Object Oriented Design Patterns

Definition

In software engineering, a design pattern is a general solution to a common problem in software design. A design pattern isn't a finished design that can be transformed directly into code; it is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Algorithms are not thought of as design patterns, since they solve computational problems rather than design problems.

Patterns are devices that allow programs to share knowledge about their design. In our daily programming, we encounter many problems that have occurred, and will occur again. The question we must ask ourselves is how we are going to solve it this time. Documenting patterns is one way that you can reuse and possibly share the information that you have learned about how it is best to solve a specific program design problem. Why Use Patterns?

They have been proven. Patterns reflect the experience, knowledge and insights of developers who have successfully used these patterns in their own work.

They are reusable. Patterns provide a ready-made solution that can be adapted to different problems as necessary.

They are expressive. Patterns provide a common vocabulary of solutions that can express large solutions succinctly.

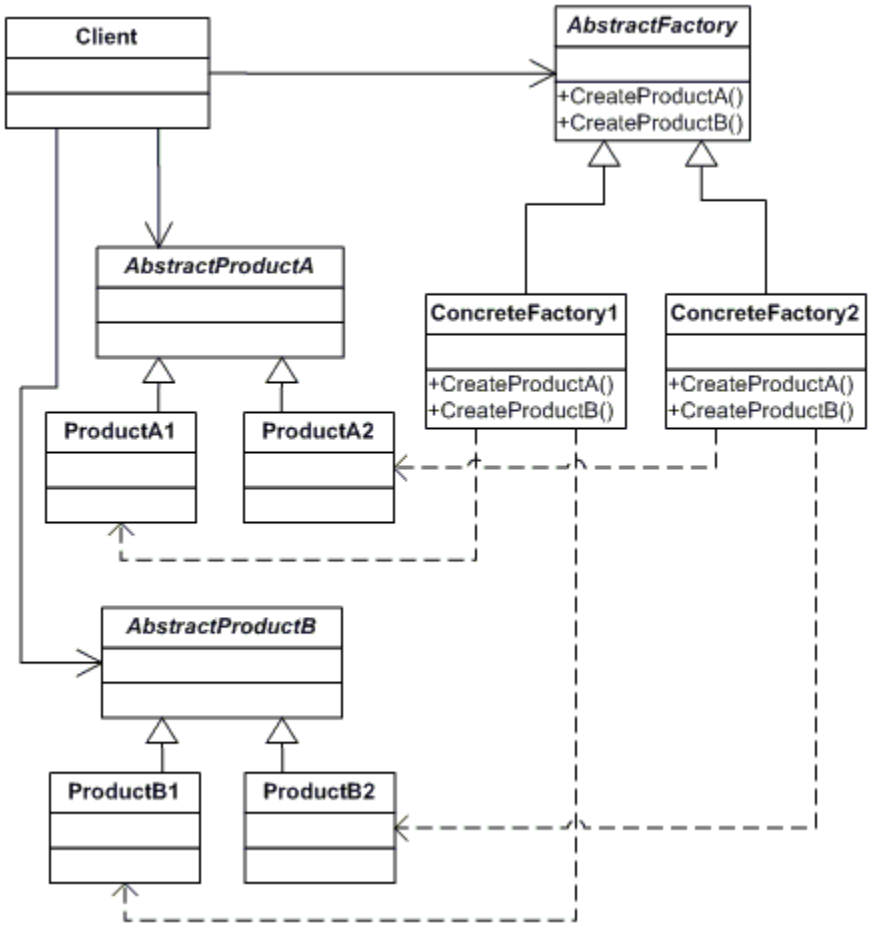
It is important remember that patterns do not guarantee success. A pattern description indicates when the pattern may be applicable, but only experience can provide understanding of when a particular pattern will improve a design.

The design patterns below are organized into three groups: Creational, Structural and Behavioural, based on the design issues they resolve.

Creational Patterns

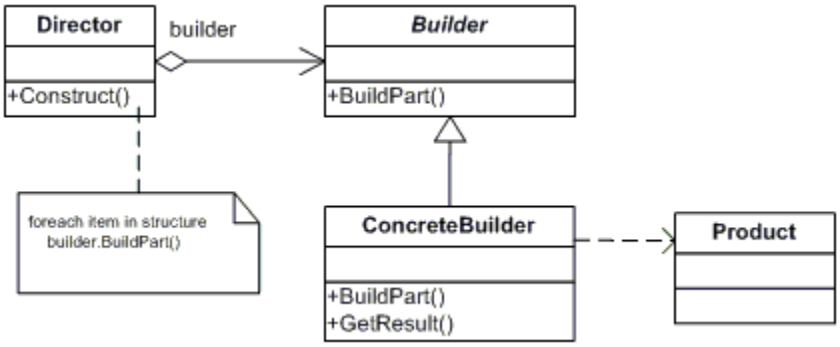
In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

Abstract Factory: Creates an instance of several families of classes
A software design pattern, the Abstract Factory Pattern provides a way to encapsulate a group of individual factories that have a common theme. In normal usage, the client software would create a concrete implementation of the abstract factory and then use the generic interfaces to create the concrete objects that are part of the theme. The client does not know (nor care) about which concrete objects it gets from each of these internal factories since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from its general usage.



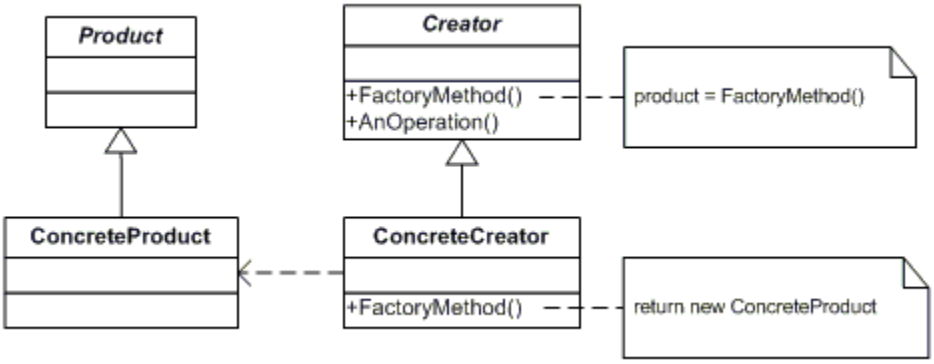
Builder: Separates object construction from its representation

A software design pattern, the builder pattern is used to enable the creation of a variety of complex objects from one source object. The source object may consist of a variety of parts that contribute individually to the creation of each complex object through a set of common interface calls of the Abstract Builder class.

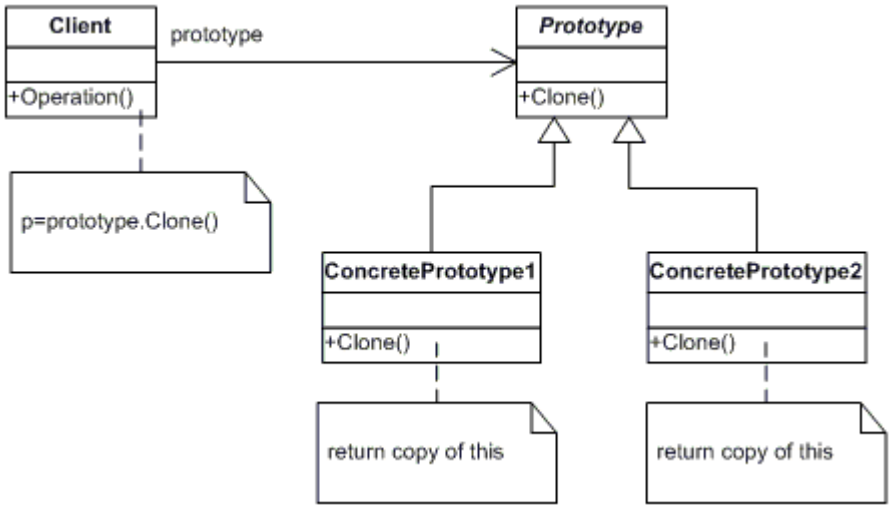


Factory Method: Creates an instance of several derived classes

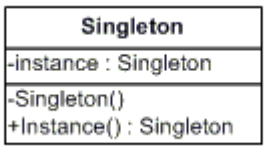
The Factory Method pattern is an object-oriented design pattern. Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. Factory Method, one of the patterns from the Design Patterns book, handles this problem by defining a separate method for creating the objects, which subclasses can then override to specify the type of product that will be created.



Prototype: A fully initialized instance to be copied or cloned
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

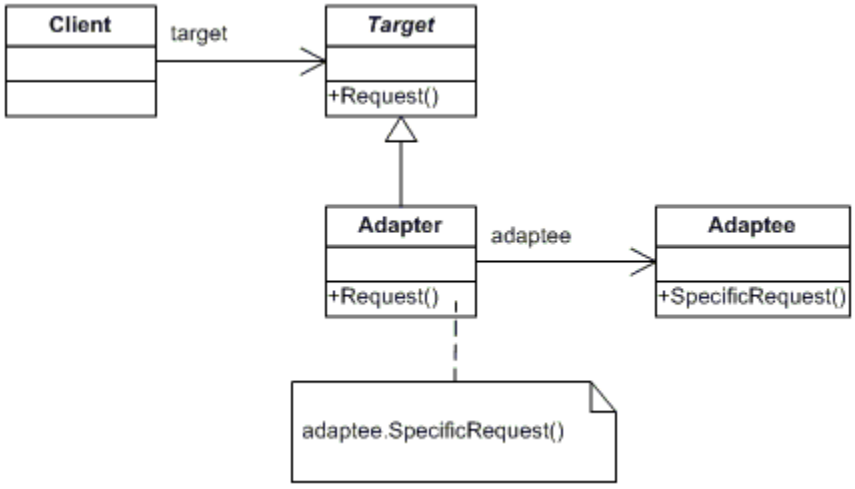


Singleton: A class of which only a single instance can exist
In computer science, the singleton design pattern is designed to restrict instantiation of a class to one (or a few) objects. This is useful when exactly one object is needed to coordinate actions across the system. Sometimes it is generalized to systems that operate more efficiently when only one or a few objects exist. It is also considered an Anti-pattern since it is often used as a politically correct term for global variable -and hence frowned upon.

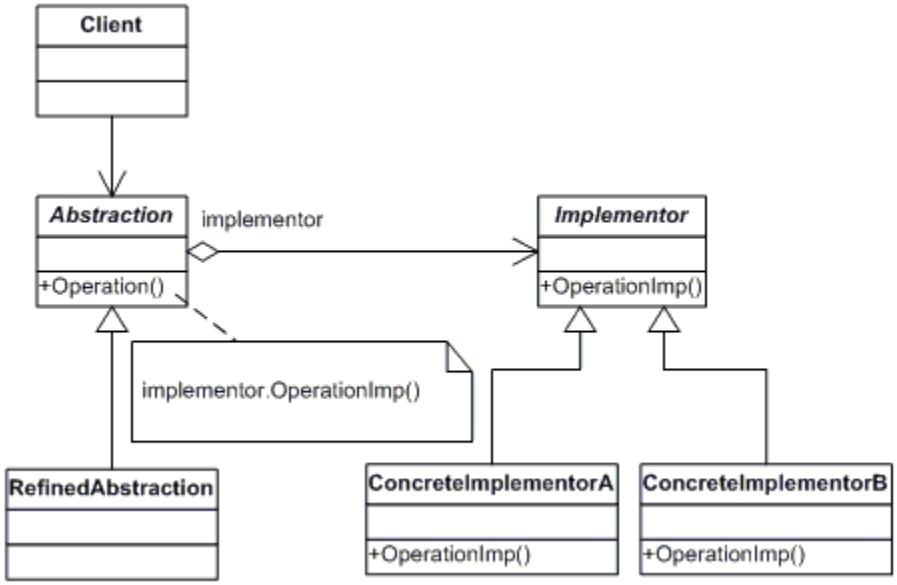


Structural Patterns

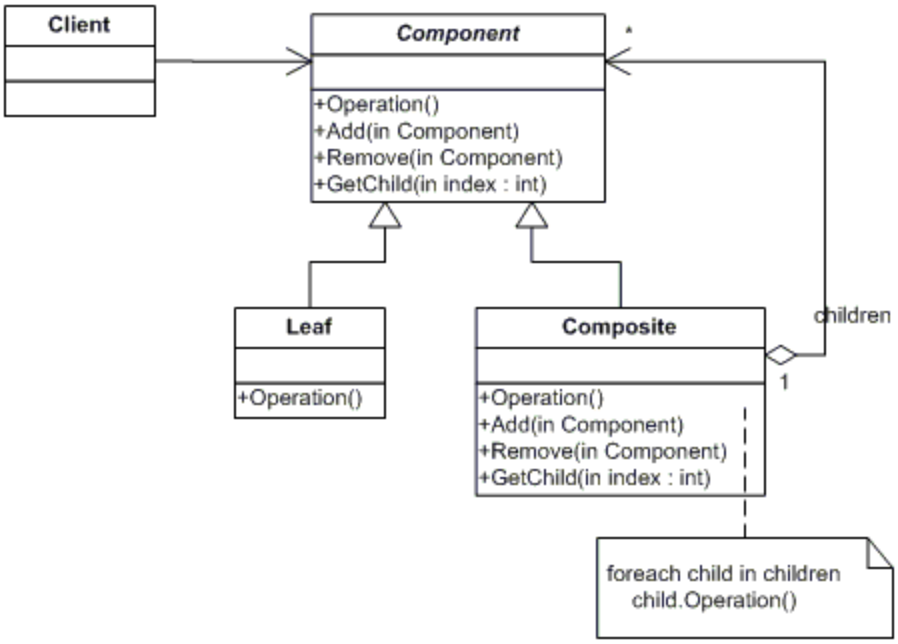
Adapter: Match interfaces of different classes



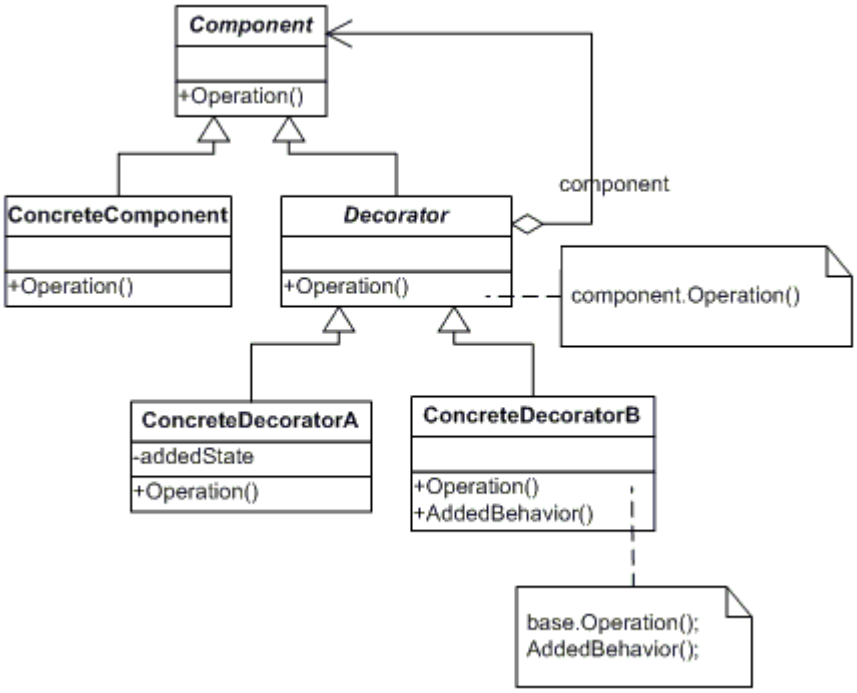
Bridge: Separates an object's interface from its implementation



Composite: A tree structure of simple and composite objects



Decorator: Add responsibilities to objects dynamically



Facade: A single class that represents an entire subsystem

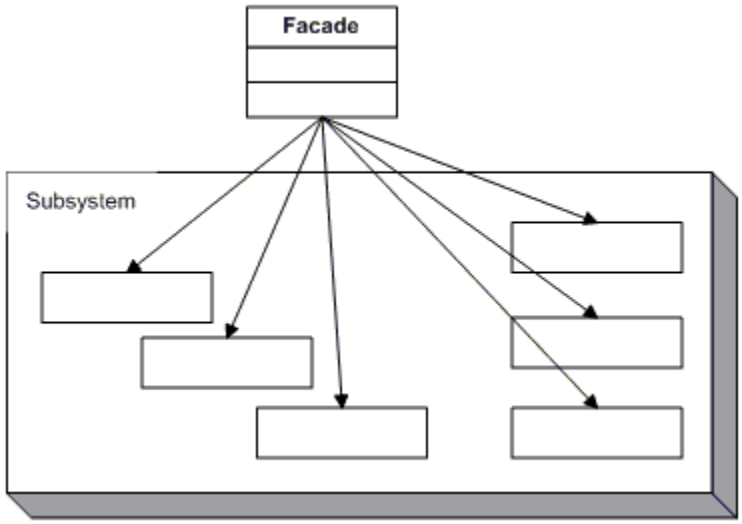
The facade pattern can make the task of accessing a large number of modules much simpler by providing an additional interface layer. When designing good programs, programmers usually attempt to avoid excess coupling between module/classes. Using this pattern helps to simplify much of the interfacing that makes large amounts of coupling complex to use and difficult to understand. In a nutshell, this is accomplished by creating a small collection of classes that have a single class that is used to access them, the facade.

Classes

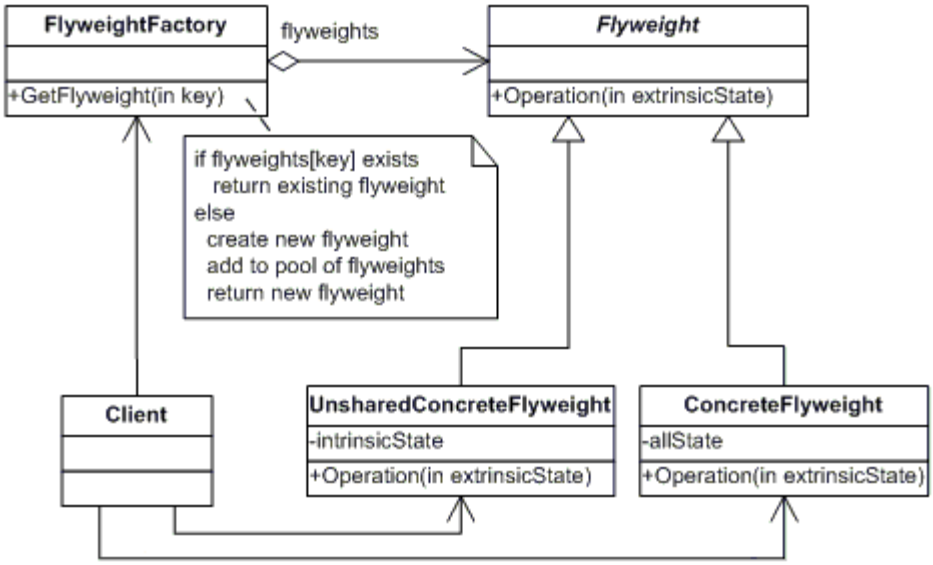
There can be any number of classes involved in this "facaded" system, but I would think that the minimum is four classes. One client, the facade, and the classes underneath the facade. In a typical situation, the facade would have a limited amount of actual code, making calls to lower layers most of the time.

Advantages/Disadvantages

As stated before, the primary advantage to using the facade is the make the interfacing between many modules or classes more managable. One possible disadvantage to this pattern is that you may lose some functionality contained in the lower level of classes, but this depends on how the facade was designed.



Flyweight: A fine-grained instance used for efficient sharing

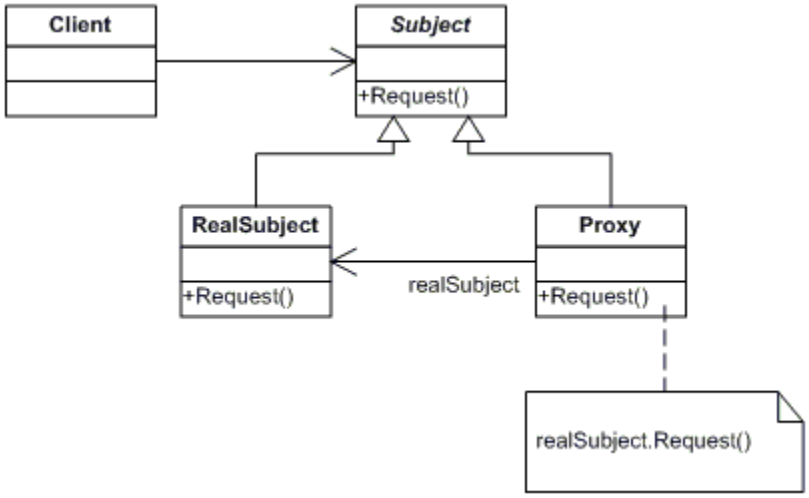


Proxy: An object representing another object

A proxy, in its most general form, is a class functioning as an interface to another thing. The other thing could be anything, a network connection, a large object in memory, a file, or other resource that is expensive or impossible to duplicate.

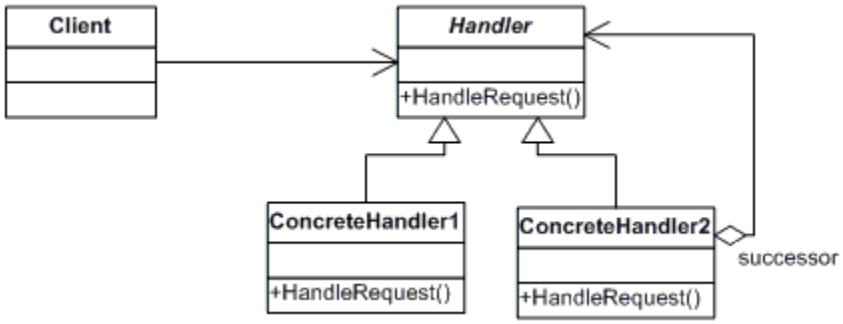
A well-known example of the proxy pattern is a reference counting pointer object, also known as an auto pointer.

The proxy pattern can be used in situations where multiple copies of a complex object must exist. In order to reduce the application's memory footprint in such situations, one instance of the complex object is created, and multiple proxy objects are created, all of which contain a reference to the single original complex object. Any operations performed on the proxies are forwarded to the original object. Once all instances of the proxy are out of scope, the complex object's memory may be deallocated.

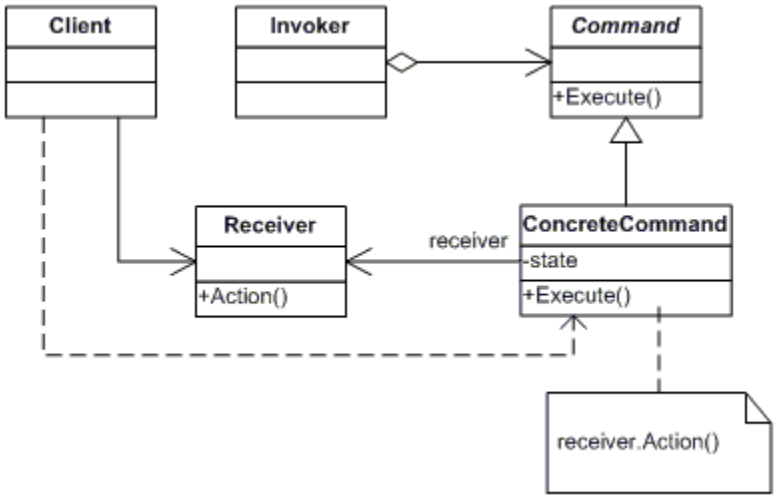


Behavioral Patterns

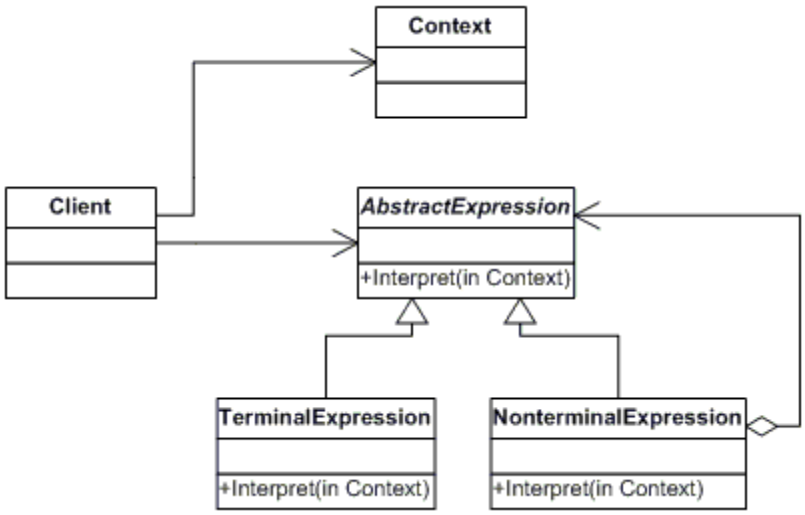
Chain of Responsibility: A way of passing a request between a chain of objects



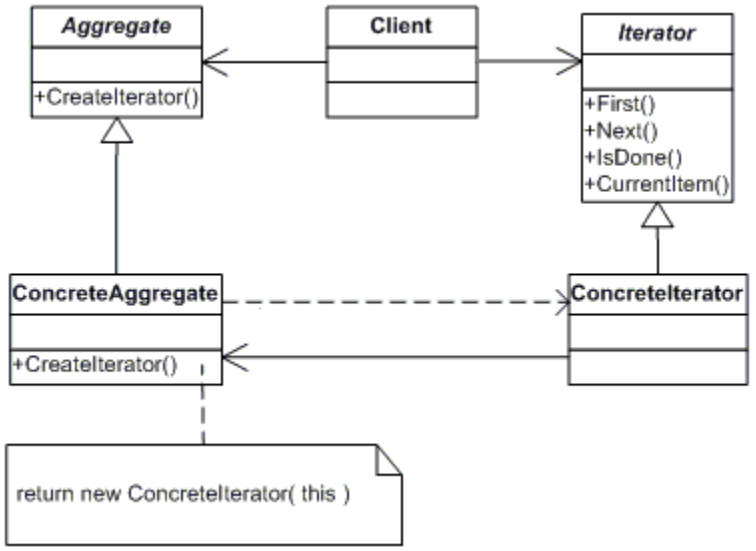
Command: Encapsulate a command request as an object



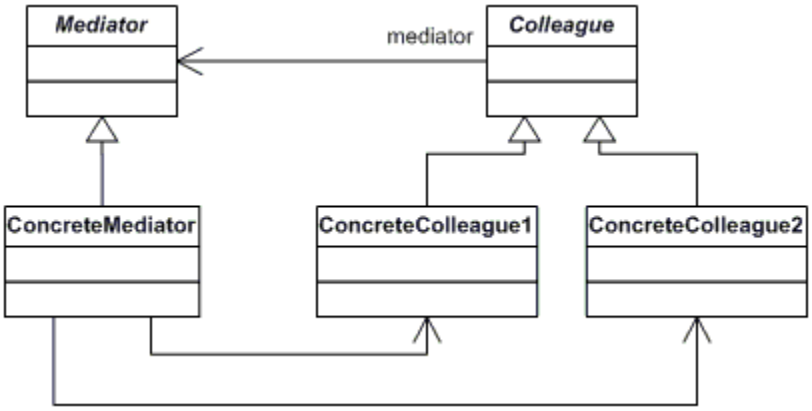
Interpreter: A way to include language elements in a program



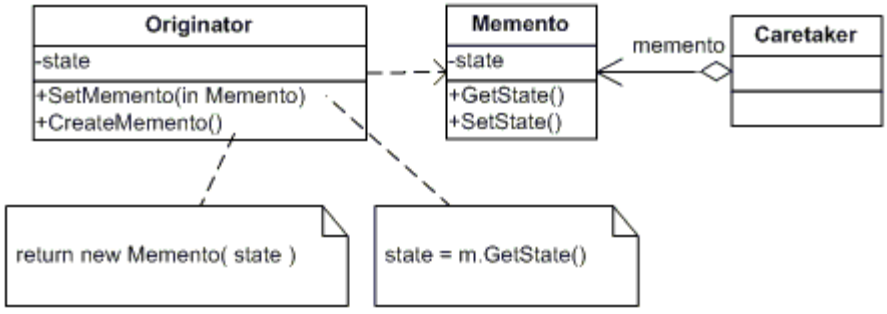
Iterator: Sequentially access the elements of a collection



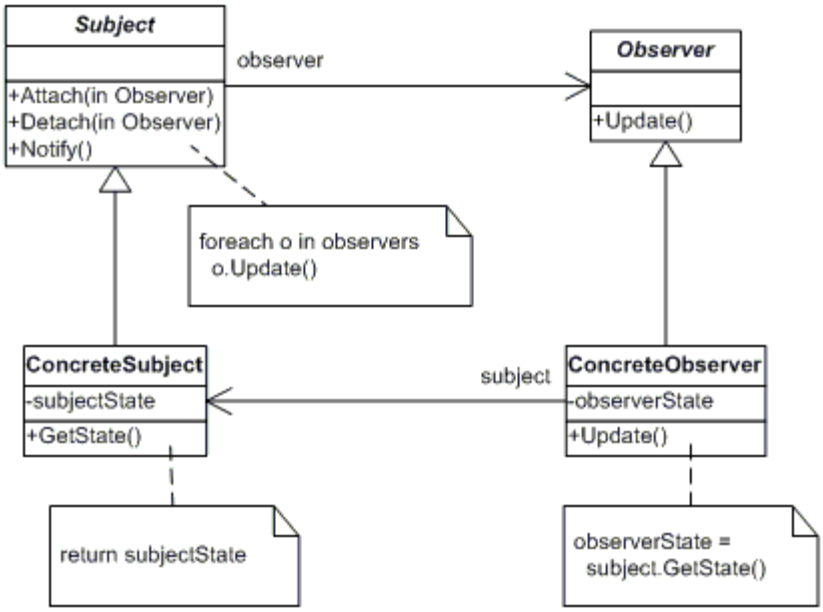
Mediator: Defines simplified communication between classes



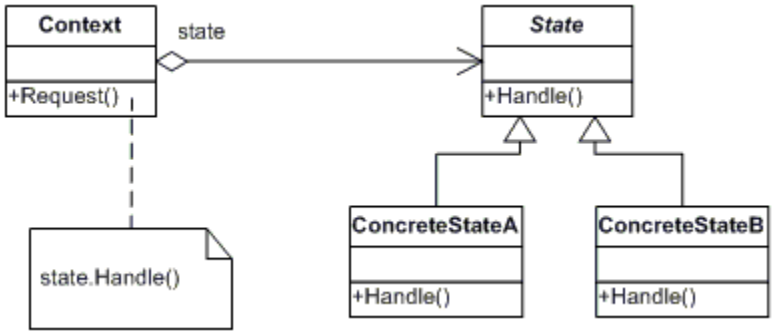
Memento: Capture and restore an object's internal state



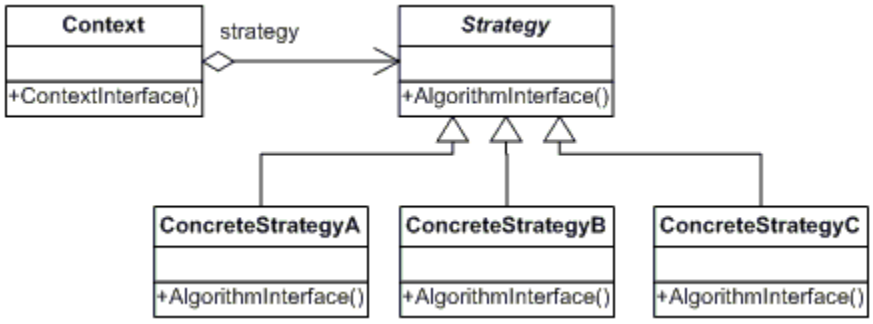
Observer: A way of notifying change to a number of classes



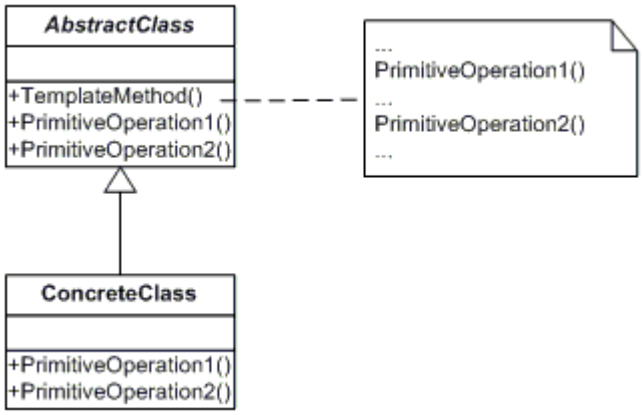
State: Alter an object's behavior when its state changes



Strategy: Encapsulates an algorithm inside a class



Template Method: Defer the exact steps of an algorithm to a subclass



Visitor: Defines a new operation to a class without change