Design Patterns Cheat Sheet

Mohammad Faisal
@mohammadfaisalkhan

What is Design Pattern

Design patterns are general, reusable solutions to common problems in software design. They are like templates that can be applied to various programming situations, providing a proven strategy for solving issues related to code structure, object creation, and system behavior.

Types of Design Patterns:-

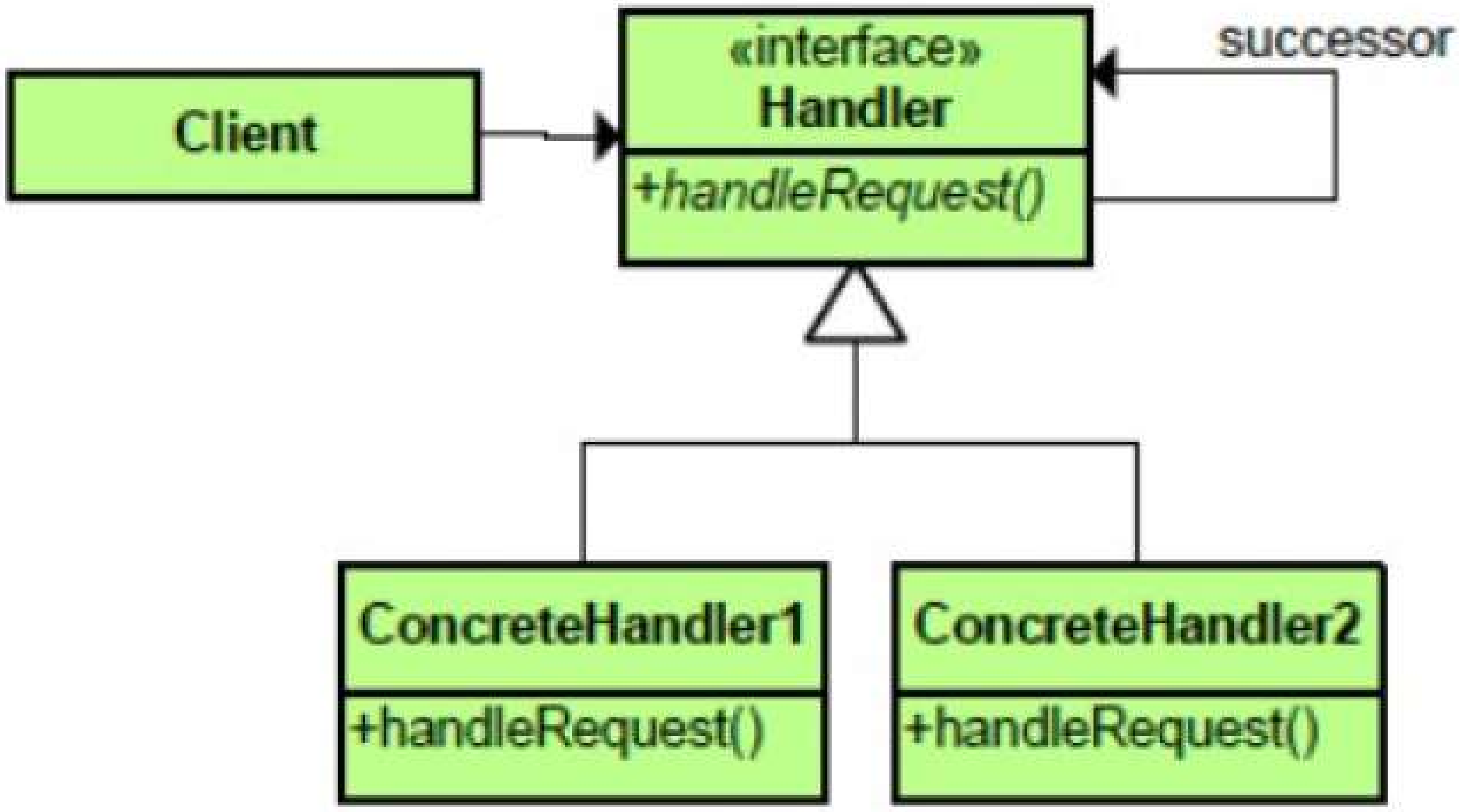
1. **Behavioral Patterns:** These focus on communication between objects, defining how objects interact and share responsibility.

2. **Structural Patterns:** These deal with object composition, helping to create relationships between objects in a flexible and efficient manner.

3. **Creational Patterns:** These deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

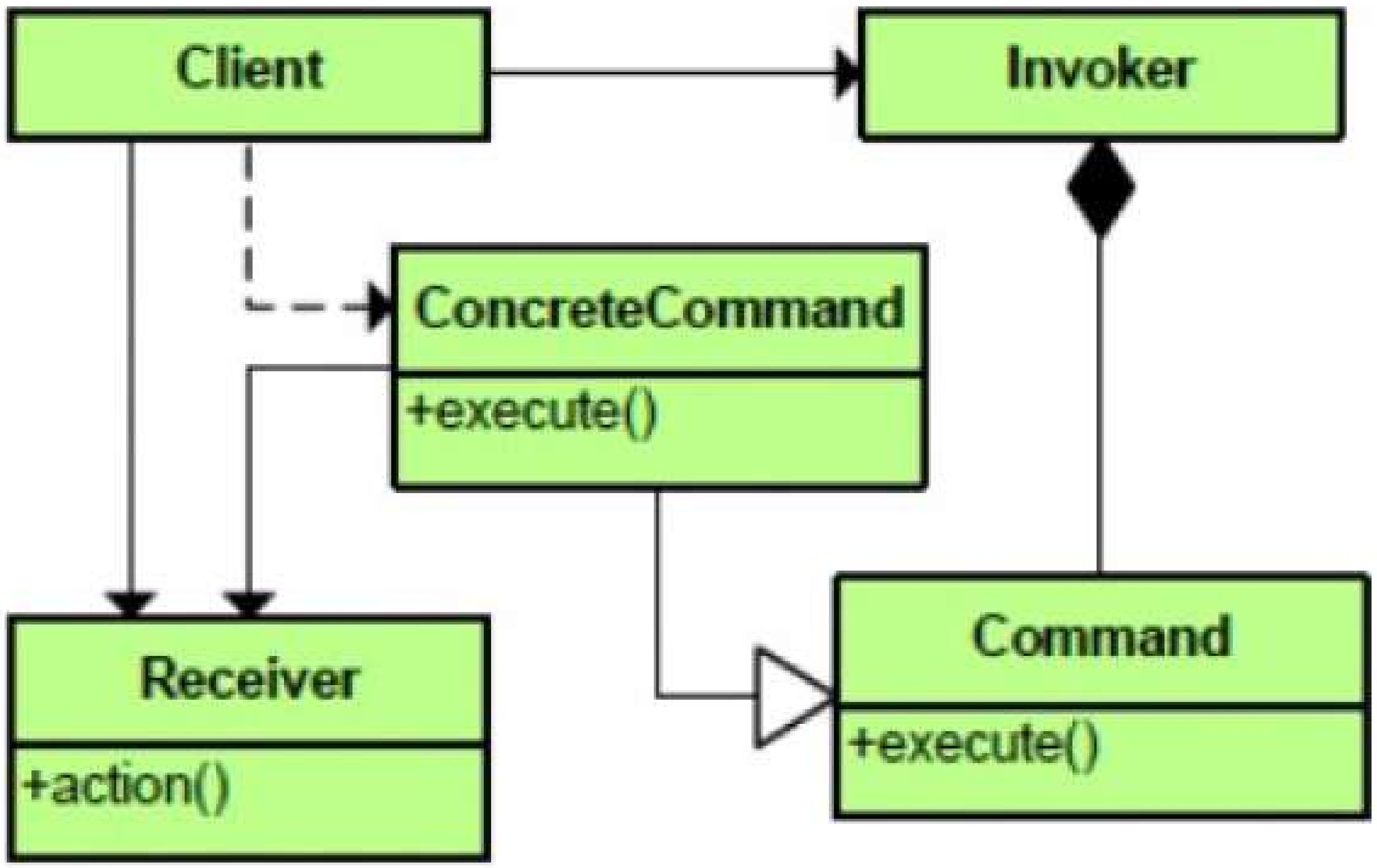
Chain of Responsibility

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.
Type: Behavioral



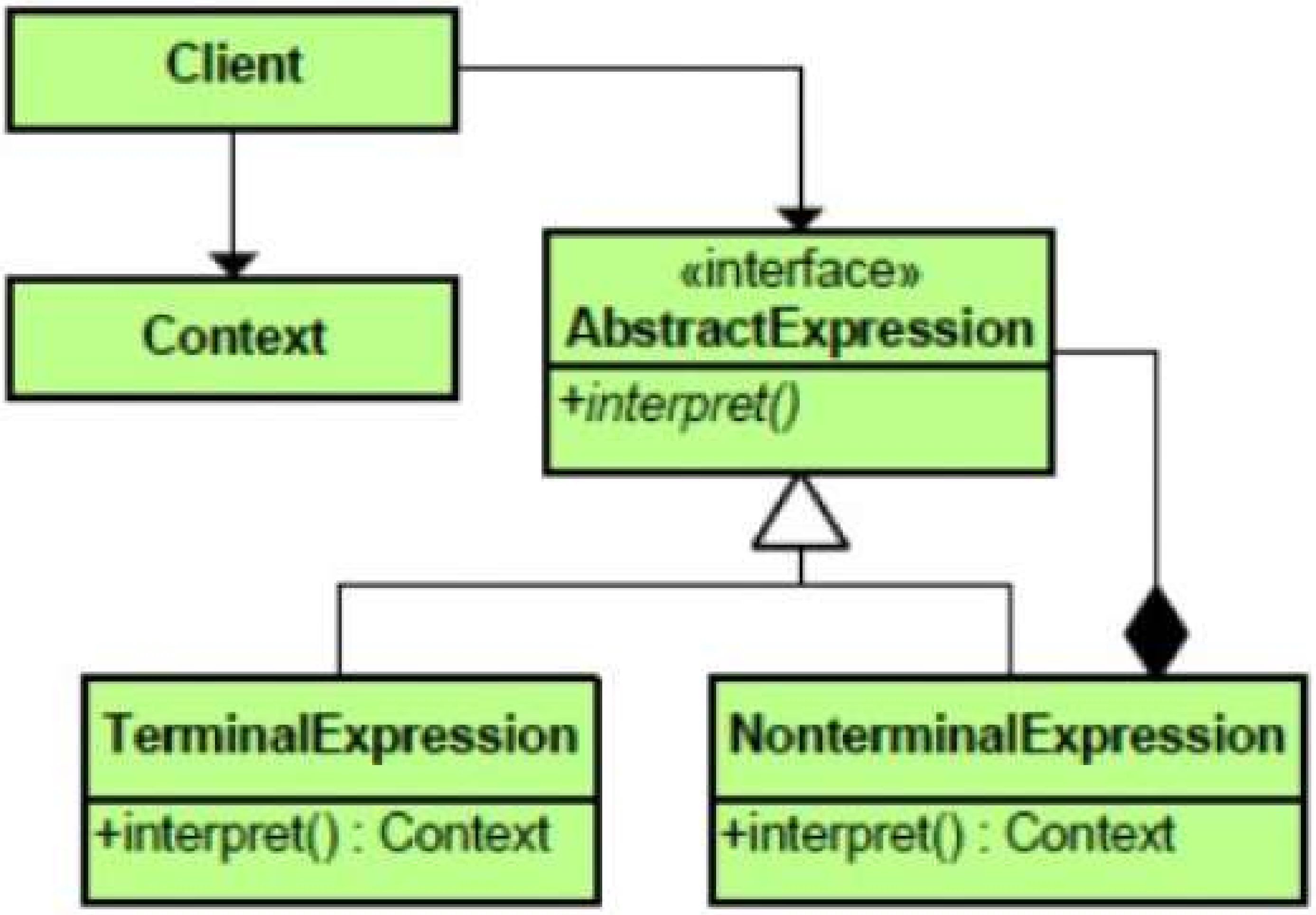
Command

Encapsulate a request as an object, allowing you to parameterize clients with different requests, queues, and enable undoable operations. This is a key aspect of the Command pattern.
Type : Behavioral



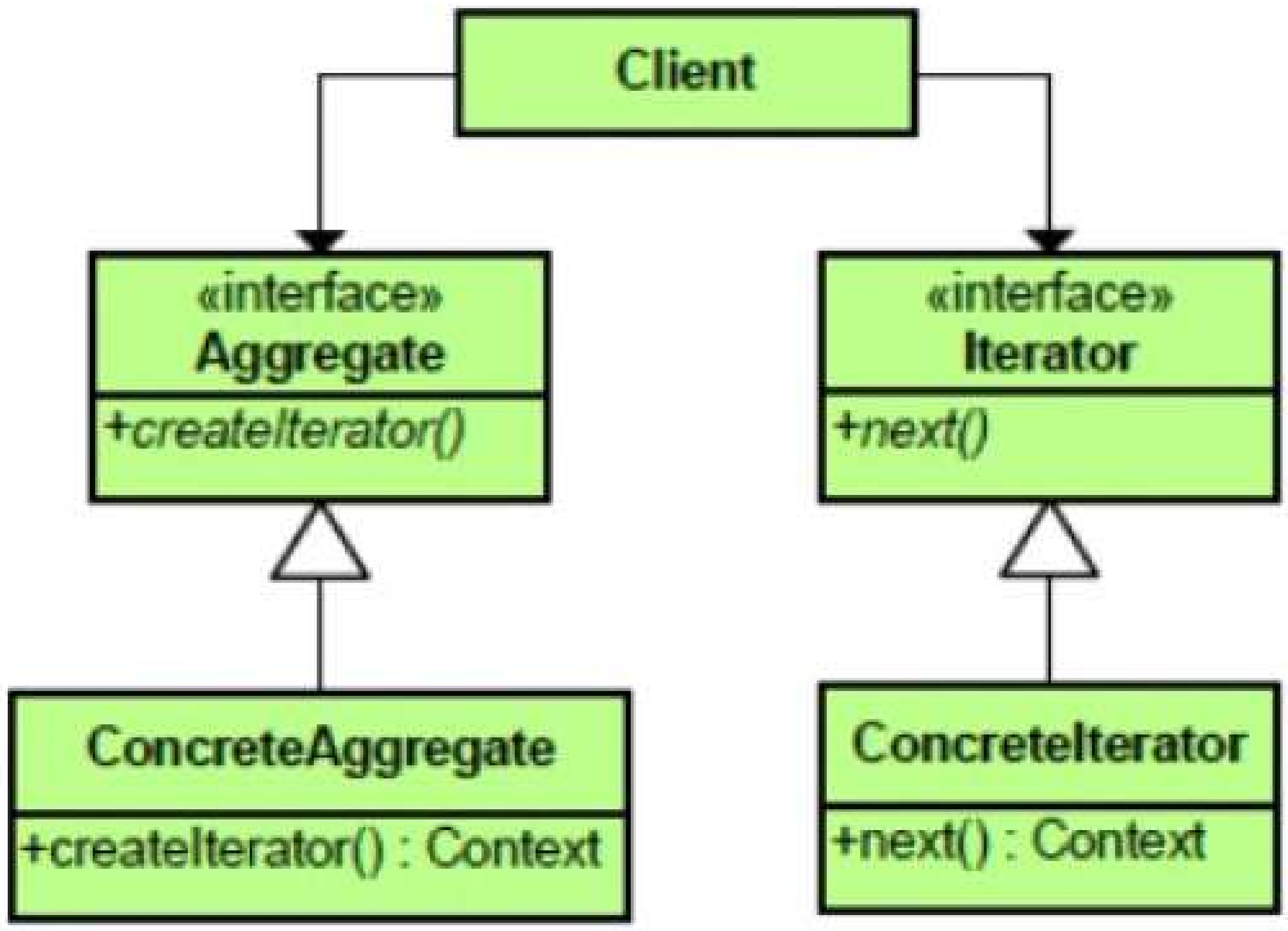
Interpreter

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



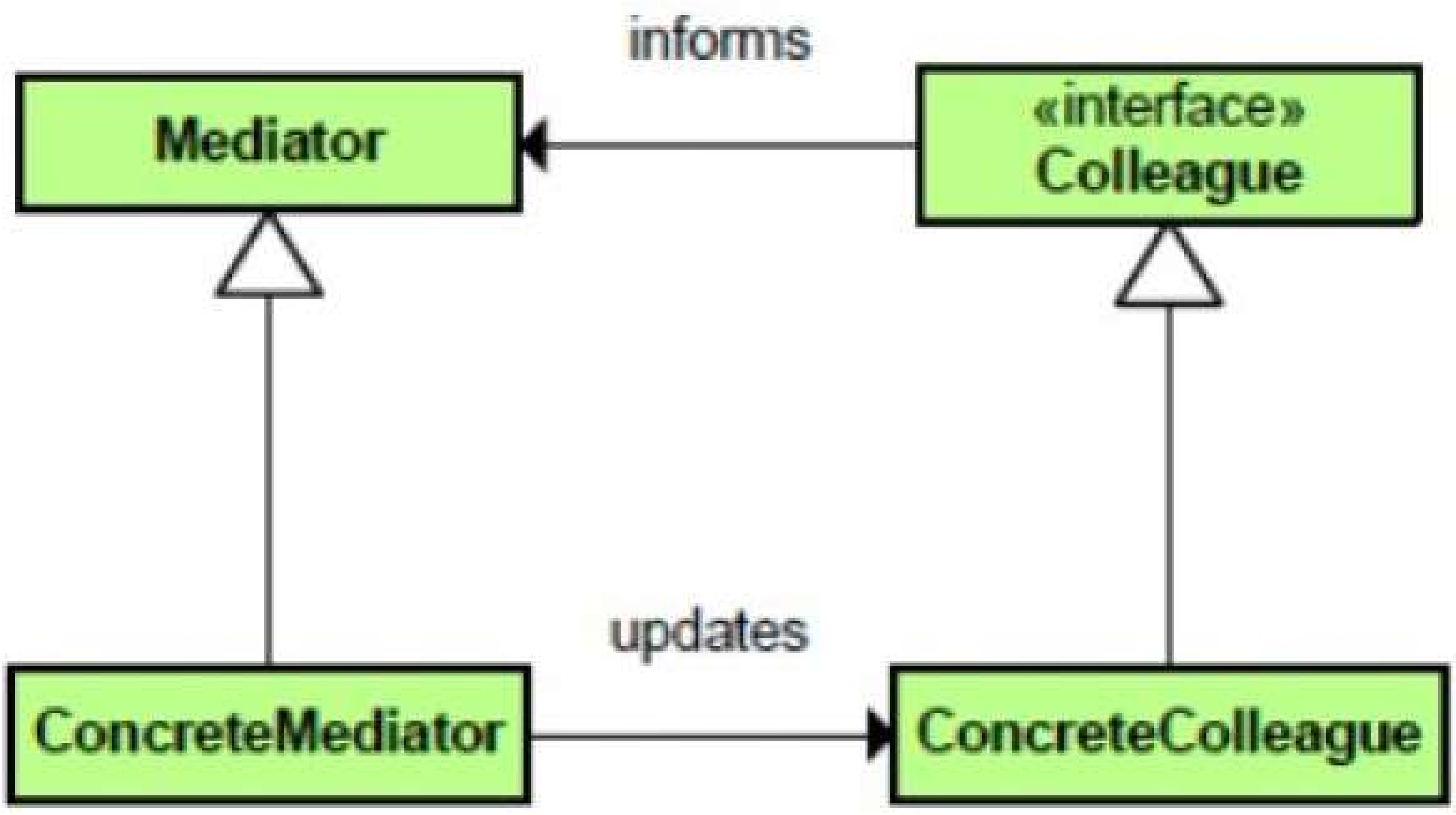
Iterator

It allows sequential access to elements of a collection without exposing its internal structure, ensuring encapsulation while providing a uniform way to traverse the collection.
Type : Behavioral



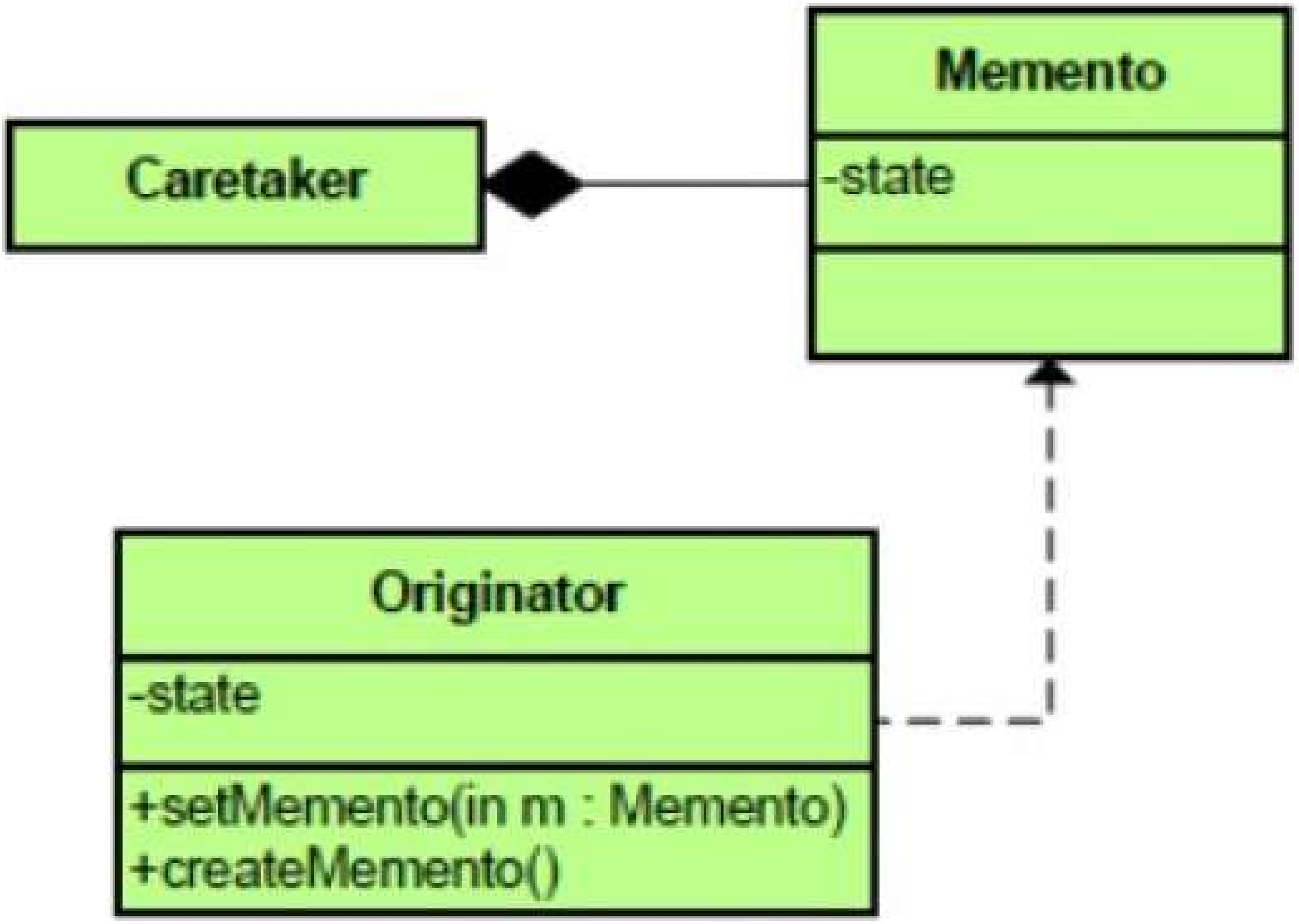
Mediator

Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interactions independently.
Type : Behavioral



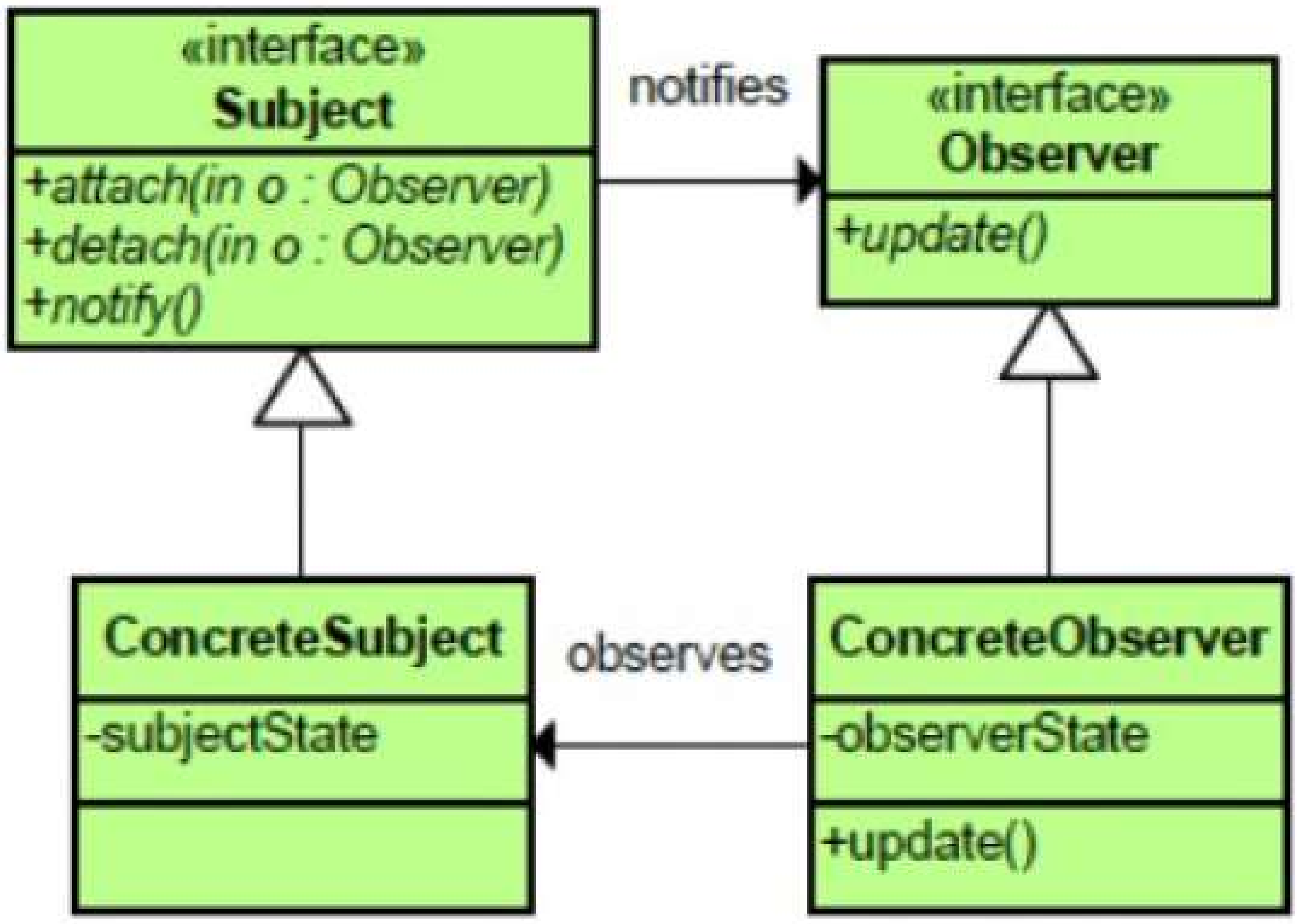
Memento

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



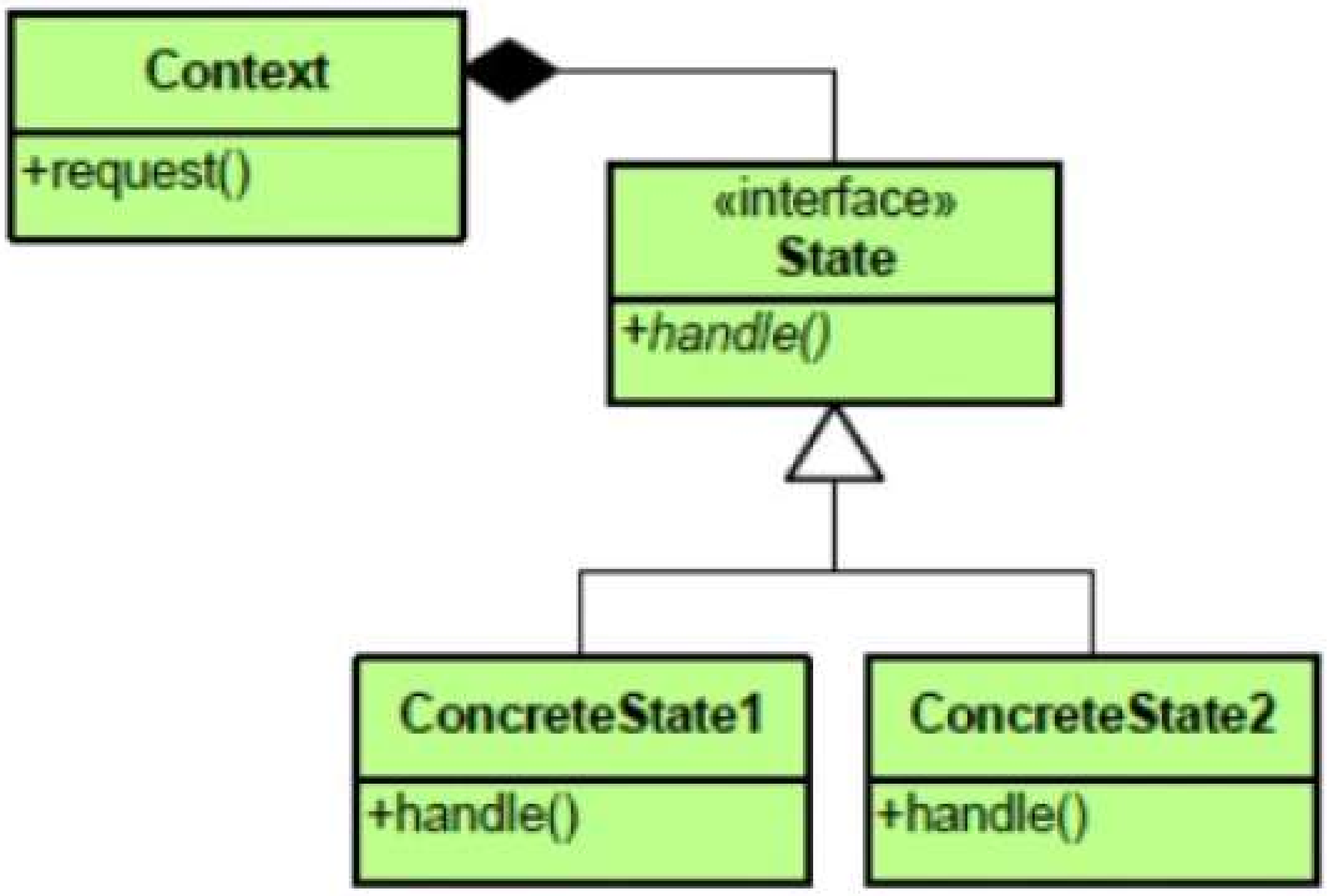
Observer

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



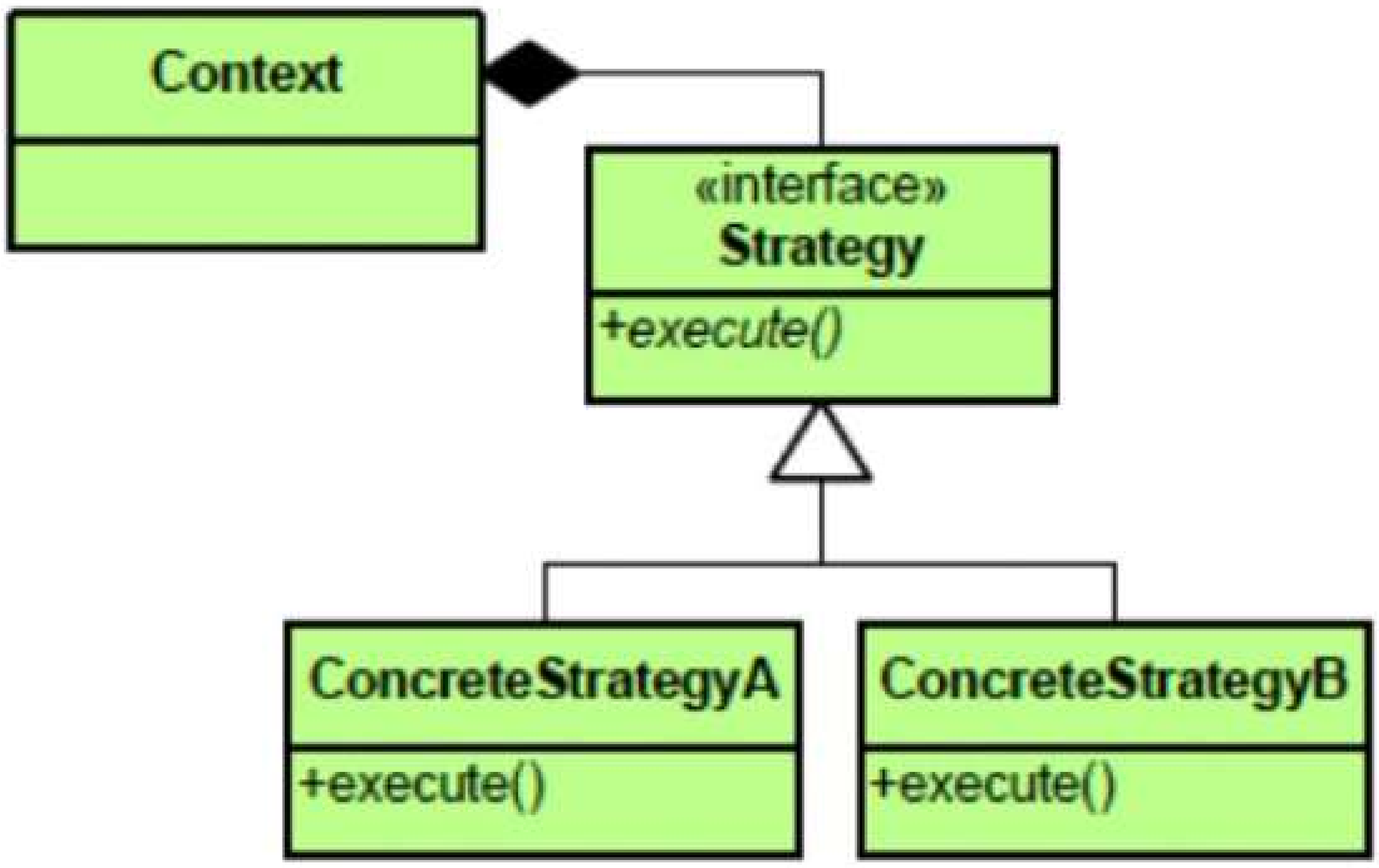
State

It enables an object to change its behavior based on its internal state, creating the illusion that the object has changed its class. This approach allows for cleaner code by encapsulating state-specific behavior and transitions.
Type : Behavioral



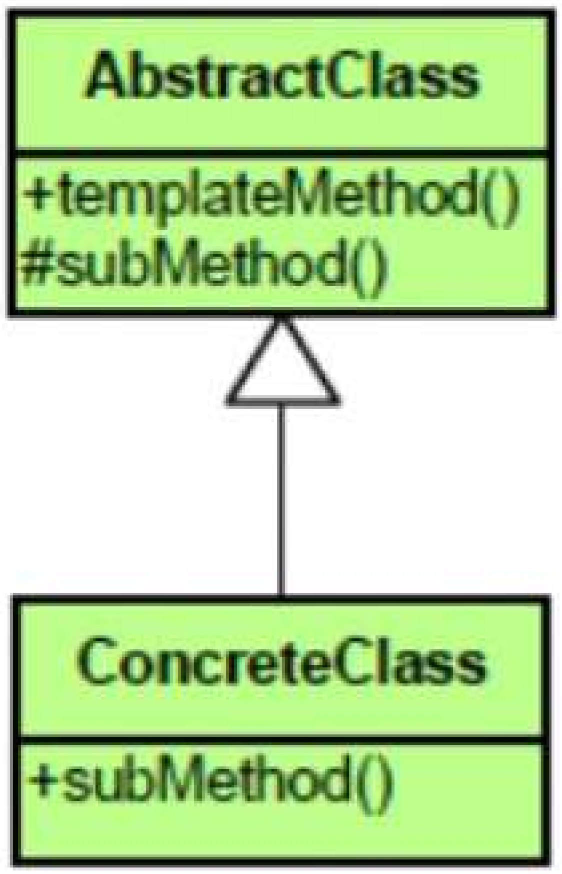
Strategy

The Strategy pattern defines a family of algorithms, encapsulates each one within its own class, and makes them interchangeable. This allows the algorithm to vary independently from the clients that utilize it, promoting flexibility and reducing code dependencies.
Type : Behavioral



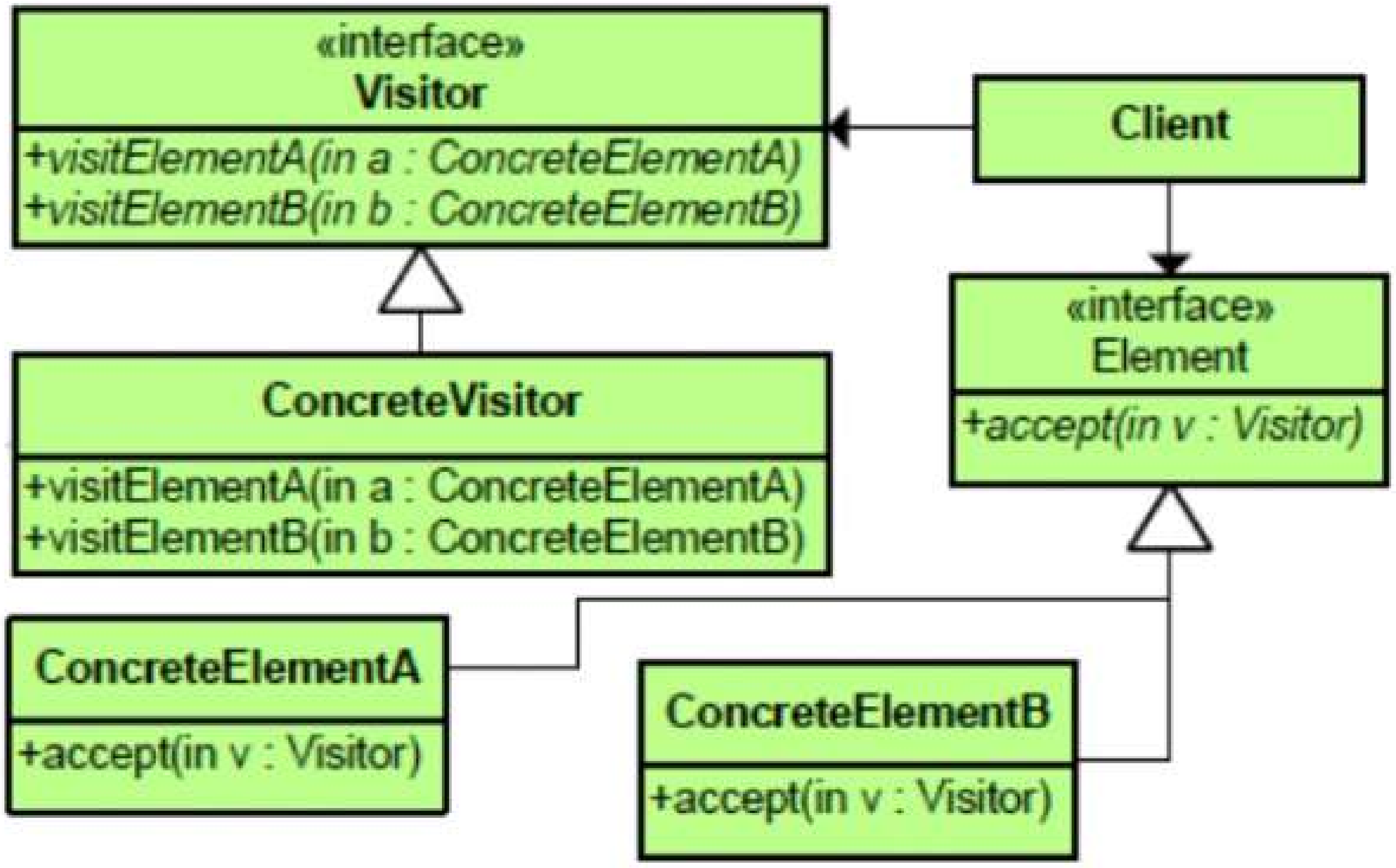
Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Type : Behavioral



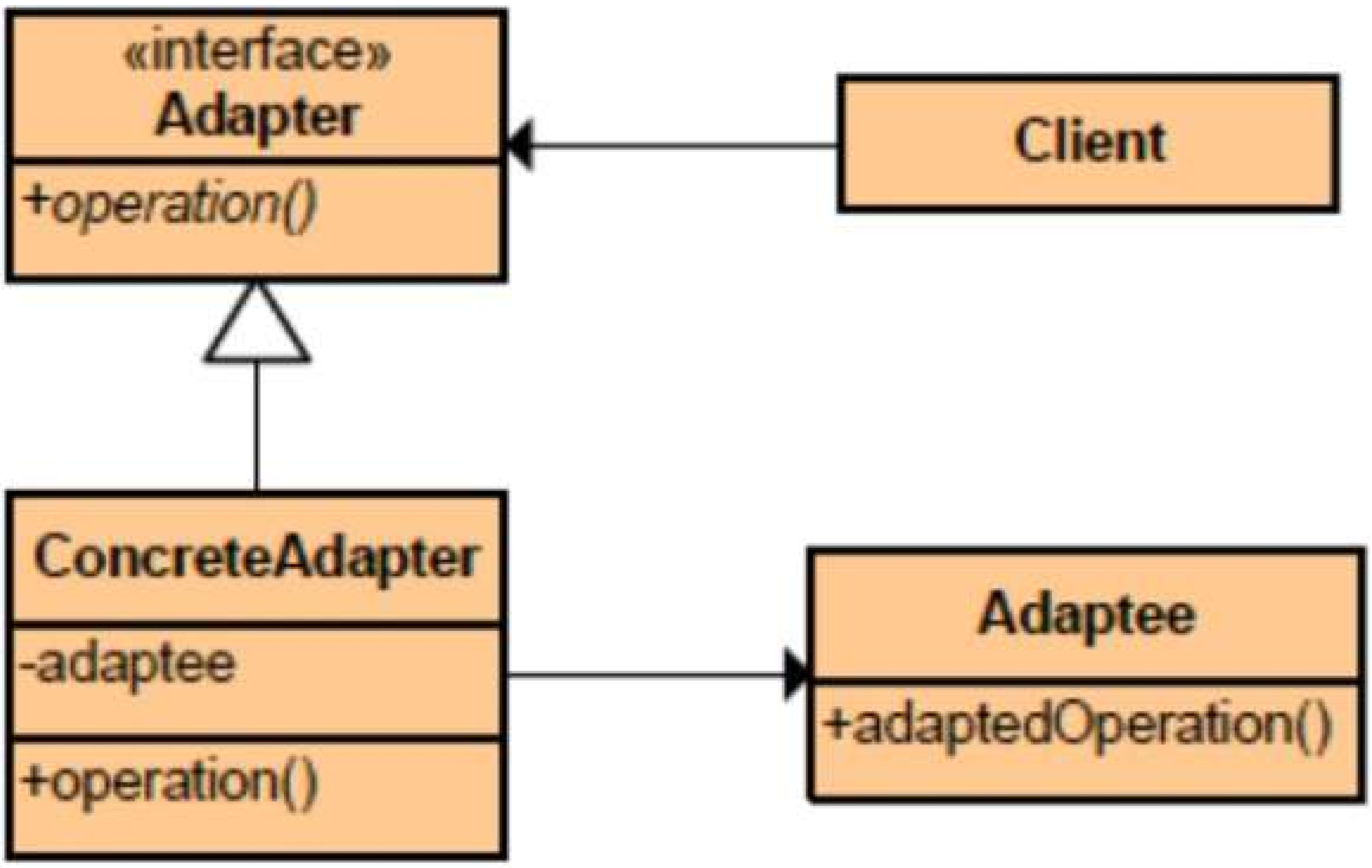
Visitor

Represent an operation to be performed on elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.
Type : Behavioral



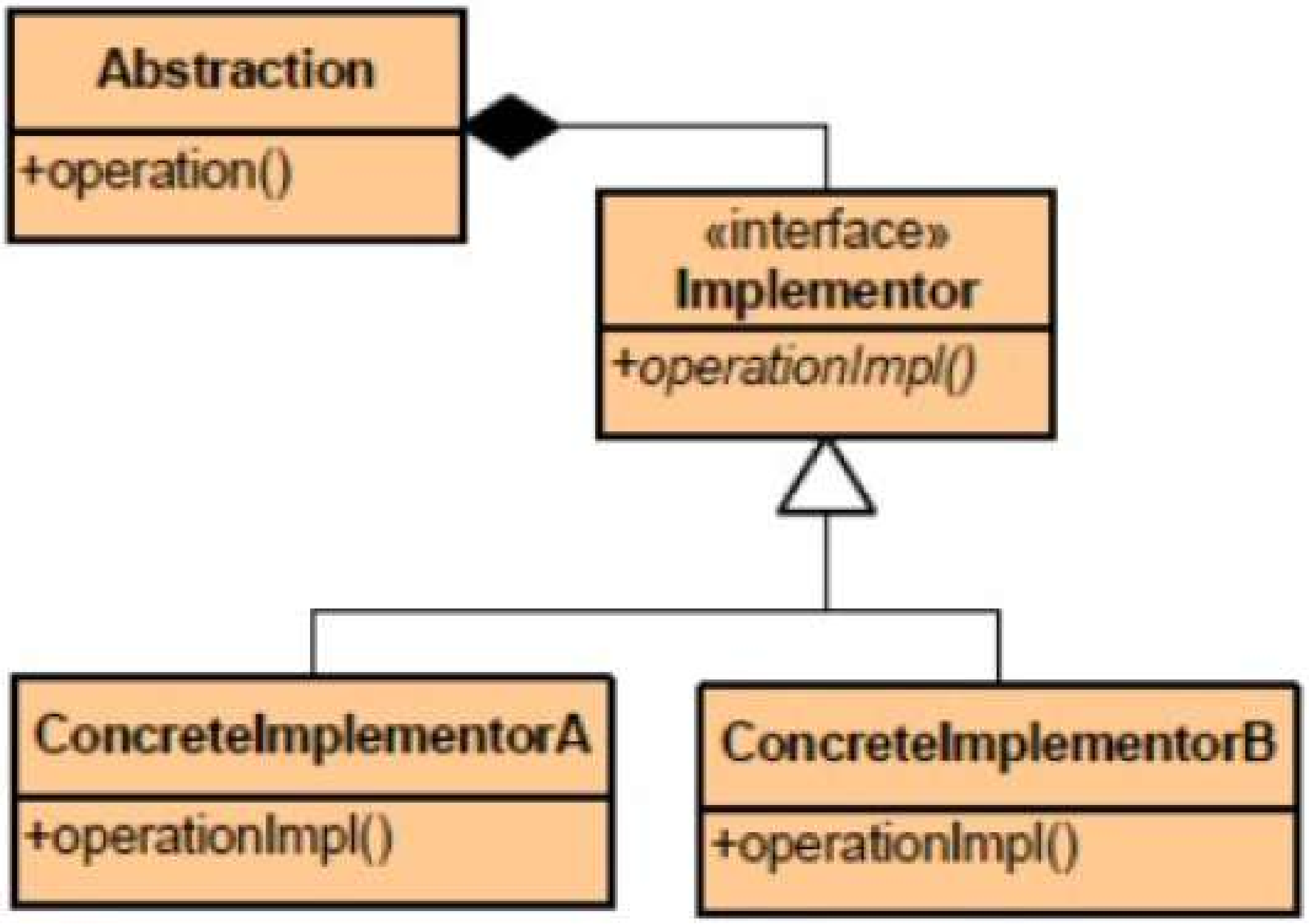
Adapter

Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.
Type : Structural



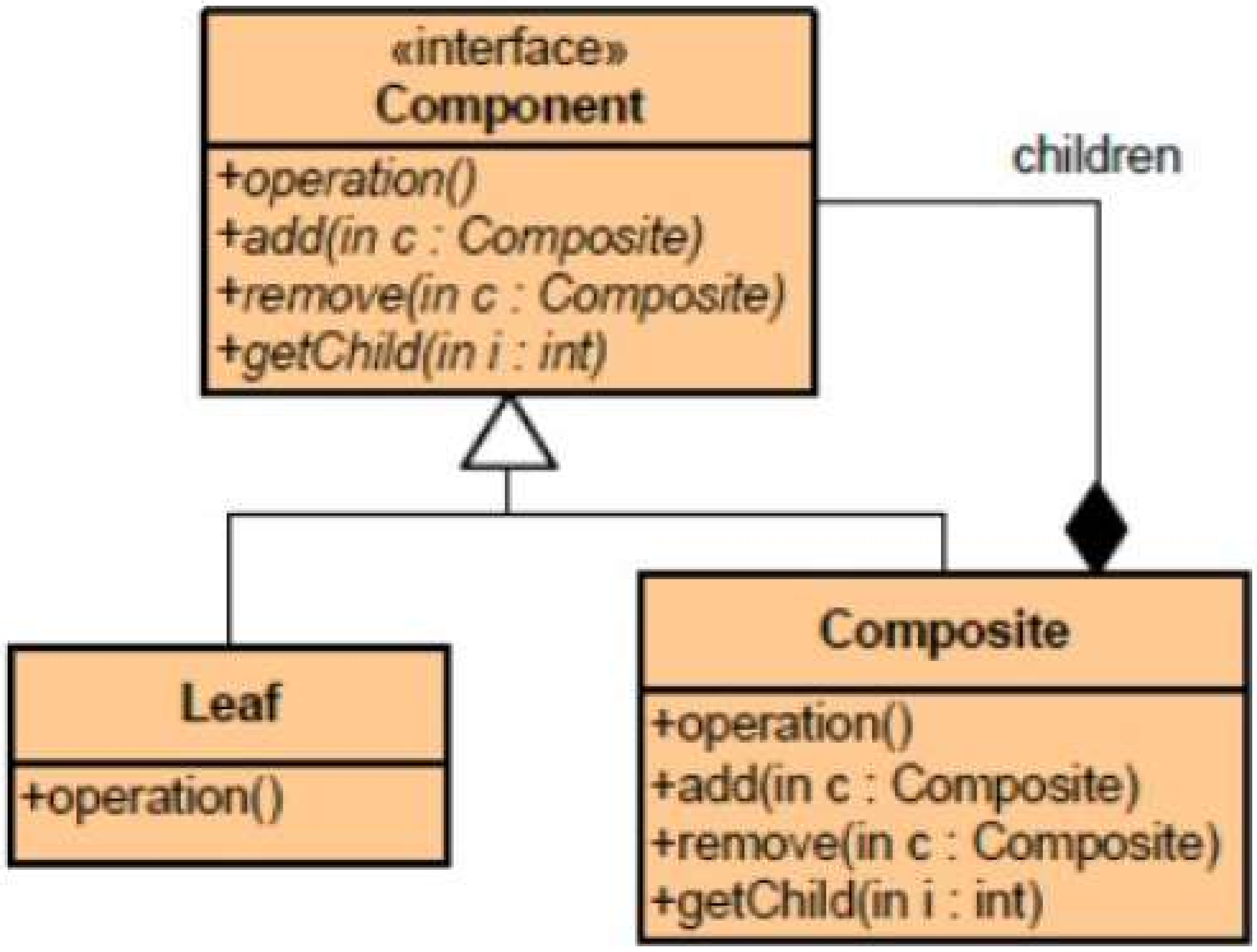
Bridge

The Bridge pattern decouples an abstraction from its implementation, allowing both to evolve independently. This separation enables flexibility, as changes in the implementation do not affect the abstraction, and vice versa.
Type : Structural



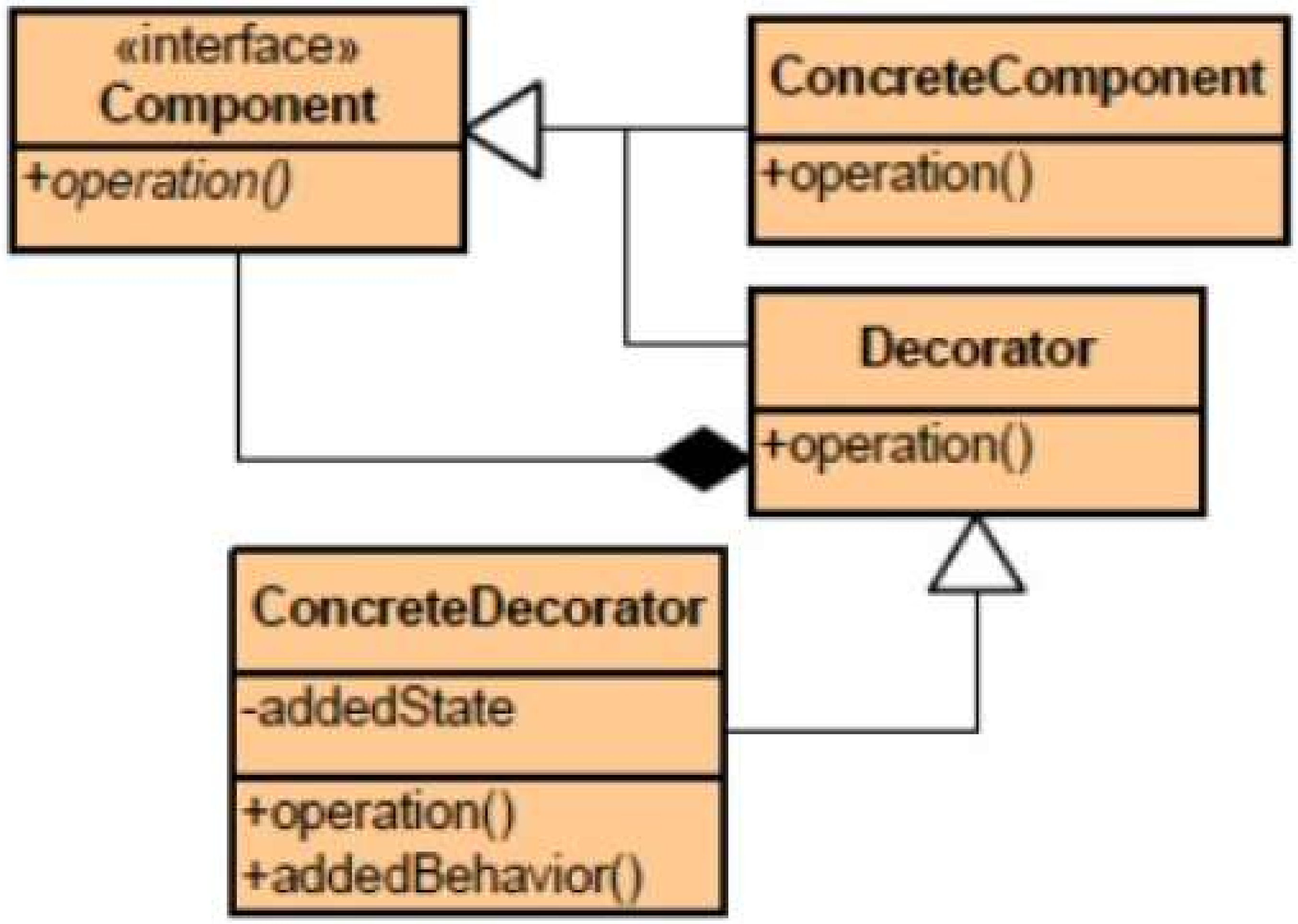
Composite

It organizes objects into tree structures to represent part-whole hierarchies, allowing clients to treat individual objects and compositions uniformly. It simplifies the client code by enabling consistent interaction with both single objects and composite groups.
Type : Structural



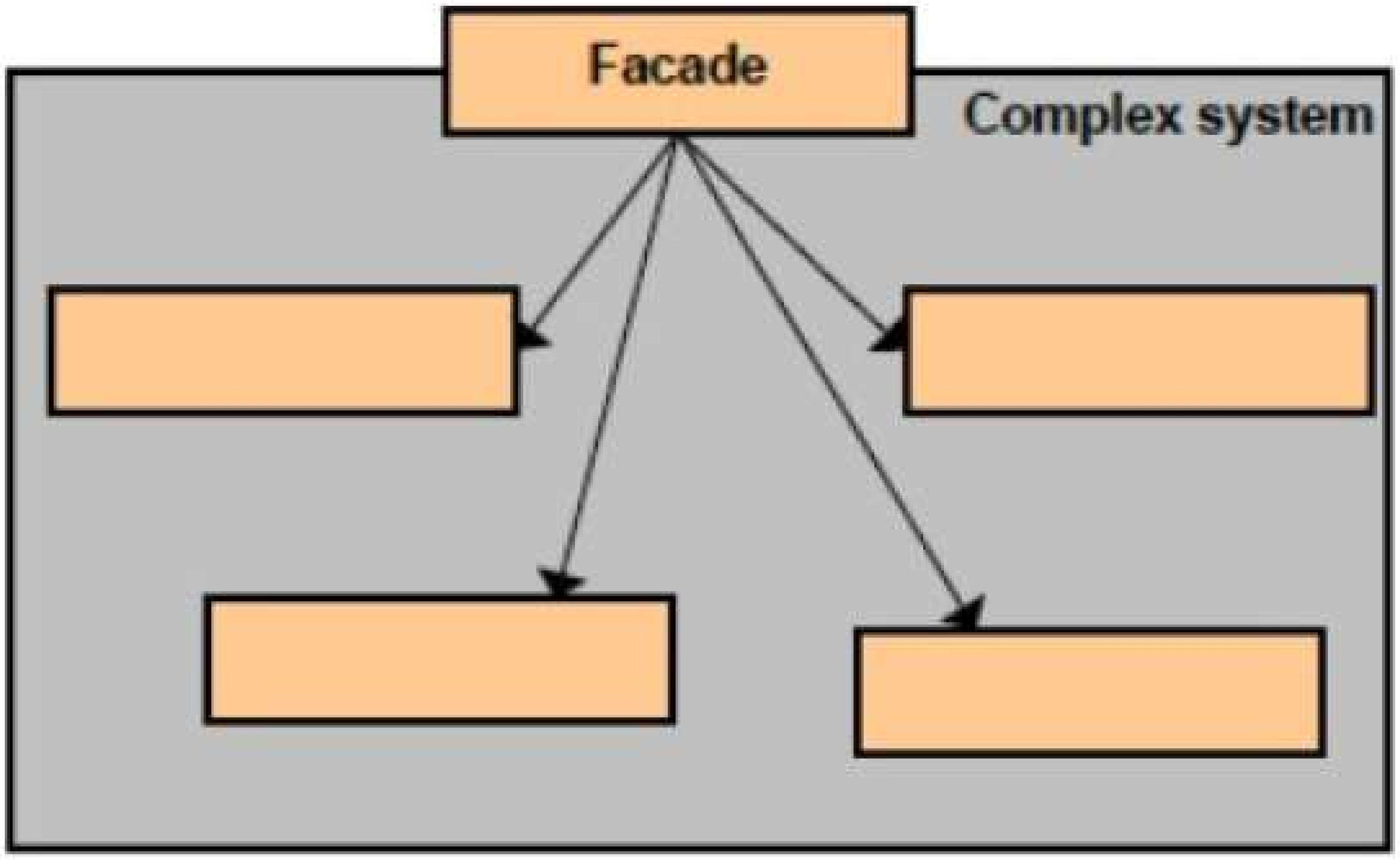
Decorator

It dynamically attaches additional responsibilities to an object, offering a flexible alternative to subclassing for extending functionality. This approach allows for enhanced features without altering the original object's structure.
Type : Structural



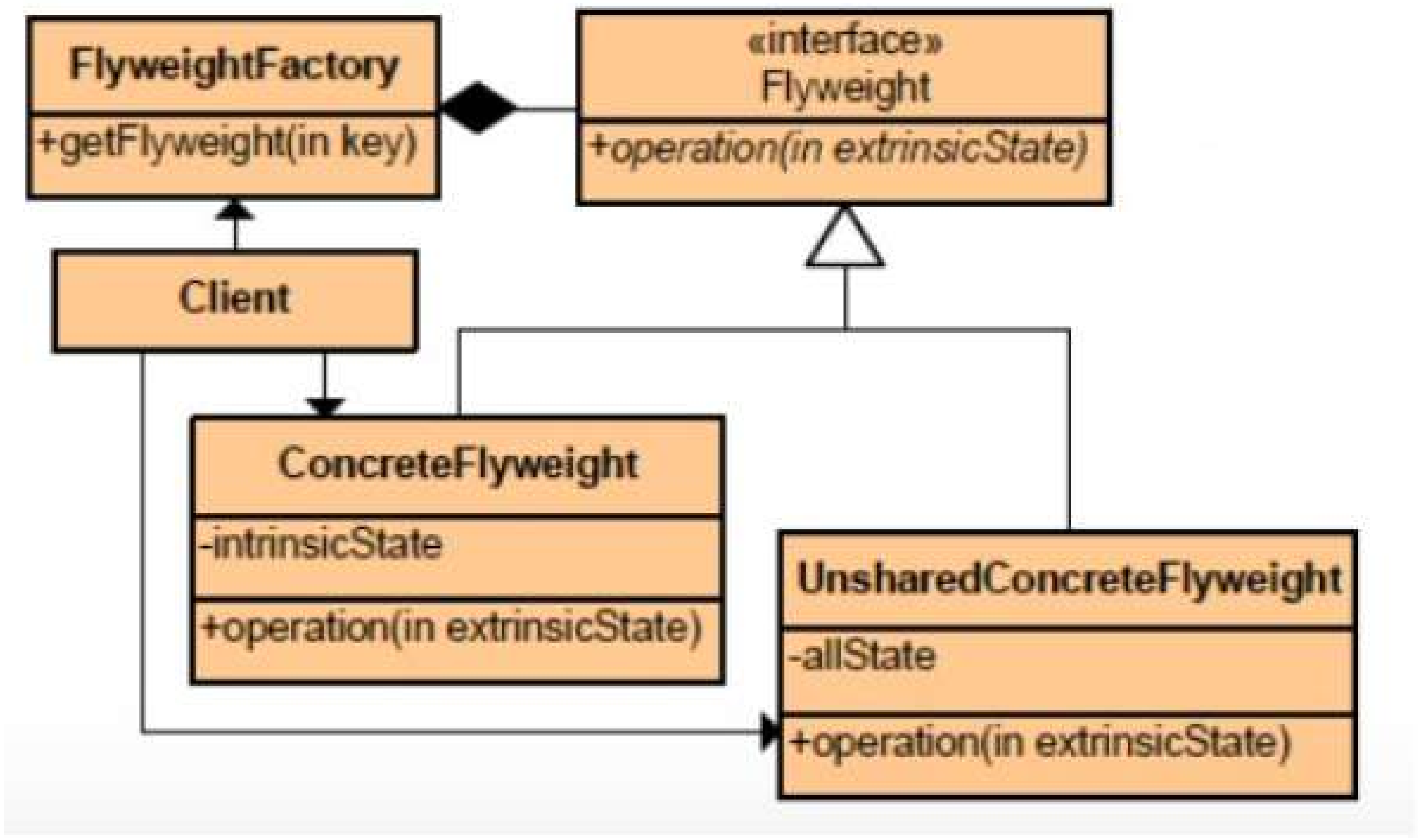
Facade

The Facade design pattern provide a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that makes the subsystem easier to use.
Type : Structural



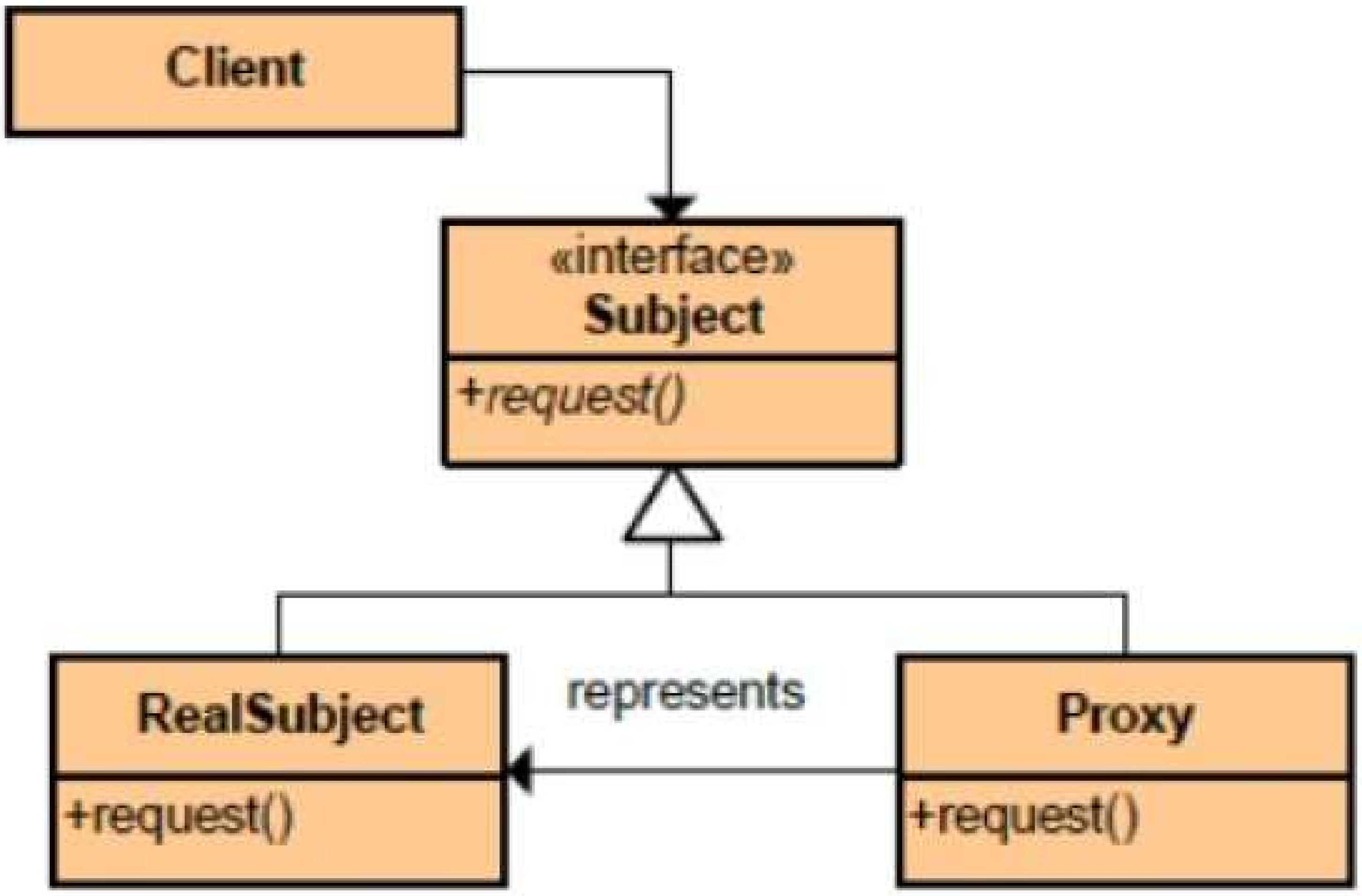
Flyweight

It uses sharing to efficiently support large numbers of fine-grained objects. By minimizing memory usage, it allows identical objects to share common data, reducing resource consumption.
Type : Structural


FlyweightFactory
+getFlyweight(in key)
«interface»
Flyweight
+operation(in extrinsicState)
Client
ConcreteFlyweight
-intrinsicState
+operation(in extrinsicState)
UnsharedConcreteFlyweight
-allState
+operation(in extrinsicState)

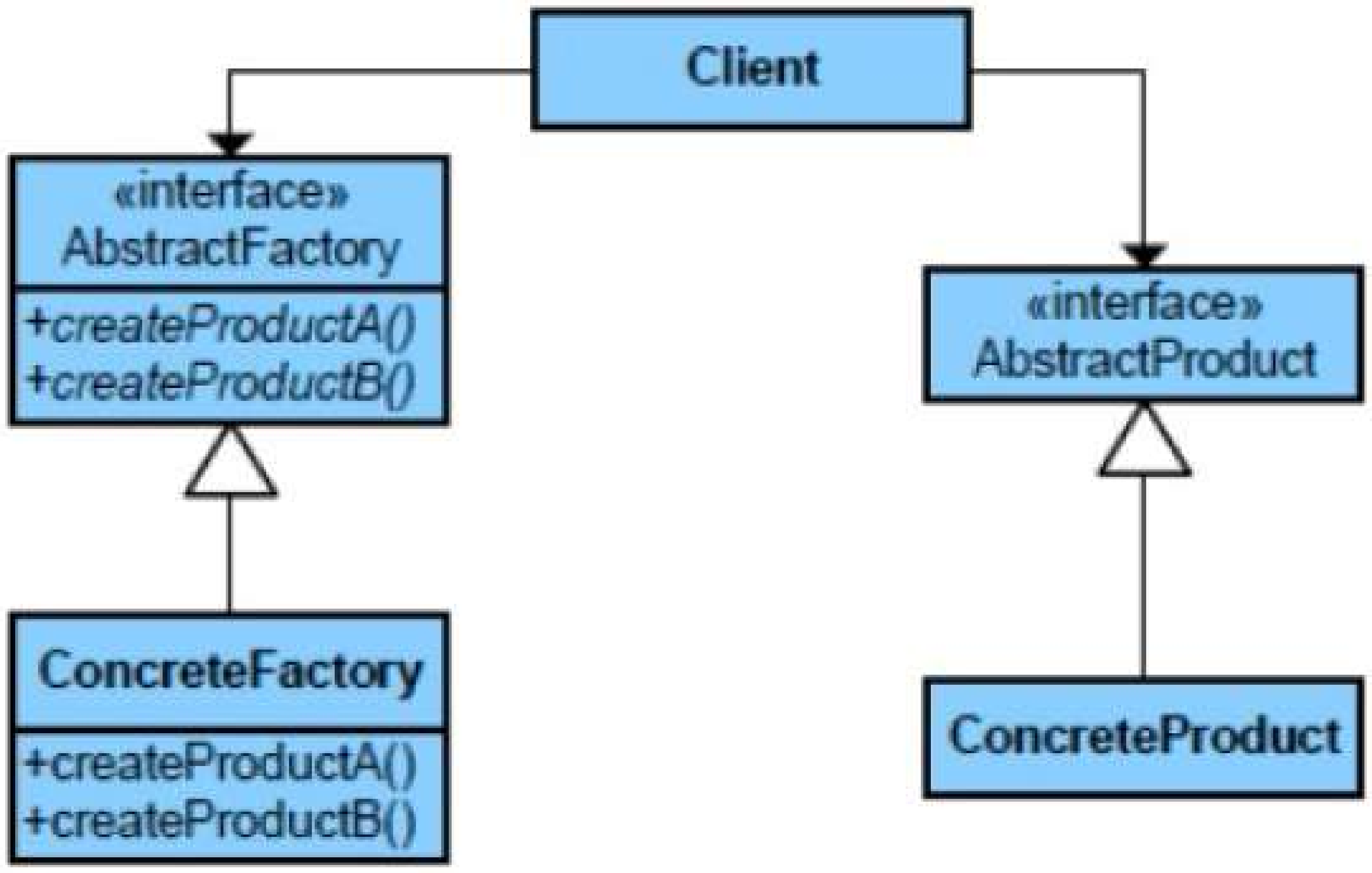
Proxy

It provides a surrogate or placeholder for another object, controlling access to it. This pattern is useful for adding a layer of control, such as lazy initialization, access control, or logging, without modifying the original object.
Type : Structural



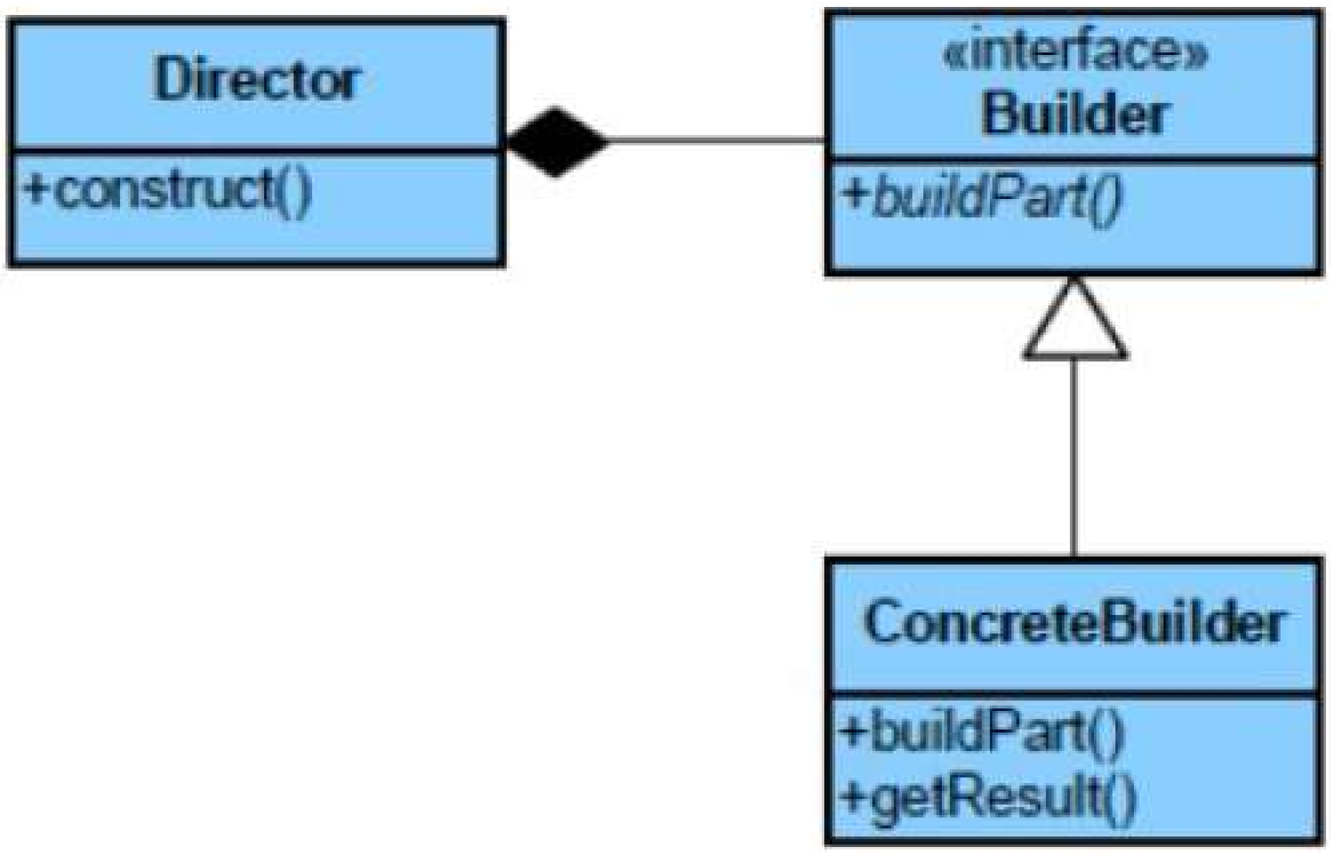
Abstract Factory

It provides an interface for creating families of related or dependent objects without specifying their concrete classes. This ensures that the client can work with a consistent set of objects, promoting flexibility and scalability.
Type : Creational



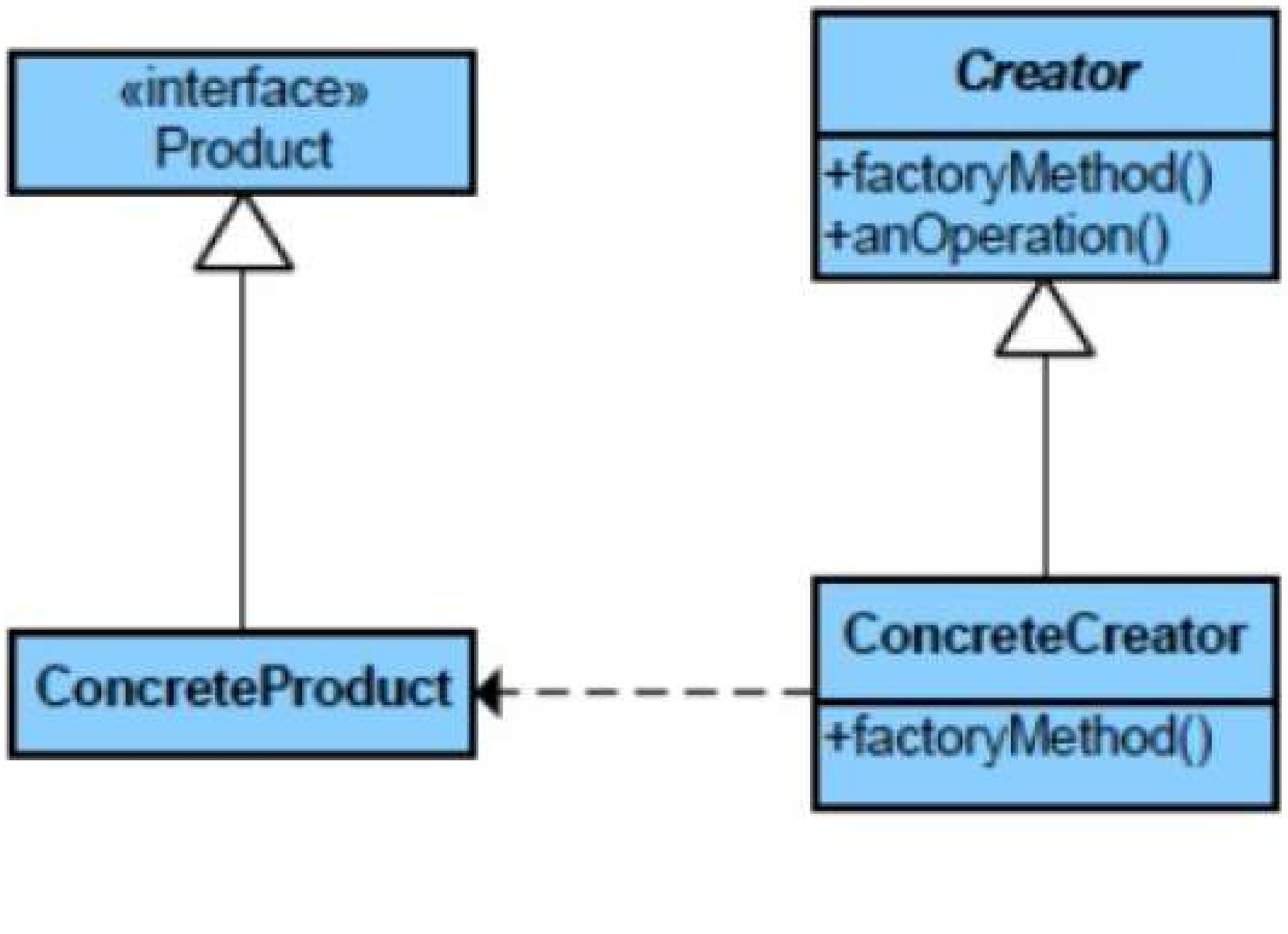
Builder

It separates the construction of a complex object from its representation, allowing the same construction process to create different representations. It enhances flexibility and simplifies the creation of complex objects.
Type : Creational



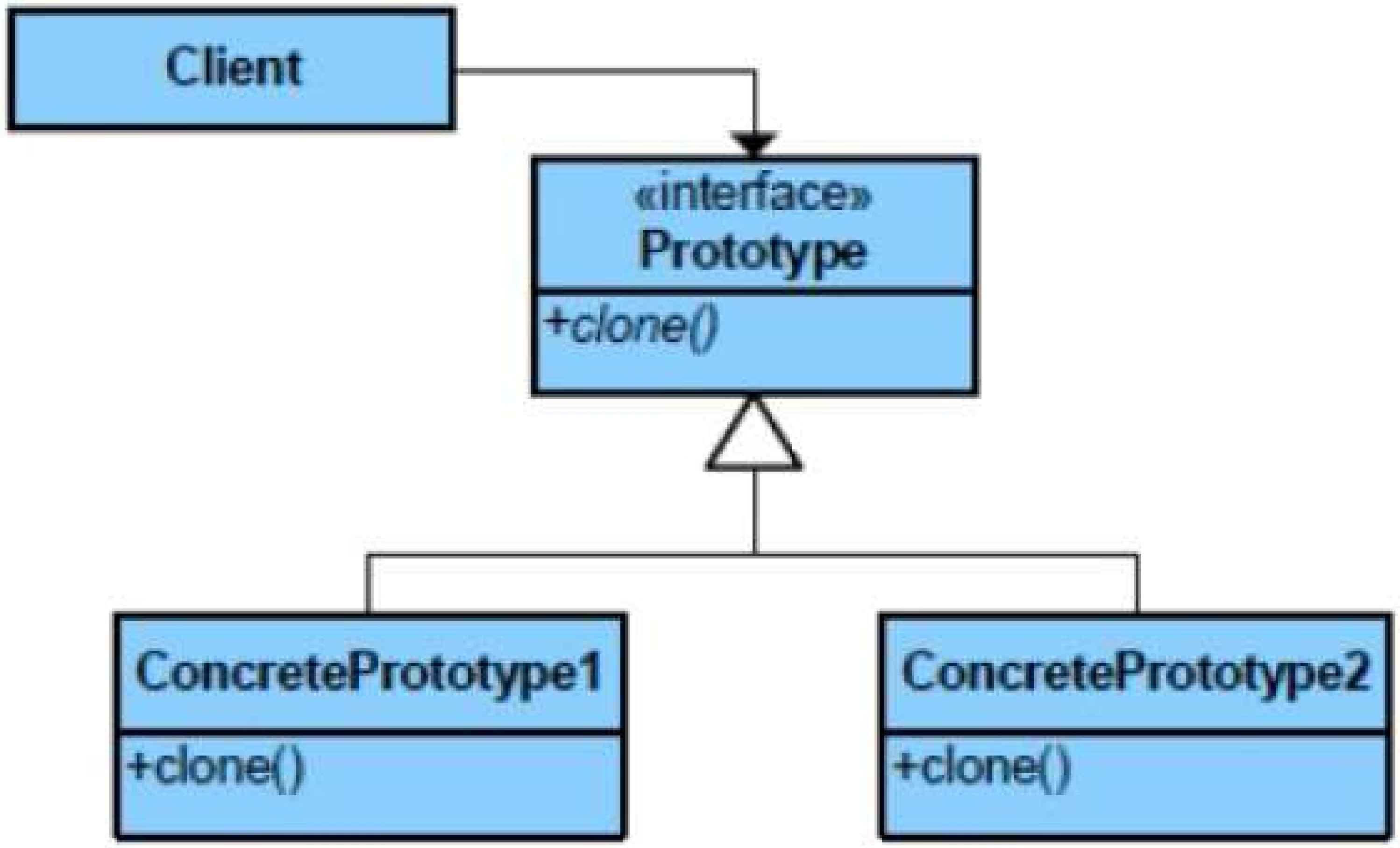
Factory Method

It defines an interface for creating an object, but allows subclasses to decide which class to instantiate. This approach lets a class defer instantiation to its subclasses, promoting flexibility and extensibility.
Type : Creational



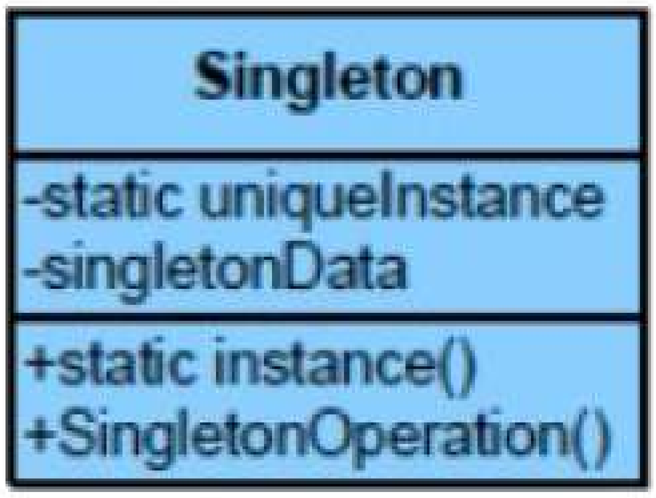
Prototype

It specifies the types of objects to create using a prototypical instance, allowing new objects to be created by copying this prototype. This approach facilitates object creation while preserving the original instance's state.
Type : Creational



Singleton

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. This pattern is useful for managing shared resources or coordinating actions across a system.
Type : Creational



Learned something?

Follow for more like this

Mohammad Faisal

@mohammadfaisalkhan