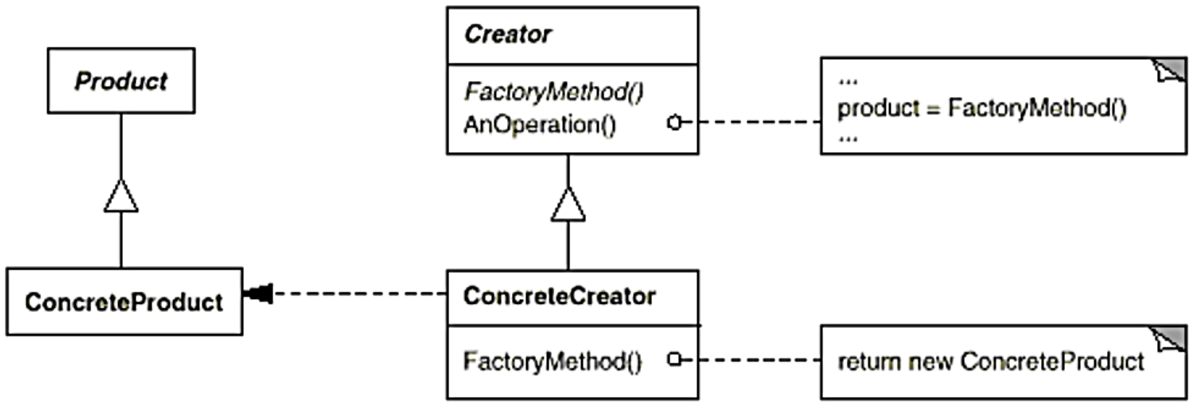
Design Patterns

This document concisely describes each design pattern and presents the UML diagram that describes it.

Written by Shlomi Ben Shushan, based on refactoring.guru website and Dr. Eli Khalasche's lectures

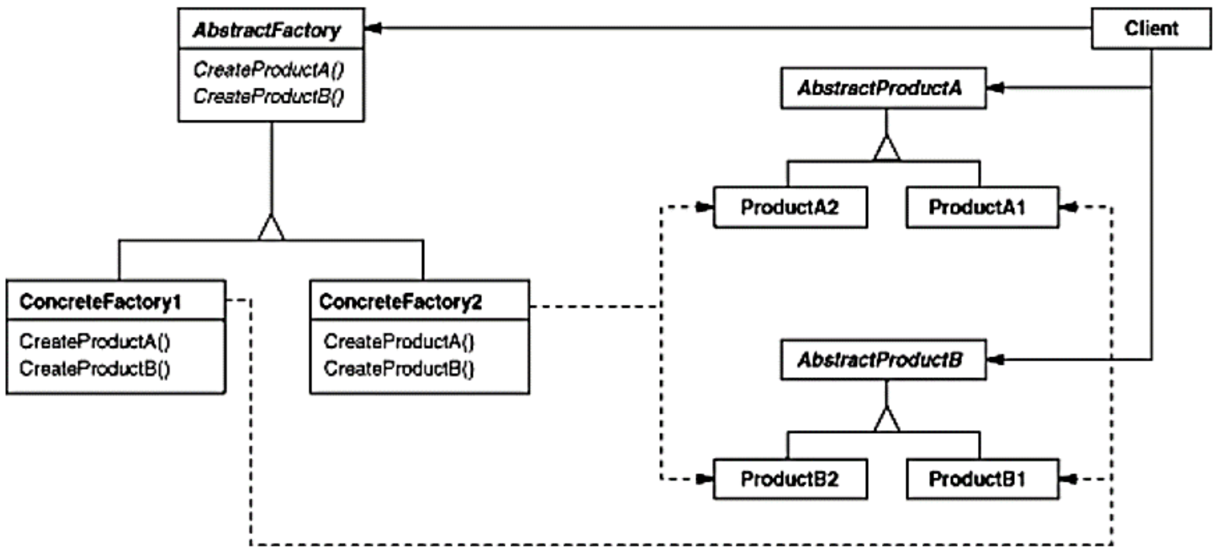
1.1 Factory

Factory method is a creational design pattern which solves the problem of creating product objects without specifying their concrete classes.



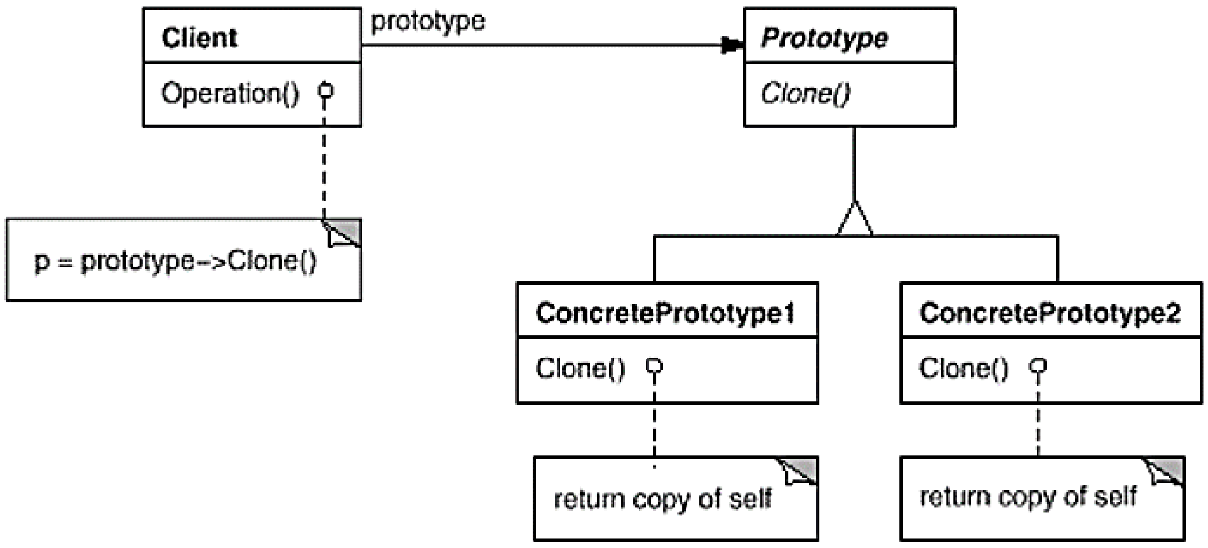
1.2 Abstract Factory

Abstract Factory is a creational design pattern, which solves the problem of creating entire product families without specifying their concrete classes.



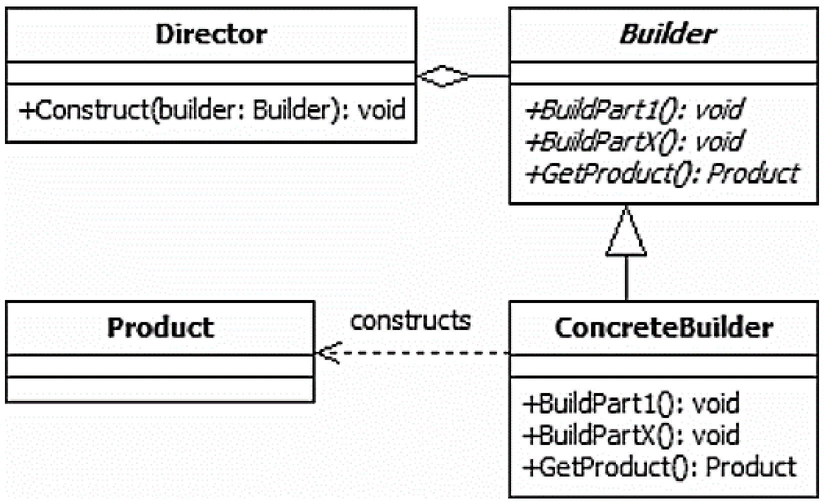
1.3 Prototype

Prototype is a creational design pattern that allows cloning objects, even complex ones, without coupling to their specific classes.



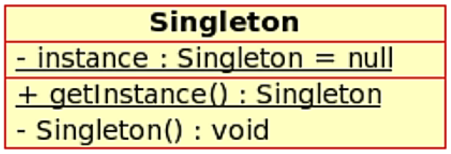
1.4 Builder

Builder is a creational design pattern, which allows constructing complex objects step by step. Note that Builder is based on Bridge DP.



1.5 Singleton

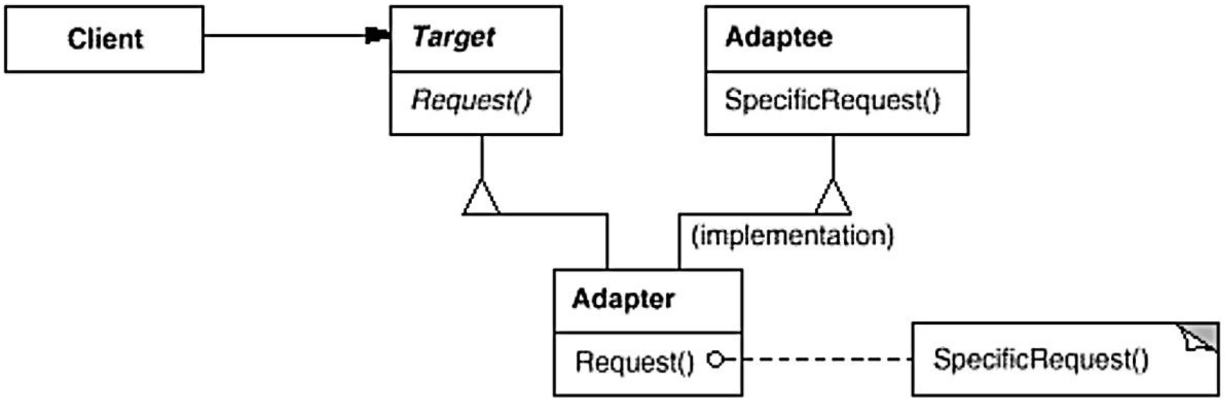
Singleton is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code.



2.1 Class Adapter

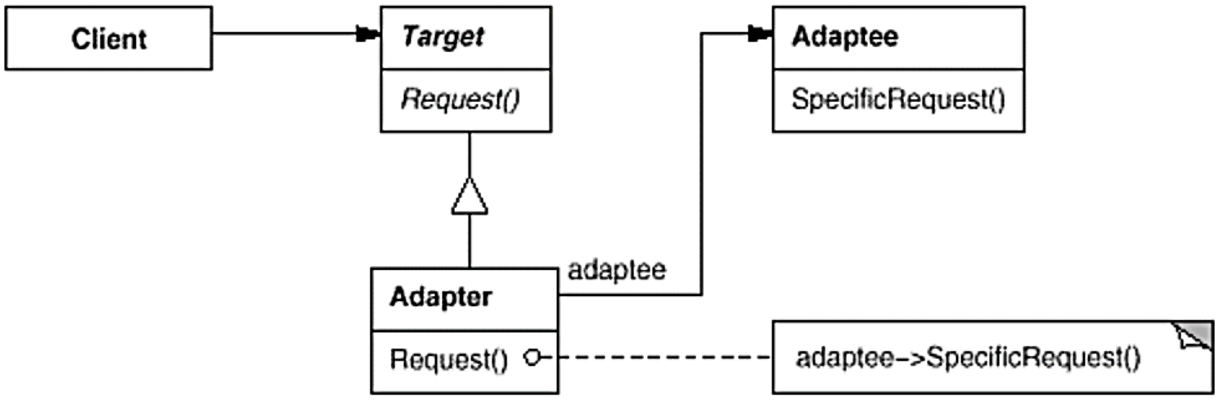
Generally, Adapter is a DP pattern that allows objects with incompatible interfaces to collaborate. There are two main Adapter Patterns – Class Adapter and Object Adapter.

Class Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate. Namely, allows the Target do an operation of the Adaptee using Class Adapter.



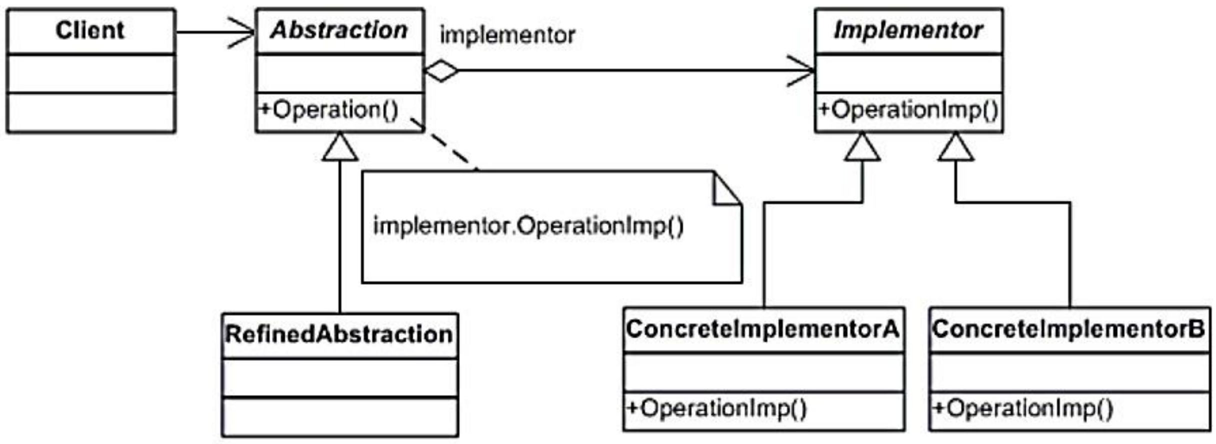
2.2 Object Adapter

Object Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate. Namely, allows the Target do operations of different Adaptees using Different object adapters.



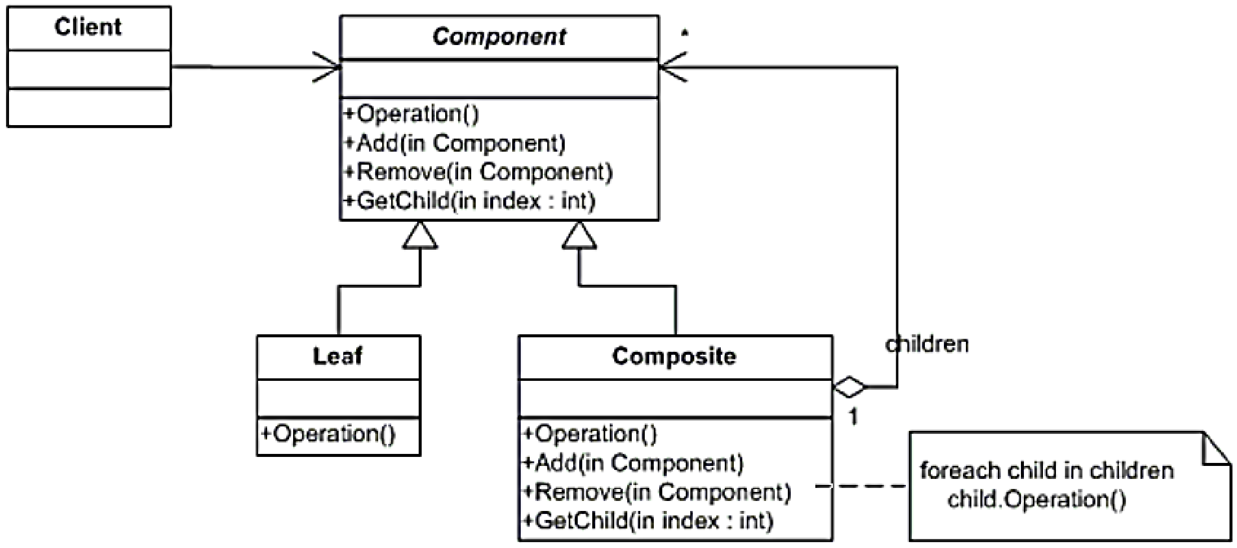
2.3 Bridge

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



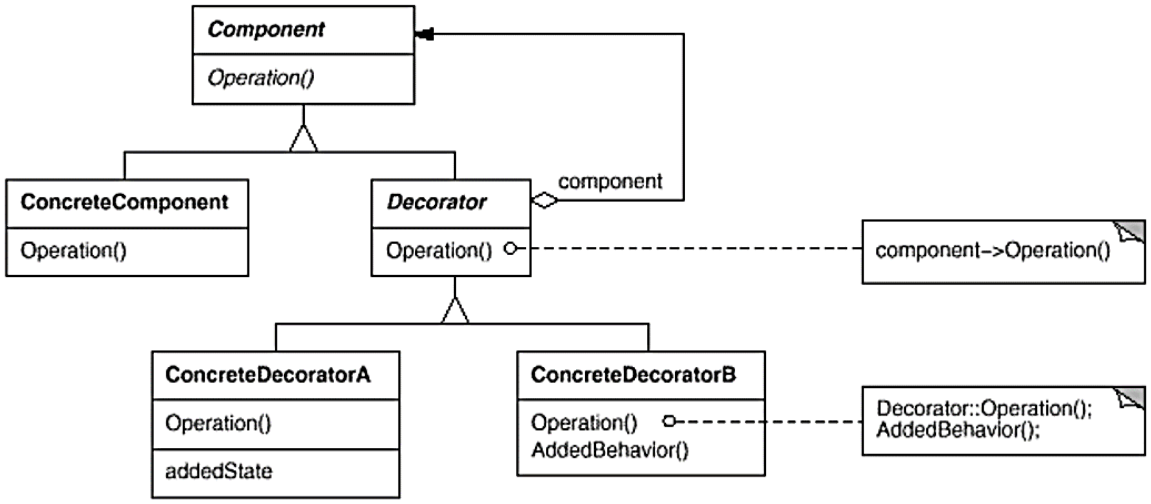
2.4 Composite

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.



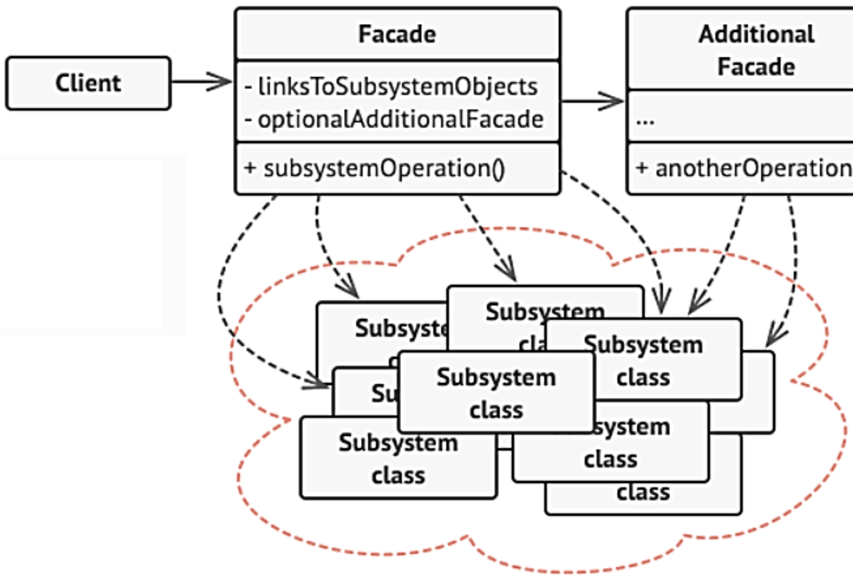
2.5 Decorator

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



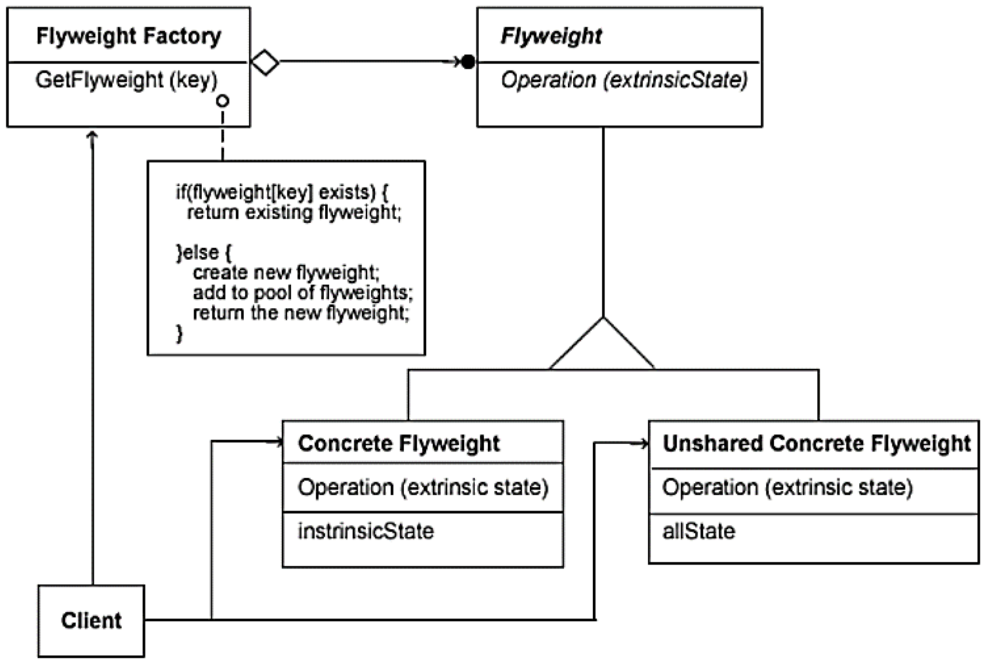
2.6 Facade

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.



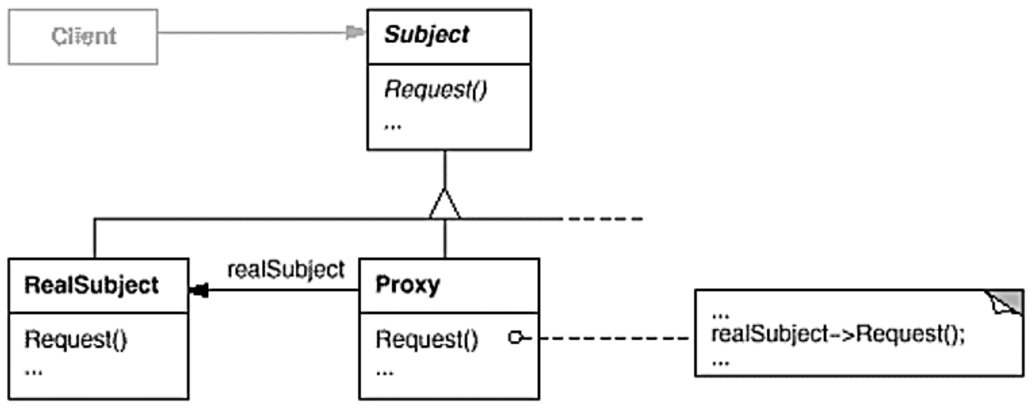
2.7 Flyweight

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all the data in each object.


Flyweight Factory
GetFlyweight (key)
Flyweight
Operation (extrinsicState)
if(flyweight[key] exists) {
return existing flyweight;
}else {
create new flyweight;
add to pool of flyweights;
return the new flyweight;
}
Concrete Flyweight
Operation (extrinsic state)
instrinsicState
Unshared Concrete Flyweight
Operation (extrinsic state)
allState
Client

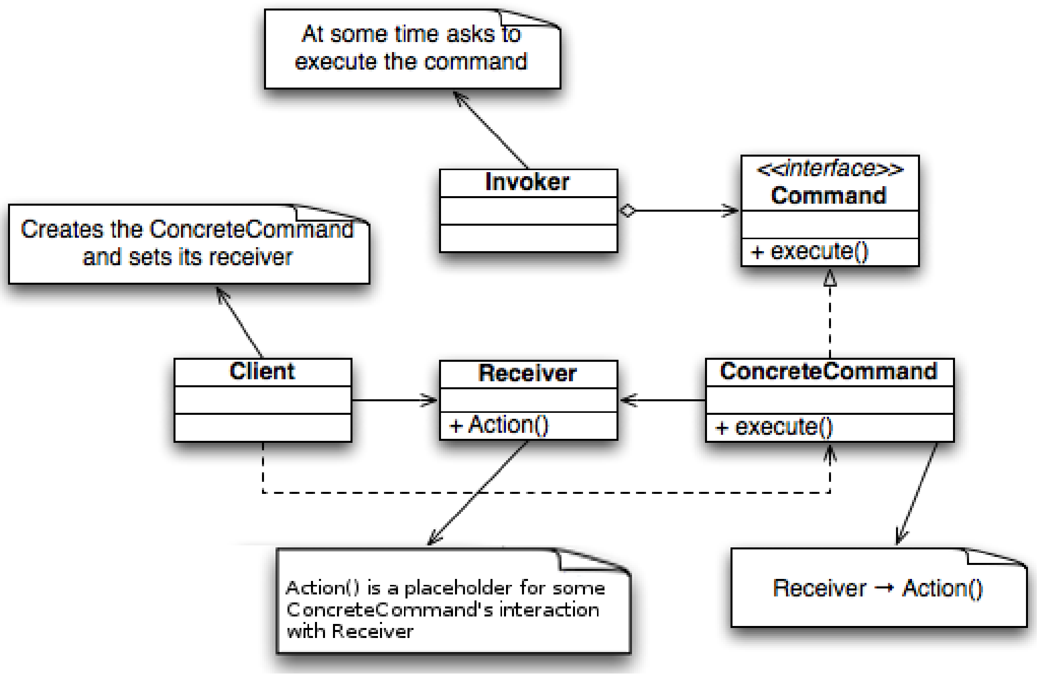
2.8 Proxy

Proxy is a structural DP that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.



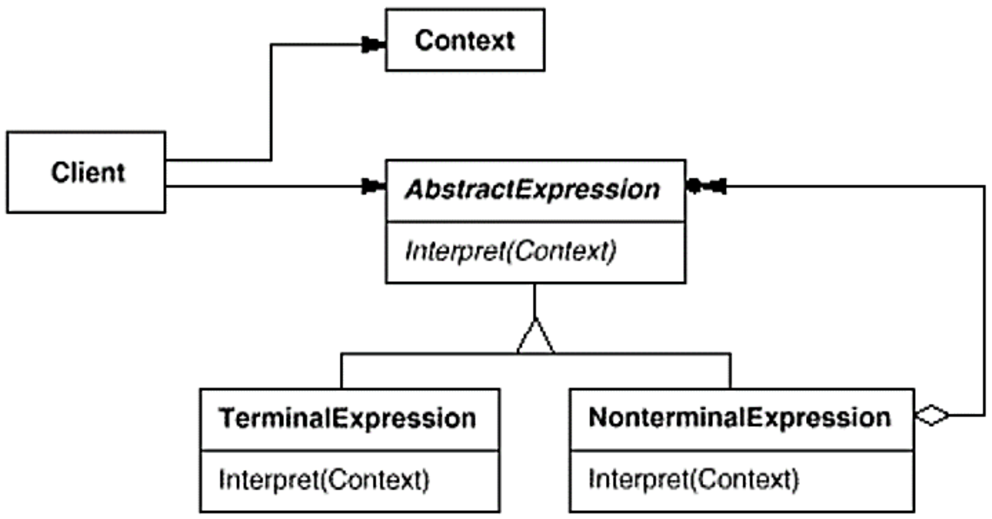
3.1 Command

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.


At some time asks to execute the command
Invoker
<<interface>>
Command
+ execute()
Creates the ConcreteCommand and sets its receiver
Client
Receiver
+ Action()
ConcreteCommand
+ execute()
Action() is a placeholder for some ConcreteCommand's interaction with Receiver
Receiver → Action()

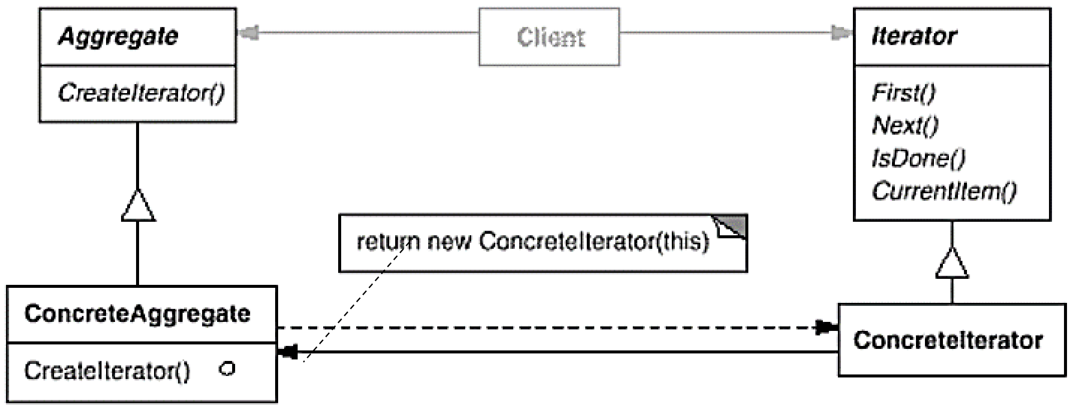
3.2 Interpreter

Interpreter design pattern is one of the behavioral design patterns. Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.



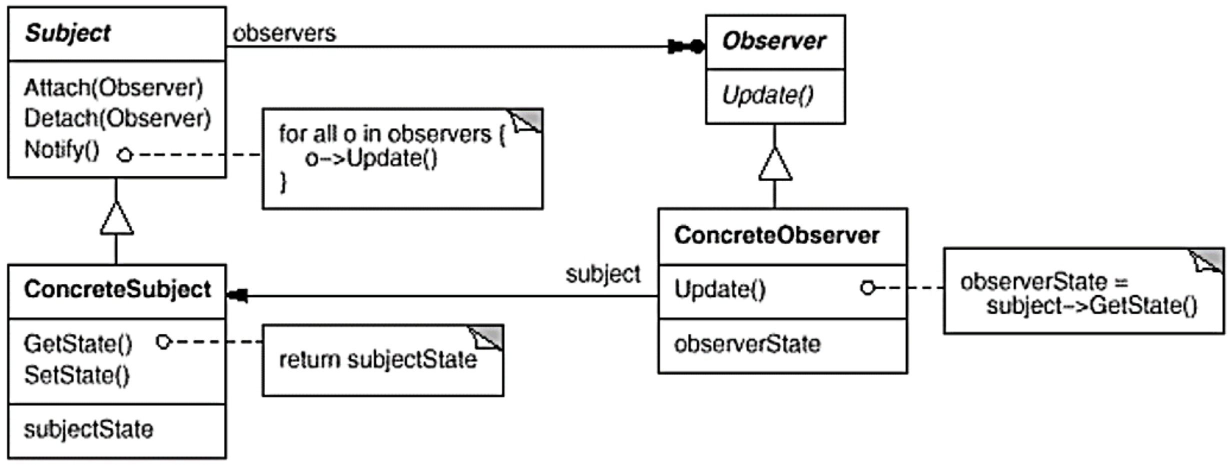
3.3 Iterator

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



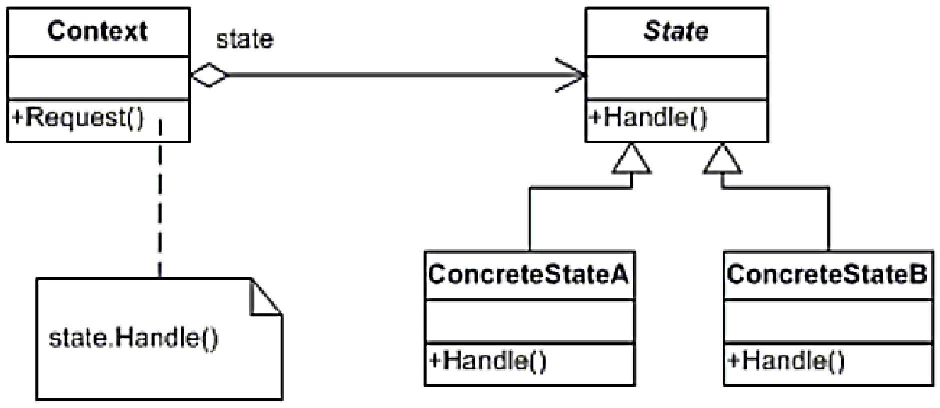
3.4 Observer

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



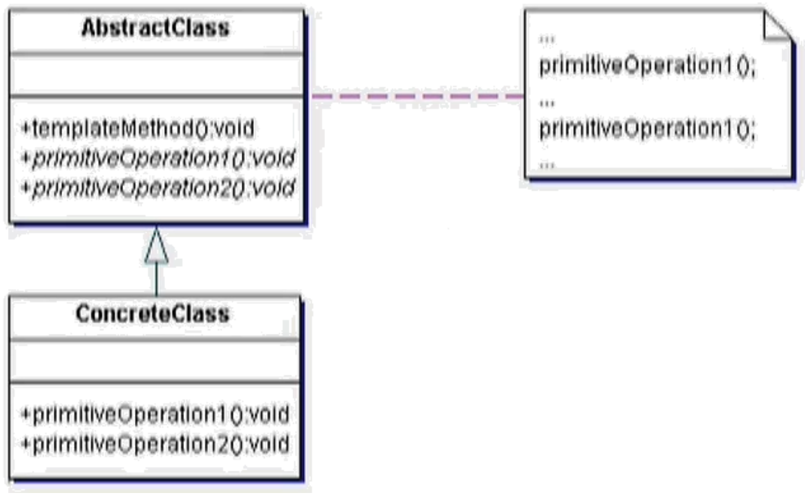
3.5 State

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



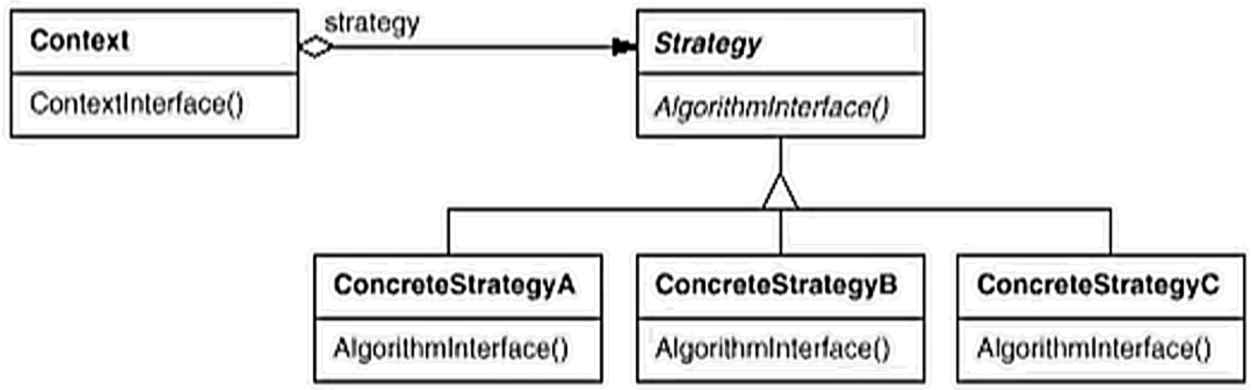
3.6 Template

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but let's subclasses override specific steps of the algorithm without changing its structure.


AbstractClass
+templateMethod():void
+primitiveOperation1():void
+primitiveOperation2():void
...
primitiveOperation1();
...
primitiveOperation1();
...
ConcreteClass
+primitiveOperation1():void
+primitiveOperation2():void

3.7 Strategy

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()