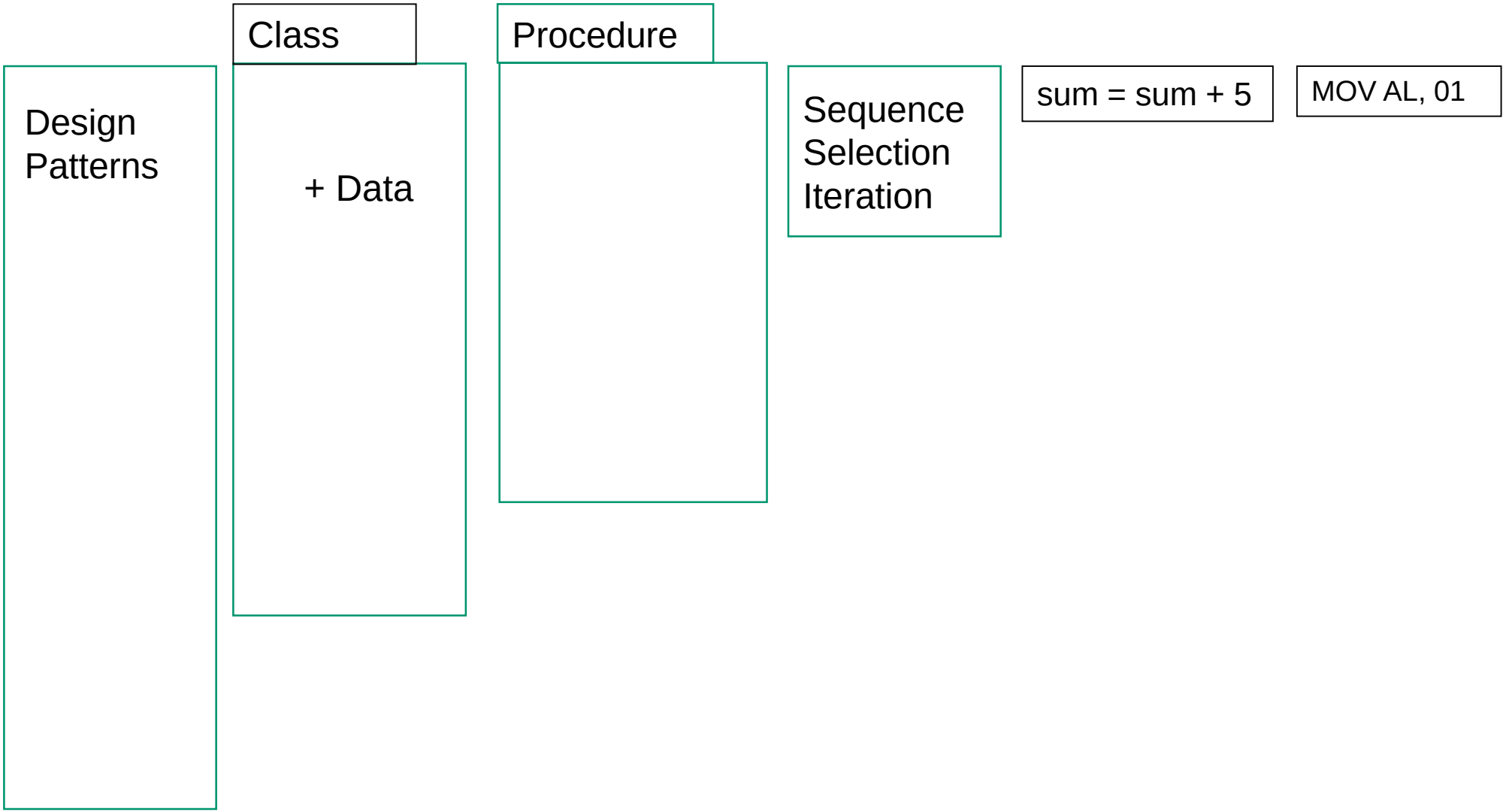
Design patterns

Jef De Smedt
Beta VZW

Overview

- Introduction
- Creational Design patterns
- Structural Design patterns
- Behavorial Design patterns

Design pattterns: higher level of abstraction



Design pattern

- Name
- Description of the problem
- Description of the solution(pattern)
- Advantages/disadvantages

General principles

Program to an interface, not to an implementation

Declaration

void ExecuteCommand(String cmd, IConnection c)

Call

ExecuteCommand("...",new OracleConnection())

ExecuteCommand("...",new MySQLConnection())

General principles

Favor object composition over class inheritance

```
class List{
    public void add(Object o){...}
 }

class PersonList: List{ //inheritance
 }

 class PersonList{ //composition
    private List l = new List();
    public void add(Person p){ l.add(p);}
}
```

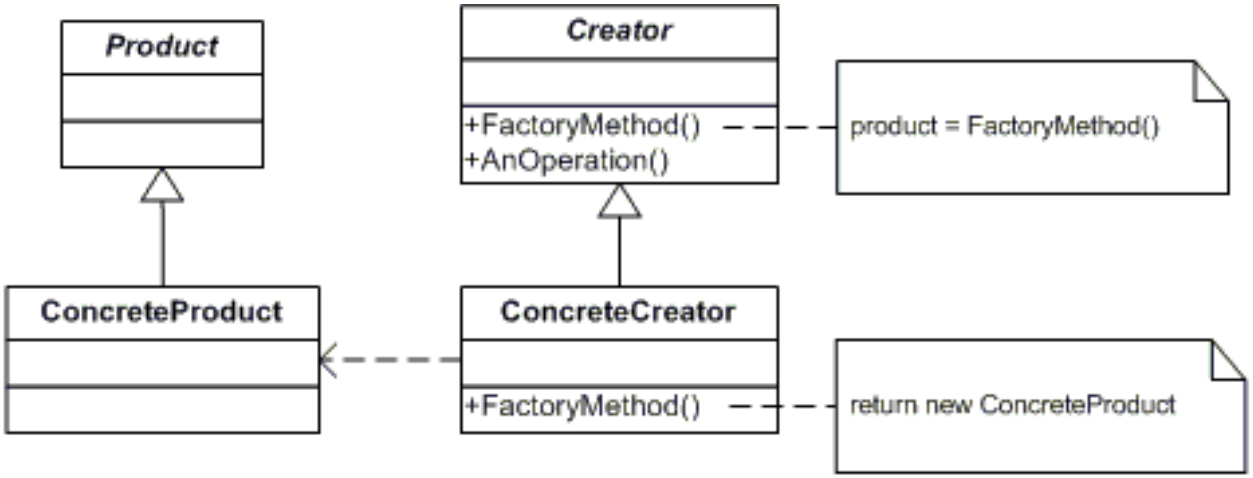
Singleton

- Name: Singleton
- Description: I want one global object
- Solution: Define private static instance + static method
- Advantages/disadvantages: Can be used to manage a pool of objects

Creational patterns

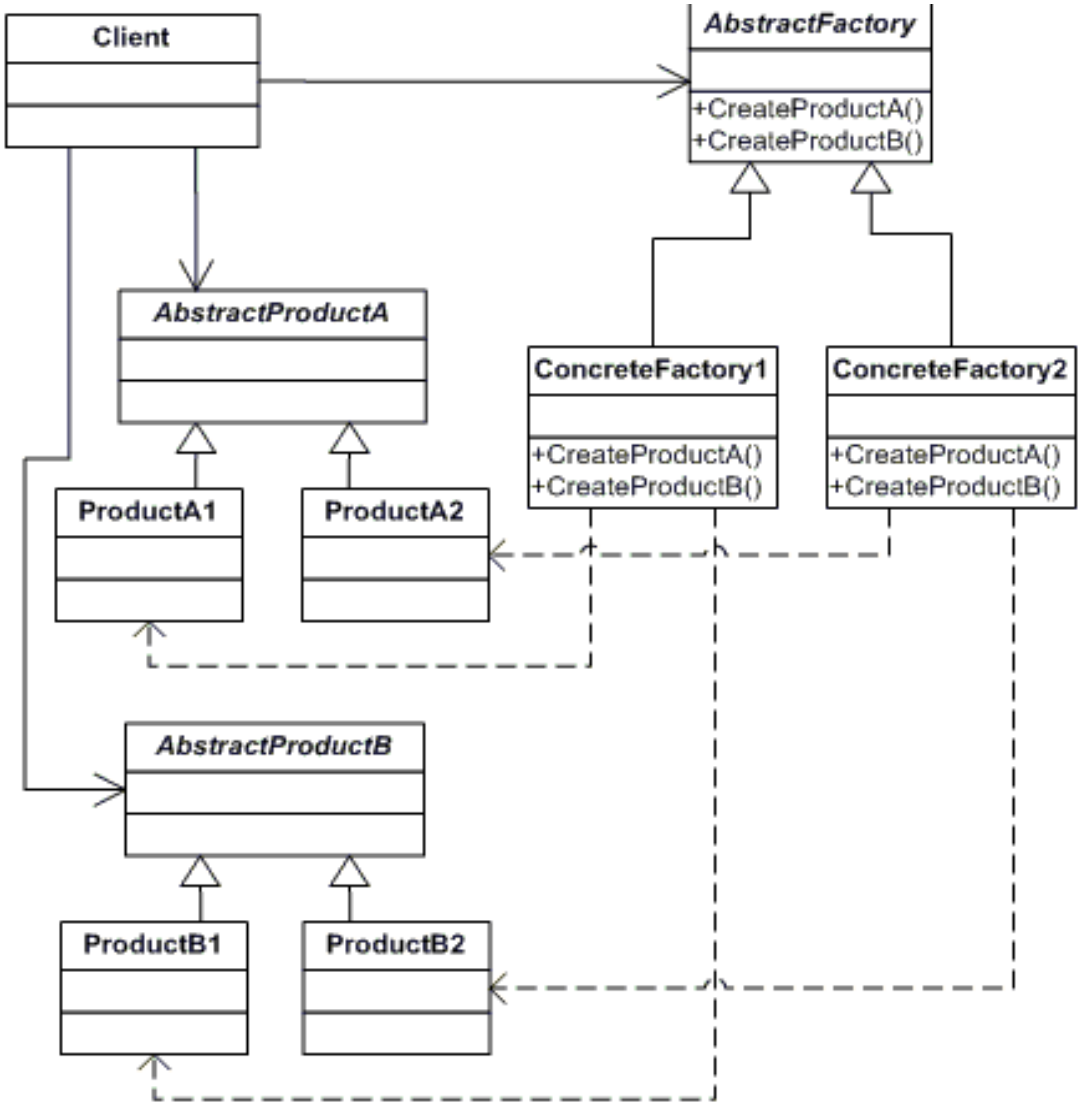
(Patterns to create an object)

Factory method



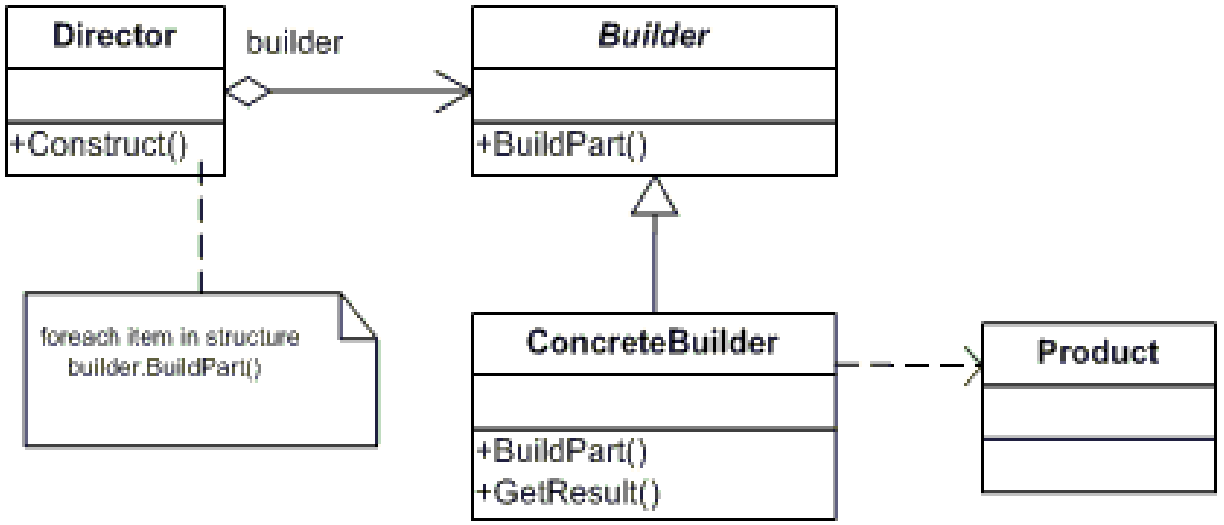
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Abstract Factory



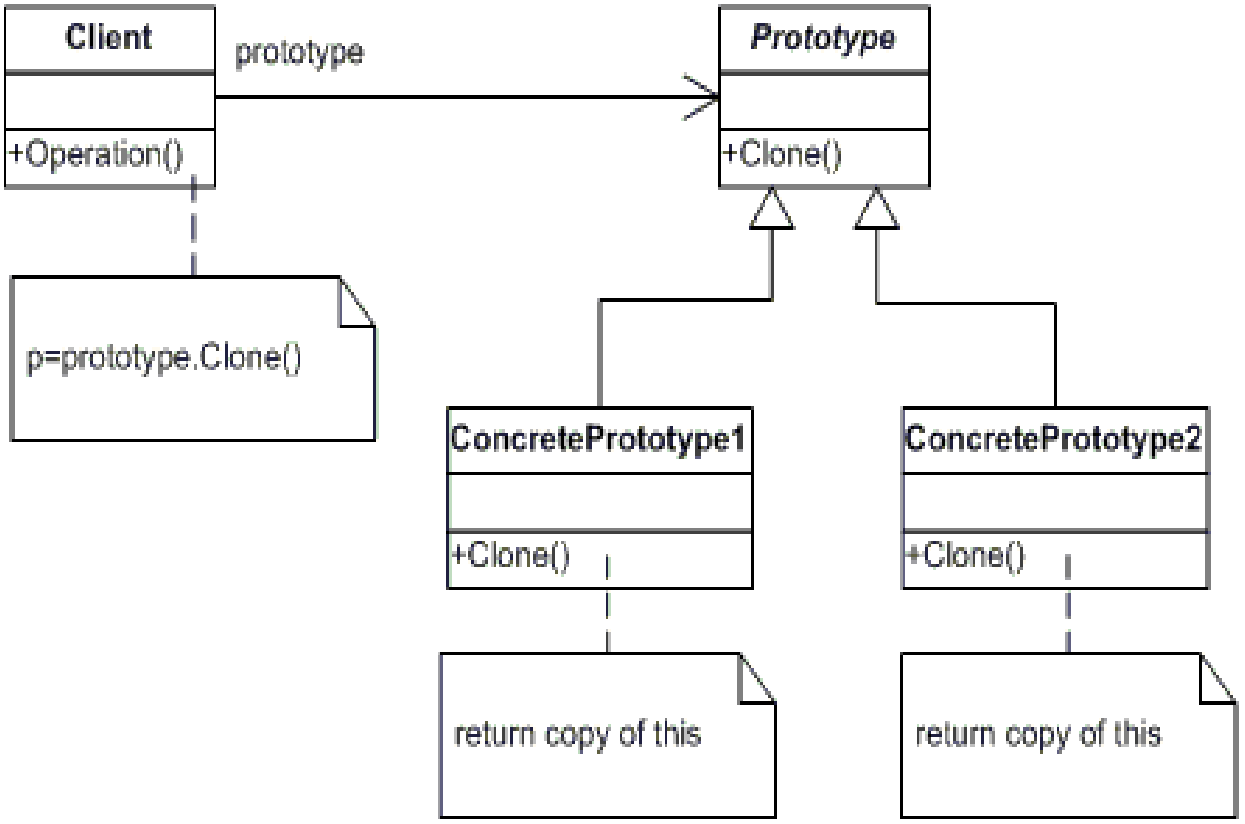
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder



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

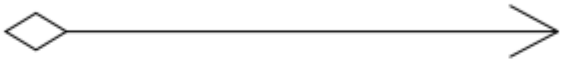
Prototype



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

UML symbols

Generalisation, inheritance



Aggregation (filled diamond =composition)

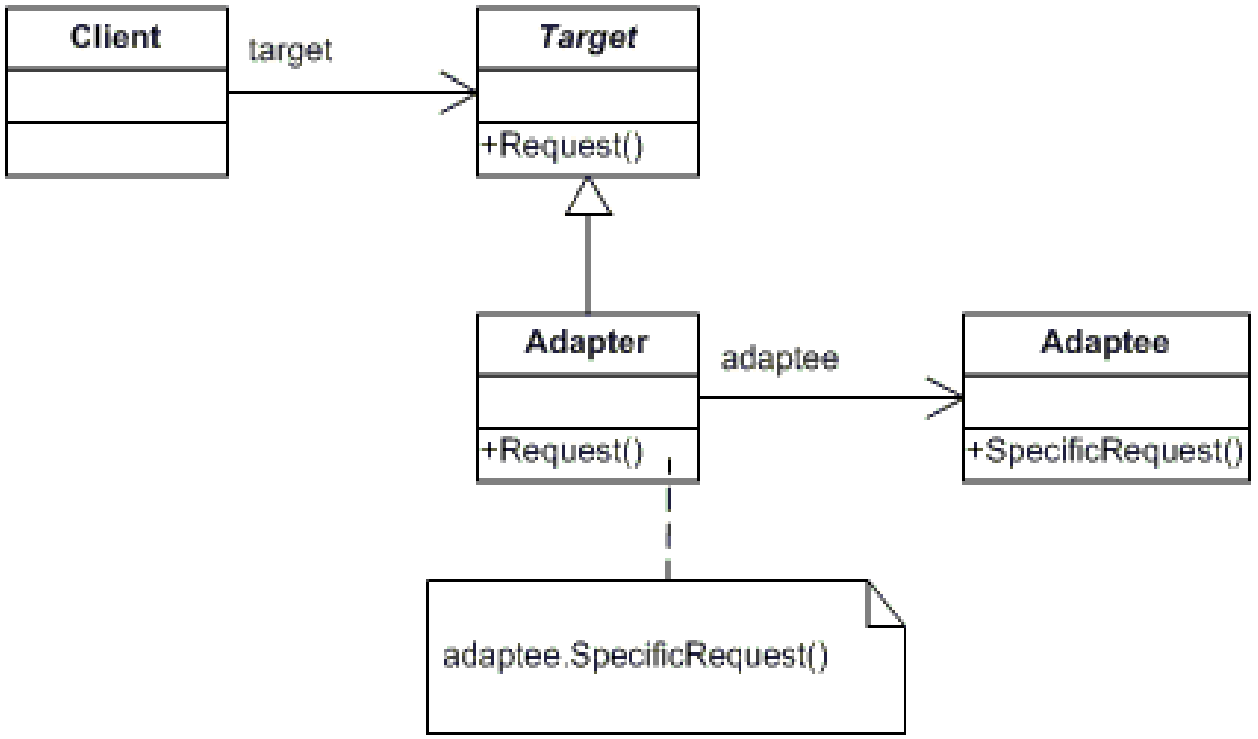
Association

Dependency

Structural patterns

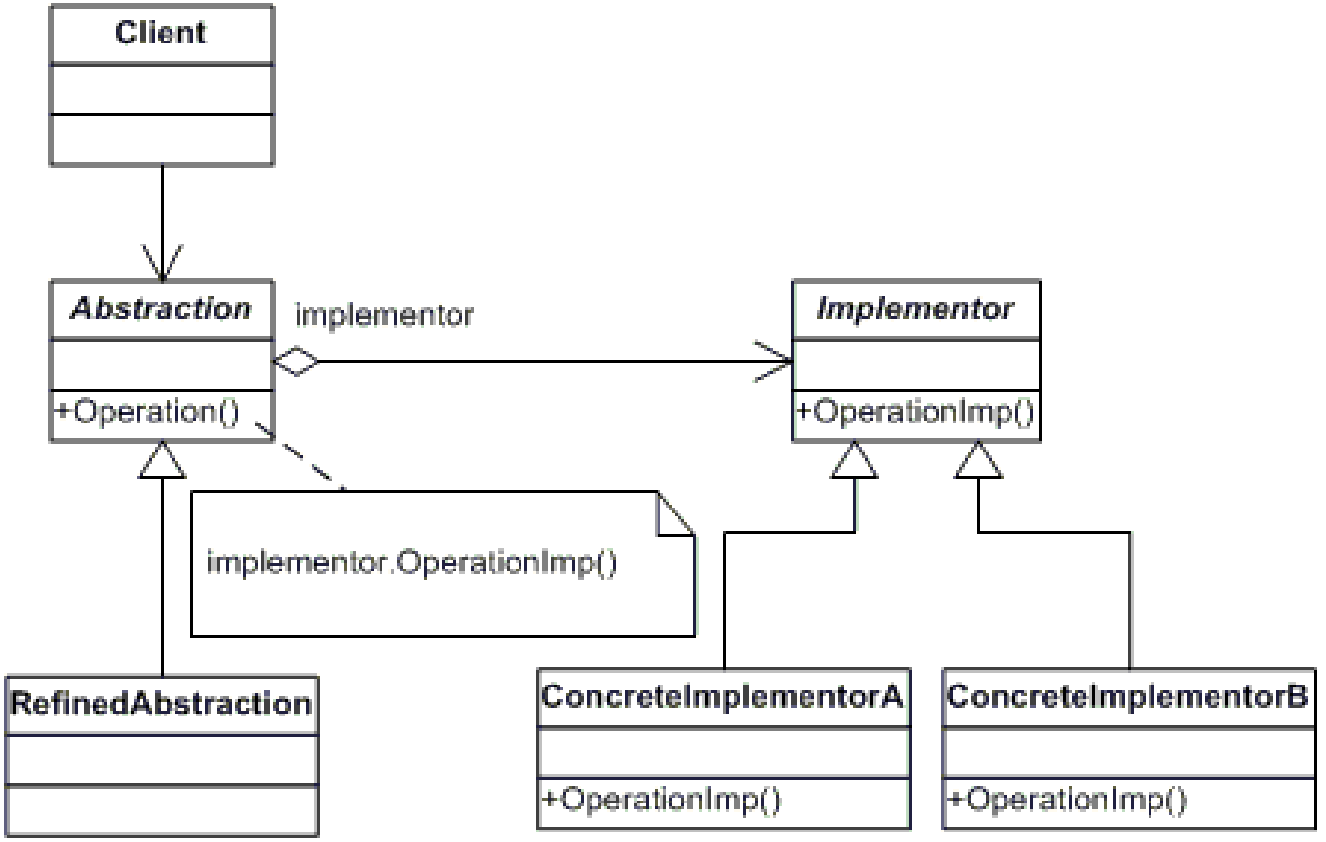
Combine classes and object to form larger structures

Adapter



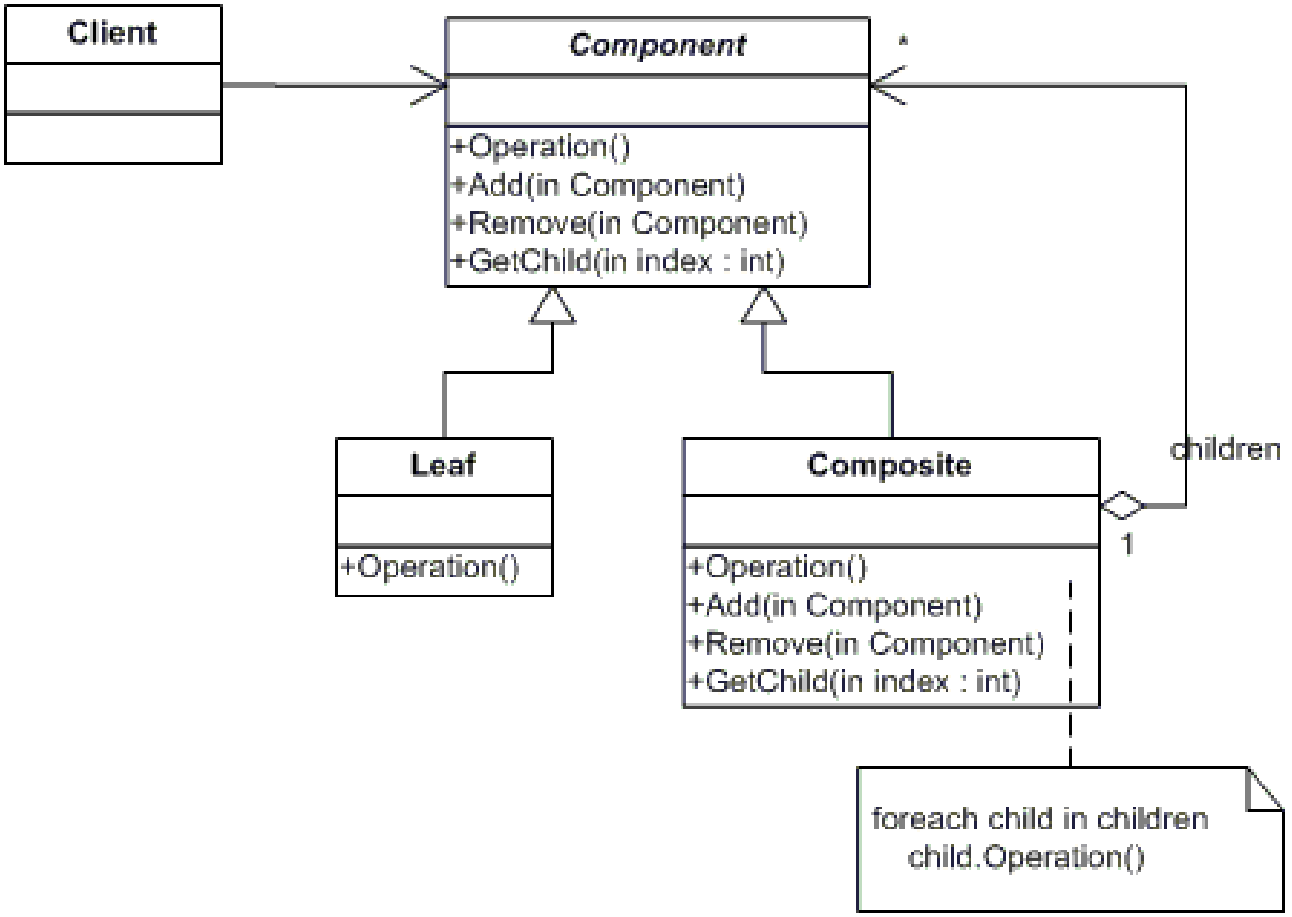
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces

Bridge



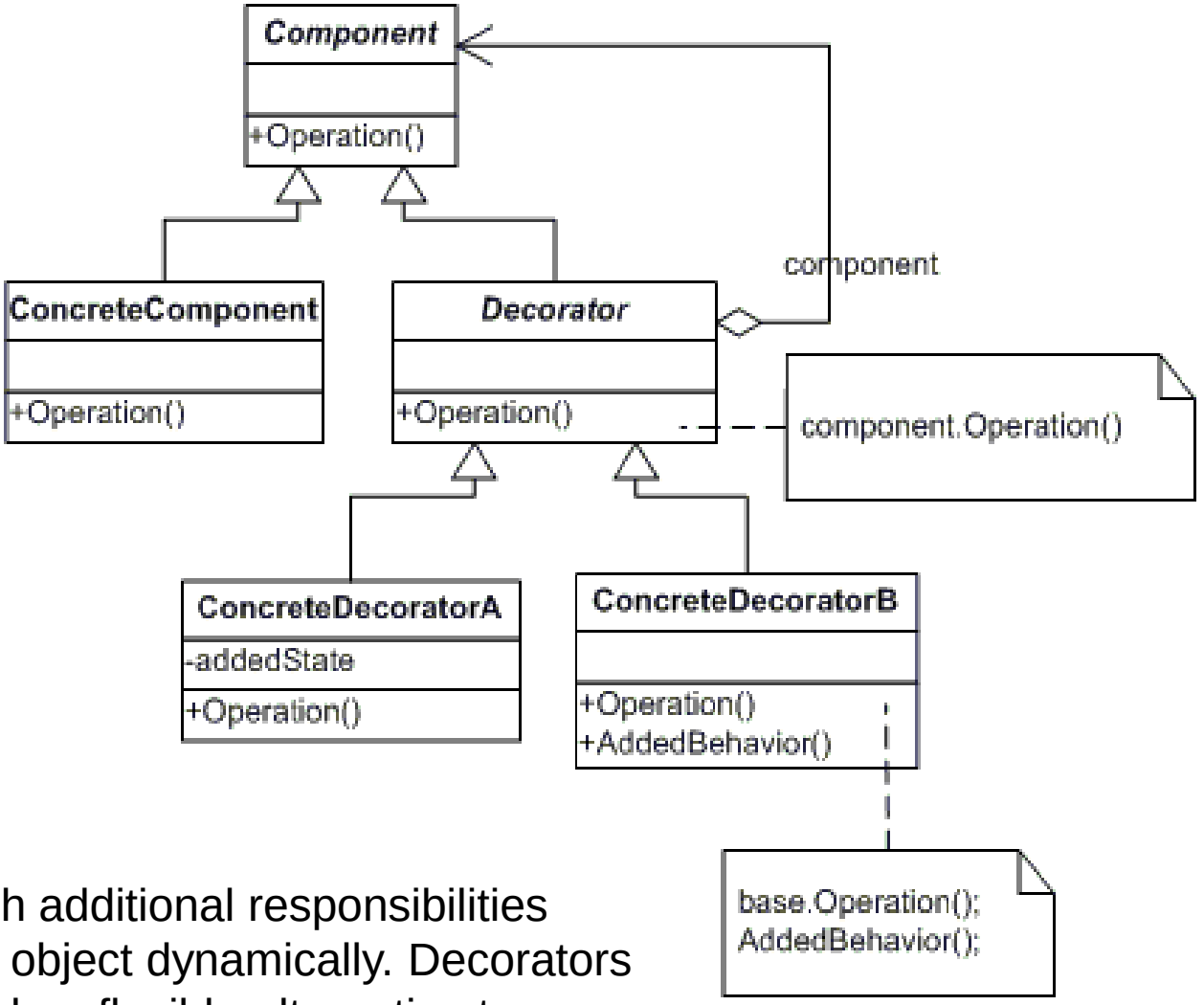
Decouple and abstractation from its implementation so that the two can vary independently.

Composite



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

Decorator

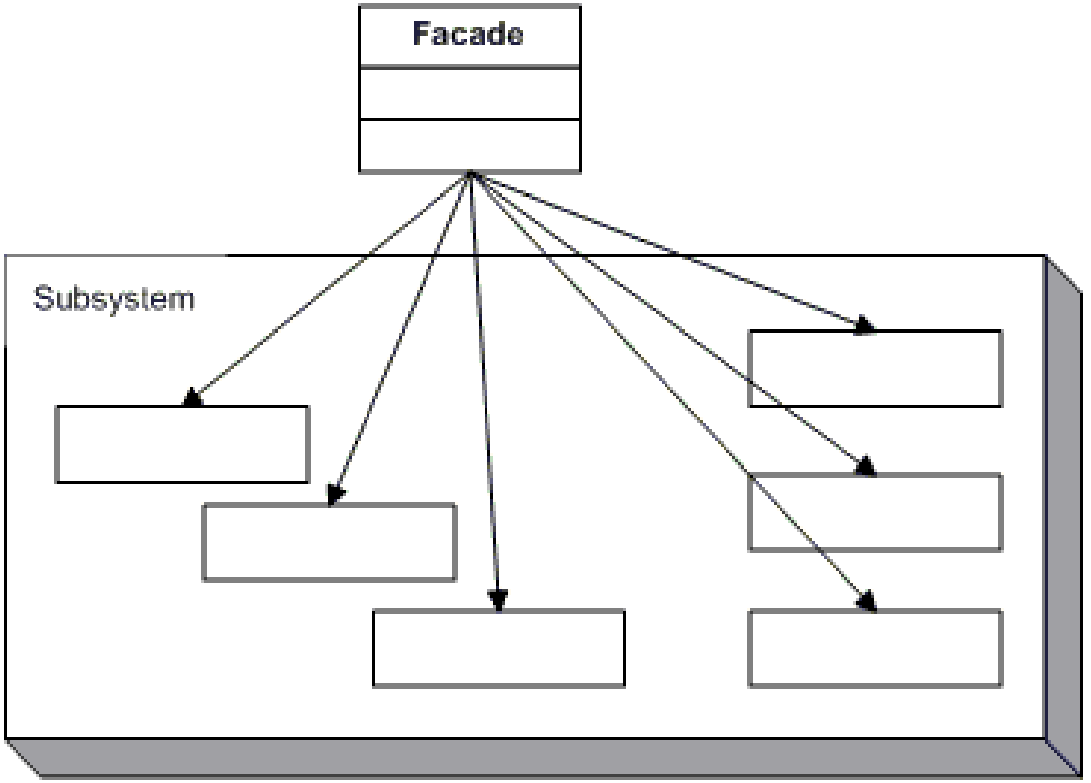


Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Problem

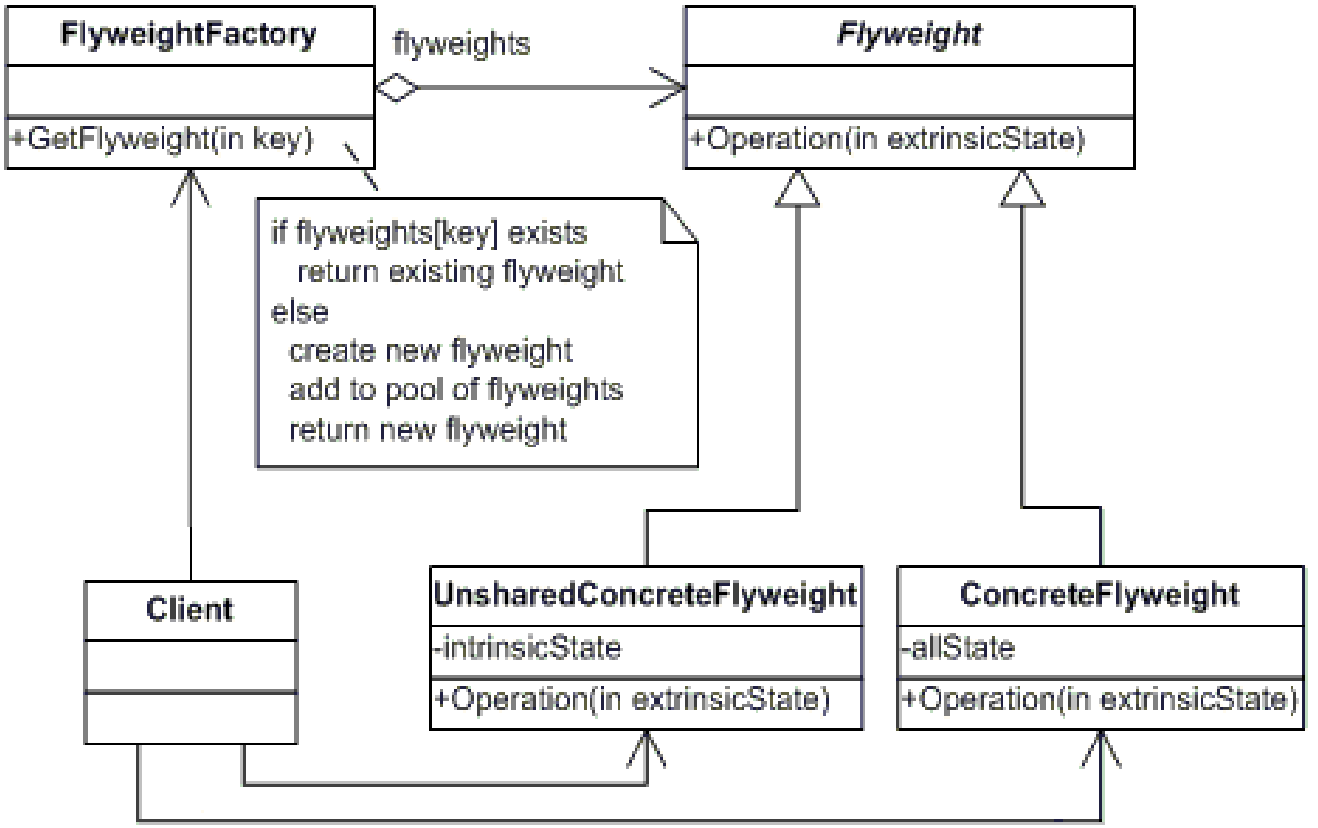
- Purchase has to be approved
  - < 10000 it can be approved by Director
  - < 25000 it can be approved by vice president
  - Else it must be approved by president
- For some purchases only vice president and president can approve
- For some purchases only director and president can approve

Facade



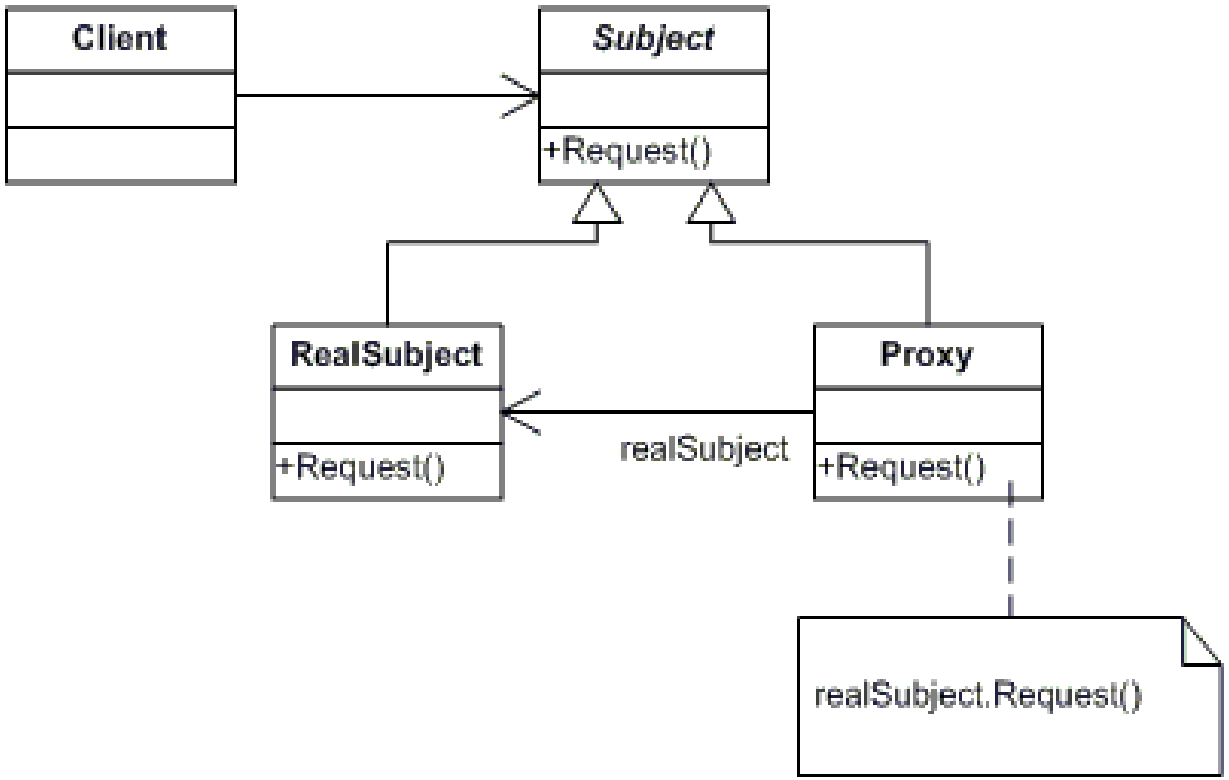
Privide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Flyweight



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

Proxy

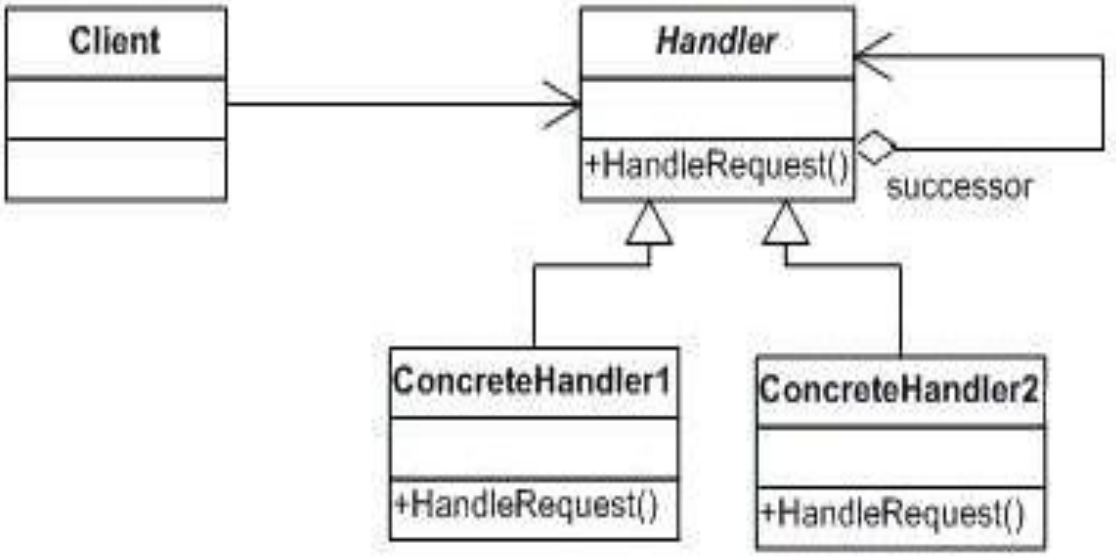


Provide a surrogate or placeholder for another object to control access to it.

Behavorial Patterns

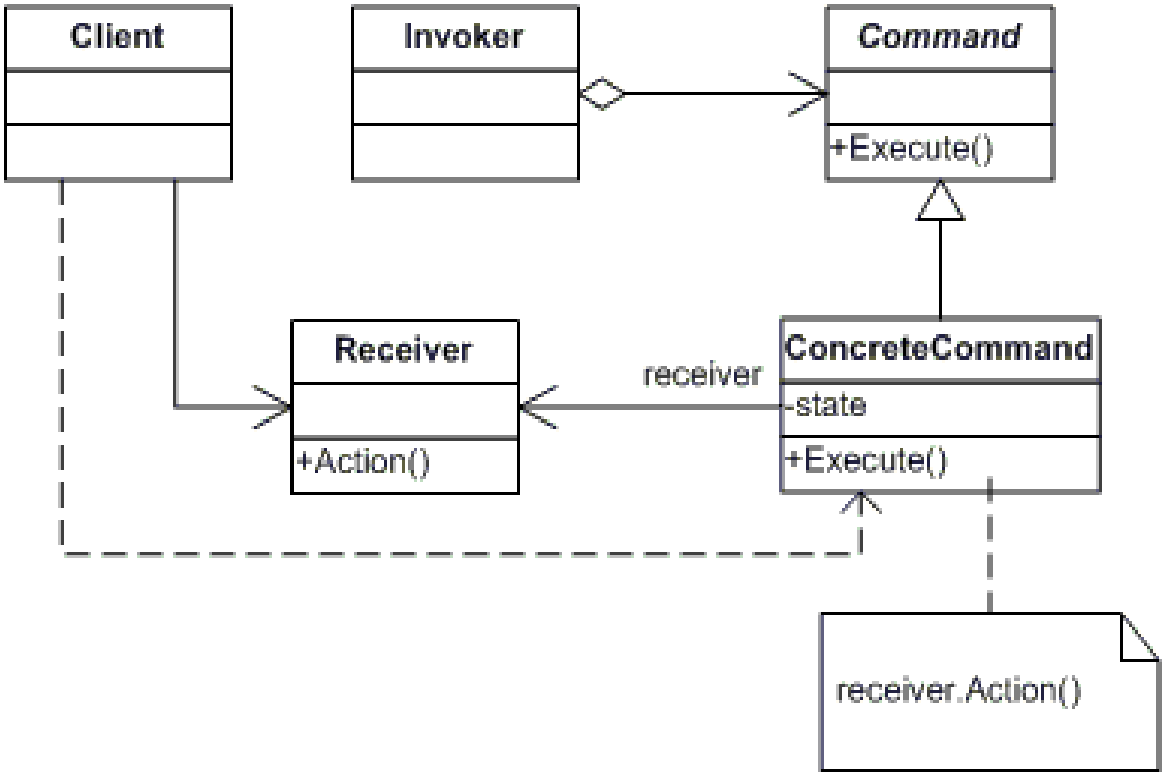
Algorithms and the assignment of responsibilities between objects

Chain of responsability



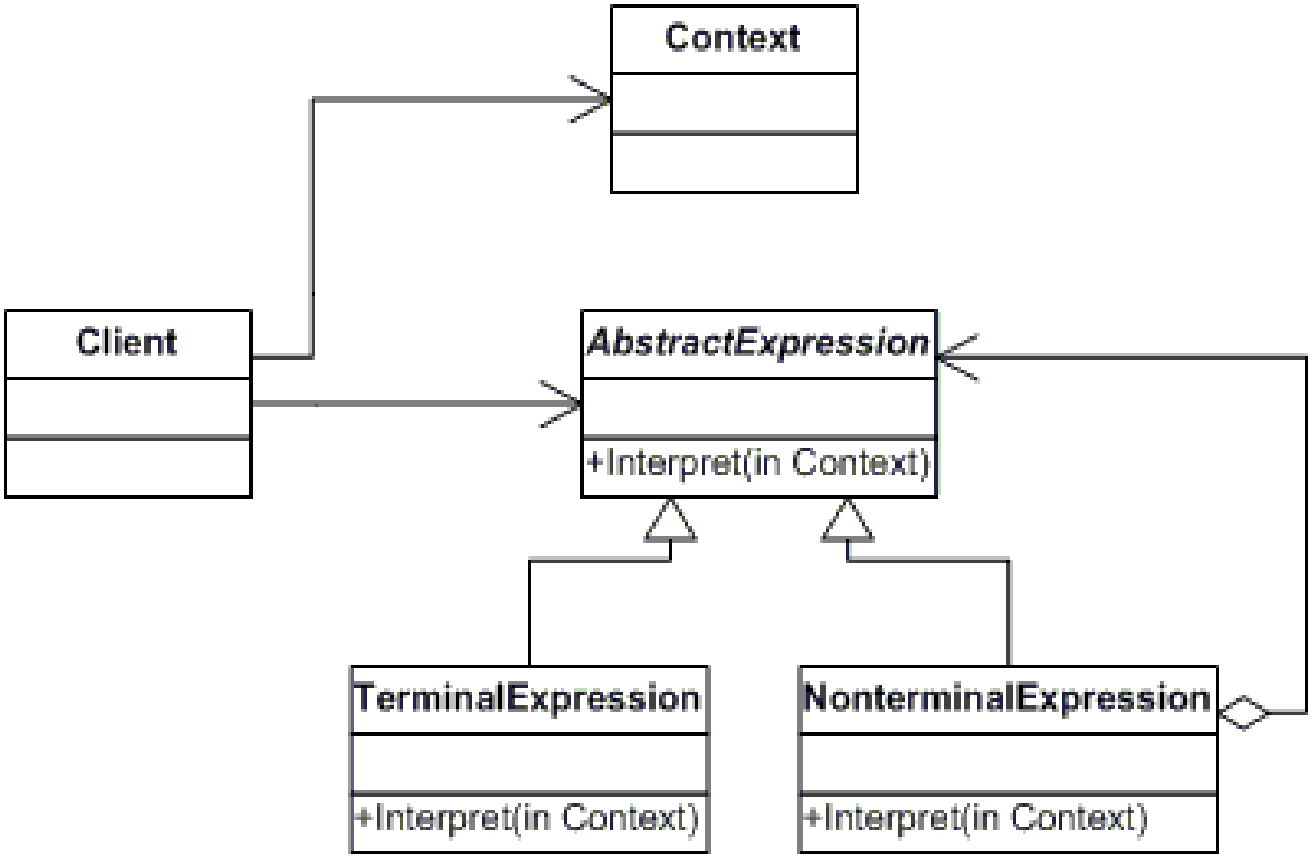
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command



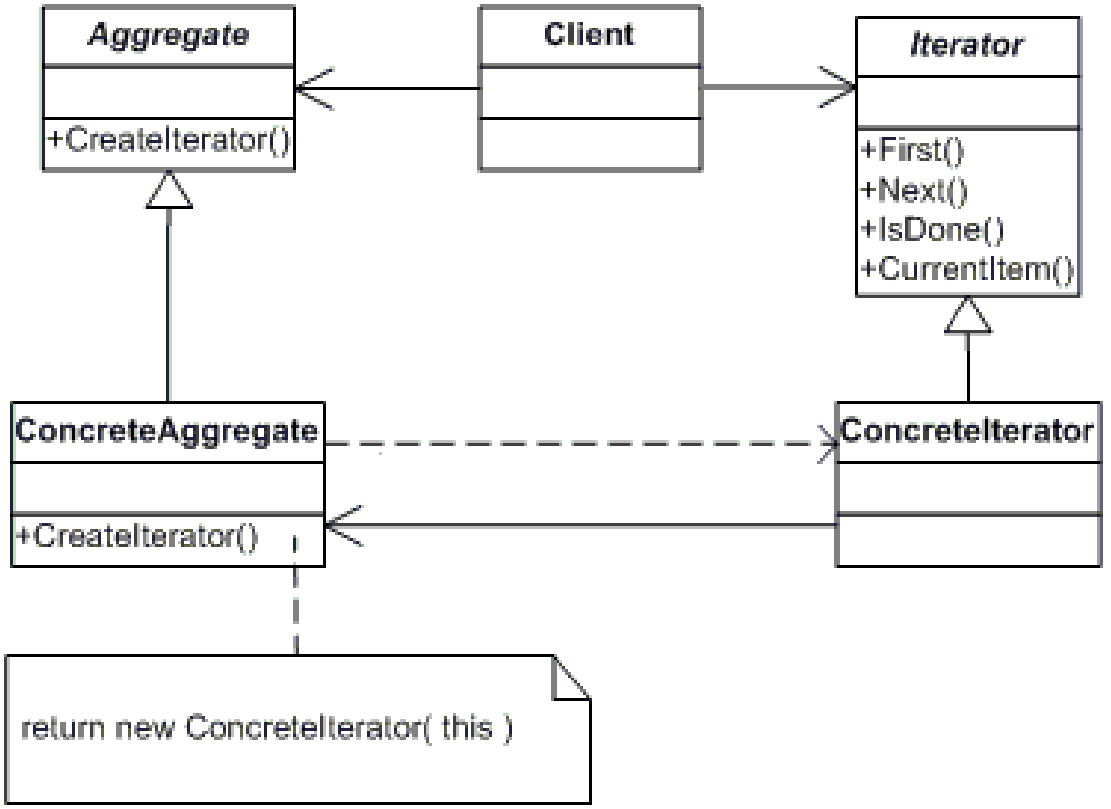
Encapsulate request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

Interpreter



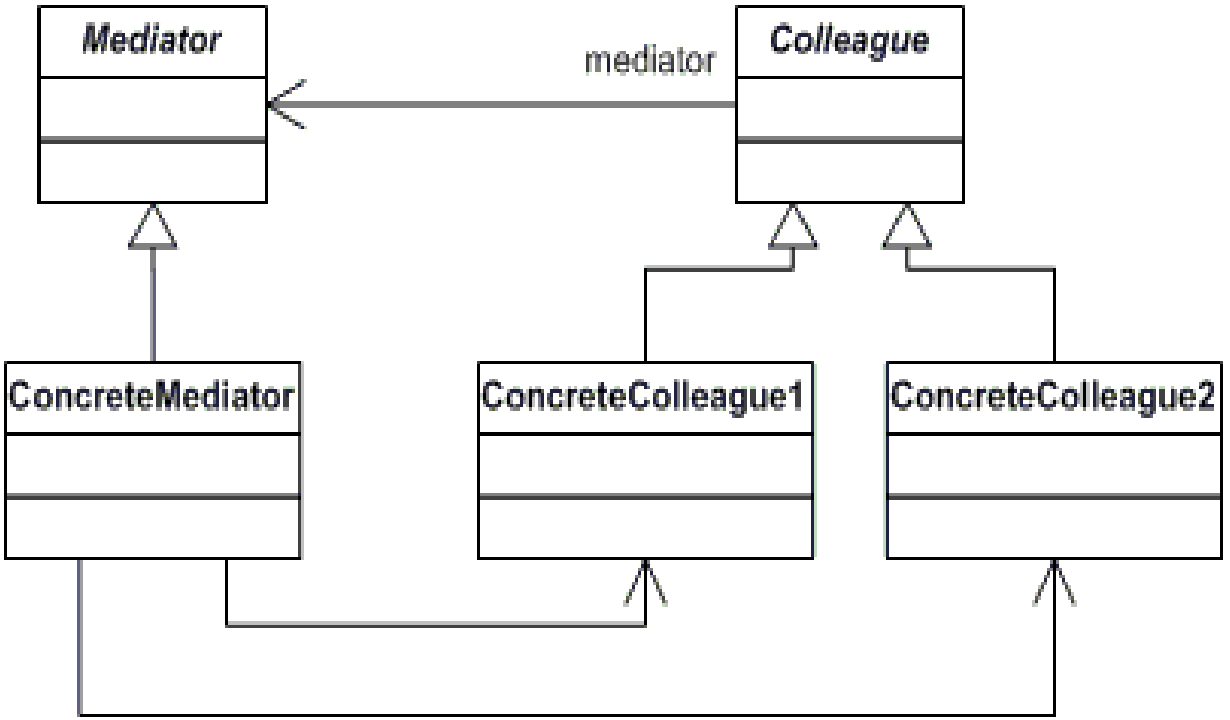
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator



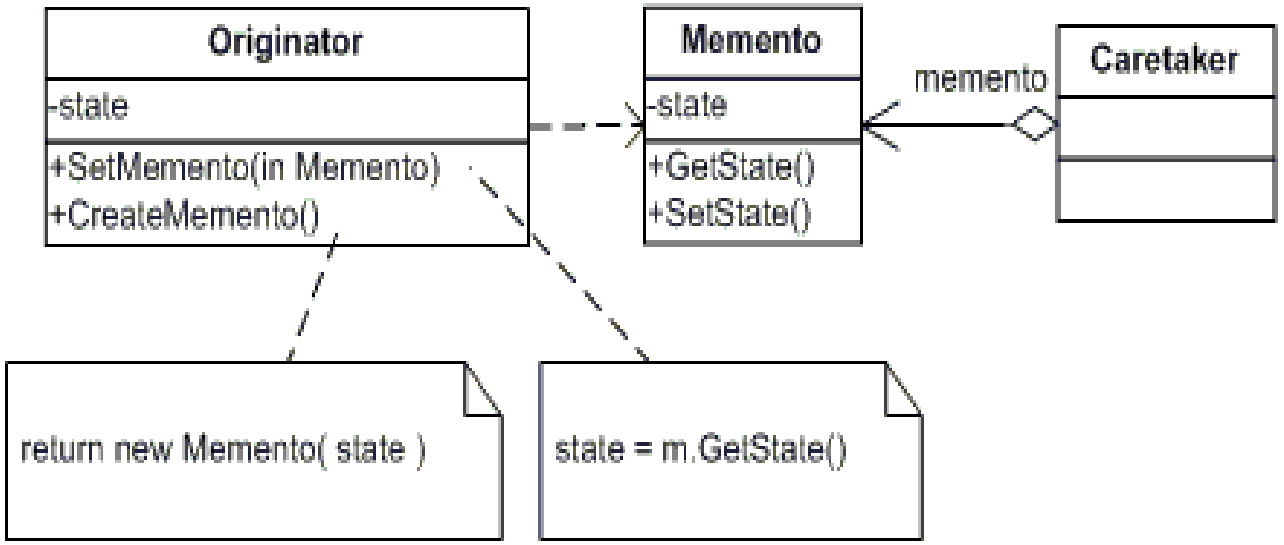
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator



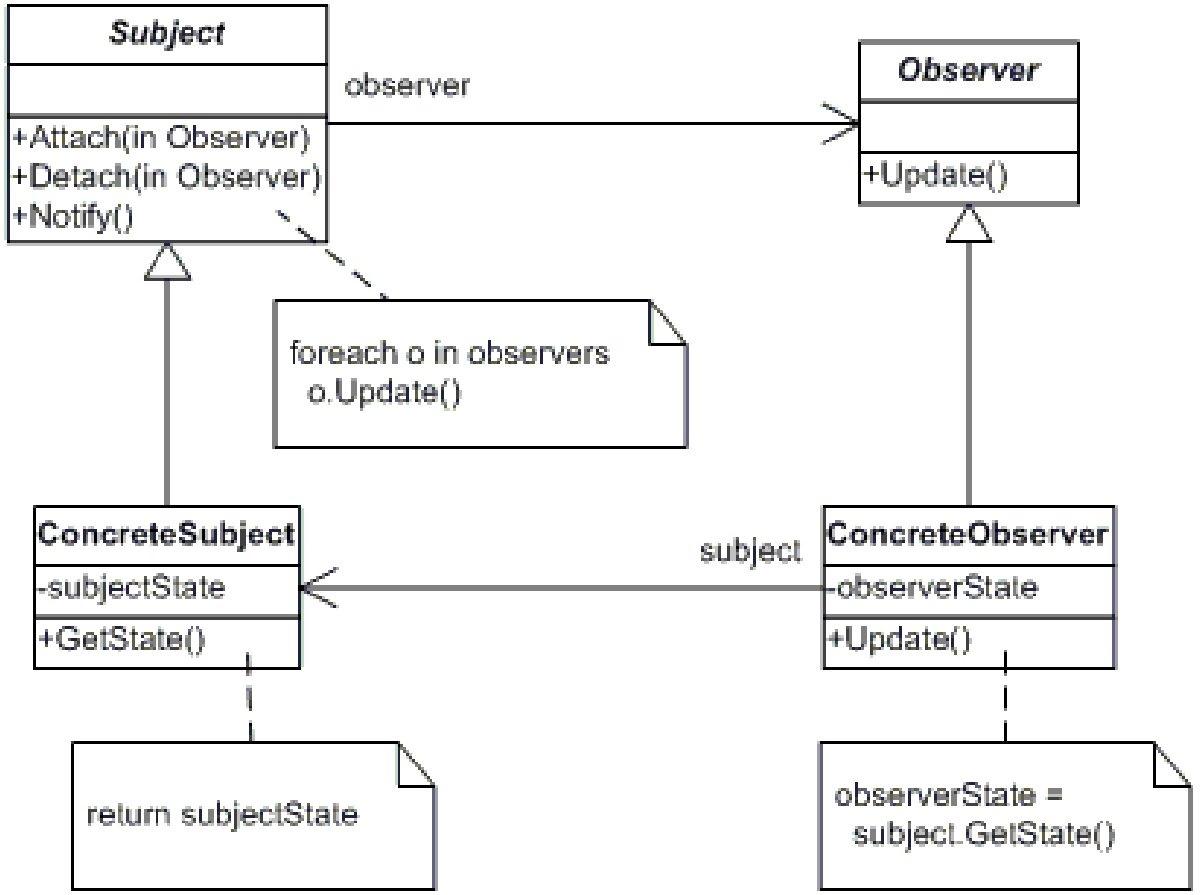
Define an object that encapsulates how a set of objects interact.
Mediator promotes loose coupling by keeping objects from referring
to each other explicitly, and it lets you vary their interaction independently.

Memento



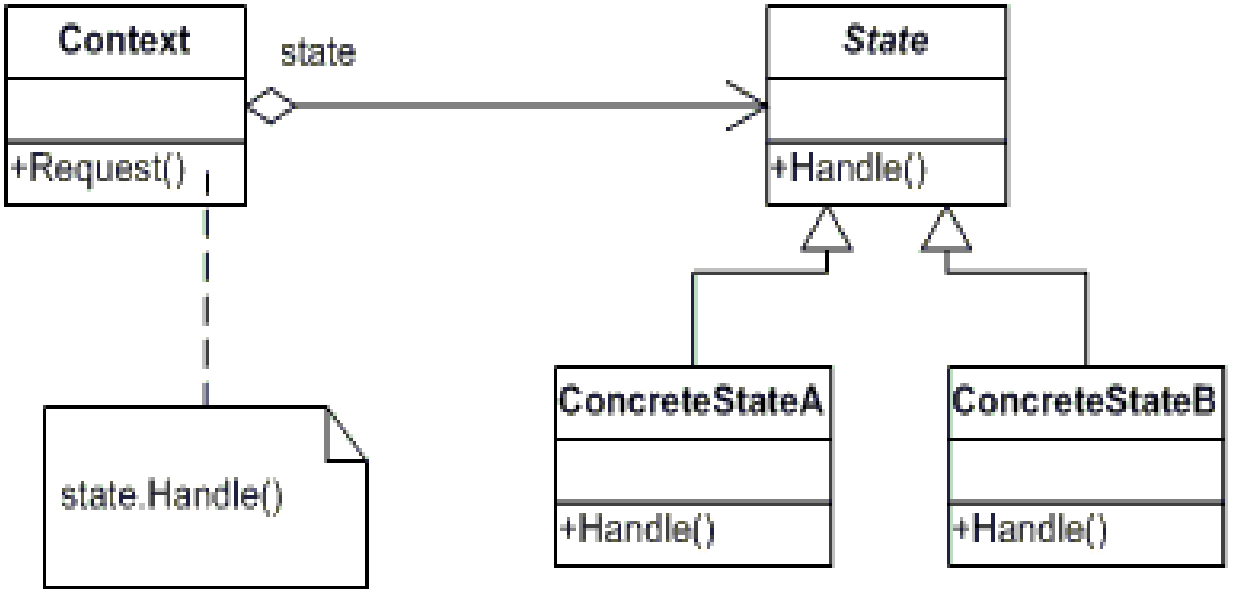
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Observer



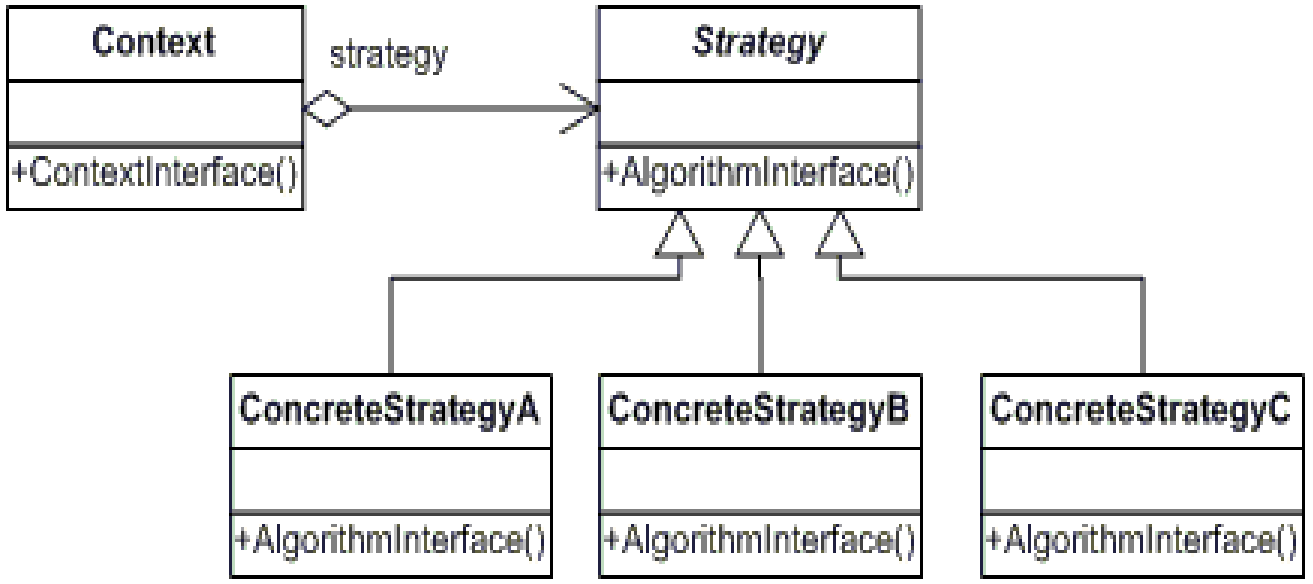
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

State



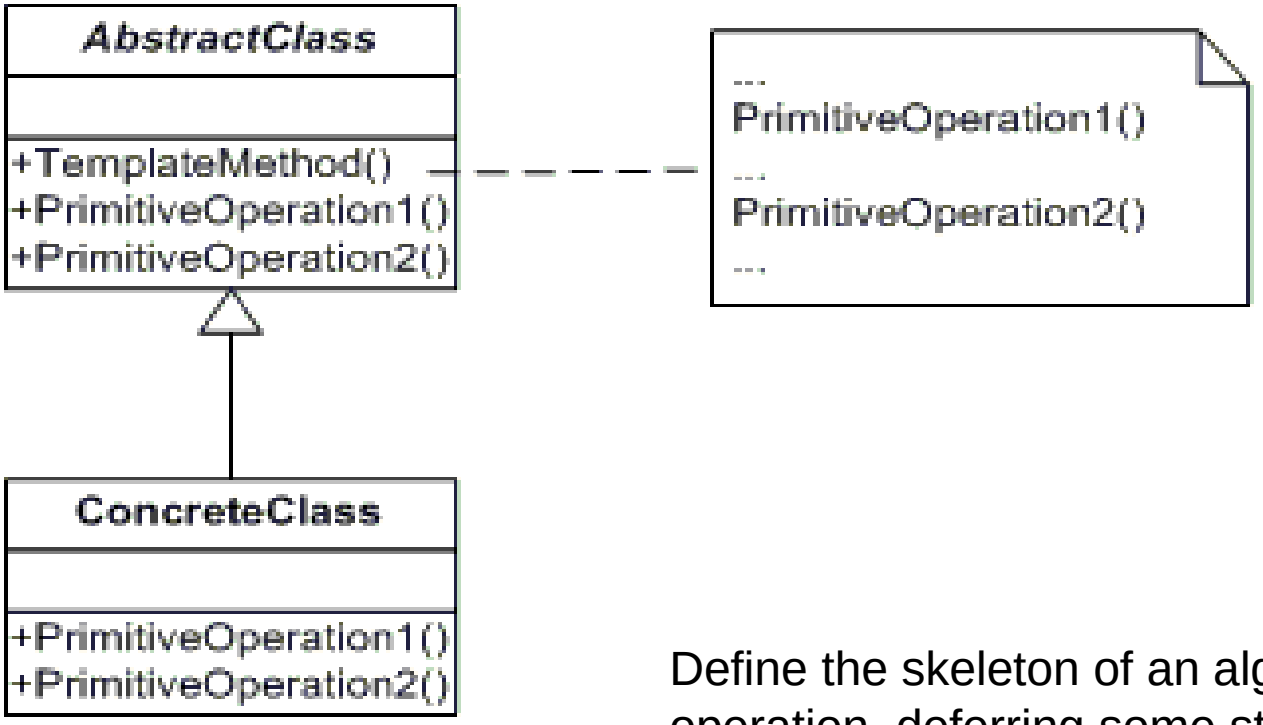
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy



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

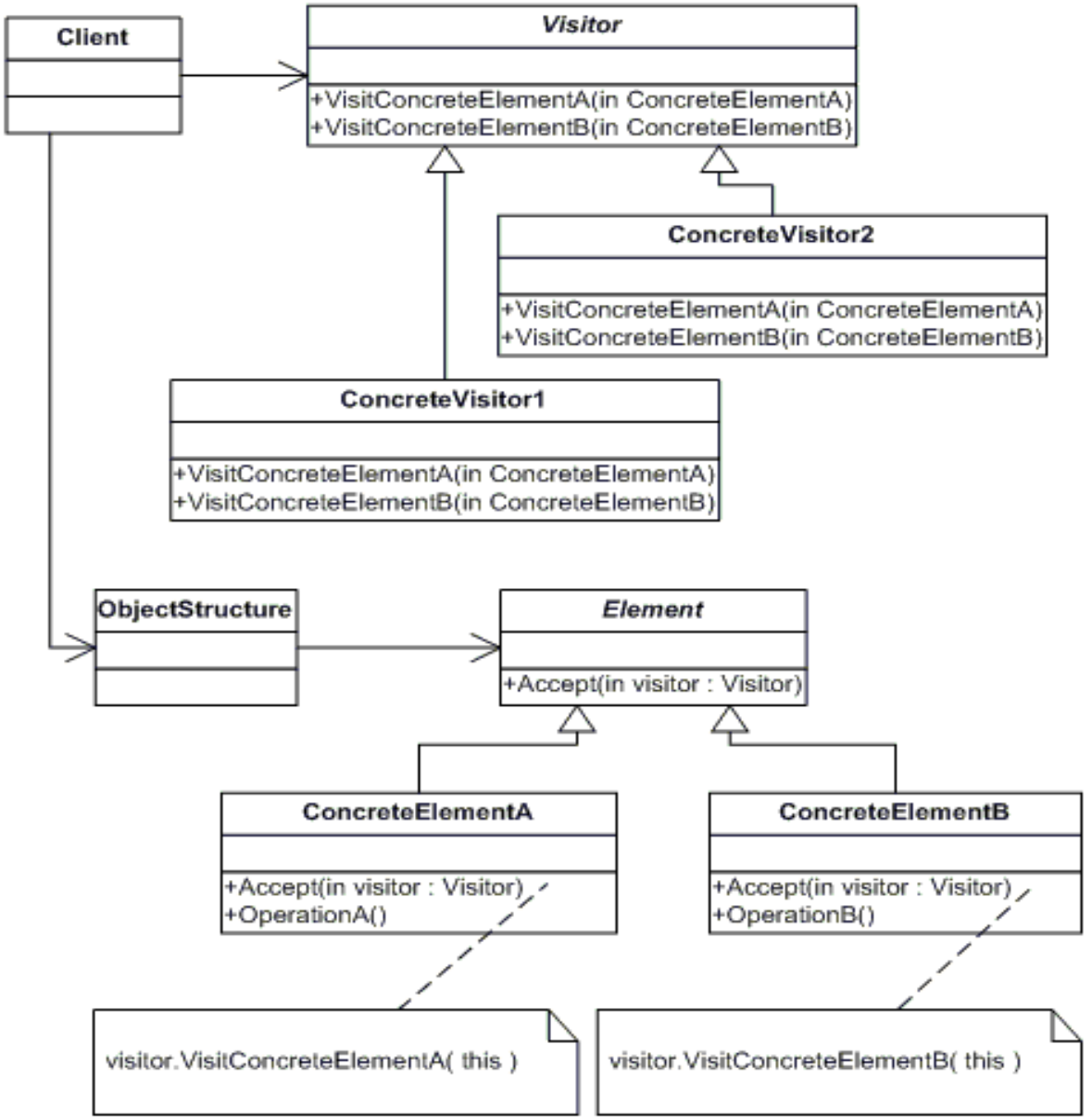
Template method



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.

Visitor

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.



And…

- Microsoft Patterns & Practices: General principles to us the .NET framework and MS Servers
- Java blueprints
- Enterprise Design patterns (Martin Fowler)
- GRASP: General Responsibility Assignment Software Patterns/Principles (Craig Larman)
  - Knowing patterns
  - Doing patterns

Grasp

- Information Expert: Which class is responsible for what
- Creator: Who is responsible for the creation of an object
- Controller: communication between UI and service layer
- Low Coupling: classes are independent
- High Cohesion: class does one thing

Grasp

- Polymorfisme: implementation belongs to a type
- Pure Fabrication: non-business domain class to implement a design principle
- Indirection: go-between class to connect two classes
- Protected Variations: Hide changes using interfaces or polymorfisme

Conclusion

- Design patterns are no “silver bullet”
- “Design patterns” is no cookbook
- Good developers use Design patterns intuitively
- Design patterns are a way of communicating