Core Design Patterns

Outline

- Introduction to design patterns
- Creational patterns (constructing objects)
- Structural patterns (controlling heap layout)
- Behavioral patterns (affecting object semantics)

Design Pattern

Design patterns describe how objects communicate without becoming entangled in each other’s data models and methods.

Some Useful Definitions

- A standard solution to a common programming problem
- A technique for making code more flexible by making it meet certain criteria
- A design or implementation structure that achieves a particular purpose
- A high-level programming idiom
- Shorthand for describing certain aspects of program organization
- Connections among program components
- The shape of a heap snapshot or object model

Pattern

- A Pattern is a **solution** to a recurring **problem** in a **context**
- **Context:**is environment, surroundings,situation or etc
- **Problem:** unsettled question, something that needs to be investigated or solved
- **Solution:** answer to the problem

Identify the problem

- It is a similar problem
- Identify the characteristics and document it
- Consider the initial documents to be a candidate patterns.
- Then form a compete pattern
- **Rule of three:**pattern should appear in atleast 3 different systems.

Patterns Vs Strategies

- Each pattern includes various strategies that provide lower level implementation details.
- Pattern exist at higher level of abstraction than strategies.
- Patters include most recommended or most common implementations strategies

Simple template of a pattern

- **Pattern name :**
- **Problem :**
  - Describes the design issue faced by the developer
- **Forces :**
  - What are the conditions which forced the pattern to occur
- **Solution :**
  - describes the solution in two forms *Structure* *& strategies*
- **Structure** :
  - uses UML diagrams to represent the problem
- **Strategies**:
  - how the pattern may be implemented

Simple template of a pattern (2)

- **Consequences:**
  - results of using the patterns notes pros and cons
- **Sample Code:**
- **Limitations:**
- **Related Patterns:**
  - other related patterns which use this pattern

How many design patterns?

- Many
- A site says at least 250 existing patterns are used in OO world, including **Spaghetti** which refers to poor coding habits
- The 23 design patterns by **GOF** are well known, and more are to be discovered on the way
- Note that the design patterns are not idioms or algorithms or components.

What is the relationship among these patterns?

- Generally, to build a system, you may need many patterns to fit together
- Different designer may use different patterns to solve the same problem, Usually:
  - Some patterns naturally fit together
  - One pattern may lead to another
  - Some patterns are similar and alternative
  - Patterns are discoverable and documentable
  - Patterns are not methods or framework
  - Patterns give you hint to solve a problem effectively

Types of Patterns

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Creational Patterns

- Creational patterns deal with the best way to create instances of objects
- **The Factory Method**
  - provides a simple decision making class that returns one of several possible subclasses of an abstract base class depending on the data that are provided.
- **The Abstract Factory Method**
  - provides an interface to create and return one of several families of related objects.
- **The Builder Pattern**
  - separates the construction of a complex object from its representation, so that several different representations can be created depending on the needs of the program.
- **The Prototype Pattern**
  - starts with an initialized and instantiated class and copies or clones it to make new instances rather than creating new instances.
- **The Singleton Pattern**
  - is a class of which there can be no more than one instance. It provides a single global point of access to that instance

Structural Patterns

- Structural patterns describe how classes and objects can be combined to form larger structures
- The difference between *class* patterns and *object* patterns is that class patterns describe how inheritance can be used to provide more useful program interfaces
- Object patterns, on the other hand, describe how objects can be composed into larger structures using object composition, or the inclusion of objects within other objects

Structural Patterns

- The **Adapter** pattern, used to change the interface of one class to that of another one.
- The **Bridge** pattern, intended to keep the interface to your client program constant while allowing you to change the actual kind of class you display or use. You can then change the interface and the underlying class separately.
- The **Composite** pattern, a collection of objects, any one of which may be either itself a Composite, or just a primitive object.

Structural Patterns

- The **Decorator** pattern, a class that surrounds a given class, adds new capabilities to it, and passes all the unchanged methods to the underlying class.
- The **Façade** pattern, which groups a complex object hierarchy and provides a new, simpler interface to access those data.
- The **Flyweight** pattern, which provides a way to limit the proliferation of small, similar class instances by moving some of the class data outside the class and passing it in during various execution methods.
- The **Proxy** pattern, which provides a simple place-holder class for a more complex class which is expensive to instantiate

Behavioral Patterns

- specifically concerned with communication between objects
- **The Observer pattern**
  - defines the way a number of classes can be notified of a change,
- **The Mediator**
  - defines how communication between classes can be simplified by using another class to keep all classes from having to know about each other

Behavioral Patterns

- **The Chain of Responsibility**
  - allows an even further decoupling between classes, by passing a request between classes until it is recognized
- **The Template pattern**
  - provides an abstract definition of an algorithm
- **The Interpreter Pattern**
  - provides a definition of how to include language elements in a program.
- **The Strategy pattern**
  - encapsulates an algorithm inside a class,

Behavioral Patterns

- **The Visitor pattern**
  - adds function to a class,
- **The State pattern**
  - provides a memory for a class’s instance variables.
- **The Command pattern**
  - provides a simple way to separate execution of a command from the interface environment that produced it, and
- **The Iterator pattern**
  - formalizes the way we move through a list of data within a class

Patterns In Detail

Factory Method

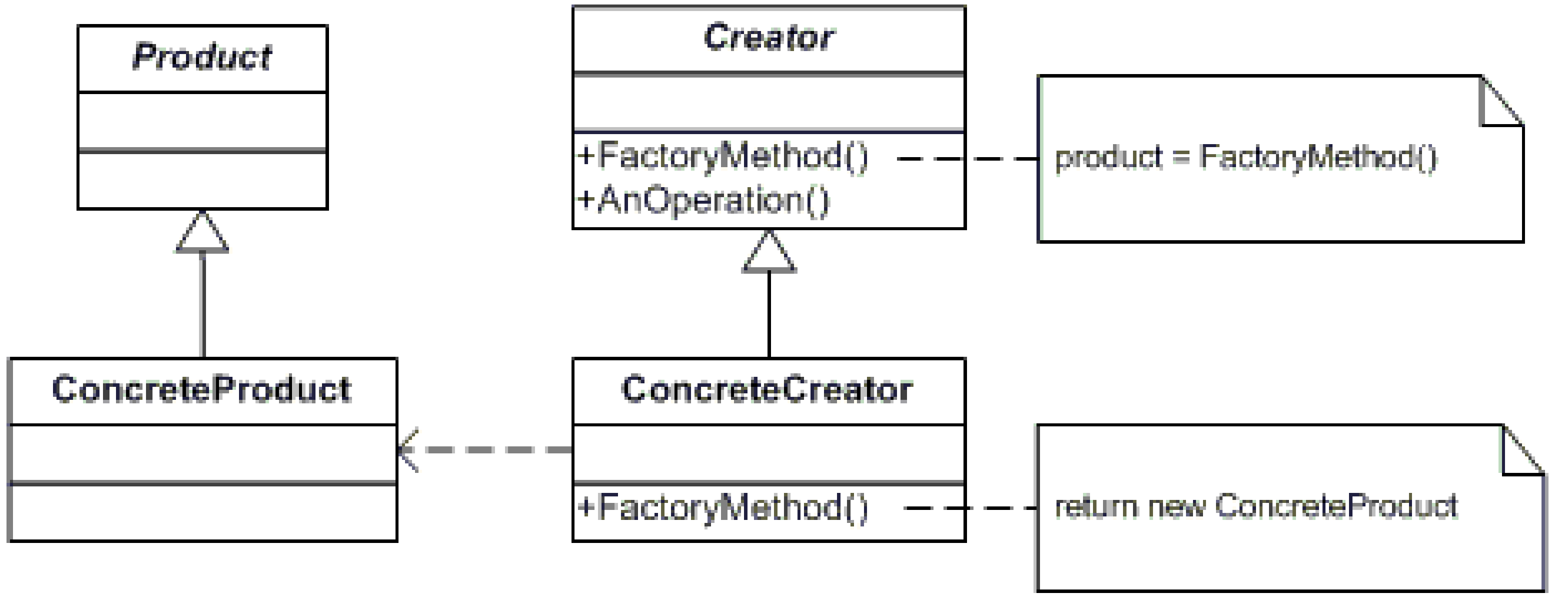
Factory Method

- Definition
  - A Factory pattern is one that returns an instance of one of several possible classes depending on the data provided to it
  - Provides an abstraction or an interface and lets subclass or implementing classes decide which class or method should be instantiated or called, based on the conditions or parameters given

Definition

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

UML Class Diagram

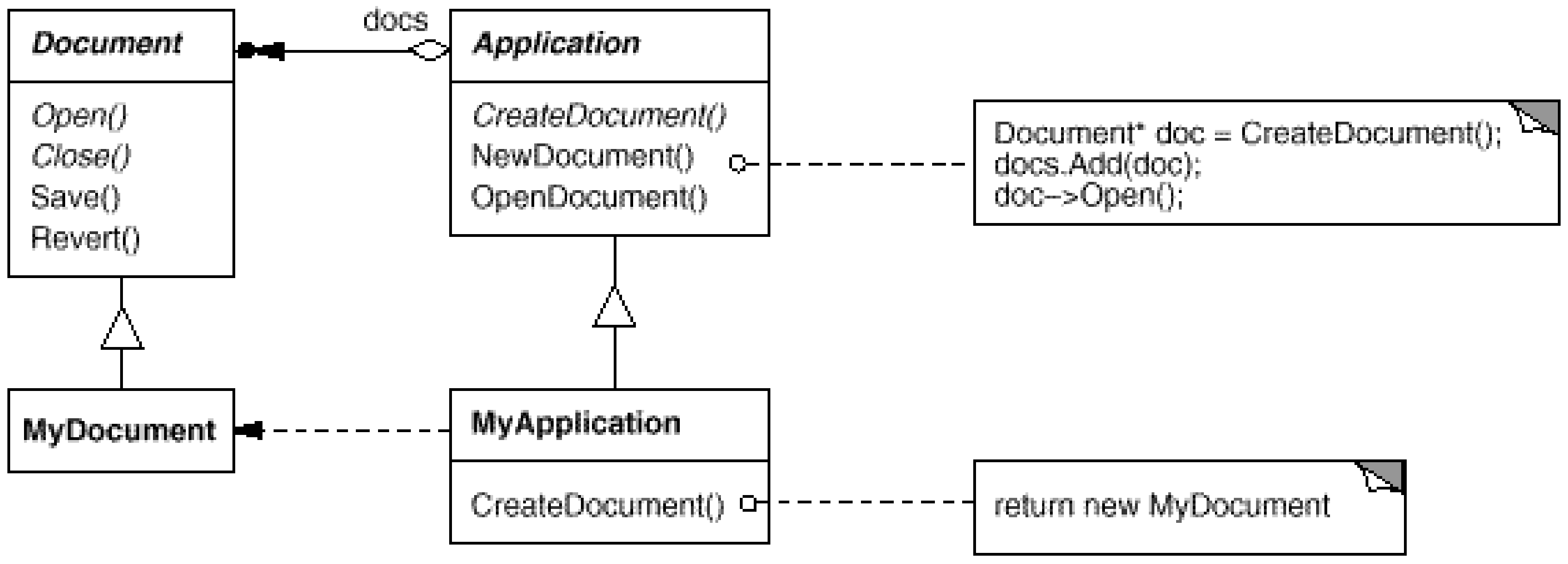


Participants

- **Product**
  - defines the interface of objects the factory method creates
- **ConcreteProduct**
  - implements the Product interface
- **Creator**
  - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.
- **ConcreteCreator**
  - overrides the factory method to return an instance of a ConcreteProduct

Motivation

- Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well.



```
Document* doc = CreateDocument();
docs.Add(doc);
doc->Open();
```

Applicability

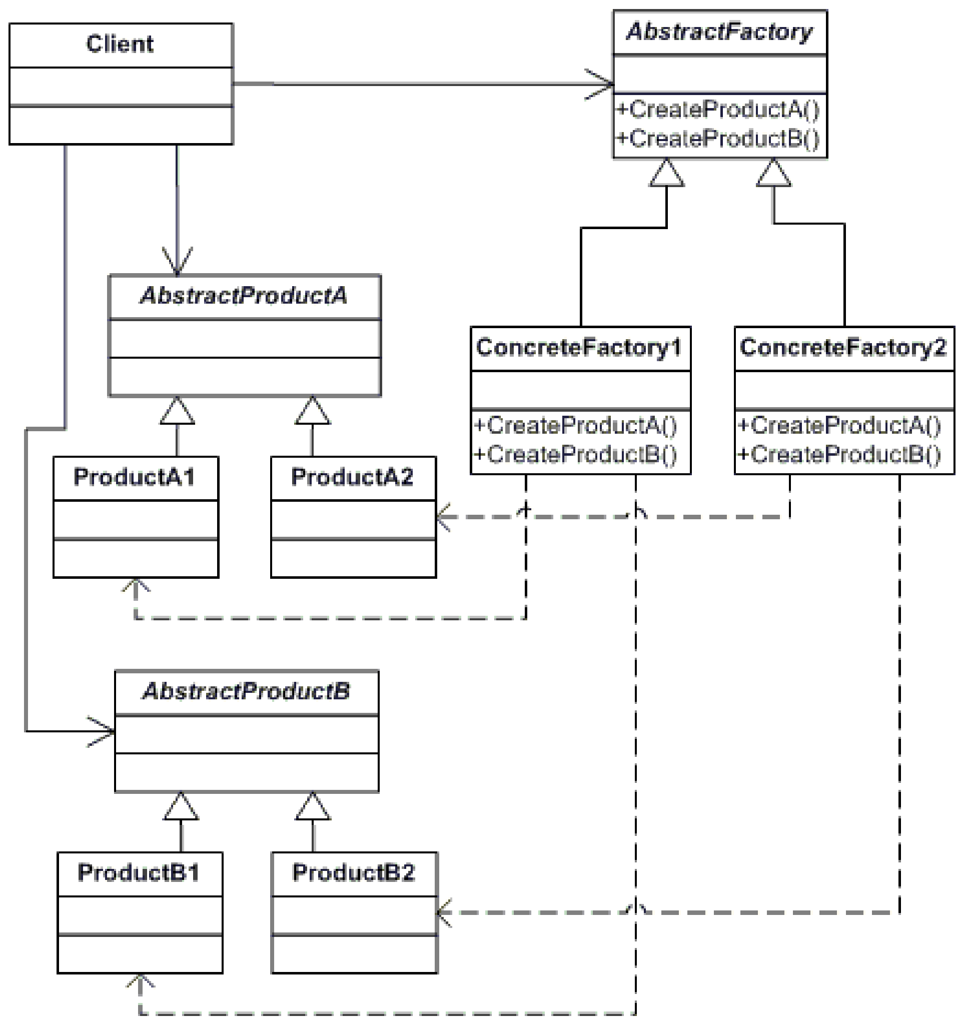
- Use the Factory Method pattern when
  - a class can't anticipate the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Abstract Factory

Definition

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

UML Class Diagram



Participants

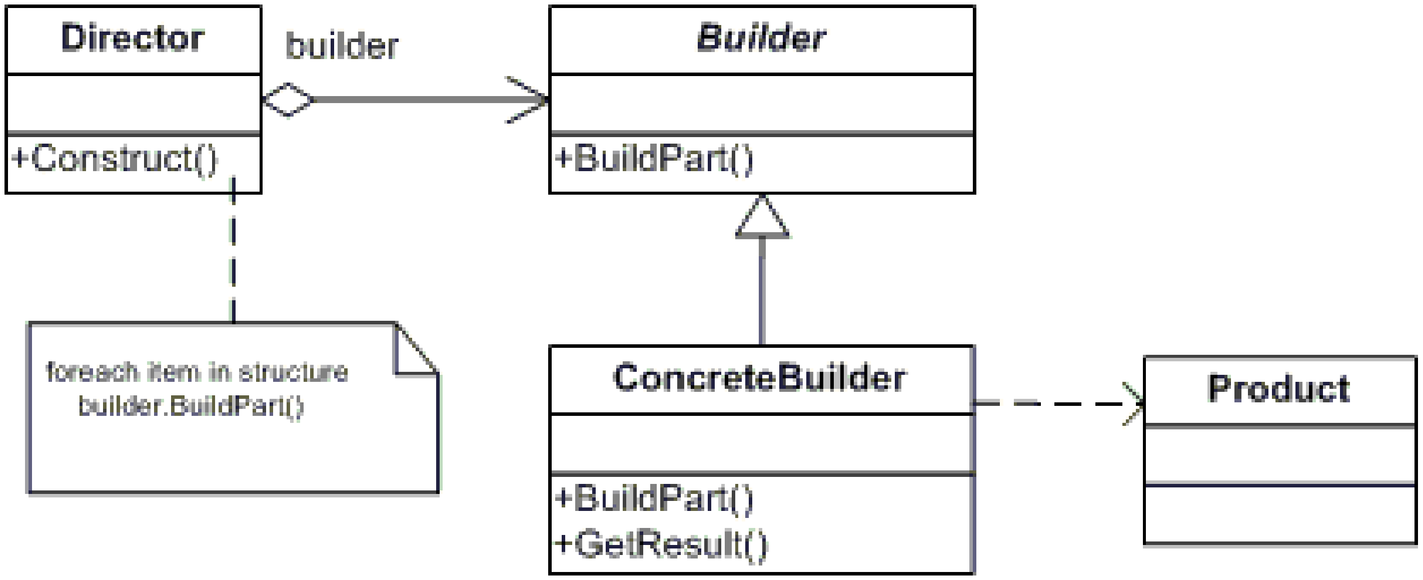
- The classes and/or objects participating in this pattern are:
  - **AbstractFactory (ContinentFactory)**
    - declares an interface for operations that create abstract products
  - **ConcreteFactory (AfricaFactory, AmericaFactory)**
    - implements the operations to create concrete product objects
  - **AbstractProduct (Herbivore, Carnivore)**
    - declares an interface for a type of product object
  - **Product (Wildebeest, Lion, Bison, Wolf)**
    - defines a product object to be created by the corresponding concrete factory implements the AbstractProduct interface
  - **Client (AnimalWorld)**
    - uses interfaces declared by AbstractFactory and AbstractProduct classes

Builder

Definition

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.

UML Class Diagram



Participants

- **Builder (VehicleBuilder)**
  - specifies an abstract interface for creating parts of a Product object
- **ConcreteBuilder (MotorCycleBuilder, CarBuilder, ScooterBuilder)**
  - constructs and assembles parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates
  - provides an interface for retrieving the product
- **Director (Shop)**
  - constructs an object using the Builder interface
- **Product (Vehicle)**
  - represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

Prototype

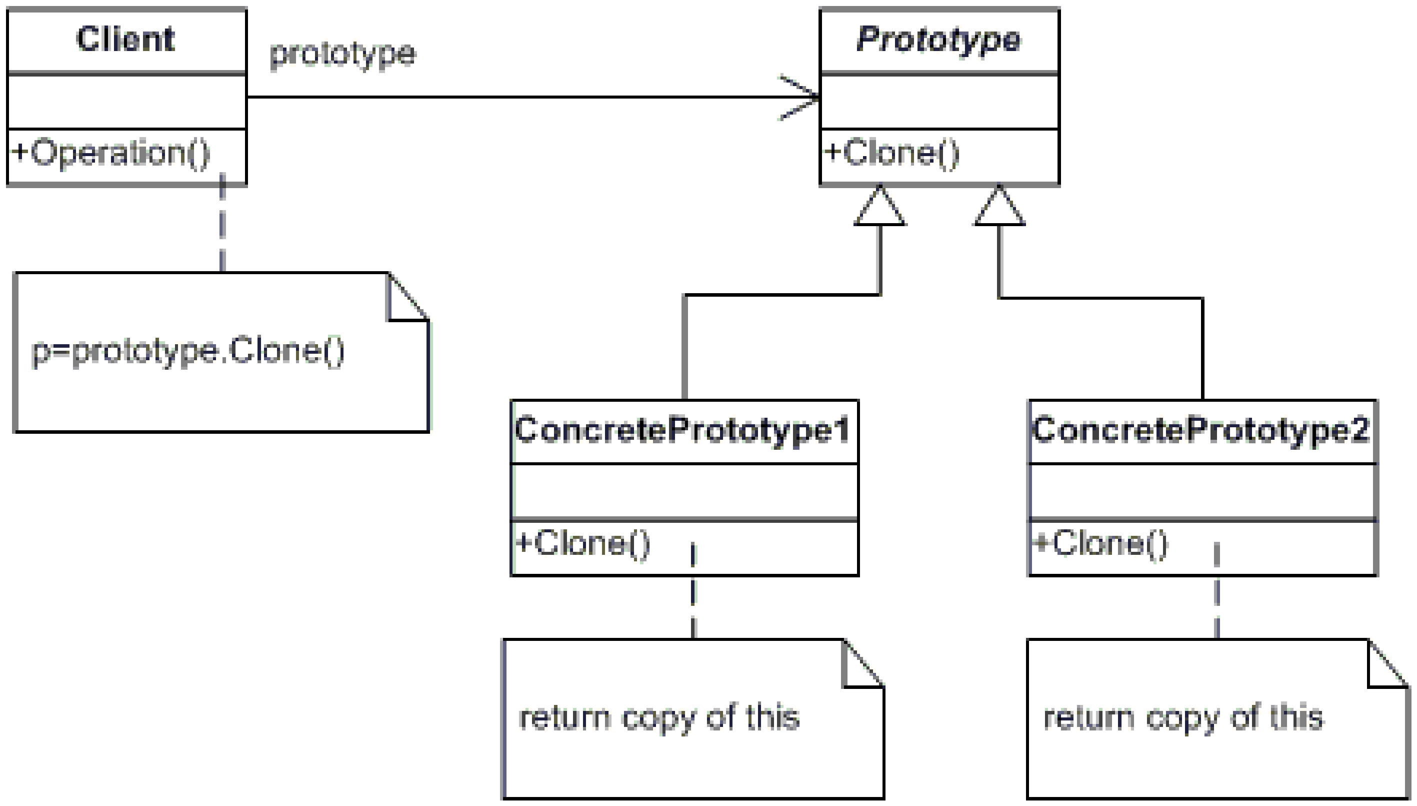
Definition

- Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

Applicability

- Use the Prototype pattern
  - when a system should be independent of how its products are created, composed, and represented
  - when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*
  - to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
  - when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

UML Class Diagram

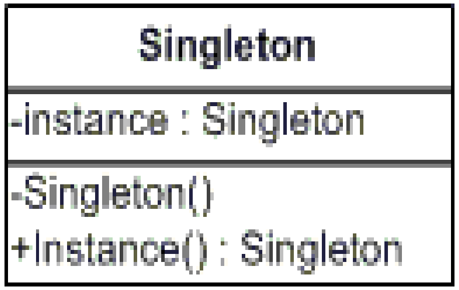


Participants

- **Prototype (ColorPrototype)**
  - declares an interace for cloning itself
- **ConcretePrototype (Color)**
  - implements an operation for cloning itself
- **Client (ColorManager)**
  - creates a new object by asking a prototype to clone itself

Singleton

- Ensure a class has only one instance and provide a global point of access to it.
- **UML Class Diagram**



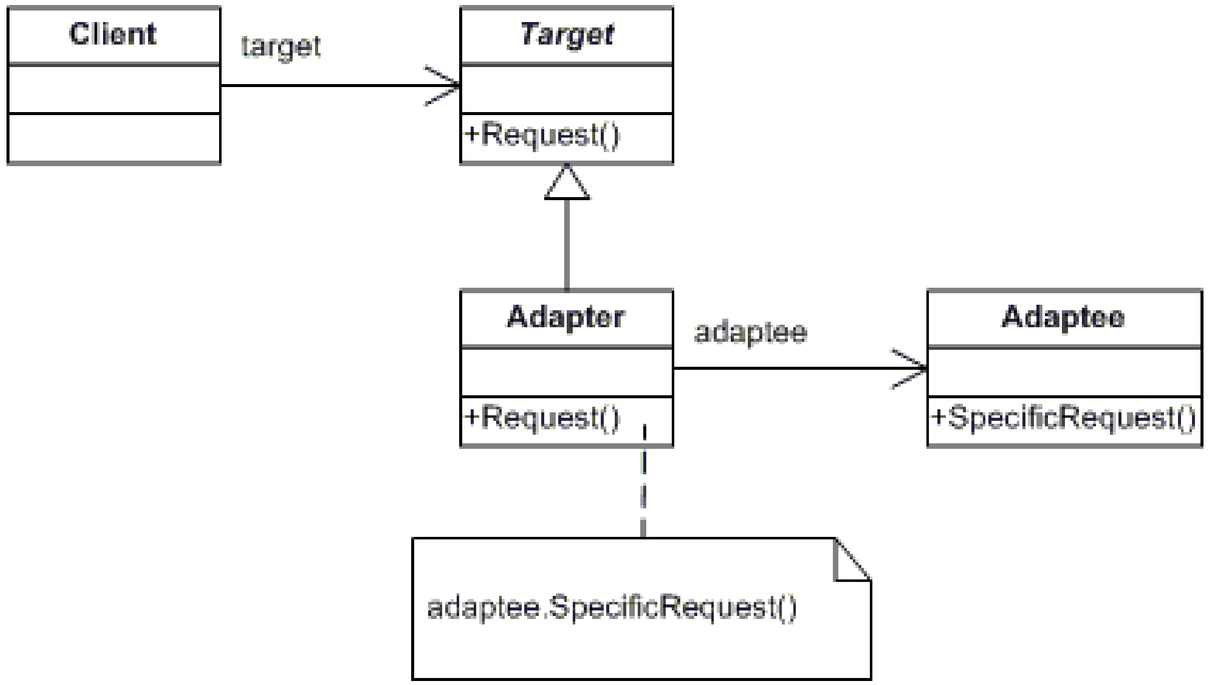
- **Participants**
  - **Singleton (LoadBalancer)**
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  - responsible for creating and maintaining its own unique instance.

Adapter

Definition

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

UML Class Diagram



Participants

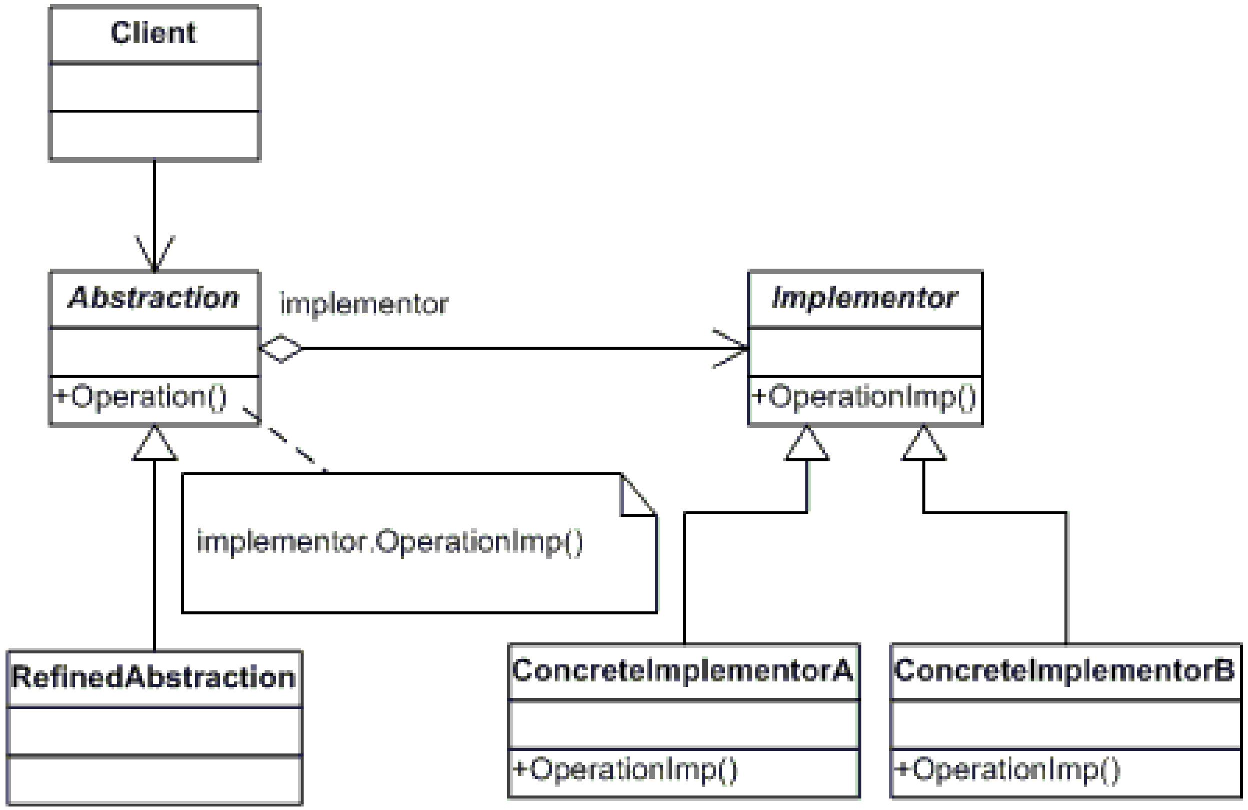
- **Target (ChemicalCompound)**
  - defines the domain-specific interface that Client uses.
- **Adapter (Compound)**
  - adapts the interface Adaptee to the Target interface.
- **Adaptee (ChemicalDatabank)**
  - defines an existing interface that needs adapting.
- **Client (AdapterApp)**
  - collaborates with objects conforming to the Target interface

Bridge

Definition

- Decouple an abstraction from its implementation so that the two can vary independently.

UML Class Diagram



Participants

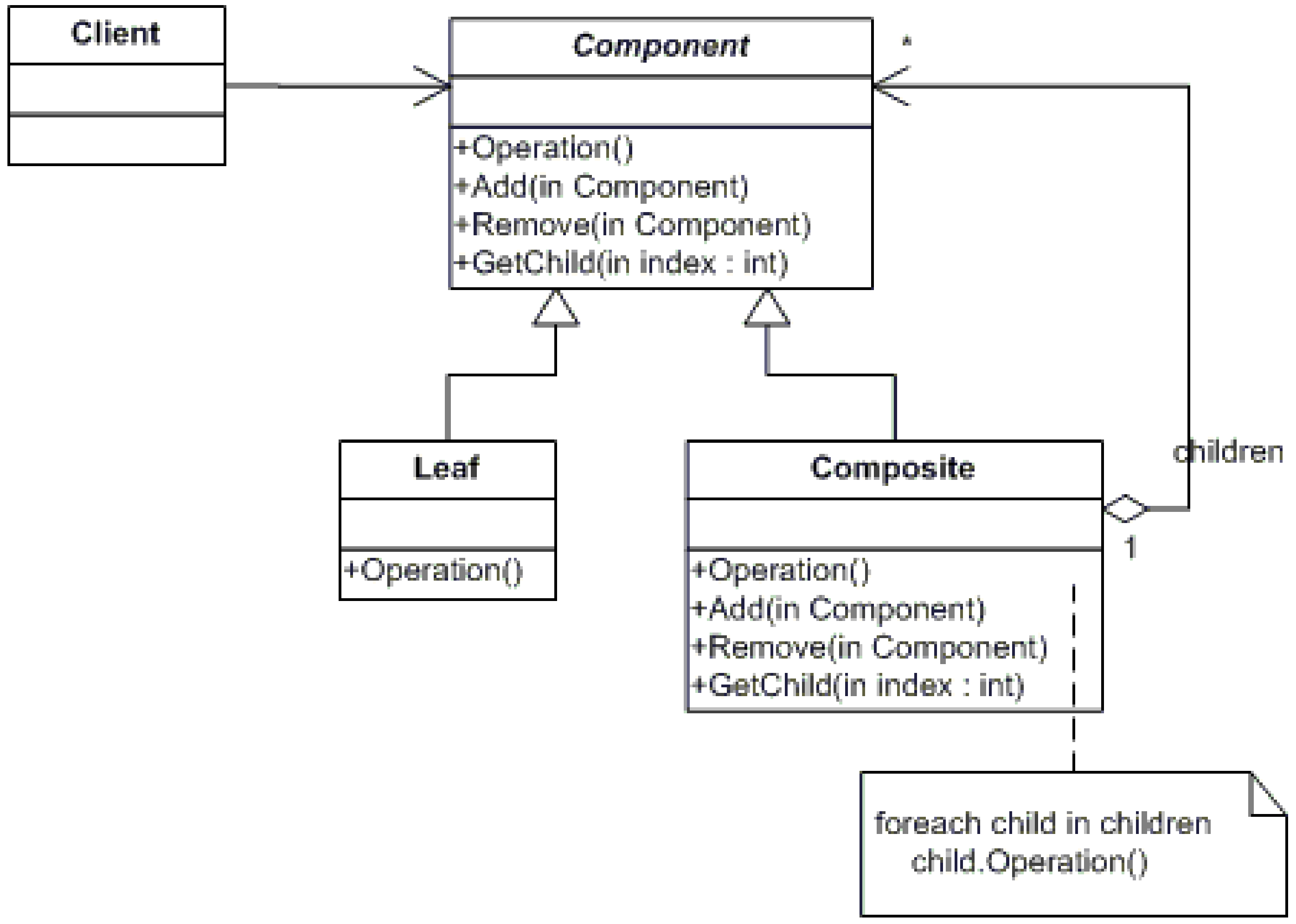
- **Abstraction (BusinessObject)**
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementor.
- **RefinedAbstraction (CustomersBusinessObject)**
  - extends the interface defined by Abstraction.
- **Implementor (DataObject)**
  - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor (CustomersDataObject)**
  - implements the Implementor interface and defines its concrete implementation.

Composite

Composite

- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

UML Class Diagram



Participants (slide1)

- **Component (DrawingElement)**
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- **Leaf (PrimitiveElement)**
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.

Participants (slide 2)

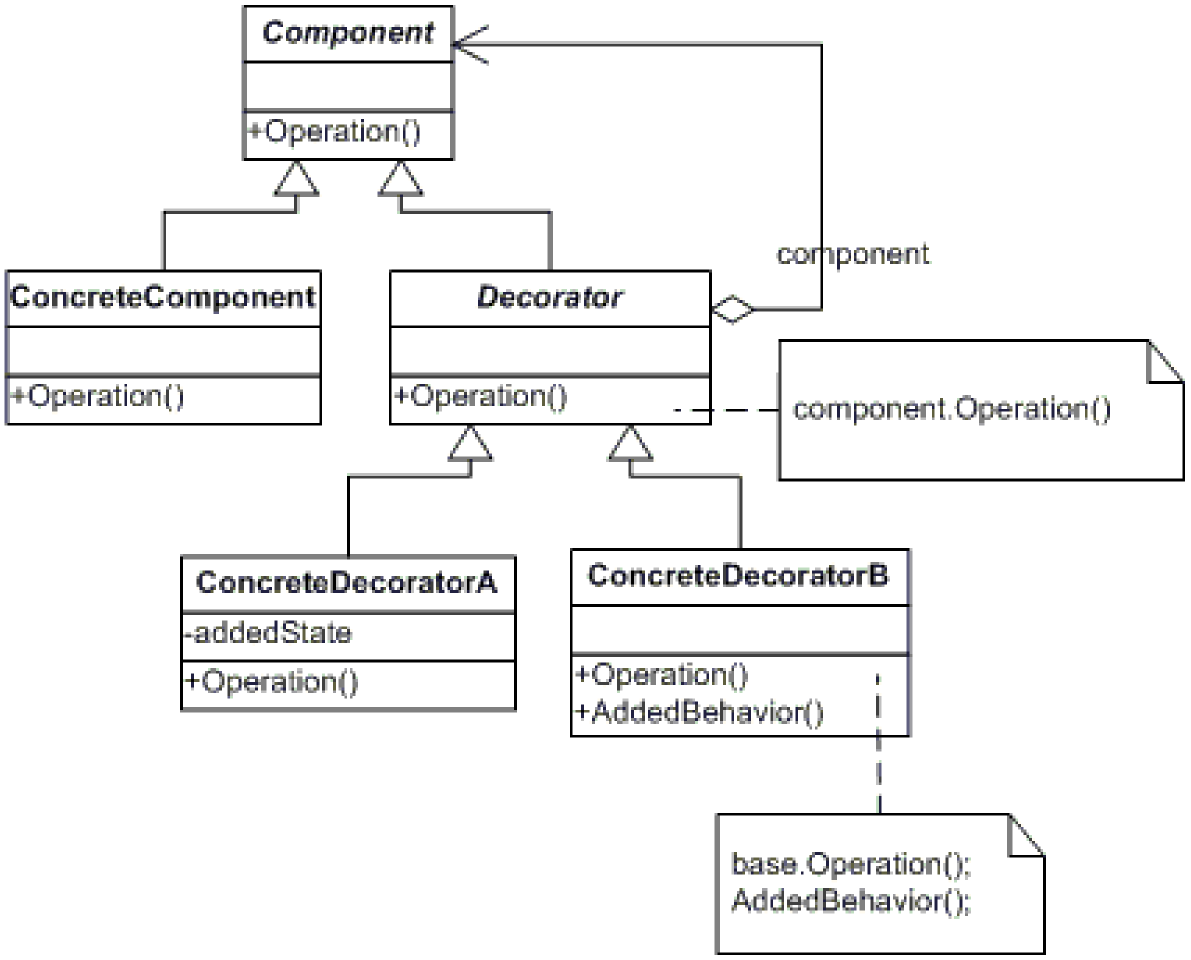
- **Composite (CompositeElement)**
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- **Client (CompositeApp)**
  - manipulates objects in the composition through the Component interface.

Decorator

Definition

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

UML Class Diagram



Participants

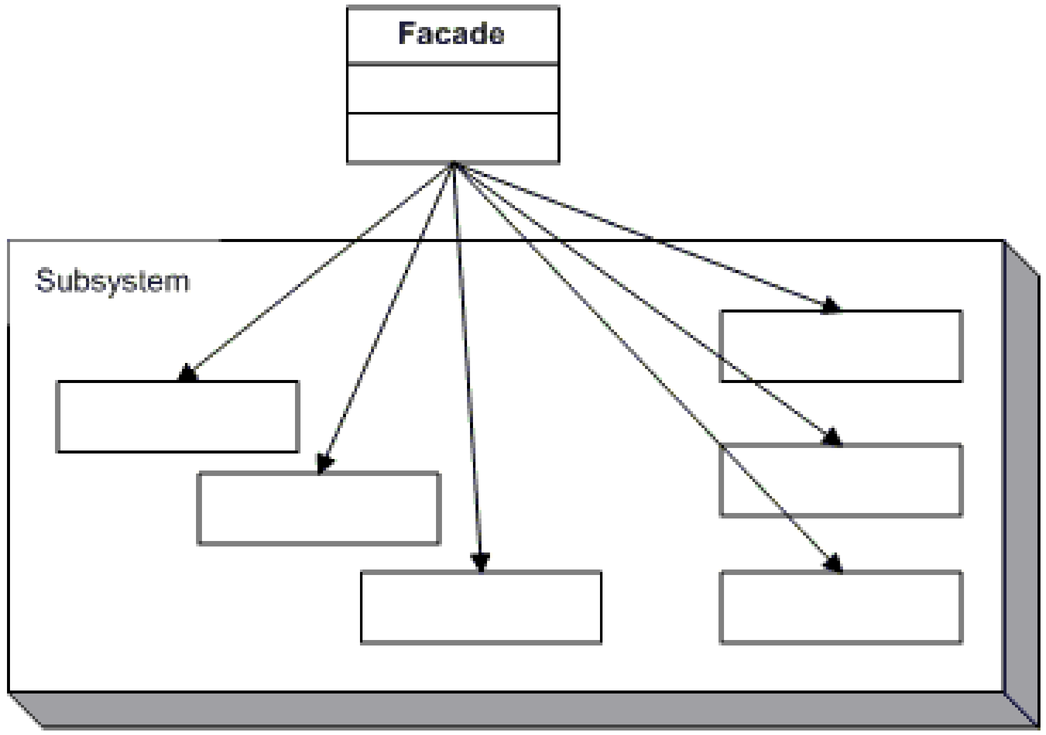
- **Component (LibraryItem)**
  - defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent (Book, Video)**
  - defines an object to which additional responsibilities can be attached.
- **Decorator (Decorator)**
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator (Borrowable)**
  - adds responsibilities to the component.

Facade

Definition

- Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

UML Class Diagram



Participants

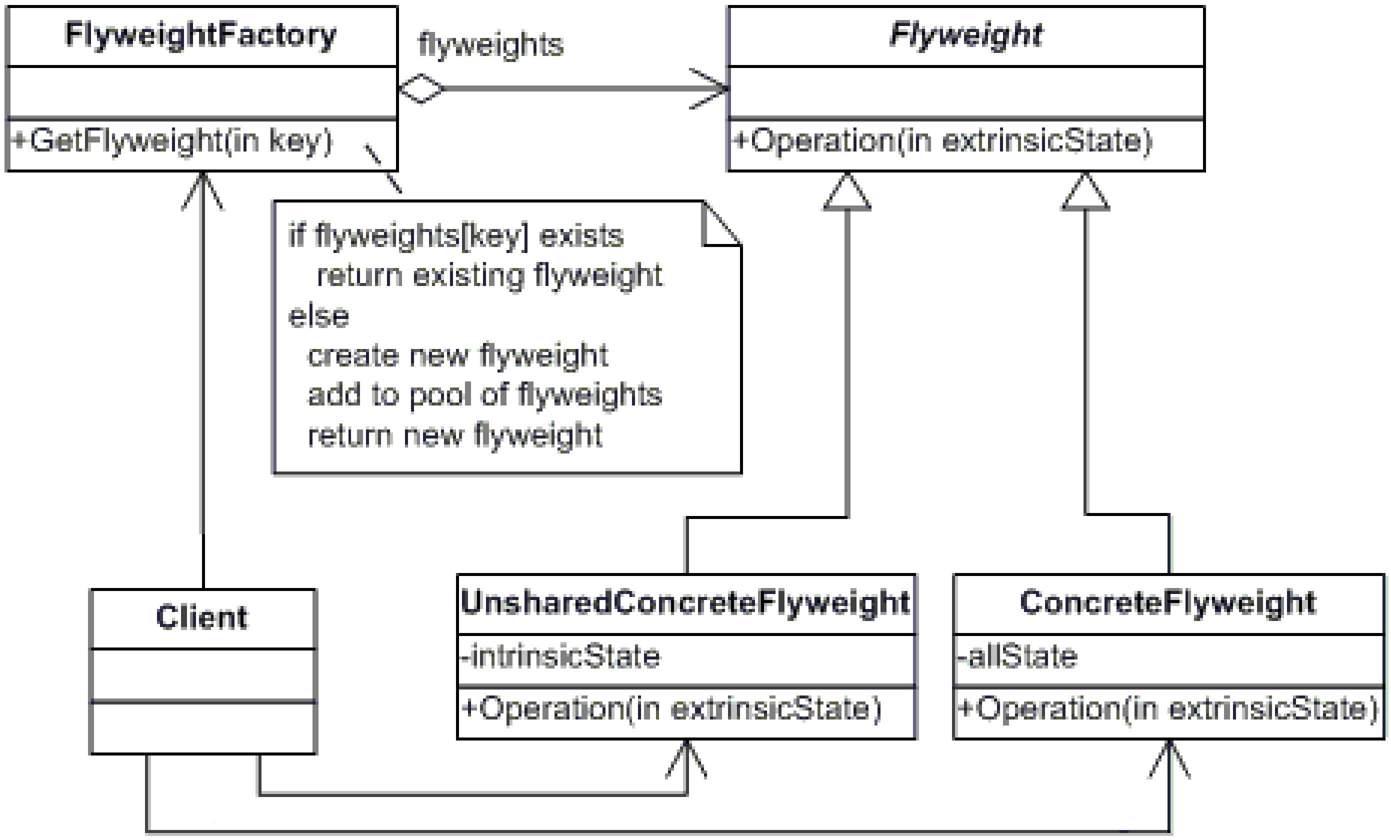
- **Facade (MortgageApplication)**
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- **Subsystem classes (Bank, Credit, Loan)**
  - implement subsystem functionality.
  - handle work assigned by the Facade object.

Flyweigt

Definition

- Use sharing to support large numbers of fine-grained objects efficiently.

UML Class Diagram


FlyweightFactory
+GetFlyweight(in key)
flyweights
Flyweight
+Operation(in extrinsicState)
if flyweights[key] exists
return existing flyweight
else
create new flyweight
add to pool of flyweights
return new flyweight
Client
UnsharedConcreteFlyweight
-intrinsicState
+Operation(in extrinsicState)
ConcreteFlyweight
-allState
+Operation(in extrinsicState)

Participants (slide1)

- **Flyweight (Character)**
  - declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight (CharacterA, CharacterB, ..., CharacterZ)**
  - implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the ConcreteFlyweight object's context.

Participants (slide2)

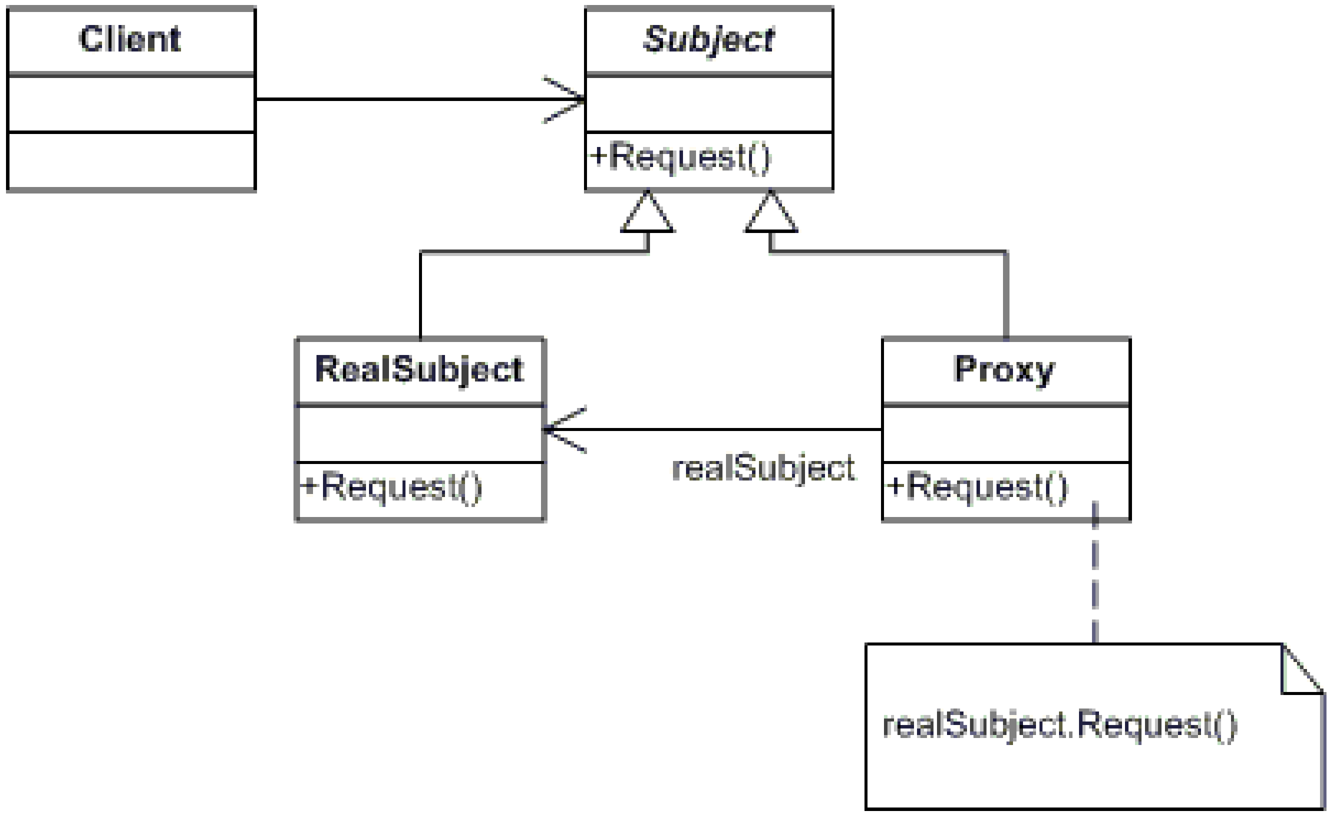
- **UnsharedConcreteFlyweight (not used)**
  - not all Flyweight subclasses need to be shared. The Flyweight interface *enables sharing, but it doesn't enforce it. It is common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).*
- **FlyweightFactory (CharacterFactory)**
  - creates and manages flyweight objects
  - ensures that flyweight are shared properly. When a client requests a flyweight, the FlyweightFactory objects supplies an existing instance or creates one, if none exists.
- **Client (FlyweightApp)**
  - maintains a reference to flyweight(s).
  - computes or stores the extrinsic state of flyweight(s).

Proxy

Definition

- Provide a surrogate or placeholder for another object to control access to it.

UML Class Diagram



Participants

- **Proxy (MathProxy)**
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can be substituted for for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
  - other responsibilites depend on the kind of proxy:
  - *remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.*
  - *virtual proxies may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real images's extent.*
  - *protection proxies check that the caller has the access permissions required to perform a request.*

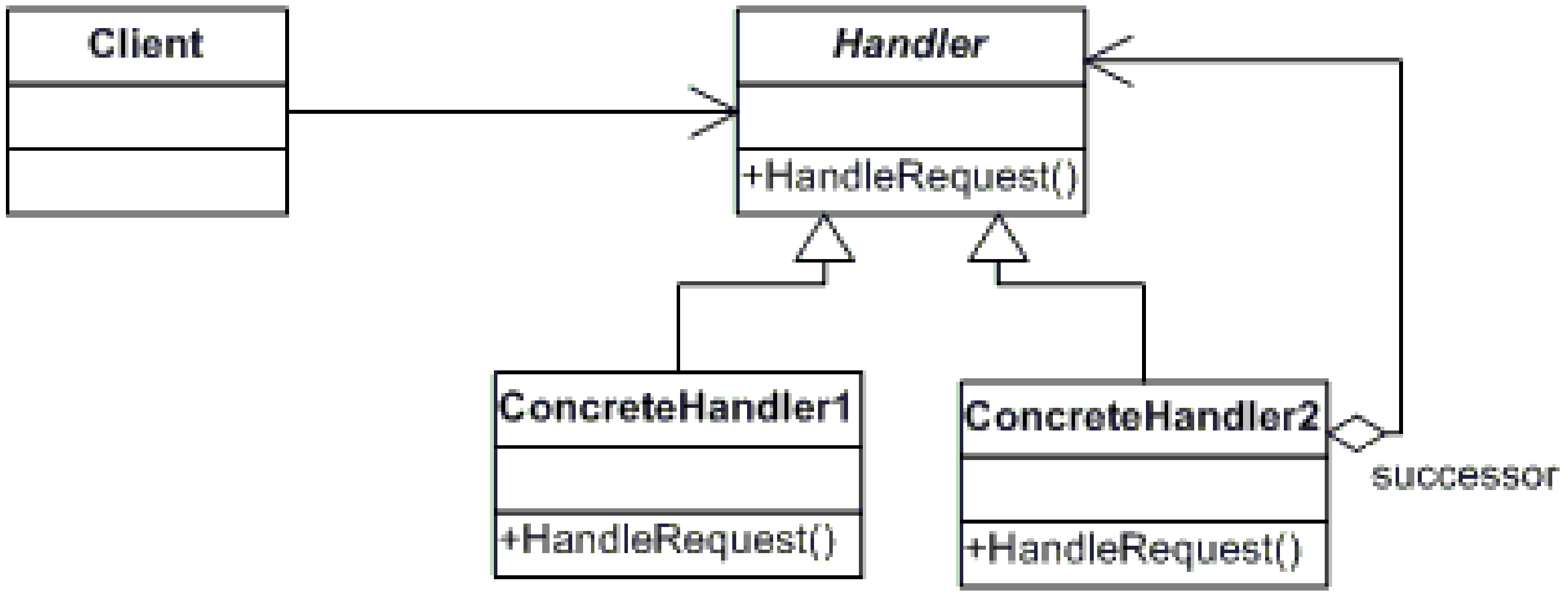
- **Subject (IMath)**
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject (Math)**
  - defines the real object that the proxy represents.

Chain of Responsibility

Definition

- 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.

UML Class Diagaram



Participants

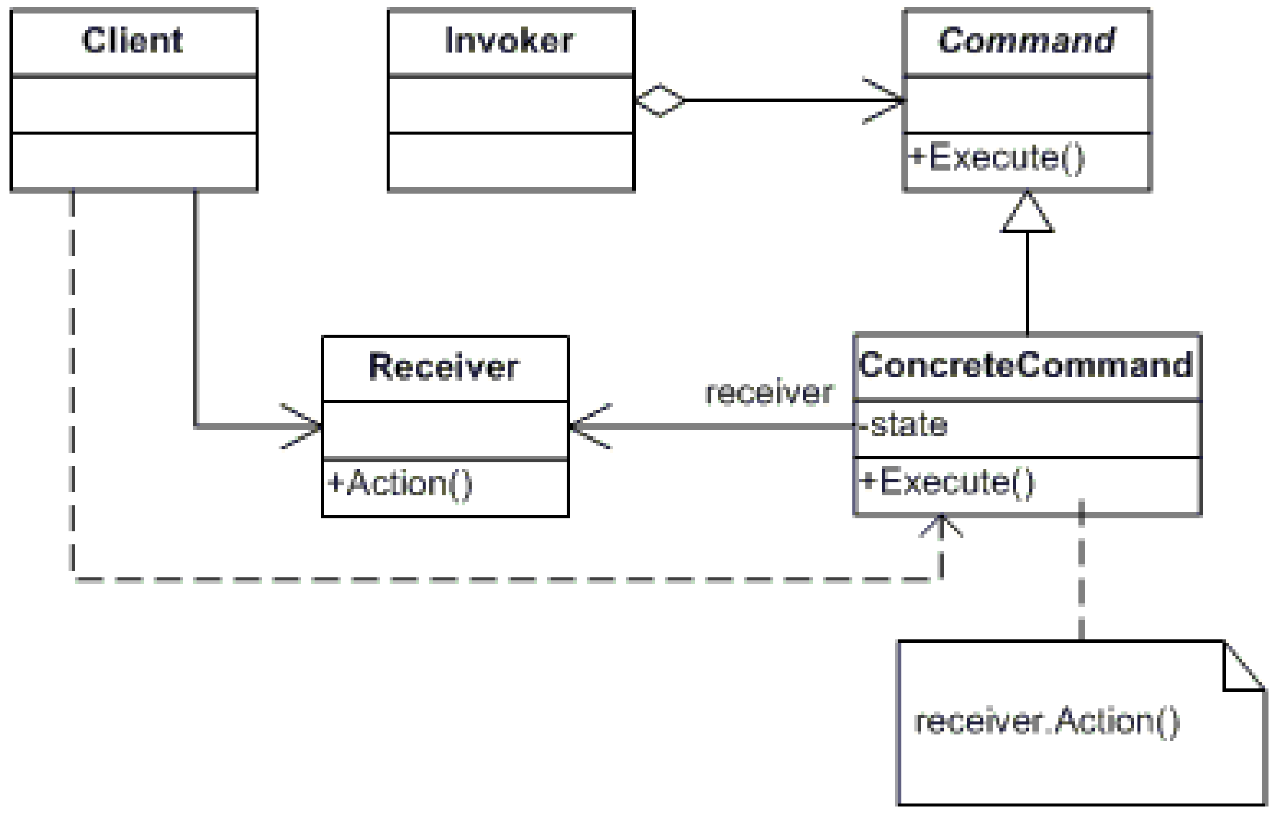
- **Handler (Approver)**
  - defines an interface for handling the requests
  - (optional) implements the successor link
- **ConcreteHandler (Director, VicePresident, President)**
  - handles requests it is responsible for
  - can access its successor
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor
- **Client (ChainApp)**
  - initiates the request to a ConcreteHandler object on the chain

Command

Definition

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

UML Class Diagram



Participants

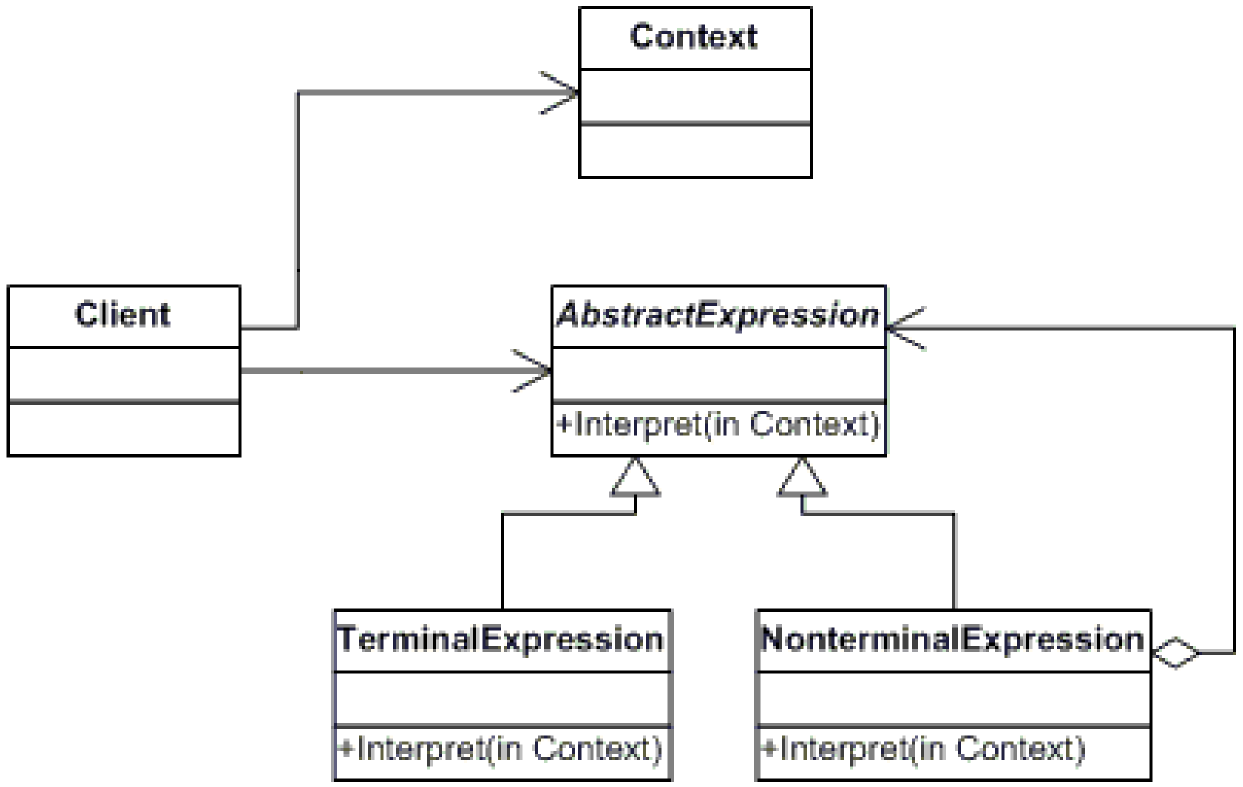
- **Command (Command)**
  - declares an interface for executing an operation
- **ConcreteCommand (CalculatorCommand)**
  - defines a binding between a Receiver object and an action
  - implements Execute by invoking the corresponding operation(s) on Receiver
- **Client (CommandApp)**
  - creates a ConcreteCommand object and sets its receiver
- **Invoker (User)**
  - asks the command to carry out the request
- **• Receiver (Calculator)**
  - knows how to perform the operations associated with carrying out the request.

Interpreter

Definition

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

UML Class Diagram



Participants

- **AbstractExpression (Expression)**
  - declares an interface for executing an operation
- **TerminalExpression (ThousandExpression, HundredExpression, TenExpression, OneExpression)**
  - implements an Interpret operation associated with terminal symbols in the grammar.
  - an instance is required for every terminal symbol in the sentence.
- **NonterminalExpression (not used)**
  - one such class is required for every rule R ::= R1R2...Rn in the grammar
  - maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
  - implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.

Participants

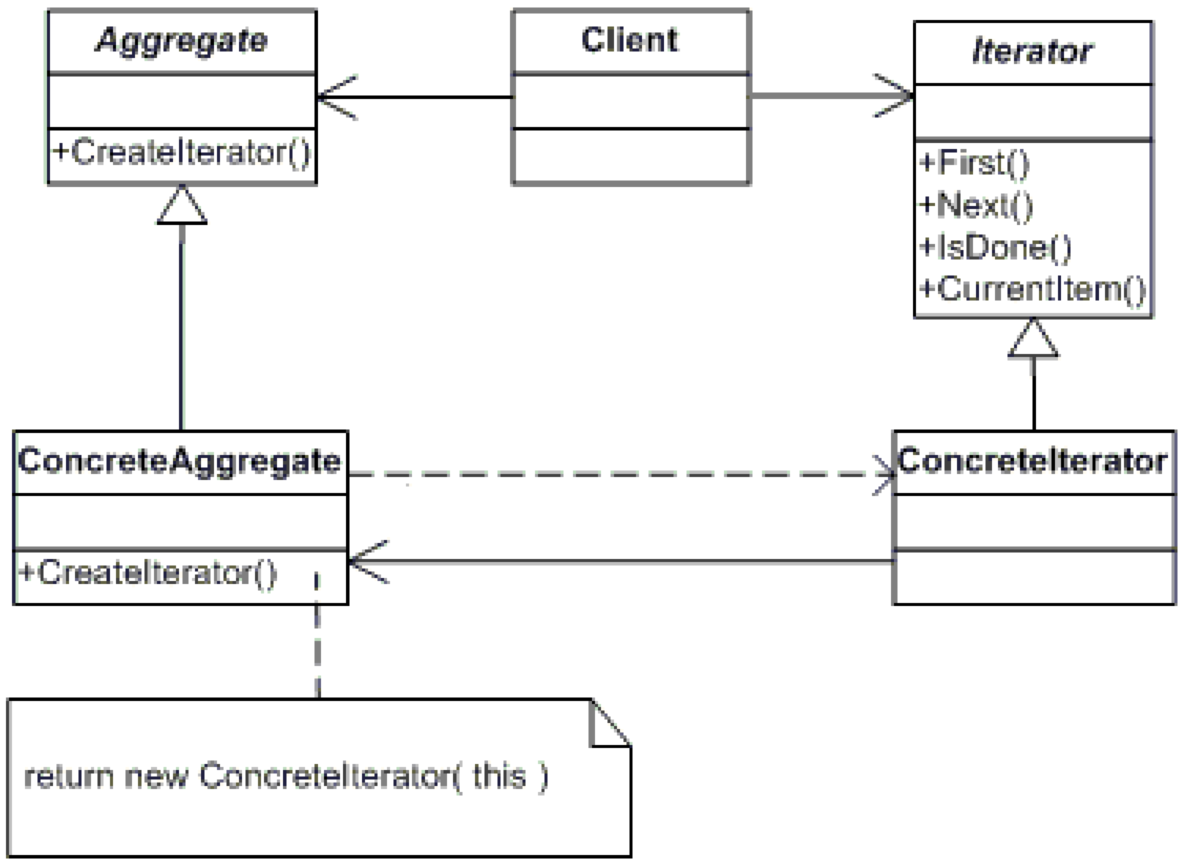
- **Context (Context)**
  - contains information that is global to the interpreter
- **Client (InterpreterApp)**
  - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes
  - invokes the Interpret operation

Iterator

Definition

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

UML Class Diagram



Participants

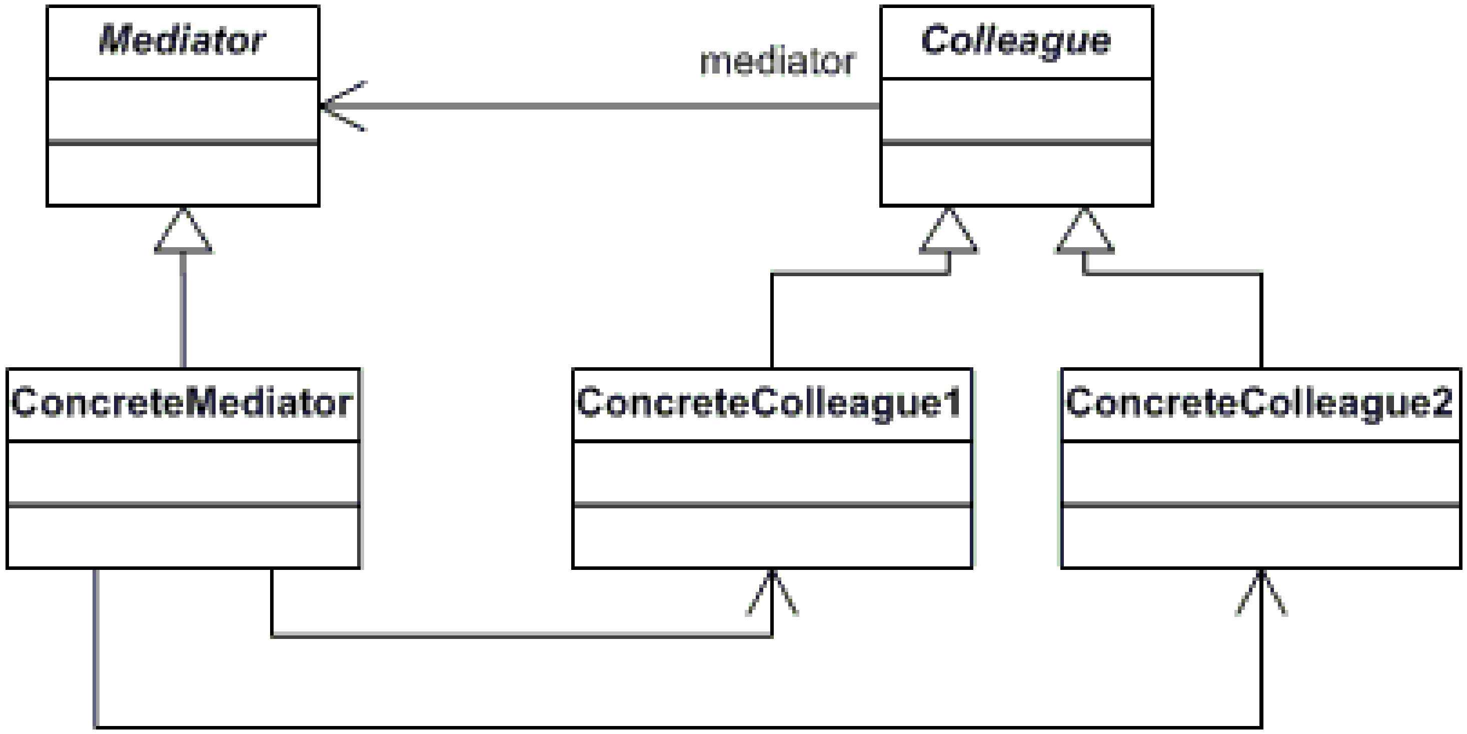
- **Iterator (AbstractIterator)**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator (Iterator)**
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **Aggregate (AbstractCollection)**
  - defines an interface for creating an Iterator object
- **ConcreteAggregate (Collection)**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Mediator

Definition

- 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 pattern** is used to reduce communication complexity between multiple objects or classes.
- This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling

UML Class diagram



Participants

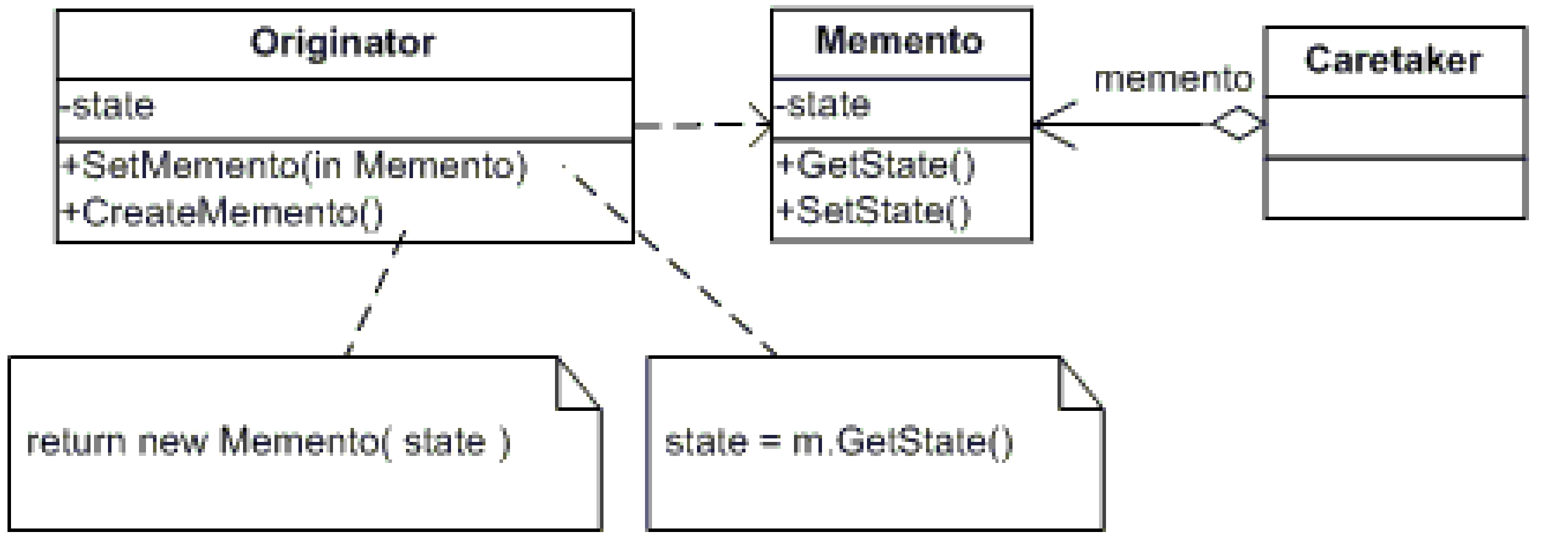
- **Mediator (IChatroom)**
  - defines an interface for communicating with Colleague objects
- **ConcreteMediator (Chatroom)**
  - implements cooperative behavior by coordinating Colleague objects
  - knows and maintains its colleagues
- **Colleague classes (Participant)**
  - each Colleague class knows its Mediator object
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

Memento

Definition

- Memento pattern is a behavioral design pattern.
- Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Memento pattern is used to restore state of an object to a previous state**.
- As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later.

UML Class Diagram



Participants

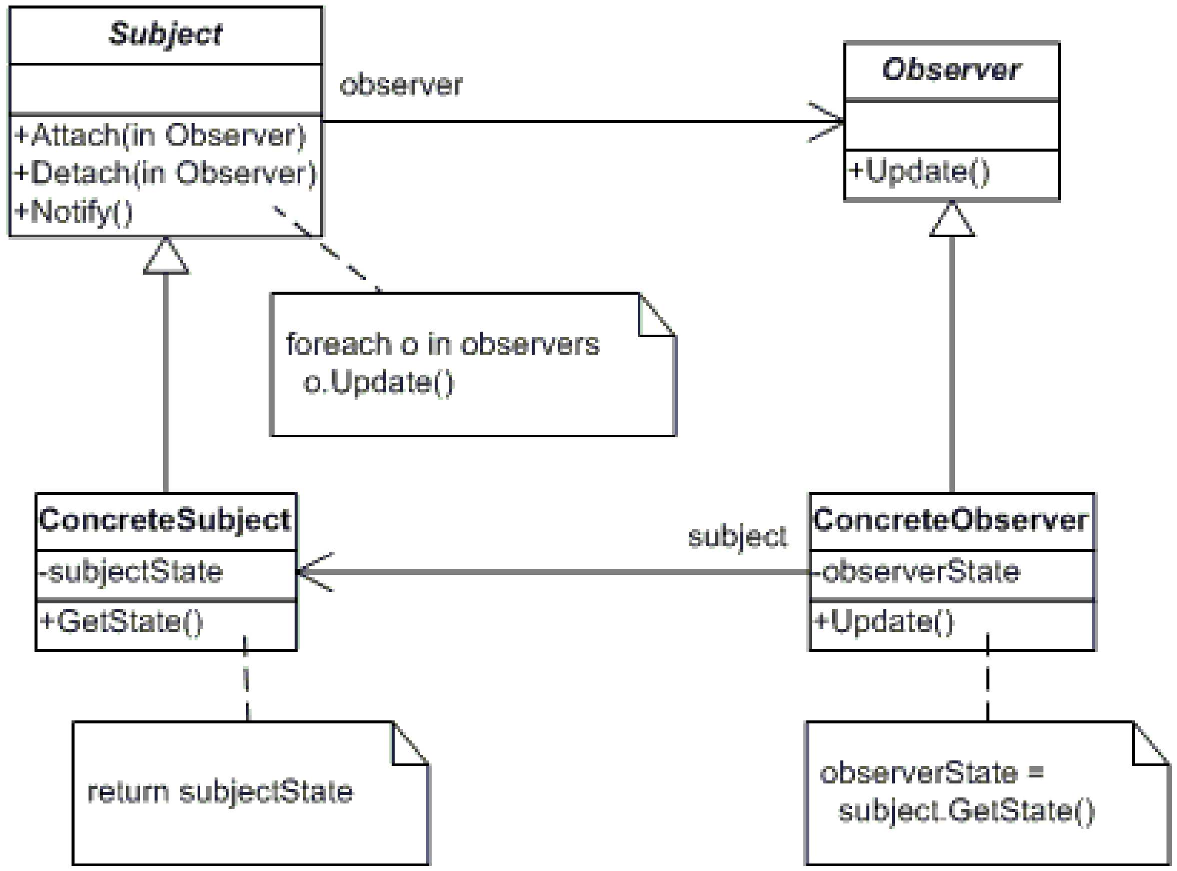
- **Memento (Memento)**
  - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - protect against access by objects of other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento -- it can only pass the memento to the other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produces the memento would be permitted to access the memento's internal state.
- **Originator (SalesProspect)**
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state
- **Caretaker (Caretaker)**
  - is responsible for the memento's safekeeping
  - never operates on or examines the contents of a memento.

Observer

Definition

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

UML Class Diagram



Participants

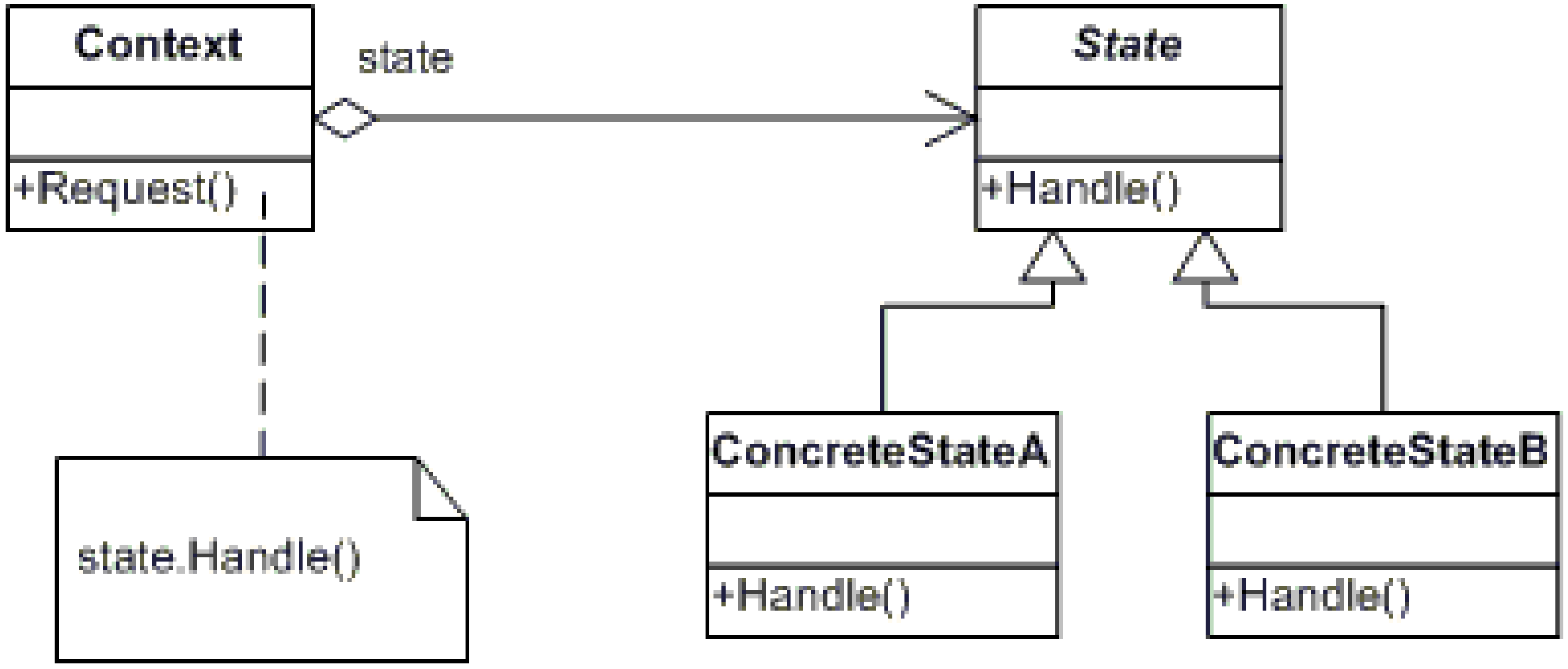
- **Subject (Stock)**
  - knows its observers. Any number of Observer objects may observe a subject
  - provides an interface for attaching and detaching Observer objects.
- **ConcreteSubject (IBM)**
  - stores state of interest to ConcreteObserver
  - sends a notification to its observers when its state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

State

23

Definition

- Allow an object to alter its behavior when its internal state changes.
- The main idea of State pattern is to allow the object for changing its behavior without changing its class.
- By implementing it, the code should remain cleaner without many if/else statements.
- The State pattern minimizes conditional complexity, eliminating the need for if and switch statements in objects that have different behavior requirements unique to different state transitions


Context
+Request()
state
State
+Handle()
state.Handle()
ConcreteStateA
+Handle()
ConcreteStateB
+Handle()

Participants

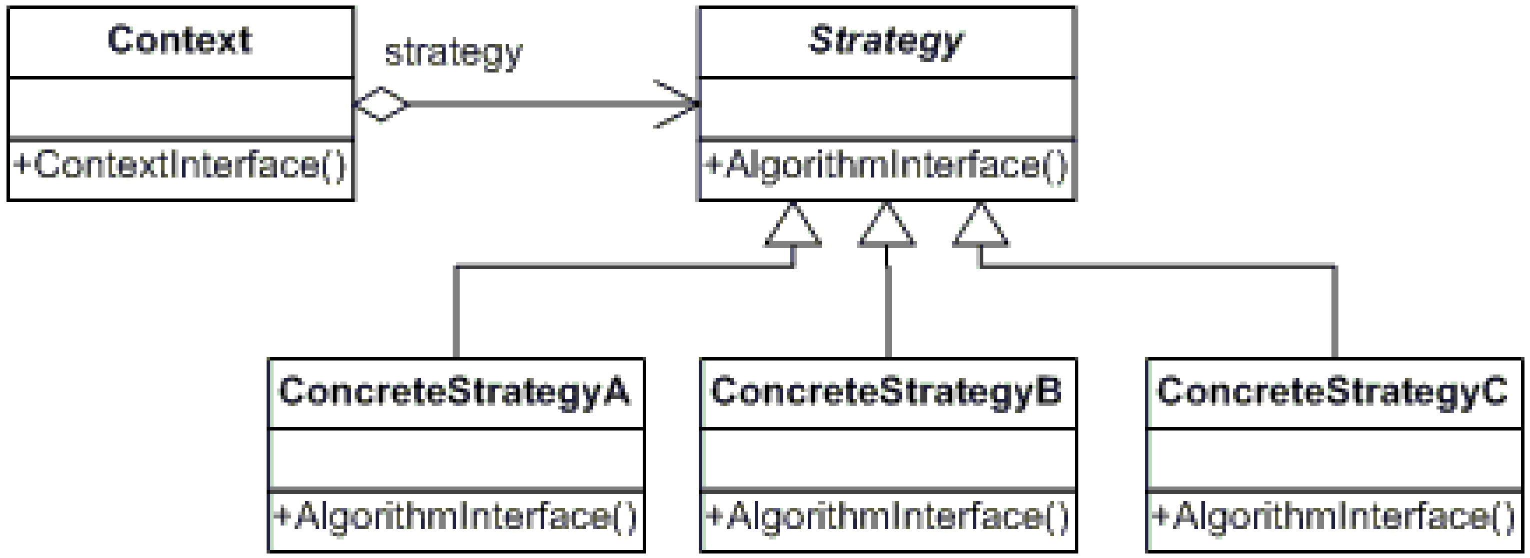
- **Context (Account)**
  - defines the interface of interest to clients
  - maintains an instance of a ConcreteState subclass that defines the current state.
- **State (State)**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **Concrete State (RedState, SilverState, GoldState)**

Strategy

Definition

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

UML Class Diagram



Participants

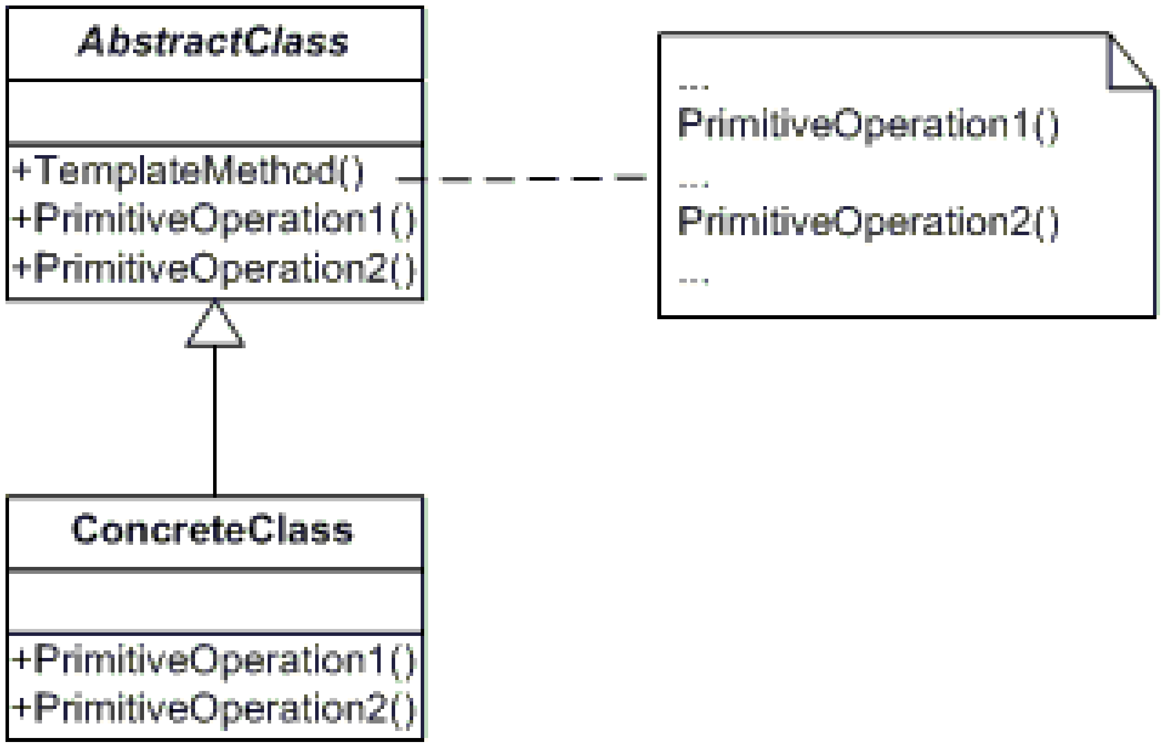
- **Strategy (SortStrategy)**
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
- **ConcreteStrategy (QuickSort, ShellSort, MergeSort)**
  - implements the algorithm using the Strategy interface
- **Context (SortedList)**
  - is configured with a ConcreteStrategy object
  - maintains a reference to a Strategy object
  - may define an interface that lets Strategy access its data.

Template Method

Definition

- 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.

Template Method



Participants

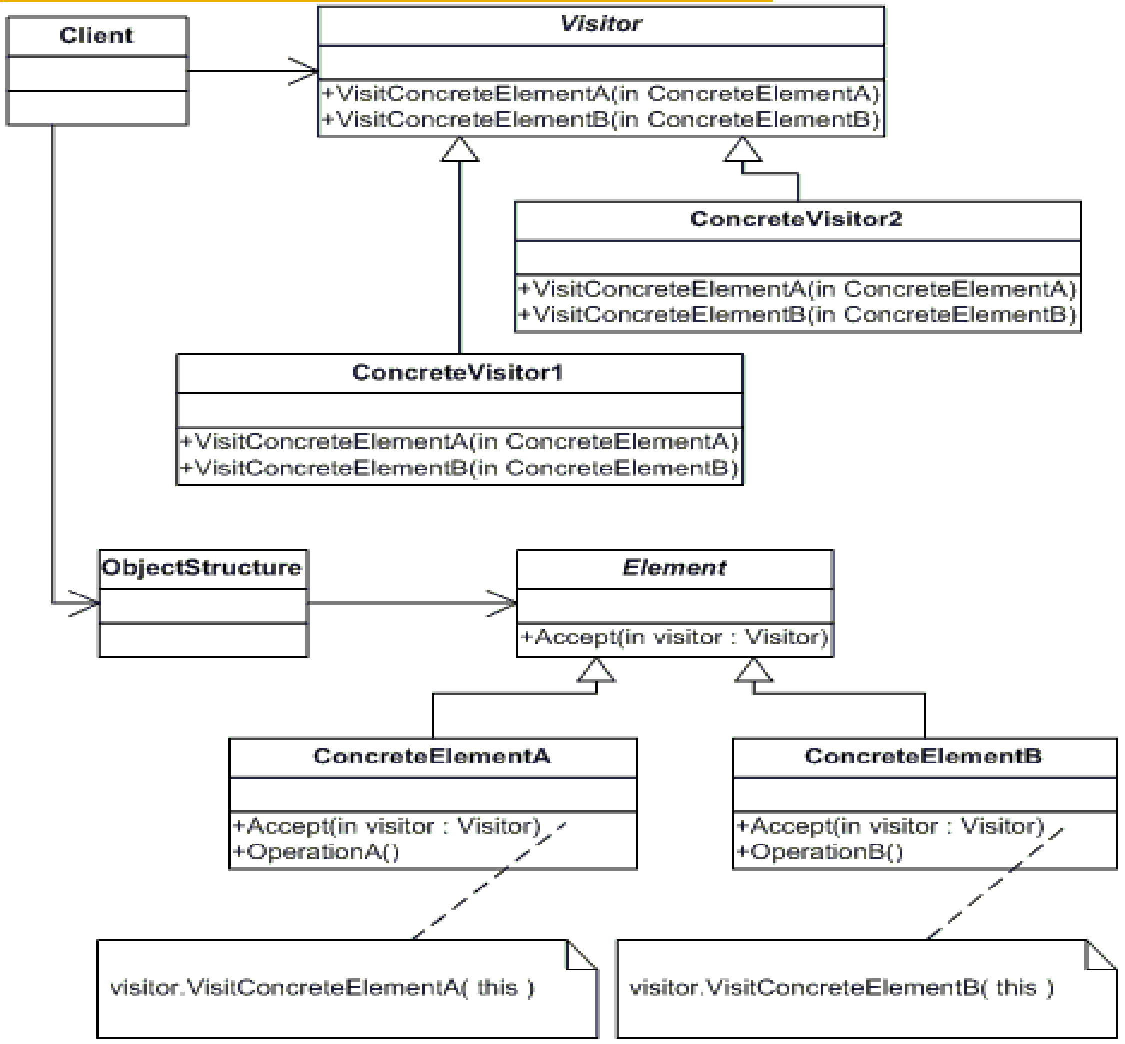
- **AbstractClass (DataObject)**
  - defines abstract *primitive operations that concrete subclasses define to implement steps of an algorithm*
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass (CustomerDataObject)**
  - implements the primitive operations ot carry out subclass-specific steps of the algorithm

Visitor

Definition

- 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.

UML Class Diagram


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)

Participants

- **Visitor (Visitor)**
  - declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the elements directly through its particular interface
- **ConcreteVisitor (IncomeVisitor, VacationVisitor)**
  - implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
- **Element (Element)**
  - defines an Accept operation that takes a visitor as an argument.
- **ConcreteElement (Employee)**
  - implements an Accept operation that takes a visitor as an argument
- **ObjectStructure (Employees)**
  - can enumerate its elements
  - may provide a high-level interface to allow the visitor to visit its elements
  - may either be a Composite (pattern) or a collection such as a list or a set

Gang Of Four- The GOF

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

The authors of *Design Patterns* have suggested that every pattern start with an abstract class and that you derive concrete working classes from that abstraction

←BACK