# OBJECT ORIENTED ANALYSIS & DESIGN
# DATA STRUCTURES & ALGORITHMS

## Structural and Behavioral Patterns

LearnOA®

# Bridge

## Definition

Decouples an abstraction from its implementation so that the two can vary independently

## Problem & Context

An abstraction can be designed as an interface with one or more concrete implementers. When sub classing the hierarchy, it could lead to an exponential number of subclasses. And since both the interface and its implementation are closely tied together, they cannot be independently varied without affecting each other

## Solution

Put both the interfaces and the implementations into separate class hierarchies. The Abstraction maintains an object reference of the Implementer type. A client application can choose a desired abstraction type from the Abstraction class hierarchy. The abstraction object can then be configured with an instance of an appropriate implementer from the Implementer class hierarchy

# Composite

## Definition

Composes objects into tree structures to represent part-whole hierarchies. Composites let clients treat individual objects and compositions of objects uniformly

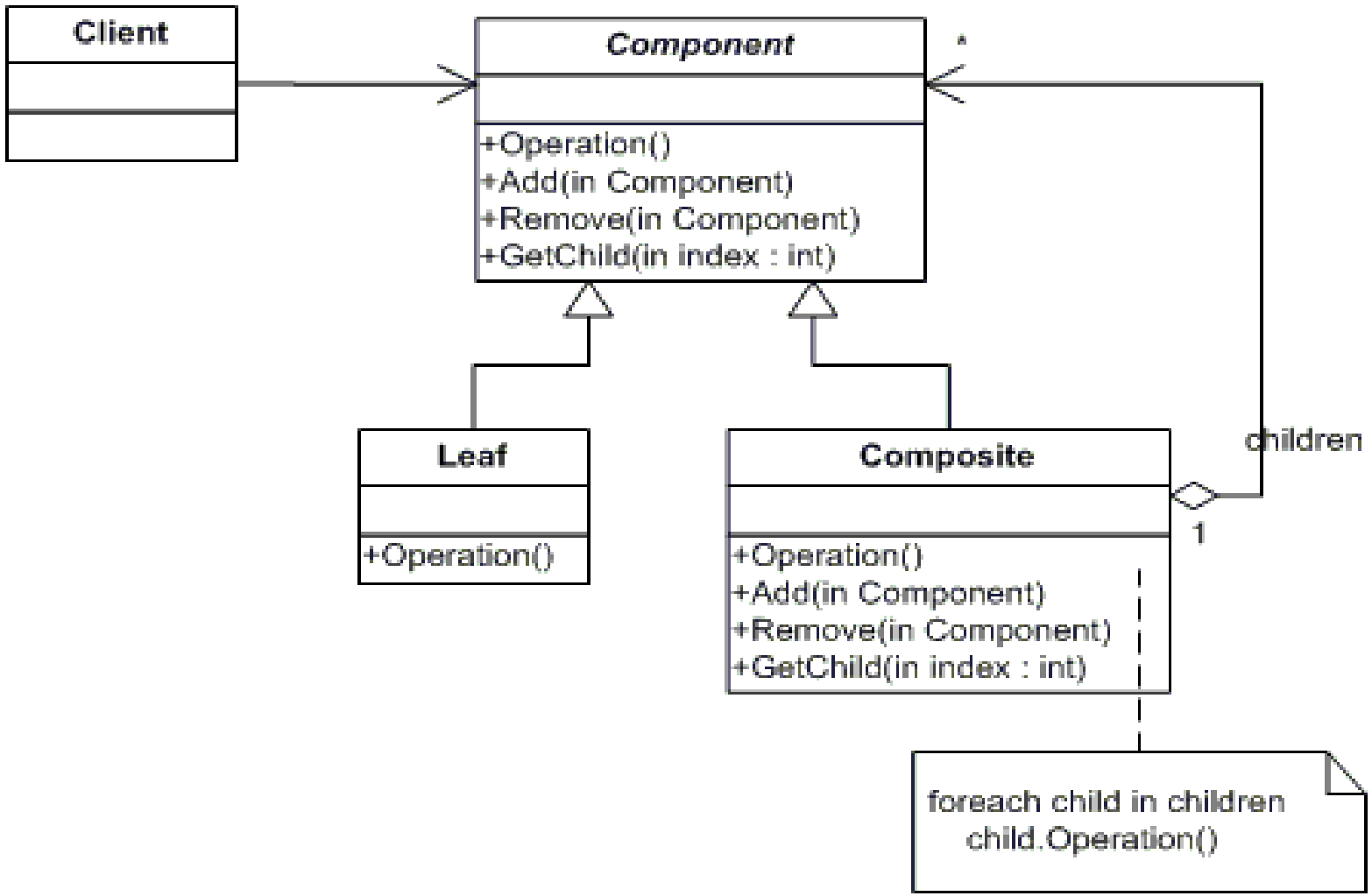
## Problem & Context

Useful in designing a common interface for both individual and composite components so that client programs can view both the individual components and groups of components uniformly

## Solution

Simulate a file system that consists of directories, files and an interface called FileSystem with methods (such as getSize(), getComponent(), addComponent()) common for both File and Directory components. File and Directory classes are the implementers of the interface. A typical client would create a set of FileSystem objects including a hierarchy of file system objects. The client can treat both the Directory and File objects identically

# Composite (Diagram)



# Decorator

## Definition

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

## Problem & Context

This pattern allows new/additional behavior to be added to an existing method of an object dynamically. Since classes cannot be created at runtime and it is typically not possible to predict what extensions will be needed at design time, a new class would have to be made for every possible combination. By contrast, decorators are objects, created at runtime, and can be combined on a per-use basis

## Solution

Pass the original object as a parameter to the constructor of the decorator, with the decorator implementing the new functionality. The interface of the original object needs to be maintained by the decorator

# Façade

## Definition

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

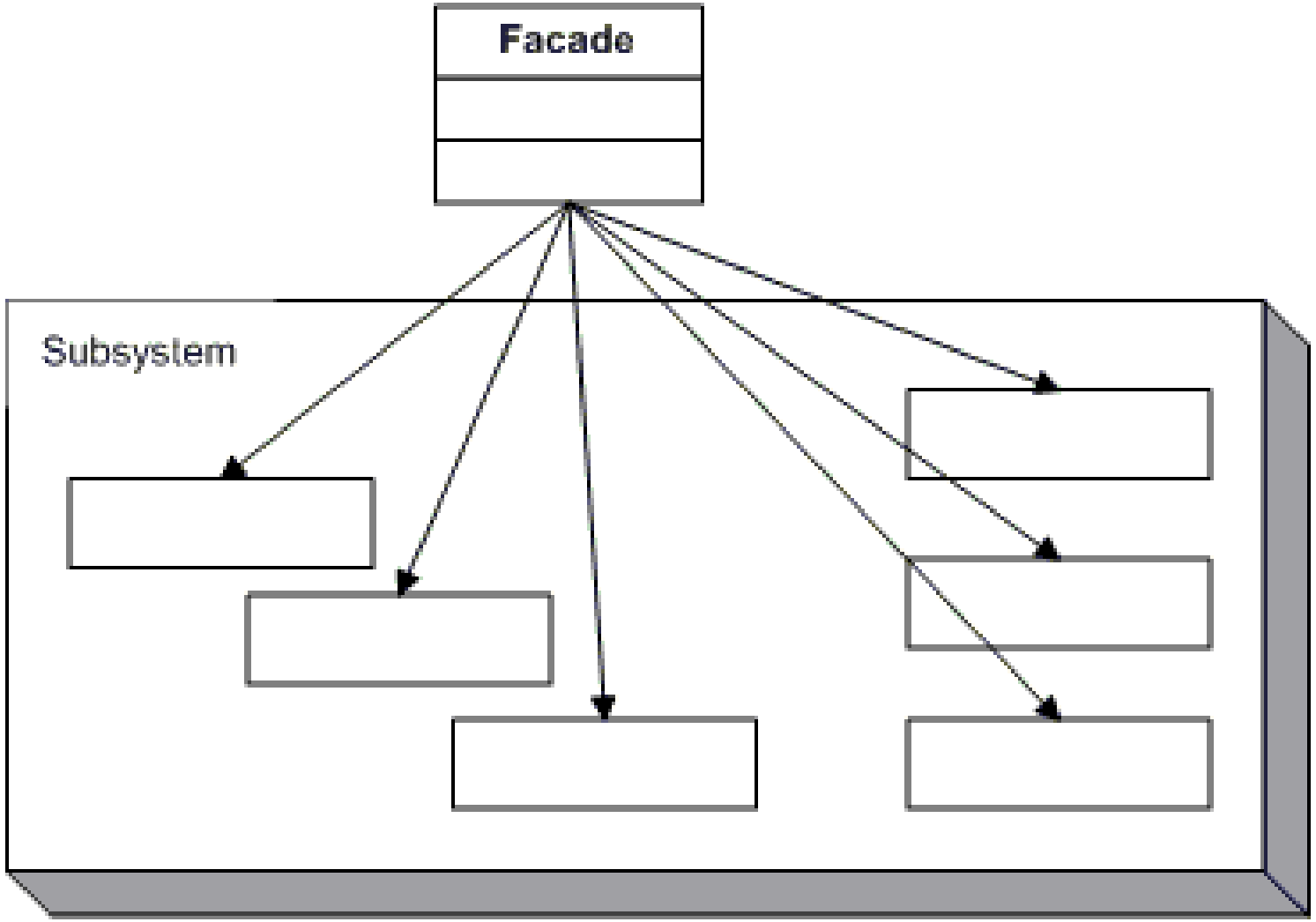
## Problem & Context

The Façade object takes up the responsibility of interacting with the subsystem classes. In effect, clients interface with the façade to deal with the subsystem. Consequently,  the Façade pattern promotes a weak coupling between a subsystem and its clients

## Solution

A façade should not be designed to provide any additional functionality. Never return subsystem components from Façade methods to clients

# Façade (Diagram)



# Flyweight

## Definition

Uses sharing to support large numbers of fine-grained objects efficiently

## Problem & Context

This pattern suggests separating all the intrinsic common data into a separate object referred to as a Flyweight object. The group of objects being created can share the Flyweight object as it represents their intrinsic state. This eliminates the need for storing the same invariant, intrinsic information in every object; instead it is stored only once in the form of a single Flyweight object

## Solution

Design flyweight as a singleton with a private constructor. The client creates an object with the exclusive extrinsic data or sends the extrinsic data as part of a method call to the Flyweight object. This results in the creation of few objects with no duplication

# Proxy

## Definition

Provides a surrogate or placeholder for another object to control access to it

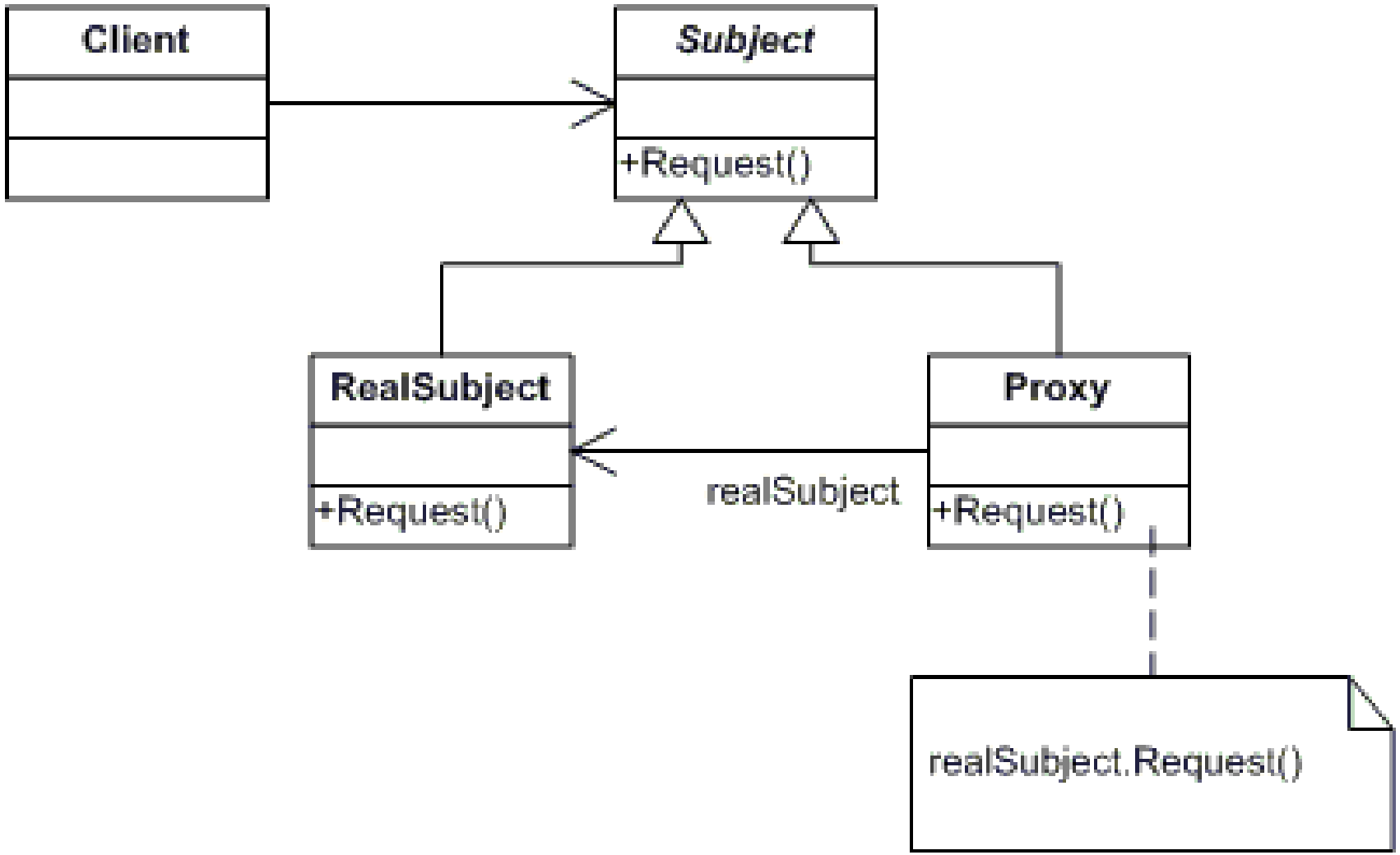
## Problem & Context

A client object can directly access a service provider object but sometimes a client object may not have access to a target object by normal means. The reasons could be: location, state of existence, and special behavior of target object. Instead of having client objects to deal with the special requirements for accessing the target object, the Proxy pattern suggests using a separate object referred to as a proxy to provide a means for different client objects to access the target object in a normal, straightforward manner

## Solution

A proxy object should represent a single object.  It provides access control to the single target object, and offers the same interface

# Proxy (Diagram)



# Chain of Responsibility

## Definition

Avoids 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

## Problem & Context

When there is more than one object that can handle or fulfill a client request, each of these potential handler objects can be arranged in the form of a chain, with each object having a pointer to the next object in the chain

## Solution

The order in which the objects form the chain can be decided dynamically at runtime by the client. All potential handler objects should provide a consistent interface. Neither the client object nor any of the handler objects in the chain need to know which object will actually fulfill the request

# Command

## Definition

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

## Problem & Context

Used when there is a proliferation of similar methods and the interface to implement an object becomes unwieldy – too many public methods for other objects to call, an interface that is unworkable and always changing

## Solution

Create an abstraction for the processing in response to client requests by declaring a common interface to be implemented by different concrete implementers referred to as Command objects. A given Command object does not contain the actual implementation of the functionality. Command objects make use of Receiver objects in offering this functionality

# Interpreter

## Definition

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

## Problem & Context

Sometimes an application may process repeated similar requests that are a combination of a set of grammar rules. These requests are distinct but similar in the sense that they are composed using the same set of rules. Instead of treating every distinct combination of rules as a separate case, it may be beneficial for the application to have the ability to interpret a generic combination of rules

## Solution

Design a class hierarchy to represent the set of grammar rules with every class in the hierarchy representing a separate grammar rule. An Interpreter module interprets the sentences constructed using the class hierarchy and carries out the necessary operations

# Iterator

## Definition

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

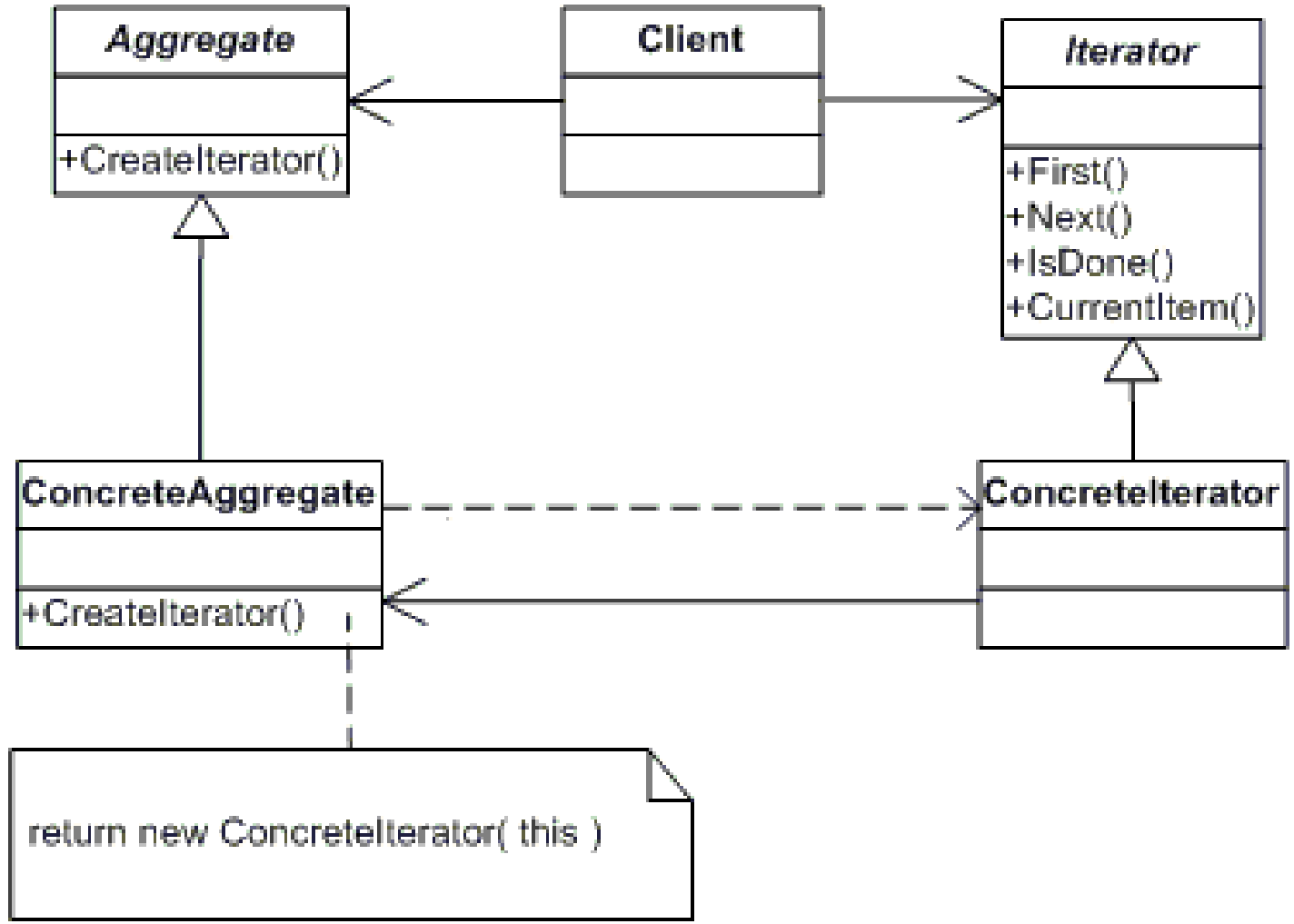
## Problem & Context

The purpose of an iterator is to process every element of a container while isolating the user from the internal structure of the container. The container allows the user to treat it as if it were a simple sequence or list while storing elements in any manner it wishes. Iterators can provide a consistent way to iterate on data structures of all kinds, making the code more readable, reusable, and less sensitive to a change in the data structure

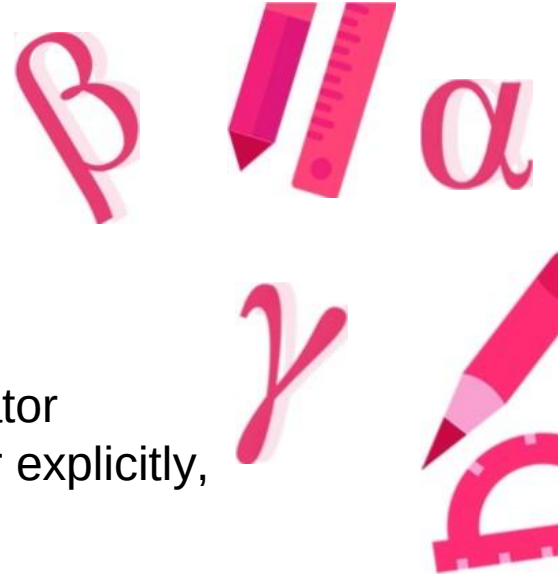
## Solution

In Java, the java.util.Iterator interface allows you to iterate container classes. Each Iterator provides a next() and hasNext() method, and may optionally support a remove() method. Iterators are created by the method iterator() provided by the corresponding container class

# Iterator (Diagram)



# Mediator



## Definition

Defines 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

## Problem & Context

Objects interact with each other for the purpose of providing a service. This interaction can be direct (point-to-point). As the number of objects increases, the interaction can lead to a complex maze of references among objects. Having an object directly referring to other objects greatly reduces the scope for reuse

## Solution

The Mediator pattern suggests abstracting all object interaction details into a separate class, referred to as a Mediator. The interaction between any two different objects is routed through the Mediator class. All objects send their messages to the mediator

# Memento

## Definition

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

## Problem & Context

The state of an object can be defined as the values of its attributes at any given point of time. The Memento pattern is useful for designing a mechanism to capture and store the state of an object so that, when needed, the object can revert to its previous state. This is more like an undo operation

## Solution

The object whose state needs to be captured is referred to as the originator. The originator stores its attributes in a separate object referred to as a Memento. A Memento object must hide the originator variable values from all objects except the originator. A Memento should be designed to provide restricted access to other objects while the originator is allowed to access its internal state

# Observer

## Definition

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

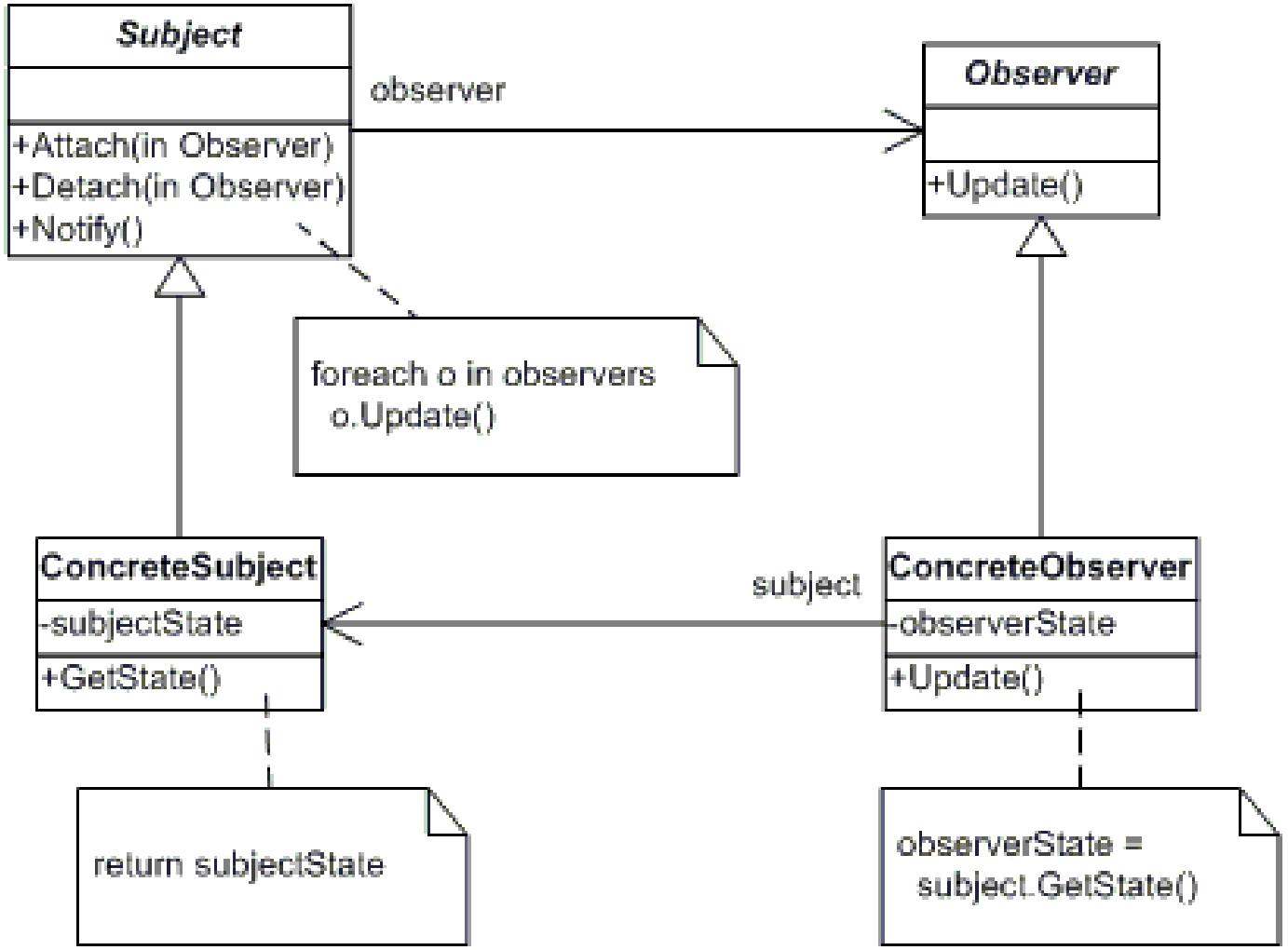
## Problem & Context

Useful for designing a consistent communication model between a set of dependent objects (observers) and an object that they are dependent on (subject). This allows the observers to have their state synchronized with the subject. Each of these observers needs to know when the subject undergoes a change in its state

## Solution

The subject should provide an interface for registering and unregistering change notifications. Observers should provide an interface for receiving notifications from the subject

# Observer (Diagram)



# State

## Definition

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

## Problem & Context

Useful in designing an efficient structure for a class, a typical instance of which can exist in many different states and exhibit different behavior depending on its state. In the case of an object of such a class, some or all of its behavior is completely influenced by its current state. Such a class is referred to as a Context class. A Context object can alter its behavior when there is a change in its internal state and is also referred as a Stateful object

## Solution

The State pattern suggests moving the state-specific behavior out of the Context class into a set of separate classes referred to as State classes. Each of the many different states in which a Context object can exist can be mapped into a separate State class

# Strategy

## Definition

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it

## Problem & Context

Useful when there is a set of related algorithms and a client object needs to be able to dynamically pick and choose an algorithm that suits its current need. The implementation of each of the algorithms is kept in a separate class referred to as a strategy. An object that uses a Strategy object is referred to as a context object. Changing the behavior of a Context object is a matter of changing its Strategy object to the one that implements the required algorithm

## Solution

All Strategy objects must be designed to offer the same interface. In Java, this can be accomplished by designing each Strategy object either as an implementer of a common interface or as a subclass of a common abstract class that declares the required common interface

# Template Method

## Definition

Defines 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

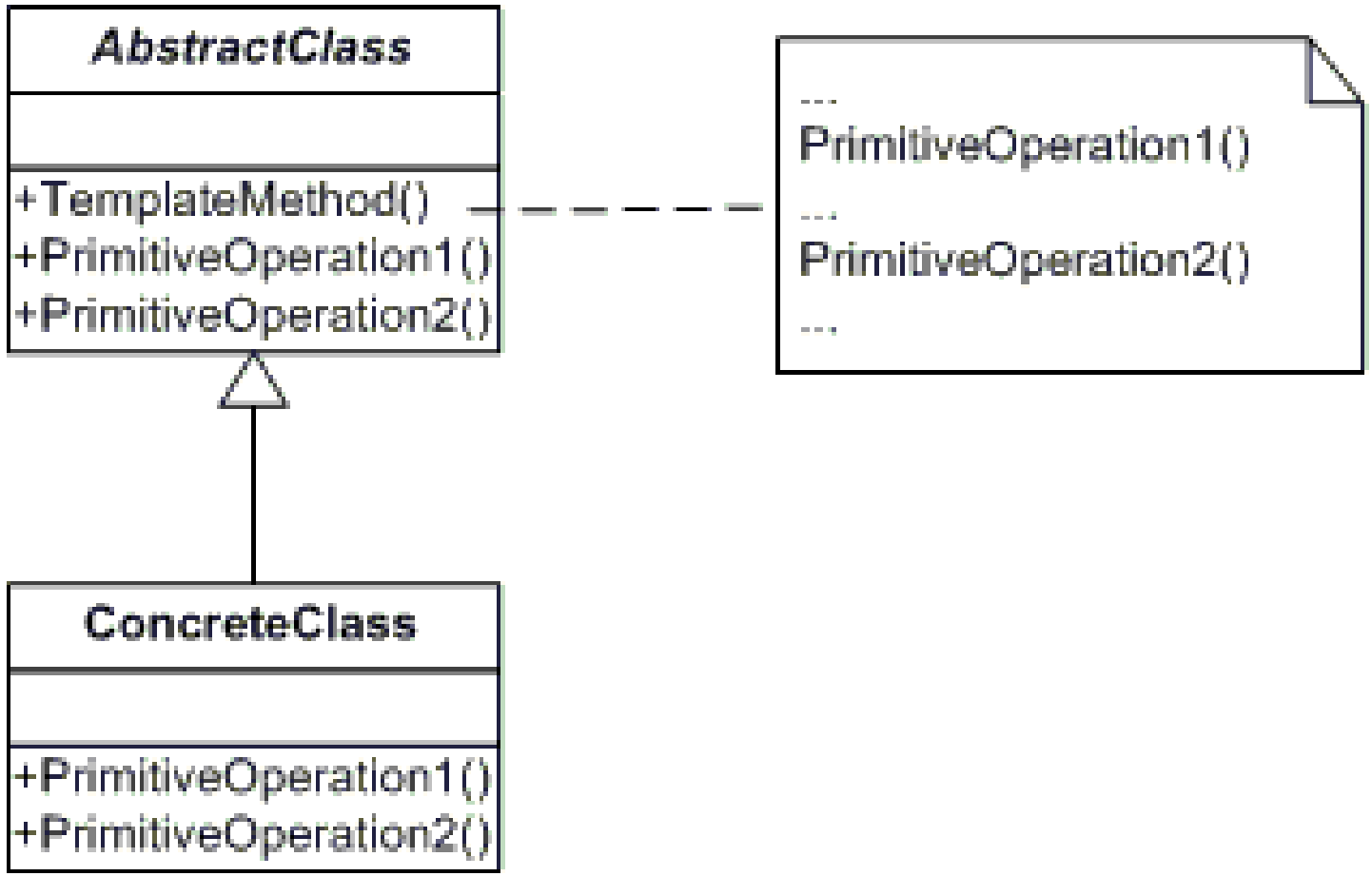
## Problem & Context

Used in situations when there is an algorithm, some steps of which could be implemented in many different ways. The outline of the algorithm is kept in a separate method referred to as a template method inside a class, referred to as a template class, leaving out the specific implementations of the variant portions of the algorithm to different subclasses of this class

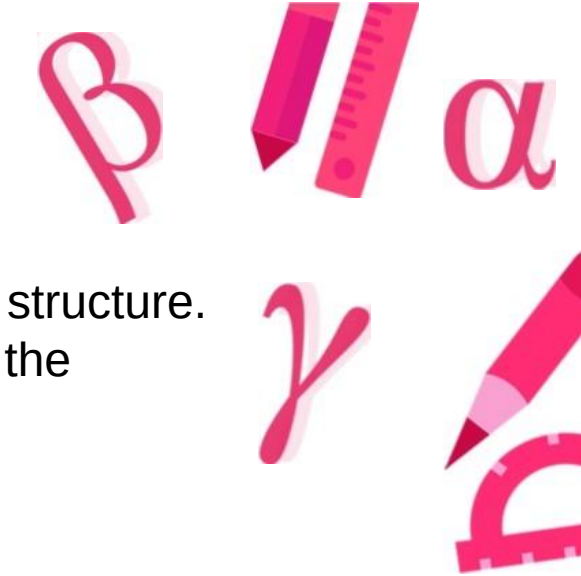
## Solution

The Template method can be a concrete, nonabstract method with calls to other methods that represent different steps of the algorithm. Specific implementations can be provided for these abstract methods inside a set of concrete subclasses of the abstract Template class

# Template Method (Diagram)



# Visitor



## Definition

Represents 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

## Problem & Context

Visitor is useful in a heterogeneous collection of objects of a class hierarchy. It allows operations to be defined without changing the class of any of the objects in the collection. The Visitor pattern suggests defining the operation in a separate class referred to as a visitor class, which separates the operation from the object collection on which it operates. For every new operation to be defined, a new visitor class is created

## Solution

Every visitor class that operates on objects of the same set of classes can be designed to implement a corresponding Visitor interface. A typical Visitor interface declares a set of visit(ObjectType) methods, one for each object type from the object collection. Each of these methods is meant for processing instances of a specific class

# Adapter

## Definition

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

## Problem & Context

Sometimes an existing class may provide the functionality required by a client, but its interface may not be what the client expects. In such cases, the existing interface needs to be converted into an interface that the client expects, preserving the reusability of the existing class

## Solution

Define a wrapper class around the object with the incompatible interface. The adapter provides the required interface expected by the client. The implementation of the adapter interface converts client requests into calls to the adaptee class interface

# Thank You!