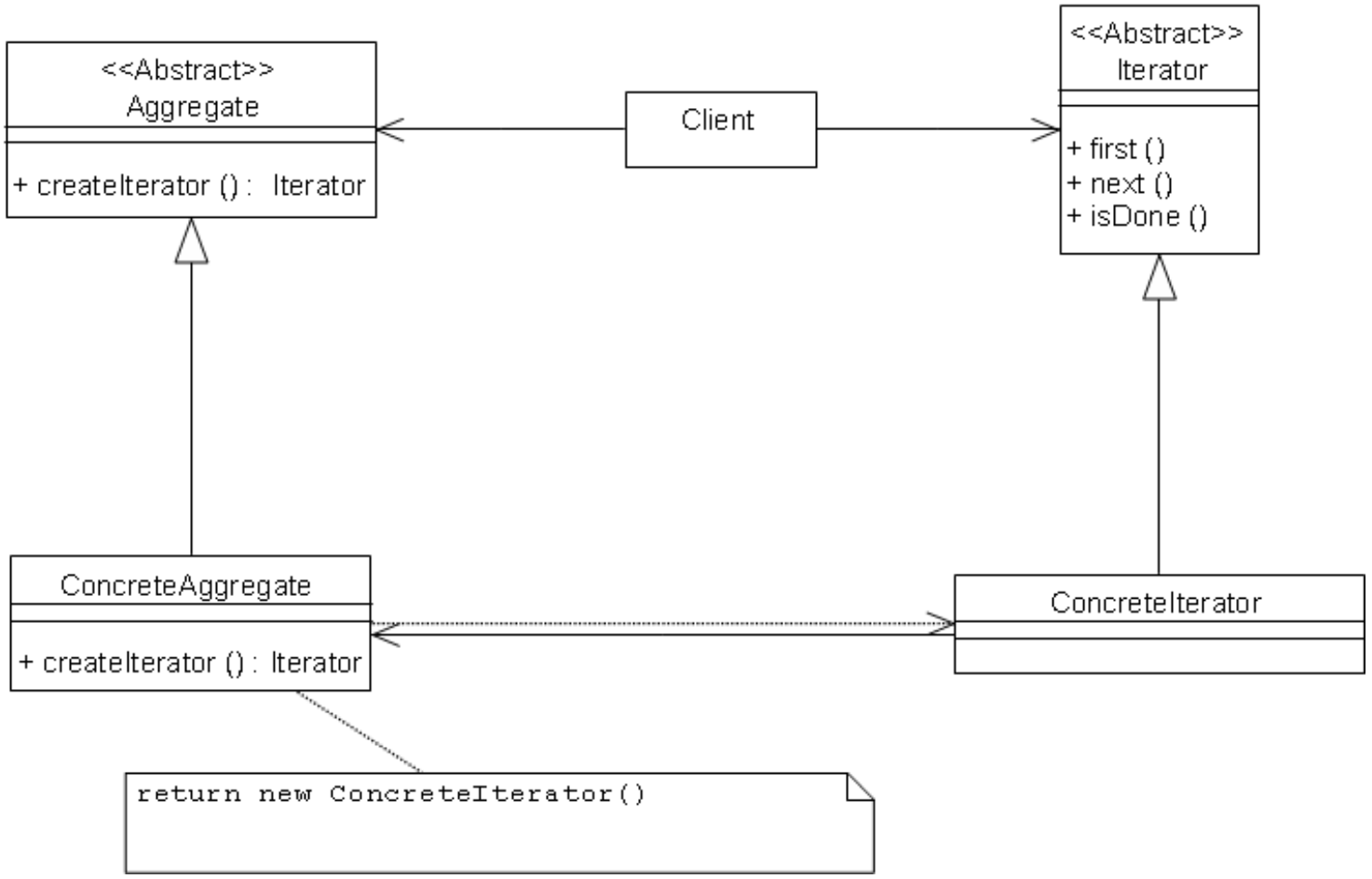
Iterator - Intent

Provide a way to access the elements of an aggregate object sequentially without exposing the underlying representation.

Iterator - Motivation

- You do not want to know the internals of a list.
- You want to traverse a list in different ways.
- You may have more than one traversal pending on the same list.
- You want to change the aggregate class without changing your client code.

Iterator - Structure



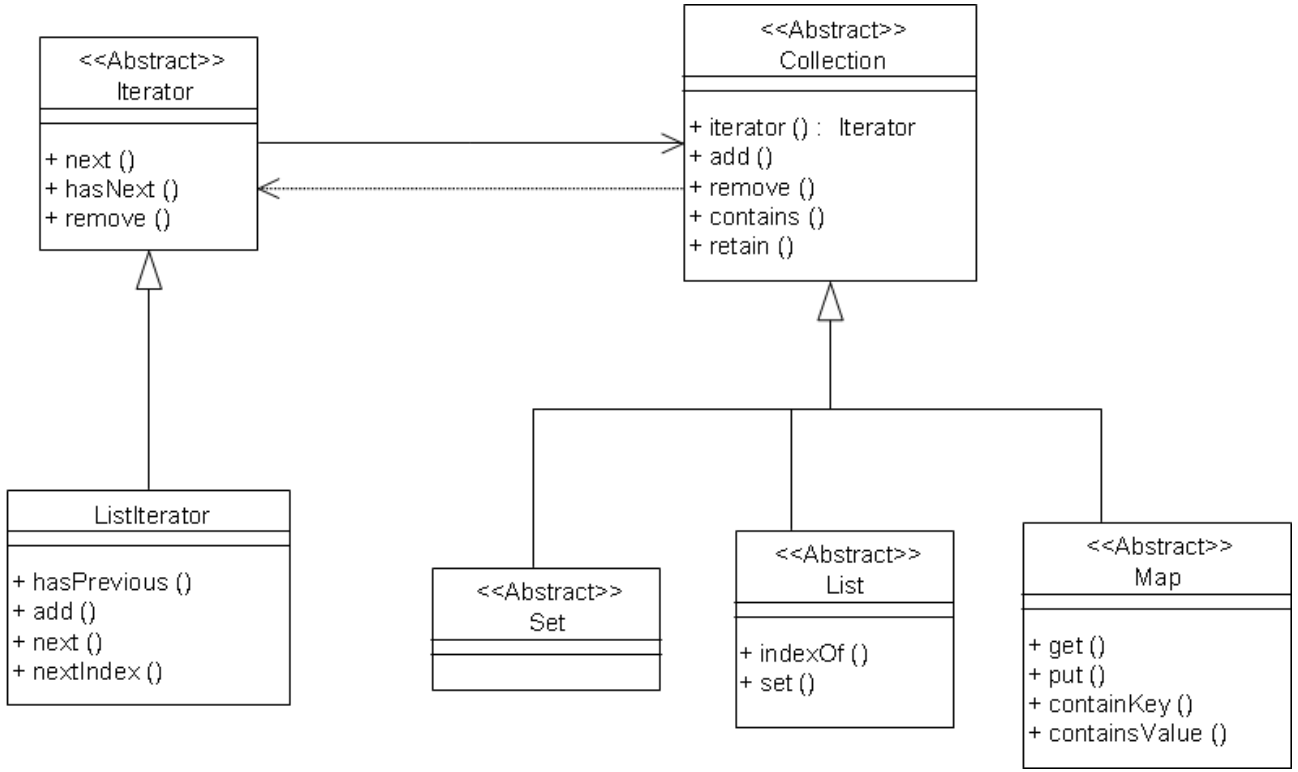
Iterator - Applicability

- To access an aggregate's contents without exposing its internal representation.
- To support multiple simultaneous traversals.
- To provide a uniform interface for traversing different aggregate structure.

Iterator - Consequences

- ✓ It supports variation in traversal algorithms.
- ✓ Iterators simplify the aggregate interface.
- ✓ There can be more than one traversal pending on the aggregate.

Iterator - Example (JDK 1.2)



Iterator - Example (JDK 1.2)

			IMPLEMENTATION		
		Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**
INTERFACES	**Set**	HashSet	ArraySet		
	List		ArrayList		Linked List
	Map	HashMap	ArrayMap	TreeMap	

Supports unmodifiable, immutable and synchronized collections.

Iterator - Implementation Issues

- External iterator vs. internal iterator
- Who defines the traversal algorithms? Iterator vs. aggregate.
- Implementing mutable, modifiable and synchronized iterators.
- NullIterator can be used for a composite.

Iterator - Related Patterns

- Iterators can be applied to recursive structure such as a Composite.
- Factory Method is used to create an iterator object.

Command - Intent

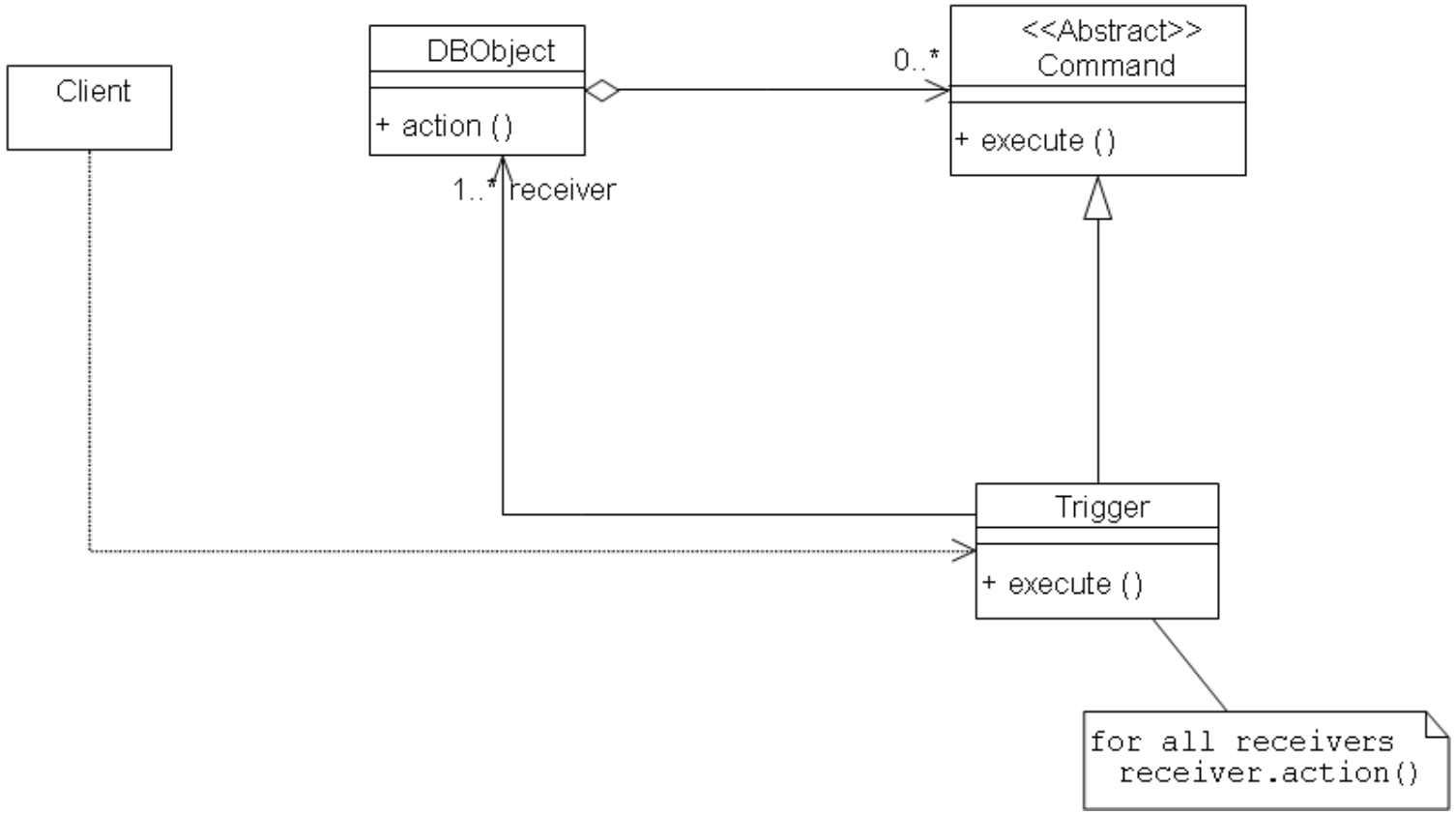
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, and support undoable operations.

Command - Motivation

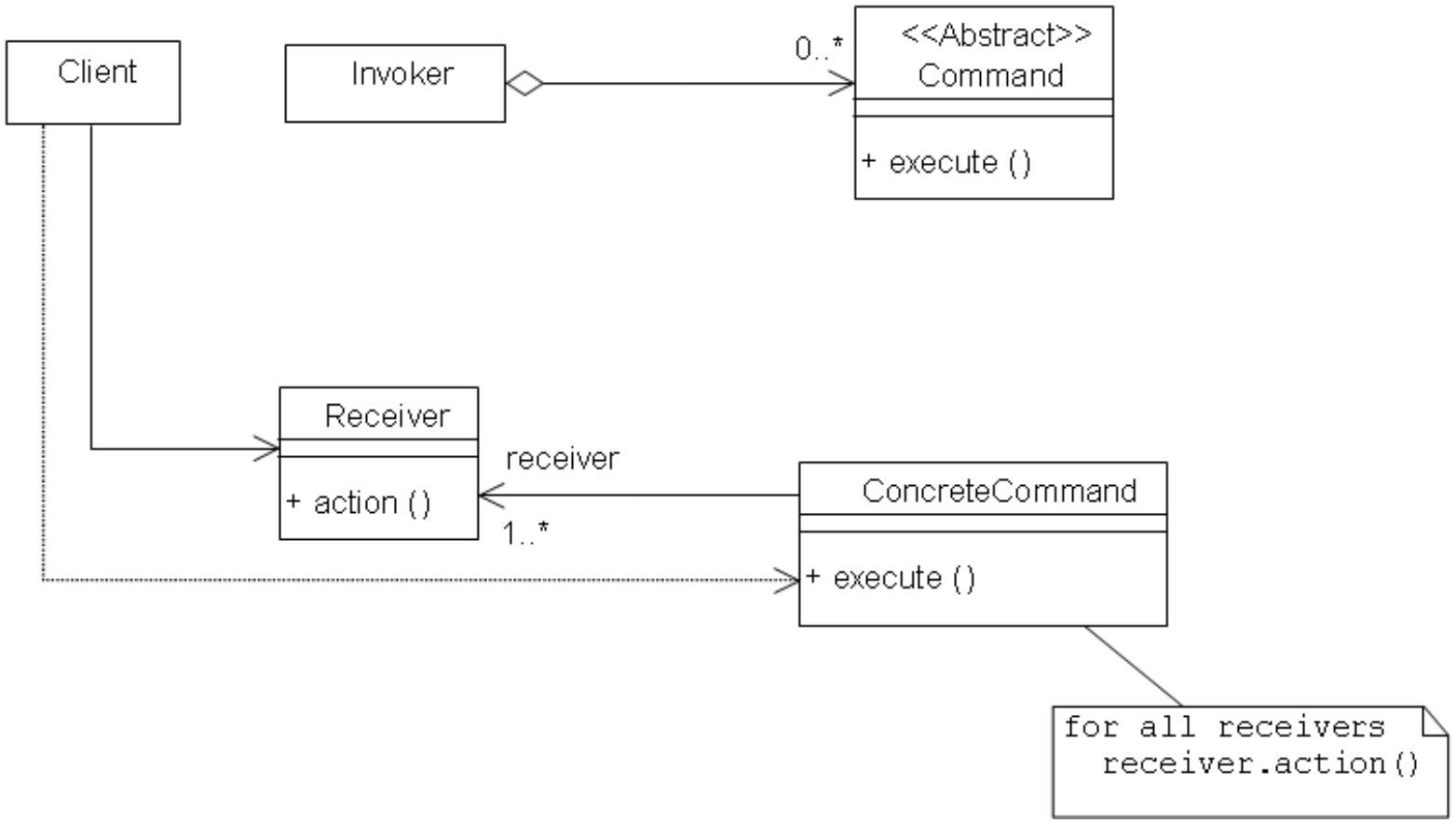
RDBMS systems provide triggers on the CRUD semantics. These triggers can be instances of Command object. A typical use-case would be :

- RDBMS will provide the abstract Command class.
- DBA's extend Command class to implement triggers.
- Users fire the triggers while using CRUD semantics.

Command - Example



Command - Structure



Command - Applicability

- To parameterize objects by an action to perform.
- To specify, queue and execute requests at different times.
- To support undo. Add unExecute() to Command.
- To support logging so that they can be reapplied at system crash.
- To model transactions.

Command - Consequences

- ✓ It decouples the object that invokes the operation from the one which knows how to perform it.
- ✓ Command can be extended and manipulated like any other object.
- ✓ Commands can be assembled to form macro-command.
- ✓ It is easy to add new commands without changing any existing classes.

Command - Related Patterns

- Composite can be used to implement MacroCommands.
- Prototype can be used to copy a command that has to be placed on the history list.
- Momento can be used to store the state of the Command object, which can be later used to undo.

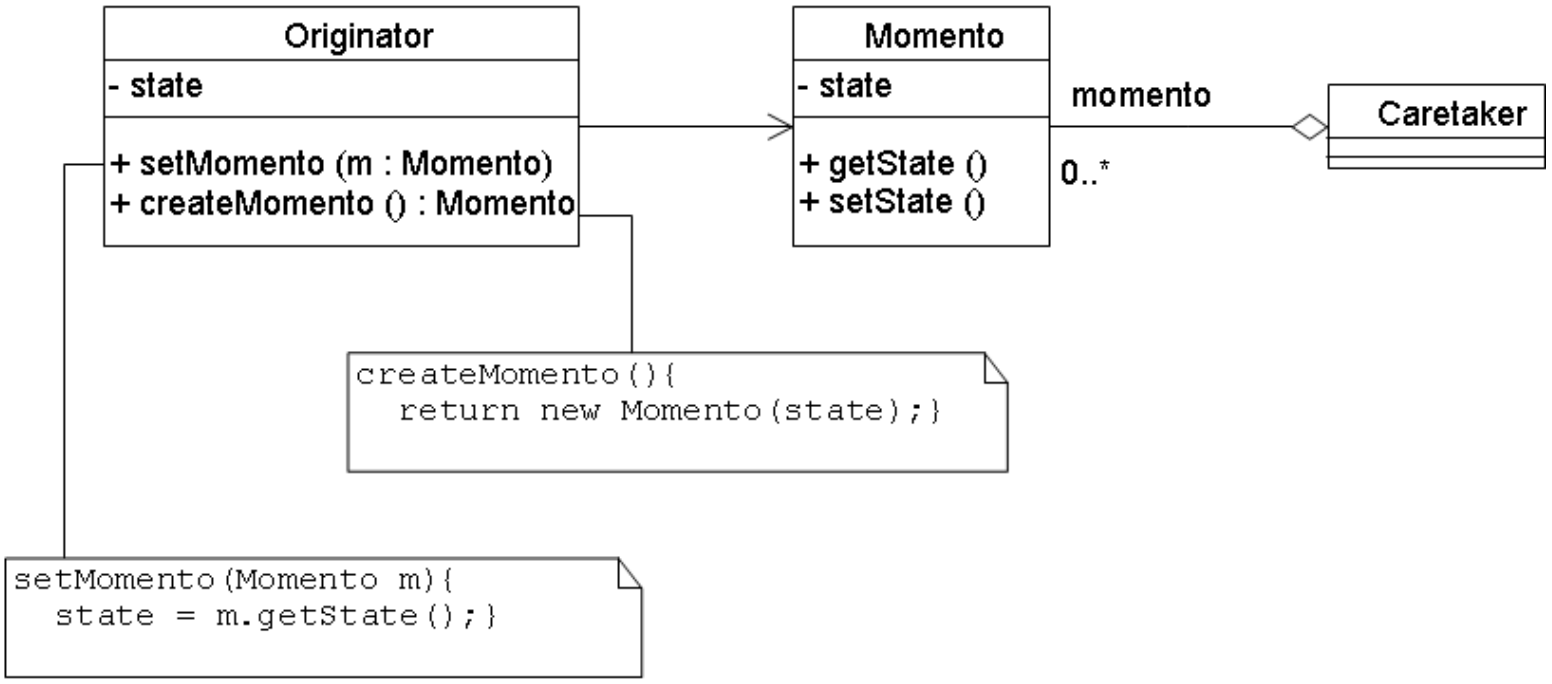
Momento - Intent

Capture and externalize an object's internal state, so that the object can be restored to its state later.

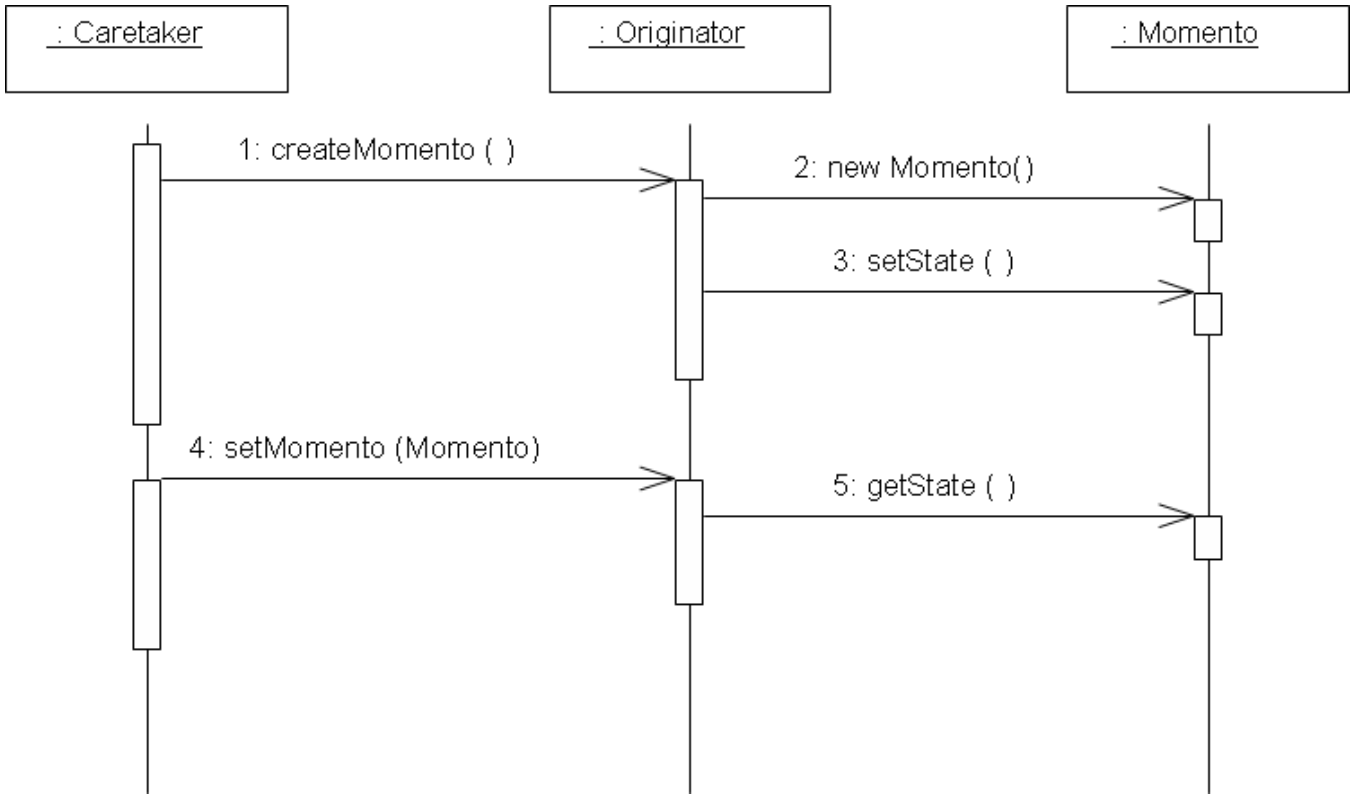
Momento - Motivation

You want to undo an executed command. This happens in a RDBMS system, when undo-redo logs are applied after the system terminates abnormally. So at every checkpoint, you want to store the state of the object as a momento, such that it can be later used for undo-redo logs.

Momento - Structure



Momento - Collaborations



Momento -Participants

- Momento stores the internal state of the Originator object.
- It provides two interfaces:
  - wide interface (with accessor and mutator) for the Originator
  - narrow interface for the Caretaker

Momento -Applicability

- To store the snapshot of an object, so that the it can be restored to the same state later; and a direct interface would expose implementation details and break object's encapsulation.

Momento - Consequences

- ✓ It preserves encapsulation boundary.
- ✓ It simplifies the Originator.
- ✗ It might be expensive to store Momento. One solution is to do incremental storage, rather than storing the full state of an object.
- ✗ It may not be possible to support two different interfaces in some implementations.
- ✗ It can incur heavy cost in caring for the Momento.

Behavioral Patterns - II

Overview

- Observer
- State
- Strategy
- Template Method
- Visitor
  - Acyclic Visitor
  - Default Visitor

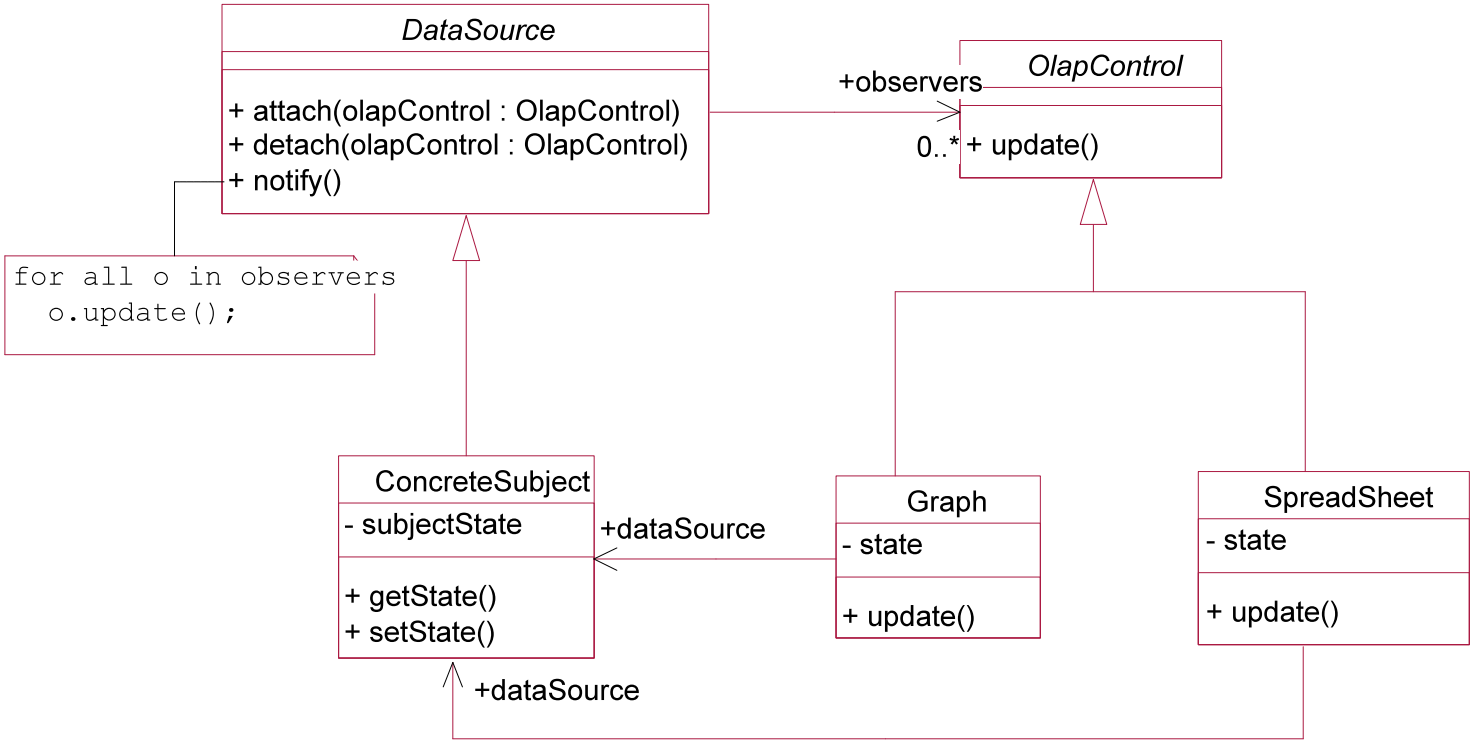
Observer - Intent

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

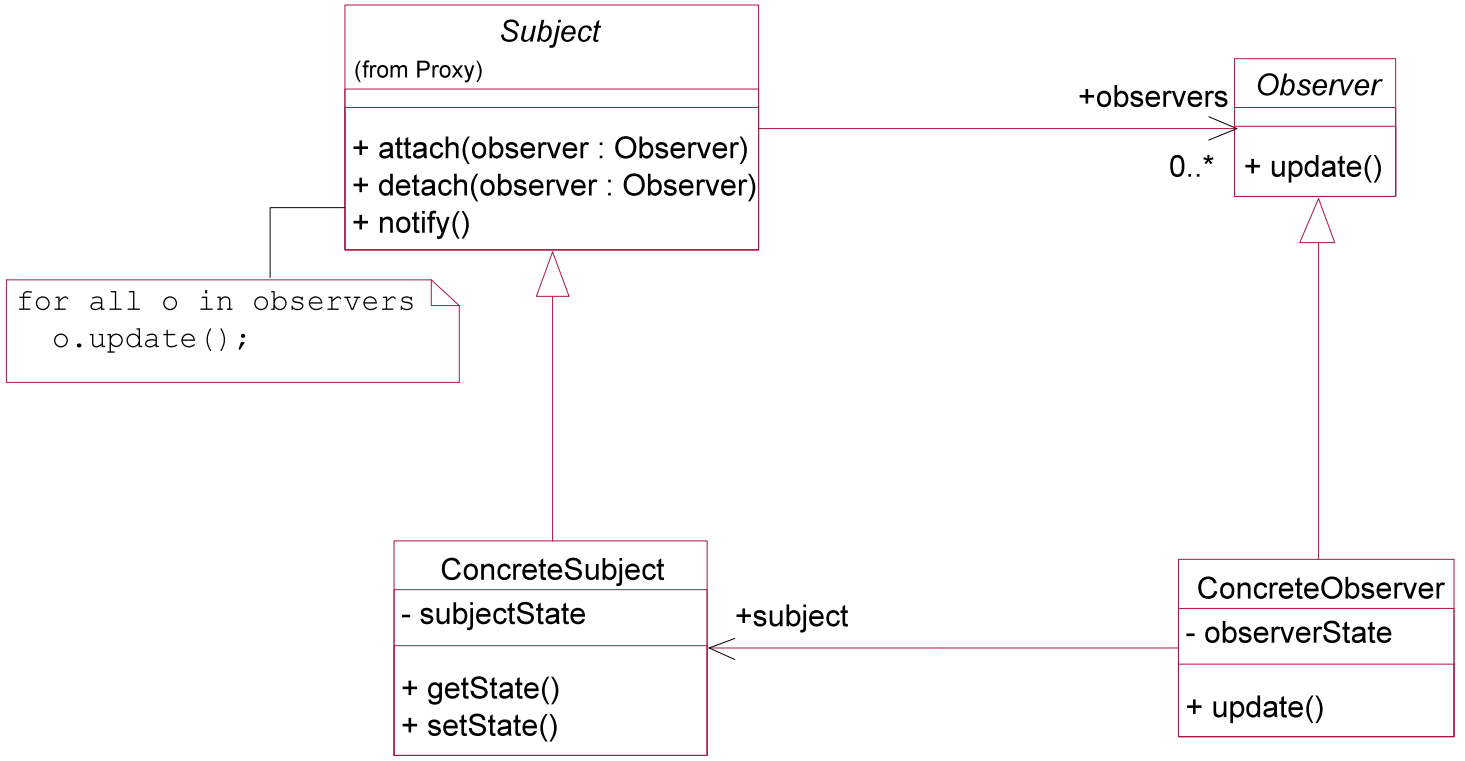
Observer - Motivation

Suppose you have a desktop OLAP (on-line analytical processing) application. A window on the application is showing a graph and a table of sales by product, geography and time. You enter the latest figures for sales of the month into the table. These changes should also be reflected in the graph. Observer pattern describes how to establish these relationships.

Observer - Example



Observer - Structure



Observer - Applicability

- When an abstraction has two aspects, the view (observer) aspect, and the document (subject) aspect.
- To notify other objects (observers) of the change
  - number of observers are not known.
  - subject and observer are to be loosely coupled.

Observers - Consequences

- ✓ Abstract coupling between subject and observer
- ✓ Support for broadcast communication.
- ✗ Unexpected updates: as observers have no knowledge of each other.
- ✗ Simple update protocols provide no detail on what changed in the subject.

Observer - Implementation Issues

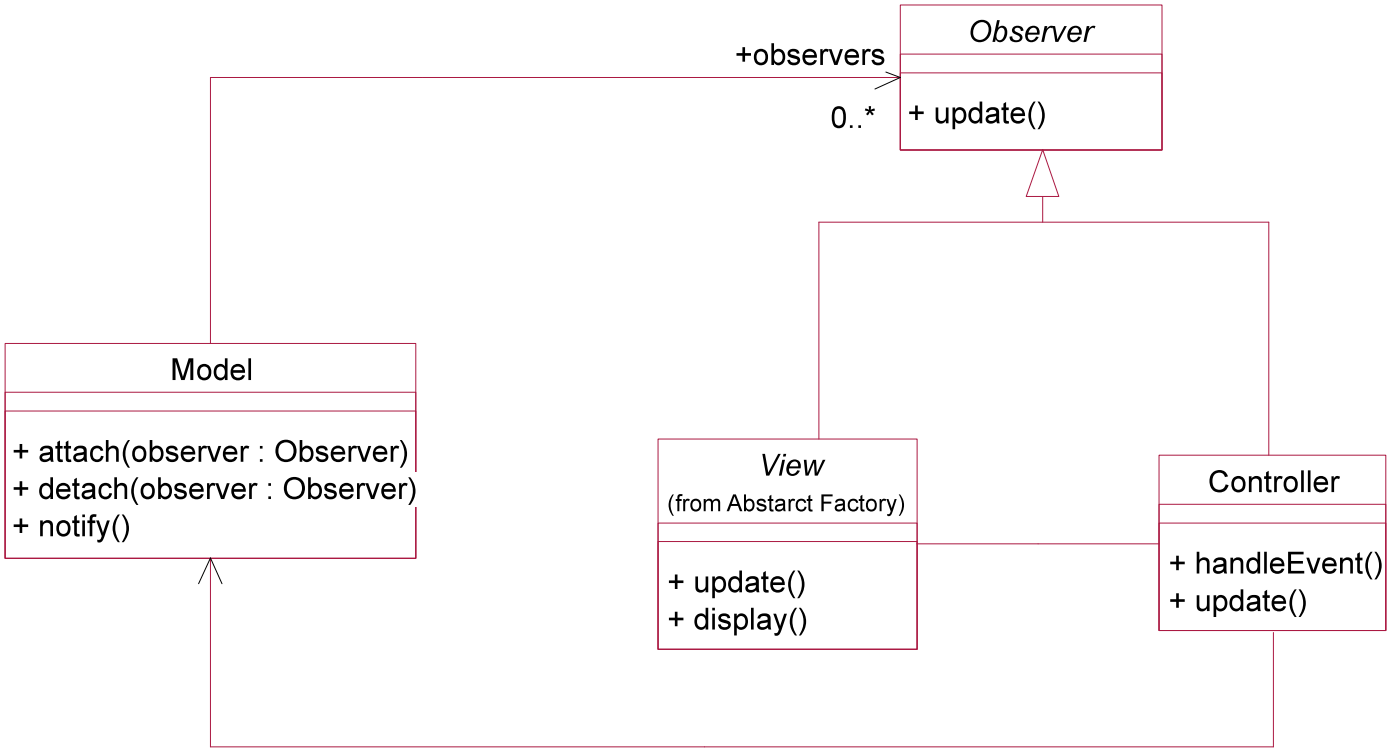
- Observing more than one subject: extend update() method to take the Subject as a parameter.
- Make sure that the Subject's state is self-consistent before calling notify().
- Push vs. the pull model for broadcasting other changes.
- Who triggers the update()?
  - state-setting on Subject.
  - client calls notify().

Observer - Implementation Issues

- Specifying modifications of interest explicitly.
- Encapsulate complex update semantics: Create a Mediator called ChangeManager.
  - it maps subject to its observers.
  - it defines particular update strategy.
  - it updates all dependent observers at the request of a subject.

Observer - Variant

MVC (Model View Controller)



MVC (Model View Controller)

- Model maps to Subject; View maps to Observer.
- Controller is new. It has the following collaborations and responsibilities:
  - every View has a Controller.
  - the user interacts through the Controller.
  - it accepts user input as events.
  - it translates an event for the Model or a display request for the View.
  - it implements the update(), if necessary.

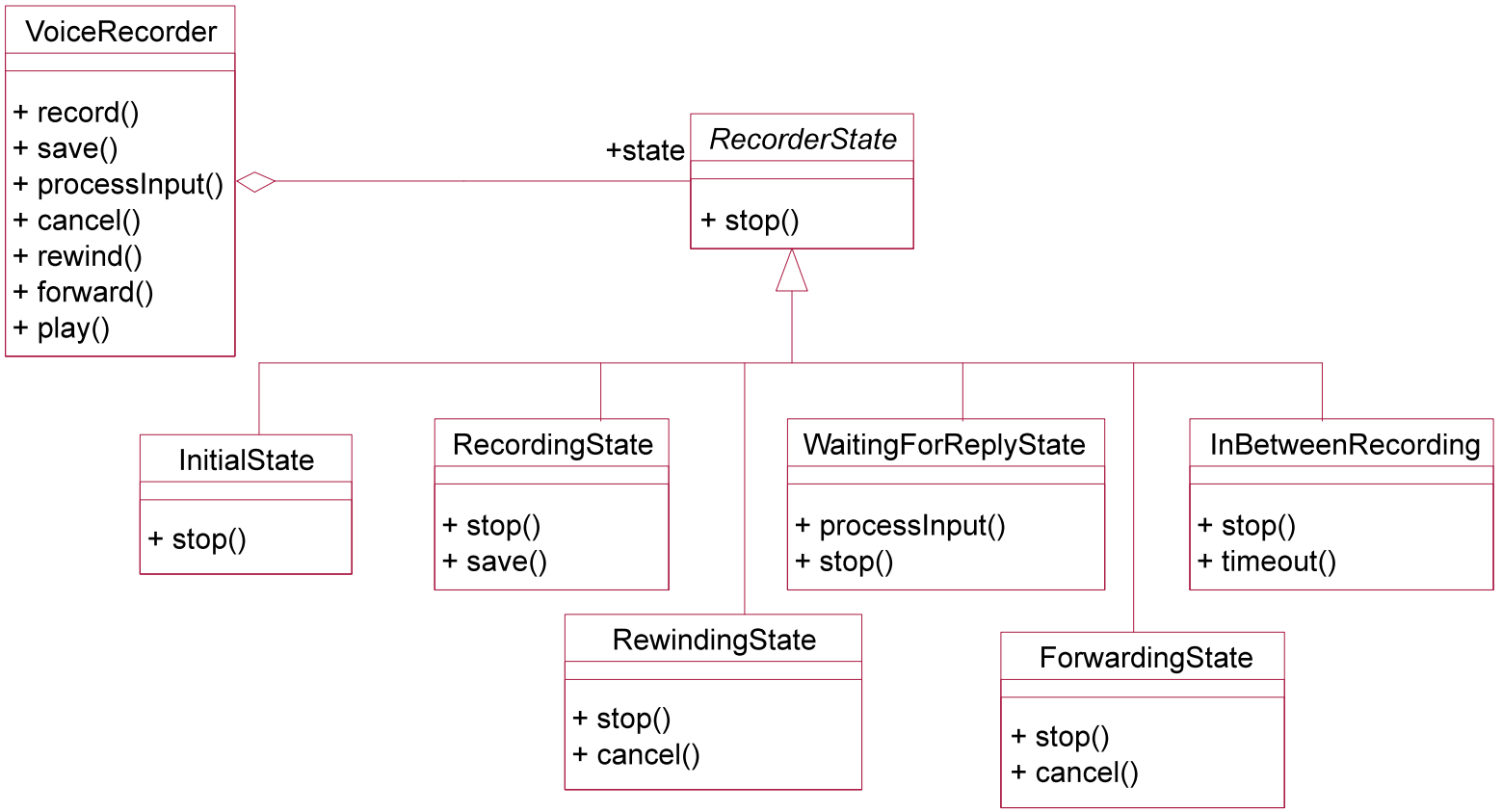
State - Intent

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

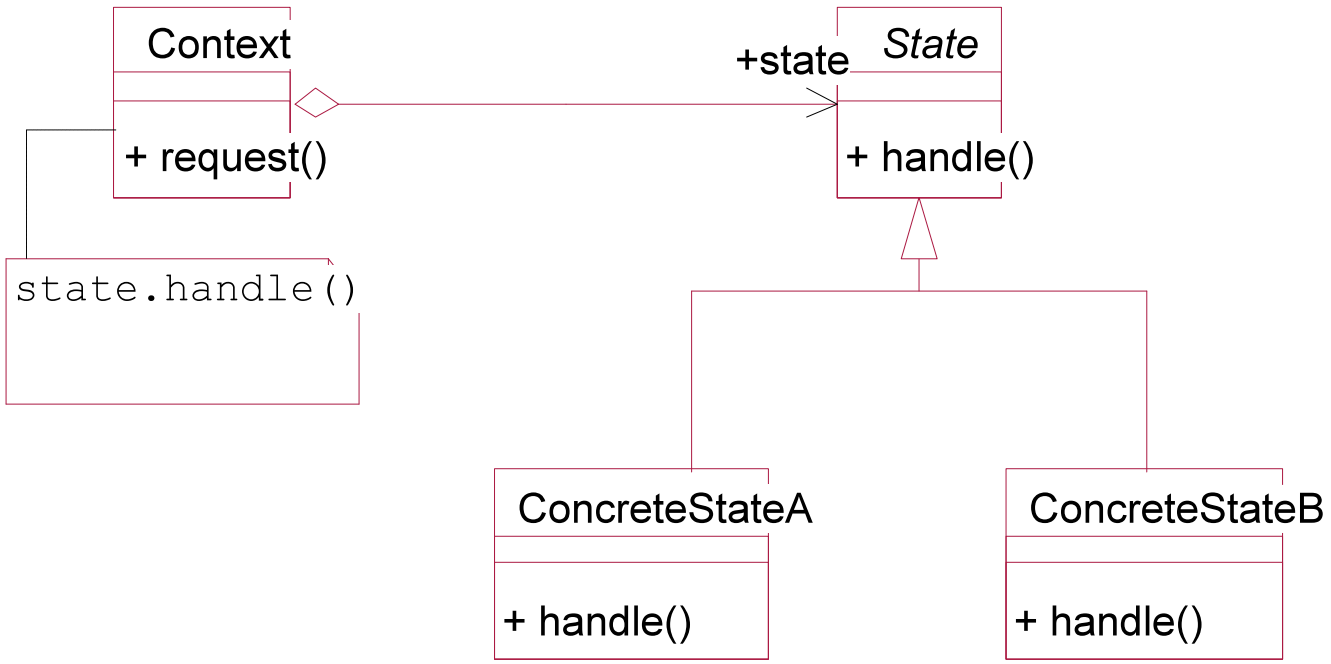
State - Motivation

Consider a voice recorder. It responds differently depending on its current state (recording, rewinding etc.). In the State pattern, this difference in behavior is captured as separate classes.

State - Example



State - Structure



State - Applicability

- When an object behavior depends on its state, and it must change its behavior at runtime depending on the state.
- To avoid large switch statements for state-specific behavior

State - Consequences

- ✓ It localizes state-specific behavior and partitions behavior for several states.
- ✓ It makes state-transition explicit and more atomic.
- ✓ State objects can be shared as Flyweights.

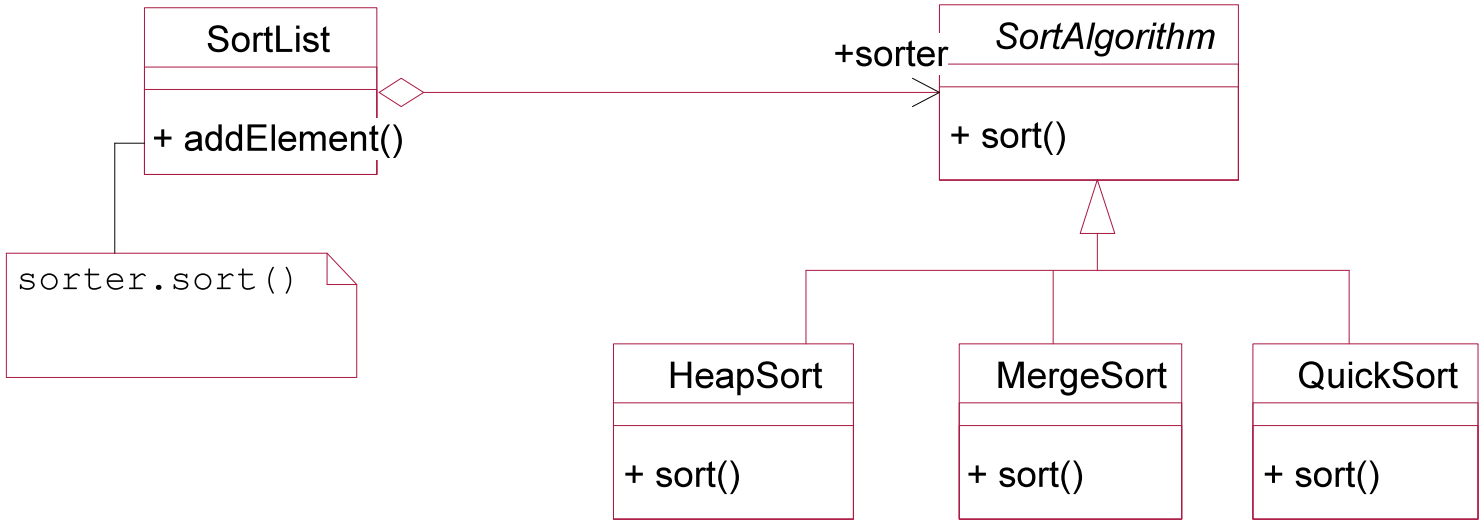
State - Related Patterns

- The Flyweight pattern can be used to share state objects.
- State objects are often Singletons.

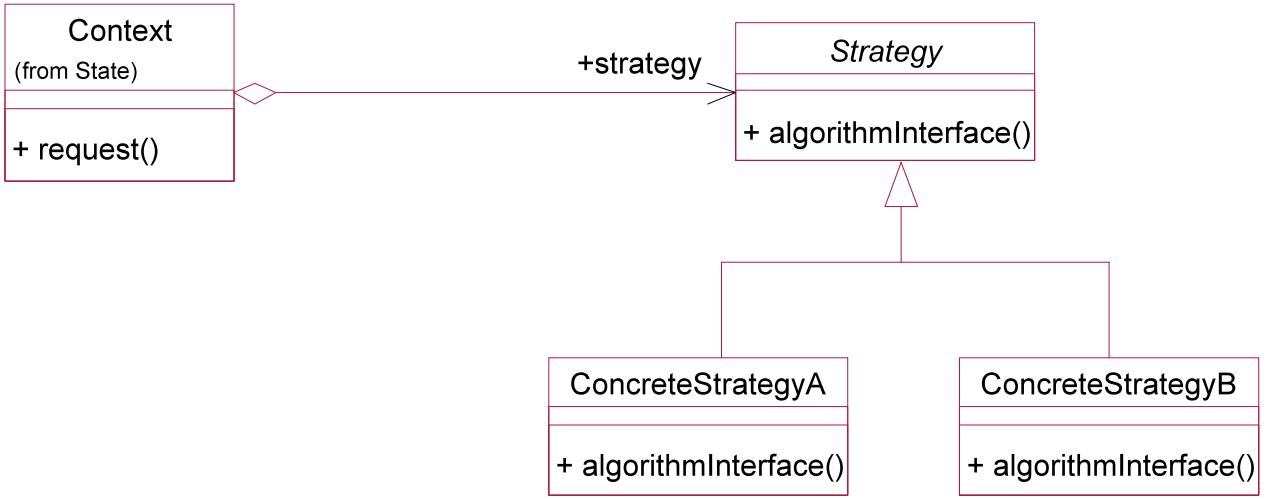
Strategy - Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

Strategy - Example



Strategy - Structure



Strategy - Applicability

- When many related classes differ only in their behavior. Strategies provide a way to configure a class with one of the many behaviors.
- When you need different variants of an algorithm.
- To avoid exposing complex, algorithm-specific data structures.
- To move partitions of a conditional statement into its classes

Strategy - Consequences

- ✓ Hierarchies of Strategy classes define a family of algorithms or behaviors for Context to reuse.
- ✓ An alternate to subclassing. Encapsulating the behavior in separate Strategy class lets you vary the algorithm independently of its Context, making it easier to switch, understand and extend.
- ✓ Eliminates conditional statements.

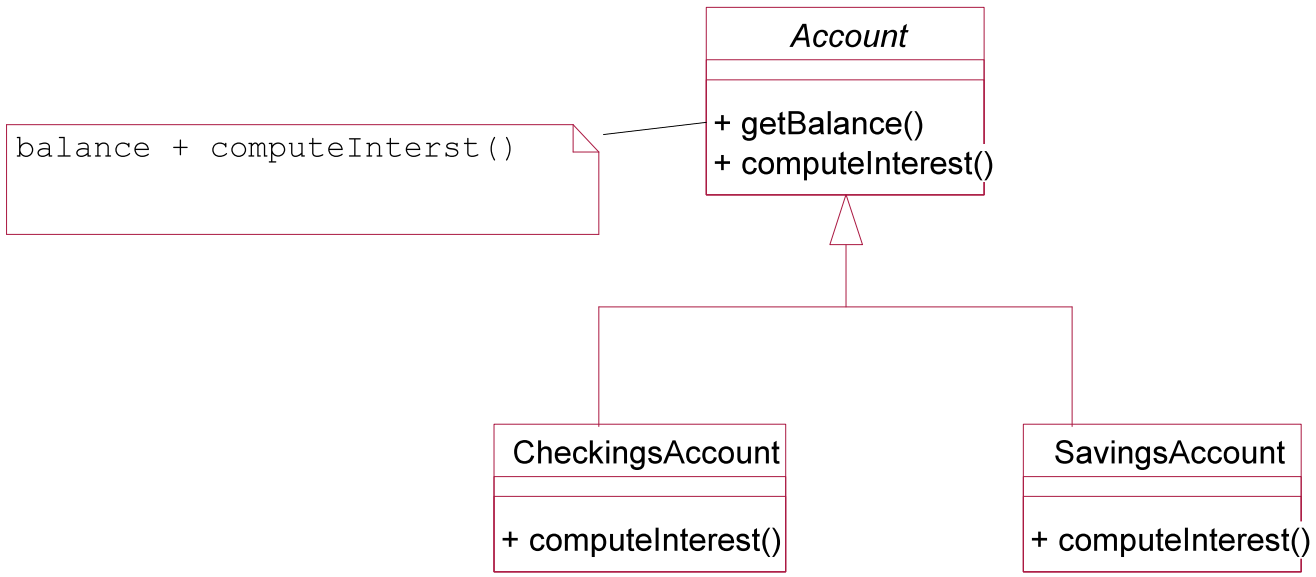
Strategy - Consequences

- Clients must be aware of different Strategies.
- Communication overhead between Strategy and Context.
- Increased number of objects. Flyweight can be used to share Strategies.

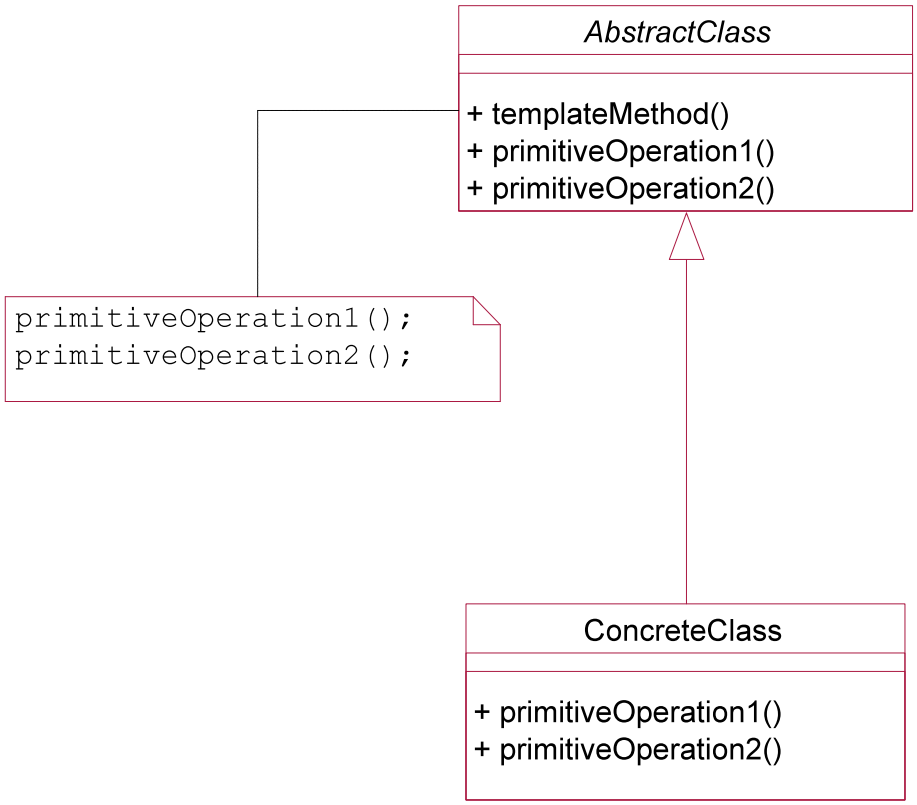
Template Method - Intent

Define a 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.

Template Method - Example



Template Method - Structure



Template Method - Applicability

- To implement the invariant part of an algorithm.
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
- To control subclass extension.

Template Method - Consequences

- ✓ It is a fundamental technique for code reuse.
- ✓ It lets the subclasses refine the (default) behavior of an operation.

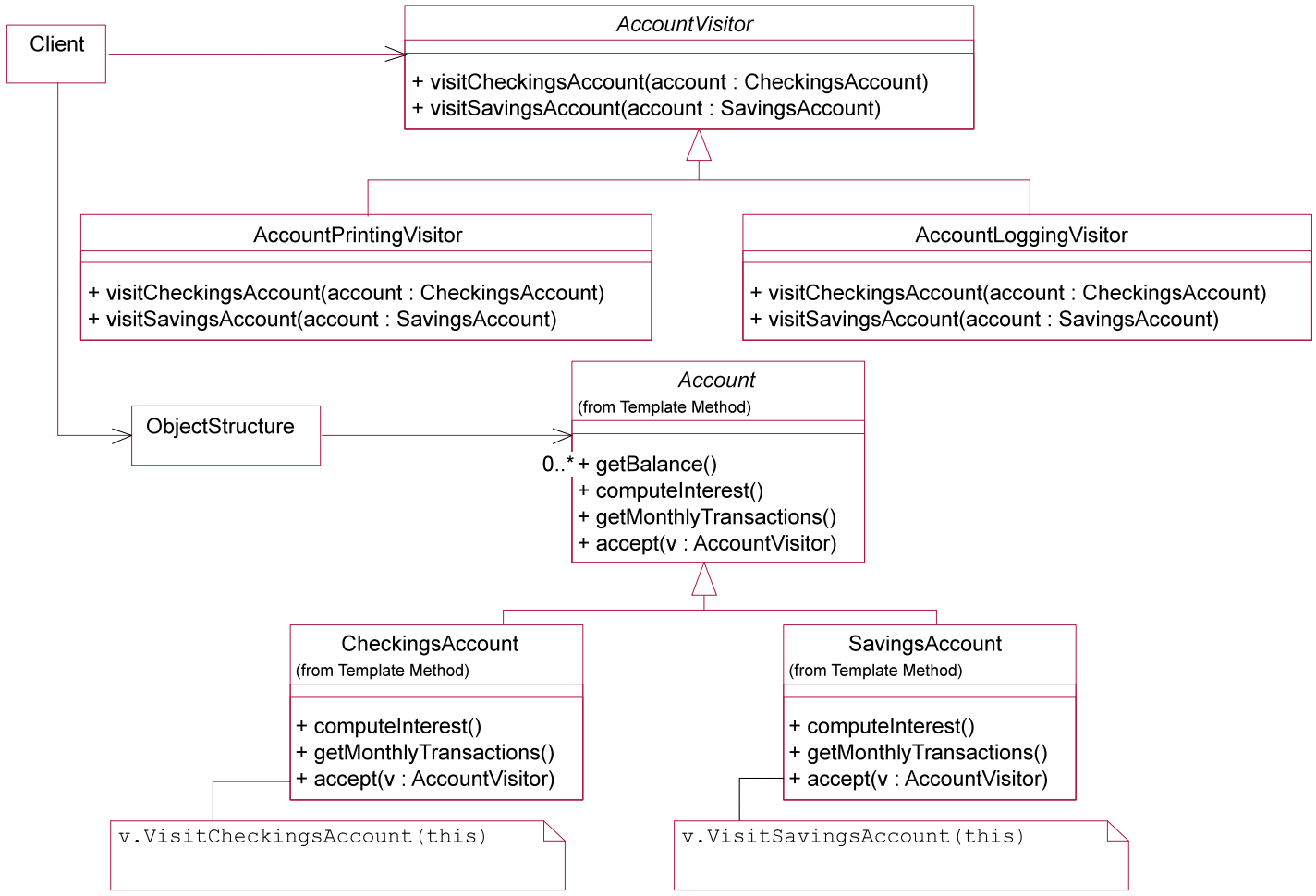
Template Method - Related Patterns

- Factory Methods are often called by Template Methods.
- Strategy uses delegation to vary the entire algorithm, while Template Method uses inheritance to vary part of an algorithm.
- Builder lets ConcreteBuilders vary the part of a construction, while Template Method lets its subclasses vry part of an operation.

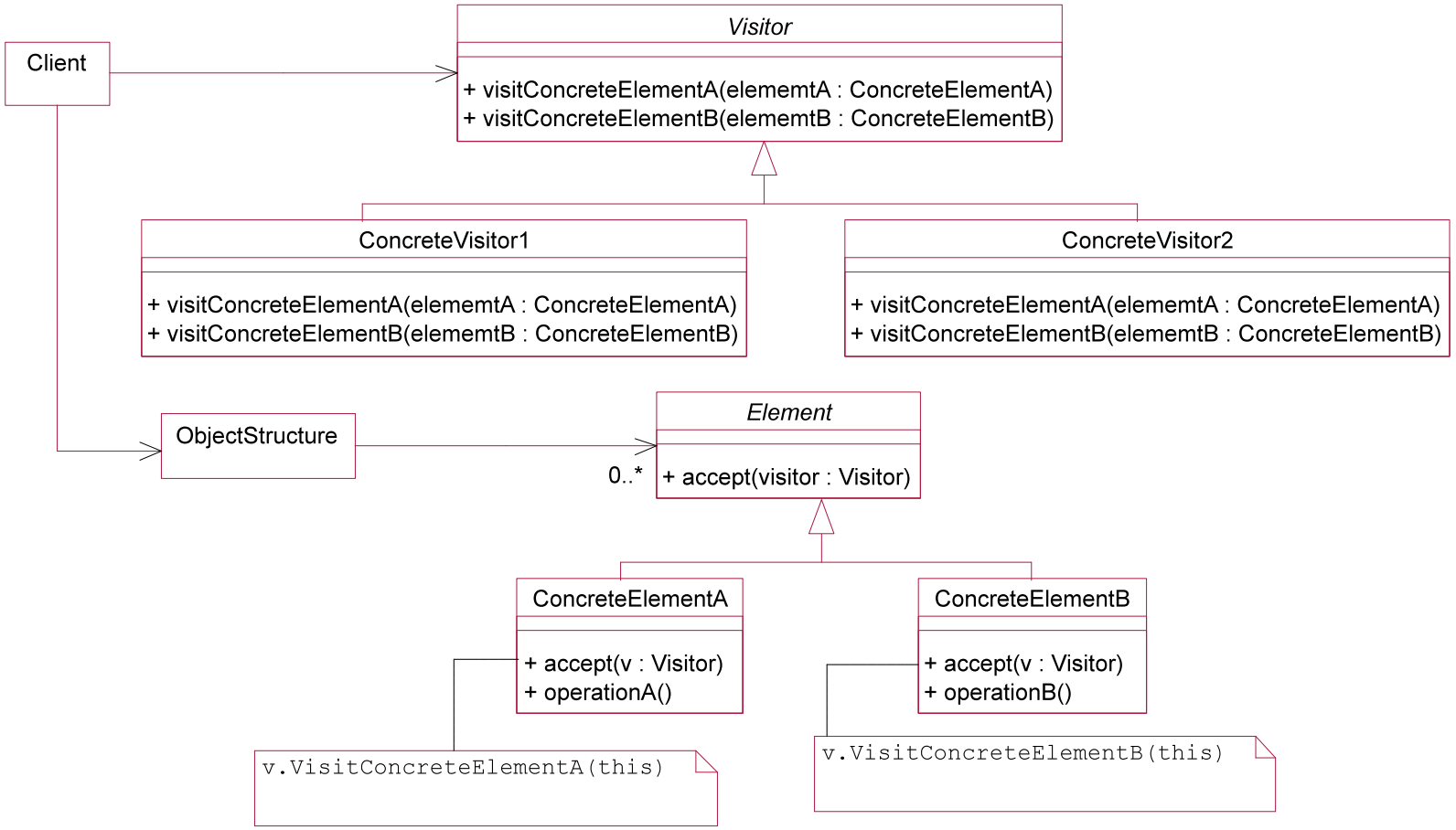
Visitor - Intent

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.

Visitor - Example



Visitor - Structure



Visitor - Applicability

- When an object structure contains many classes of objects with different interfaces, and you want to perform operations on objects that depend on their concrete classes.
- To avoid cluttering of classes. Visitor lets you keep related operations together.
- When the classes defining the object structure rarely change.

Visitor - Consequences

- ✓ It makes it easy to add new operations to an object structure.
- ✓ It gathers related operations and separates unrelated ones.
- ✓ Visitor can visit objects that do not have a common parent class.
- ✓ It can accumulate state?

Visitor - Consequences

- It makes it hard to add new classes in the object structure.
- It breaks encapsulation, by forcing you to provide public operations that access an element's internal state.

Visitor - Dependency Cycle Problem

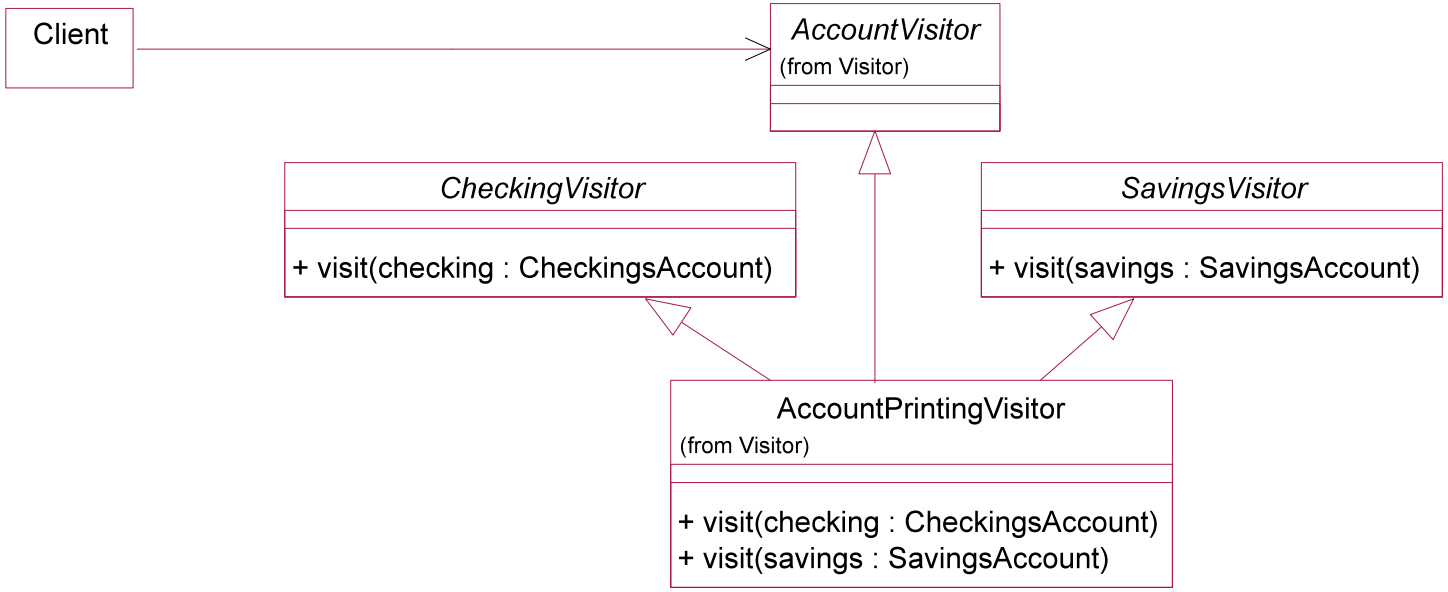
- The base class of the Element hierarchy depends on the base class of the corresponding Visitor hierarchy.
- The Visitor base class has methods for each of the derivatives of the Element base class.
- Each derivative of Element depends on the base Element.

Thus there is a cycle of dependencies that causes elements to transitively depend upon all its derivatives. So, a change in derivative hierarchy asks for massive recompilation.

Acyclic Visitor - Intent

It is to allow new functions to be added to existing class hierarchies without affecting those hierarchies, and without creating the troublesome dependency cycles inherent in GoF Visitor.

Acyclic Visitor - Example




```
void CheckingsAccount::accept(AccountVisitor& visitor){
  CheckingVisitor* checkingVisitor = dynamic_cast<CheckingsVisitor *> (&visitor);
  if (checkingVisitor)
    checkingVisitor->visit(*this);
  else
    // visitor cannot visit CheckingVisitor
```

Acyclic Visitor - Example

- There are no dependency cycles anywhere in the structure.
- New Account derivatives have no effect on existing Account Visitors unless those Visitors must implement their operations for those derivatives.

Acyclic Visitor - Example

- New Account derivatives can be added at anytime without affecting the users of Account, the derivatives of Account, or the users of the derivatives of Account.
- The need for massive recompilation is completely eliminated.

Acyclic Visitor - Applicability

- When the Visited class hierarchy can be frequently extended with new derivatives of the Element class.
- When the recompilation, relinking, retesting and redistribution of derivatives of Element is very expensive

Acyclic Visitor - Consequences

- ✓ Elimination of dependency cycle.
- ✓ Derived elements of an object structure do not depend on each other.
- ✓ Recompilation is minimized.
- ✓ Partial visitation is natural and does not require additional code overhead.

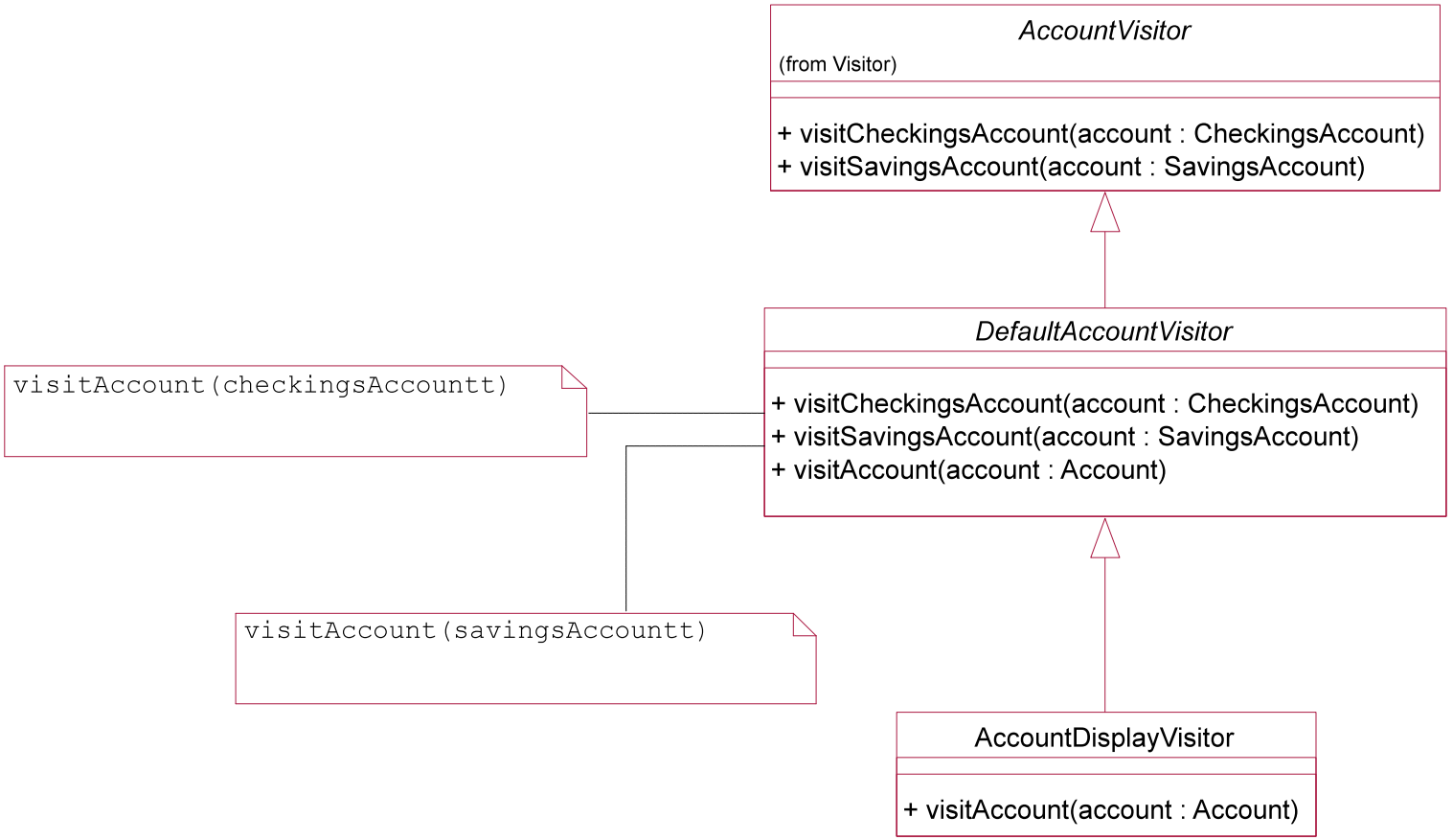
Acyclic Visitor - Consequences

- Dynamic cast can be expensive.
- Multiple inheritance is necessary.
- There will be an abstract Visitor for every derived class of Element which needs to be visited.

Default Visitor - Intent

It is to allow new functions to be added to existing class hierarchies without affecting those hierarchies, and without creating the troublesome dependency cycles inherent in GoF Visitor.

Default Visitor - Example



Default Visitor - Applicability

- When the elements to be visited come from a small set of polymorphic class hierarchies.
- When several ConcreteVisitors can employ default handlers for a small set of abstract elements.

Default Visitor - Consequences

- ✓ It makes it easier to add new ConcreteElement classes.
- ✓ It provides a base method.
- × It is difficult to change the abstract classes of the Element class hierarchy.
- × It requires more work in maintaining the DefaultVisitor class in return for less work in maintaining each ConcreteVisitor classes.