DESIGN PATTERNS

Based on the

Gang of Four Design Patterns

and

dofactory.com

by

NeptuneLuke

Introduction

This is **solely a summary** on Design Patterns, it is not meant to be a complete guide/explanation of patterns and it does not substitute academic or professional sources.
All images and text are taken either from the **Gang of Four Design Patterns** book or from **dofactory.com**.
Consider this as a cheatsheet or a quick-to-consult file.

Design patterns are solutions to software design problems you find again and again in real-world application development.
Patterns are about design and interaction of objects, as well as providing a communication platform concerning elegant, reusable solutions to commonly encountered programming challenges.

The **Gang of Four** (GoF) **patterns are generally considered the foundation for all other patterns**. A total of 23 GoF patterns exist.
They are categorized in three groups:

- **Creational**
- **Structural**
- **Behavioral**

Here you will find information on each of these patterns, including UML diagrams and C# source codes (dofactory.com).

Remember that patterns are powerful tools, but they may not always be the most effective solution. When applying patterns in your own projects, it is best to keep an open mind and, when in doubt, run some simple performance tests.

Patterns Summary

CREATIONAL Patterns

Abstract Factory	Creates an instance of several families of classes
Builder	Separates object construction from its representation
Factory Method	Creates an instance of several derived classes
Prototype	A fully initialized instance to be copied or cloned
Singleton	A class of which only a single instance can exist

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
Flyweight	A fine-grained instance used for efficient sharing
Proxy	An object representing another object

BEHAVIORAL Patterns

Chain of Responsibilities	A way of passing a request between a chain of objects

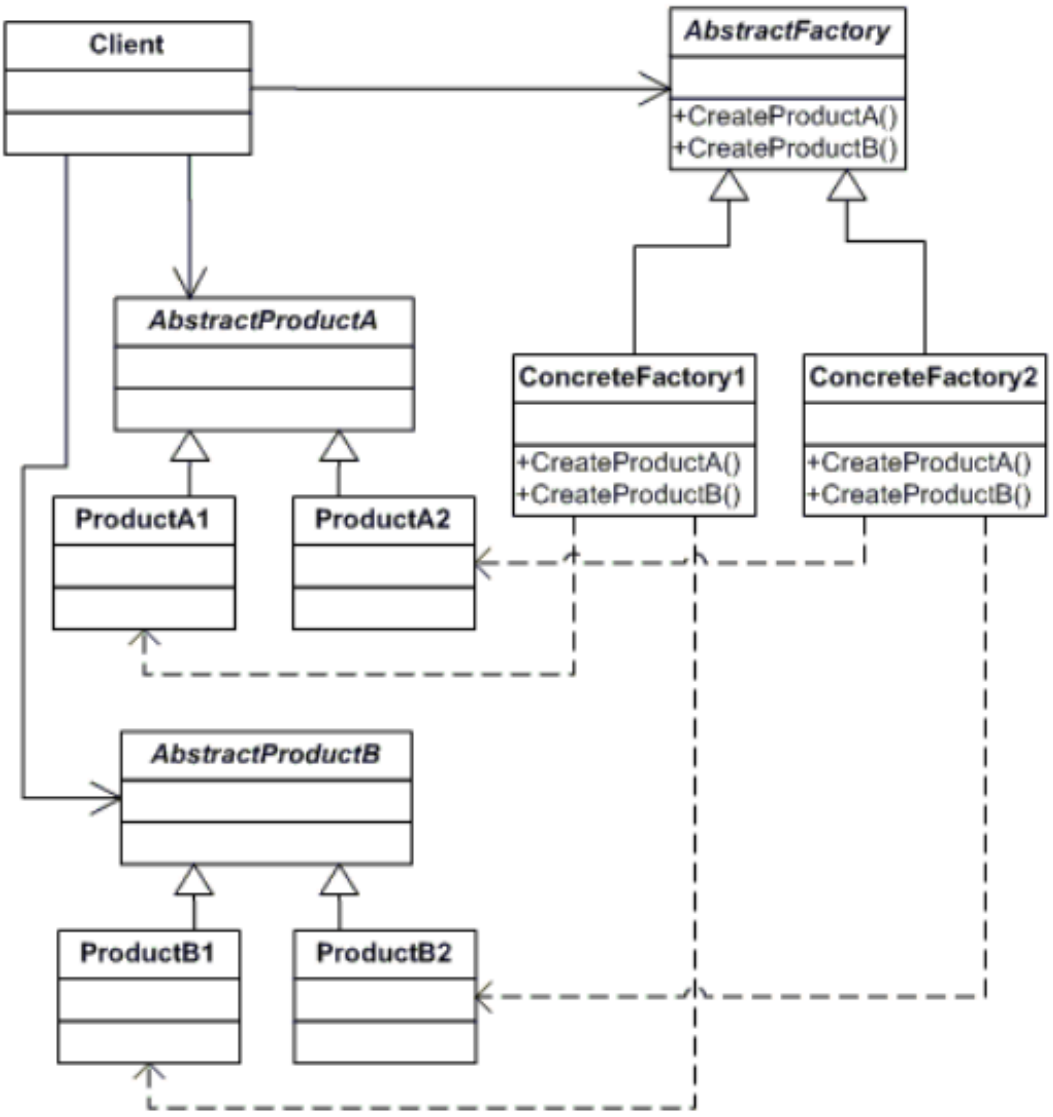
Command	Encapsulate a command request as an object
Interpreter	A way to include language elements in a program
Iterator	Sequentially access the elements of a collection
Mediator	Defines simplified communication between classes
Memento	Capture and restore an object's internal state
Observer	A way of notifying 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

CREATIONAL Patterns

Abstract Factory

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

UML Class Diagram



- **Abstract Factory** (ContinentFactory):
  - declares an interface for operations that create abstracts products

- **ConcreteFactory** (AfricaFactory, AmericaFactory):
  - implements the operations to create concrete product objects

- **AbstractProduct** (Herbivore, Carnivore):
  - declares an interface for a type of product object

- **Product** (Cow, Bison, Lion, Wolf):
  - defines a product object to be created by the corresponding concrete factory implements the AbstractProduct interface

- **Client** (AnimalWorld):
  - uses interfaces declared by AbstractFactory and AbstractProduct classes

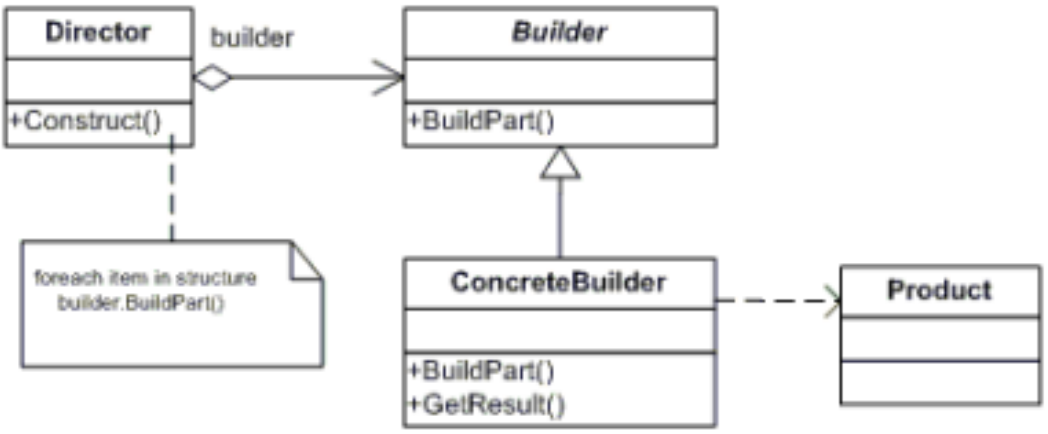
Code example

C# **Abstract** Factory Design Pattern (dofactory.com)

Builder

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

UML Class Diagram



- **Builder** (VehicleBuilder):
    - specifies an abstract interface for creating parts of a Product object

- **ConcreteBuilder** (MotorCycleBuilder, CarBuilder, ScooterBuilder):
    - constructs and assembles parts of the product by implementing the Builder interface
    - defines and keeps track of the representation it creates
    - provides an interface for retrieving the product

- **Director** (Shop):
    - constructs an object using the Builder interface

- **Product** (Vehicle):

- represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled
- includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

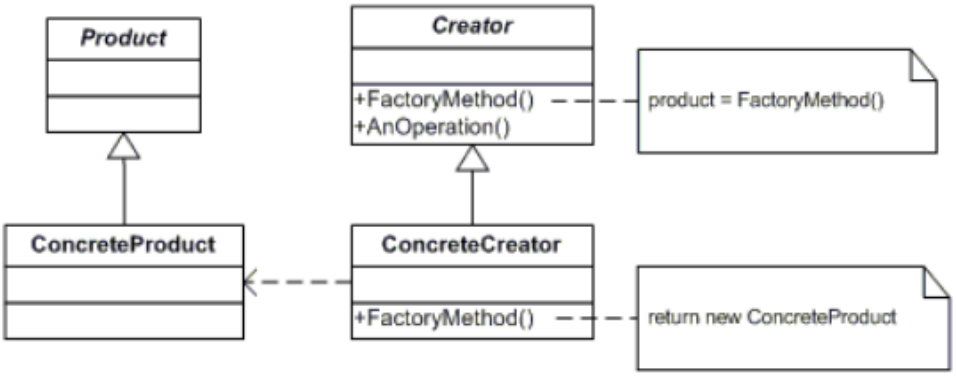
Code example

C# **Builder** Design Pattern (dofactory.com)

Factory Method

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

UML Class Diagram



- **Product** (Page):
  - defines the interface of objects the factory method creates

- **ConcreteProduct** (SkillsPage, EducationPage, ExperiencePage):
  - implements the Product interface

- **Creator** (Document):
  - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object
  - may call the factory method to create a Product object

- **ConcreteCreator** (Report, Resume):
  - overrides the factory method to return an instance of a ConcreteProduct

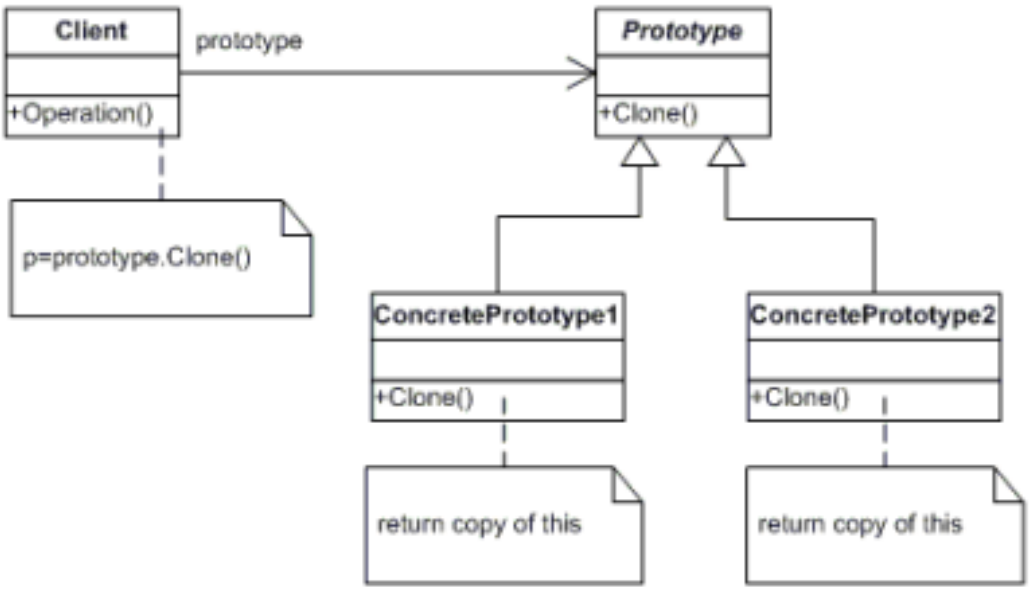
Code example

C# **Factory** Method Design Pattern (dofactory.com)

Prototype

Definition
Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.
Medium frequency of use.

UML Class Diagram



- **Prototype** (ColorPrototype):
  - declares an interface for cloning itself

- **ConcretePrototype** (Color):
  - implements an operation for cloning itself

- **Client** (ColorManager):
  - creates a new object by asking a prototype to clone itself

Code example
C# **Prototype** Design Pattern (dofactory.com)

Singleton

Definition
Ensure a class has only one instance and provide a global point of access to it.
Medium-high frequency of use.

UML Class Diagram

- **Singleton** (LoadBalancer):
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation
  - responsible for creating and maintaining its own unique instance

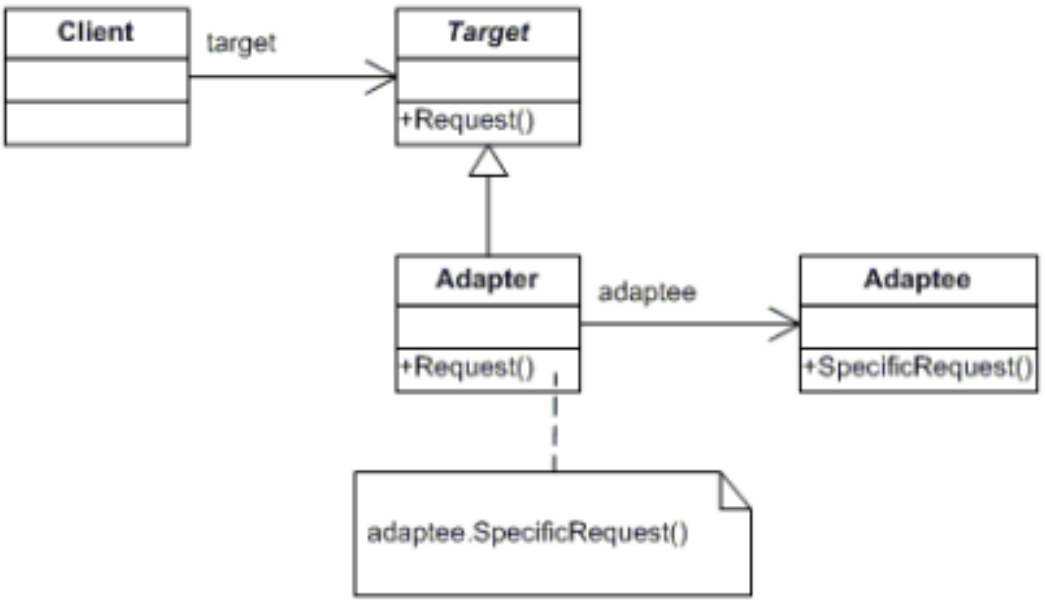
Code example
C# **Singleton** Design Pattern (dofactory.com)

STRUCTURAL Patterns

Adapter

Definition
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
Medium-high frequency of use.

UML Class Diagram



- **Target** (ChemicalCompound):
  - defines the domain-specific interface that Client uses

- **Adapter** (Compound):
  - adapts the interface Adaptee to the Target interface

- **Adaptee** (ChemicalDatabank):
  - defines an existing interface that needs adapting

- **Client** (AdapterApp):
    - collaborates with objects conforming to the Target interface

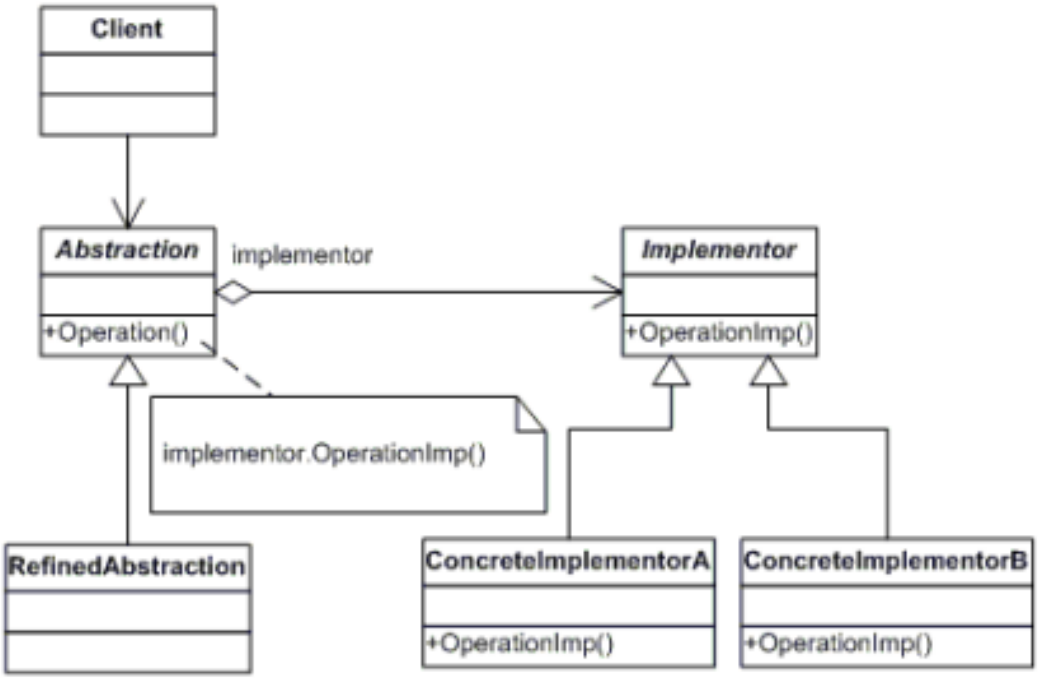
Code example

C# **Adapter** Design Pattern (dofactory.com)

Bridge

Definition
Decouple an abstraction from its implementation so that the two can vary independently.
Medium frequency of use.

UML Class Diagram



- **Abstraction** (BusinessObject):
    - defines the abstraction's interface
    - maintains a reference to an object of type Implementor

- **RefinedAbstraction** (CustomersBusinessObject):
    - extends the interface defined by Abstraction

- **Implementor** (DataObject):
    - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the

Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives

- **ConcreteImplementor** (CustomersDataObject):
    - implements the Implementor interface and defines its concrete implementation

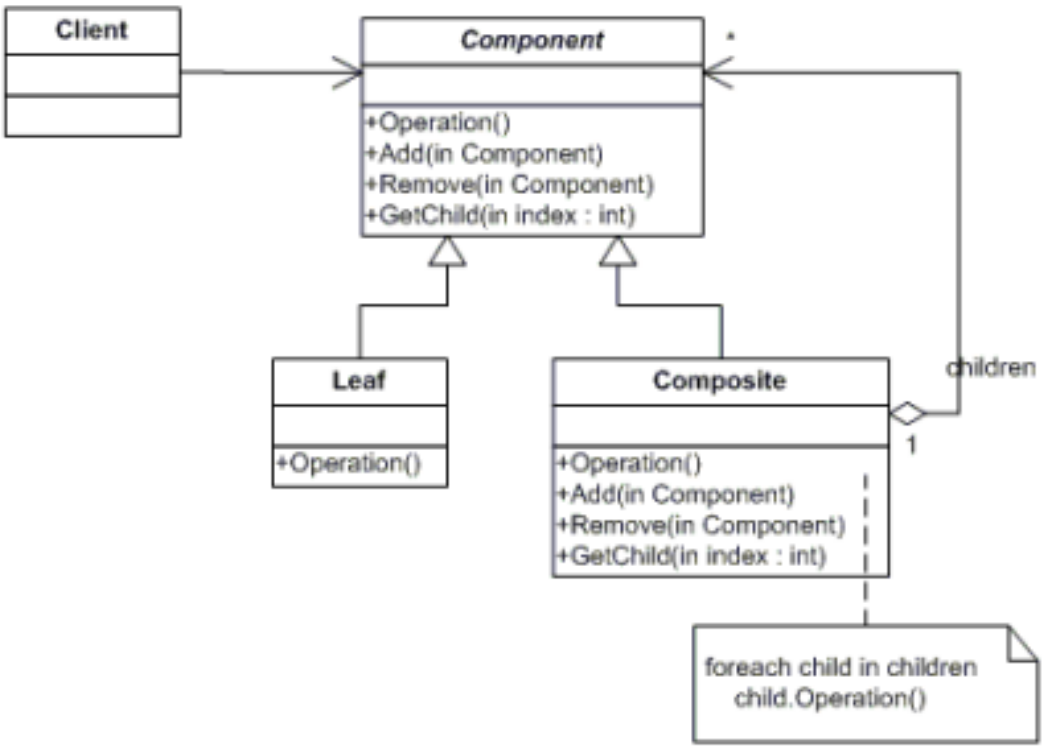
Code example

C# **Bridge** Design Pattern (dofactory.com)

Composite

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

UML Class Diagram



- **Component** (DrawingElement):
  - declares the interface for objects in the composition
  - implements default behavior for the interface common to all classes, as appropriate
  - declares an interface for accessing and managing its child components
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate

- **Leaf** (PrimitiveElement):
  - represents leaf objects in the composition. A leaf has no children
  - defines behavior for primitive objects in the composition

- **Composite** (CompositeElement):
  - defines behavior for components having children
  - stores child components
  - implements child-related operations in the Component interface

- **Client** (CompositeApp):
  - manipulates objects in the composition through the Component interface

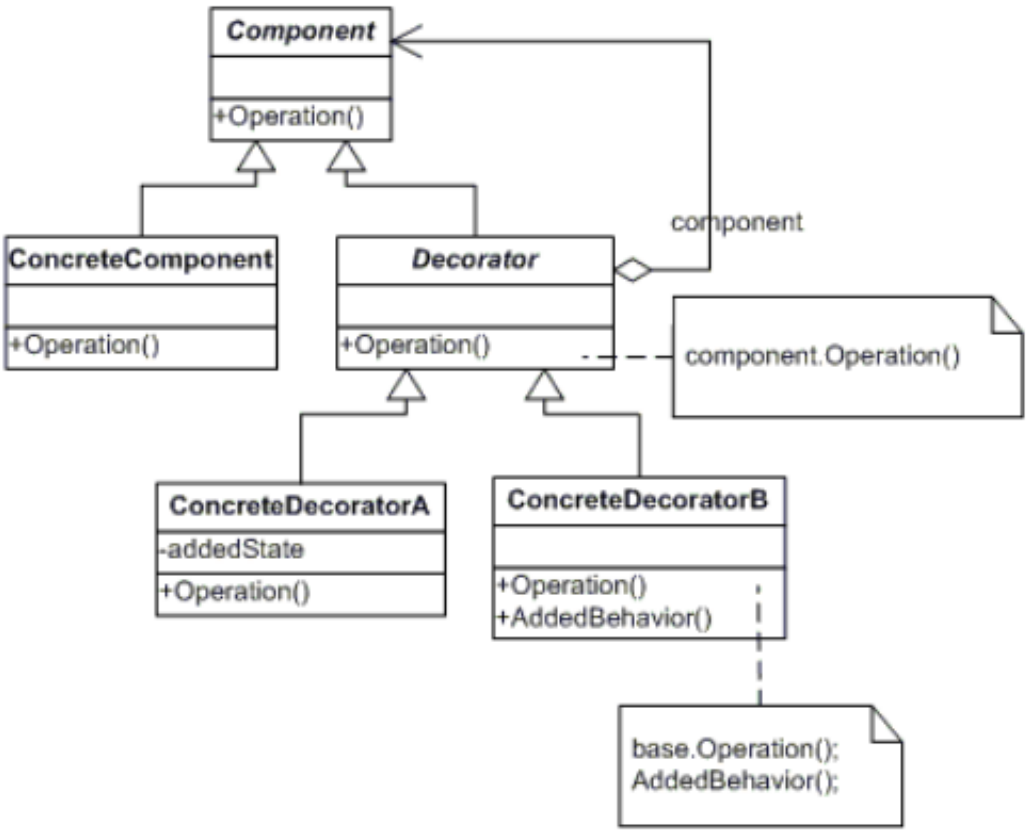
Code example

C# **Composite** Design Pattern (dofactory.com)

Decorator

Definition
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. Medium frequency of use.

UML Class Diagram



- **Component** (LibraryItem):
    - defines the interface for objects that can have responsibilities added to them dynamically

- **ConcreteComponent** (Book, Video):
    - defines an object to which additional responsibilities can be attached

- **Decorator** (Decorator):
    - maintains a reference to a Component object and defines an interface that conforms to Component's interface

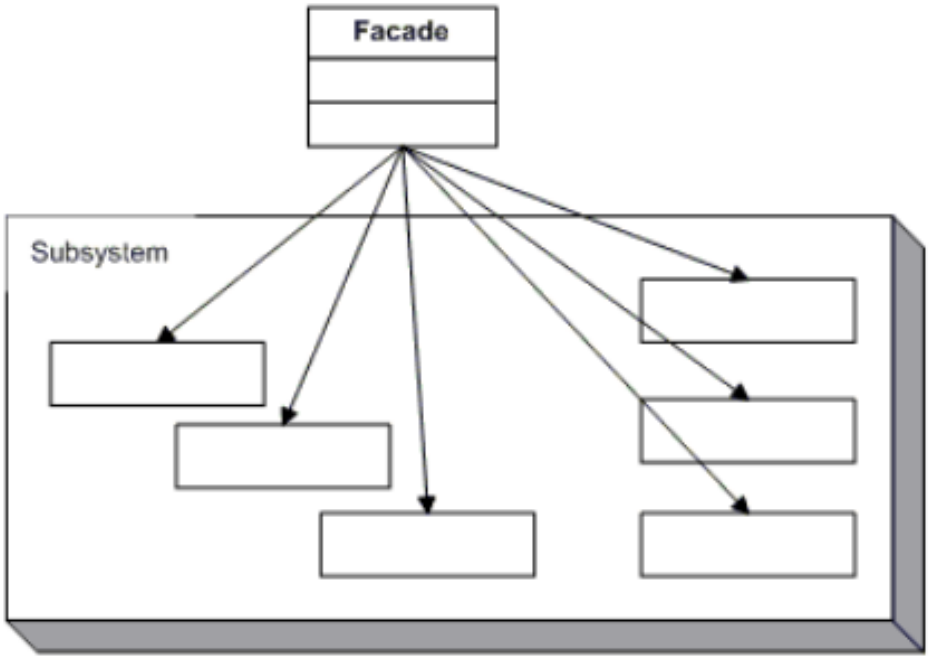
- **ConcreteDecorator** (Borrowable):
    - adds responsibilities to the component

Code example
C# **Decorator** Design Pattern (dofactory.com)

Facade

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

UML Class Diagram



- **Facade** (MortgageApplication):
  - knows which subsystem classes are responsible for a request
  - delegates client requests to appropriate subsystem objects

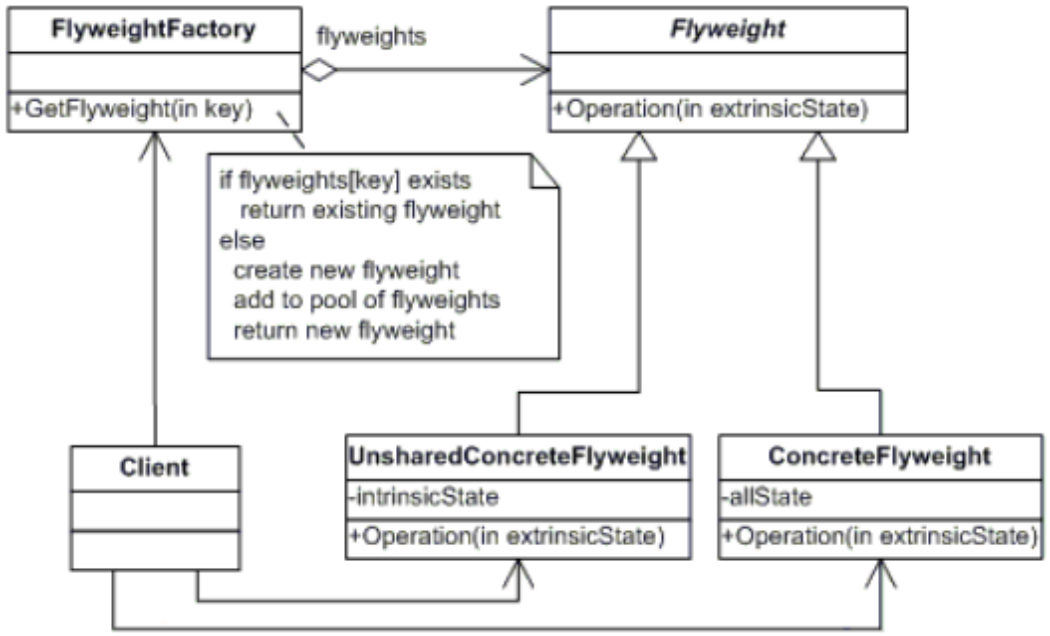
- **Subsystem classes** (Bank, Credit, Loan):
  - implement subsystem functionality
  - handle work assigned by the Facade object
  - have no knowledge of the facade and keep no reference to it

Code example
C# **Facade** Design Pattern (dofactory.com)

Flyweight

Definition
Use sharing to support large numbers of fine-grained objects efficiently.
Low frequency of use.

UML Class Diagram



- **Flyweight** (Character):
    - declares an interface through which flyweights can receive and act on extrinsic state

- **ConcreteFlyweight** (CharacterA, CharacterB, ..., CharacterZ):
    - implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the ConcreteFlyweight object's context

- **UnsharedConcreteFlyweight** (not used):
    - not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing, but it doesn't enforce it. It is common

for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have)

- **FlyweightFactory** (CharacterFactory):
  - creates and manages flyweight objects
  - ensures that flyweight are shared properly. When a client requests a flyweight, the FlyweightFactory objects supplies an existing instance or creates one, if none exists

- **Client** (FlyweightApp):
  - maintains a reference to flyweight(s)
  - computes or stores the extrinsic state of flyweight(s)

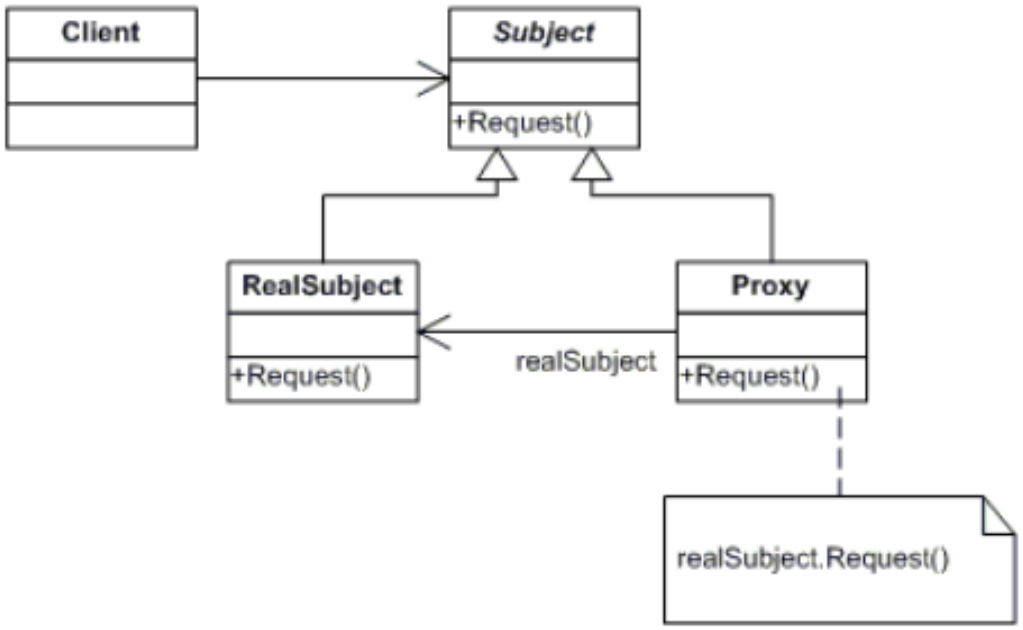
Code example

C# **Flyweight** Design Pattern (dofactory.com)

Proxy

Definition
Provide a surrogate or placeholder for another object to control access to it.
Medium-high frequency of use.

UML Class Diagram



- **Proxy** (MathProxy):
    - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same
    - provides an interface identical to Subject's so that a proxy can be substituted for for the real subject
    - controls access to the real subject and may be responsible for creating and deleting it
    - other responsibilites depend on the kind of proxy:
        - remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space

        - virtual proxies may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real images extent
        - protection proxies check that the caller has the access permissions required to perform a request

- **Subject** (IMath):
    - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected

- **RealSubject** (Math):
    - defines the real object that the proxy represents

Code example

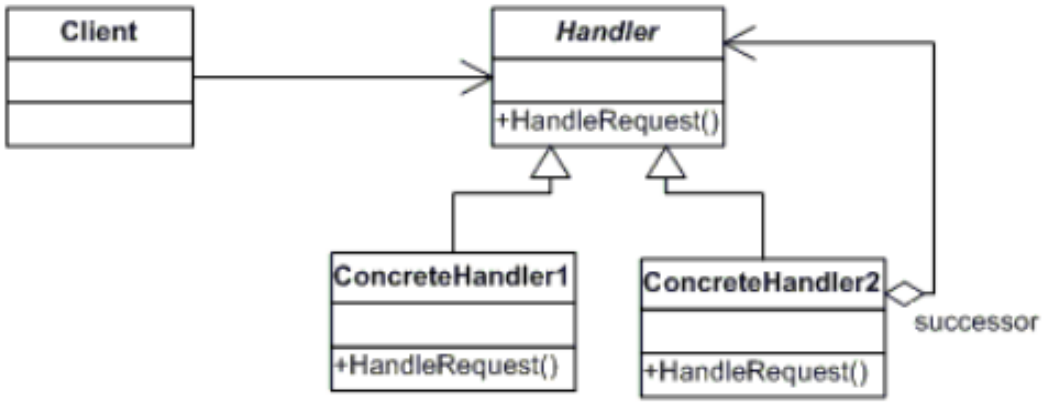
C# **Proxy** Design Pattern (dofactory.com)

BEHAVIORAL Patterns

Chain of Responsibility

Definition
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.
Medium-low frequency of use.

UML Class Diagram



- **Handler** (Approver):
  - defines an interface for handling the requests
  - (optional) implements the successor link

- **ConcreteHandler** (Director, VicePresident, President):
  - handles requests it is responsible for o can access its successor
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor

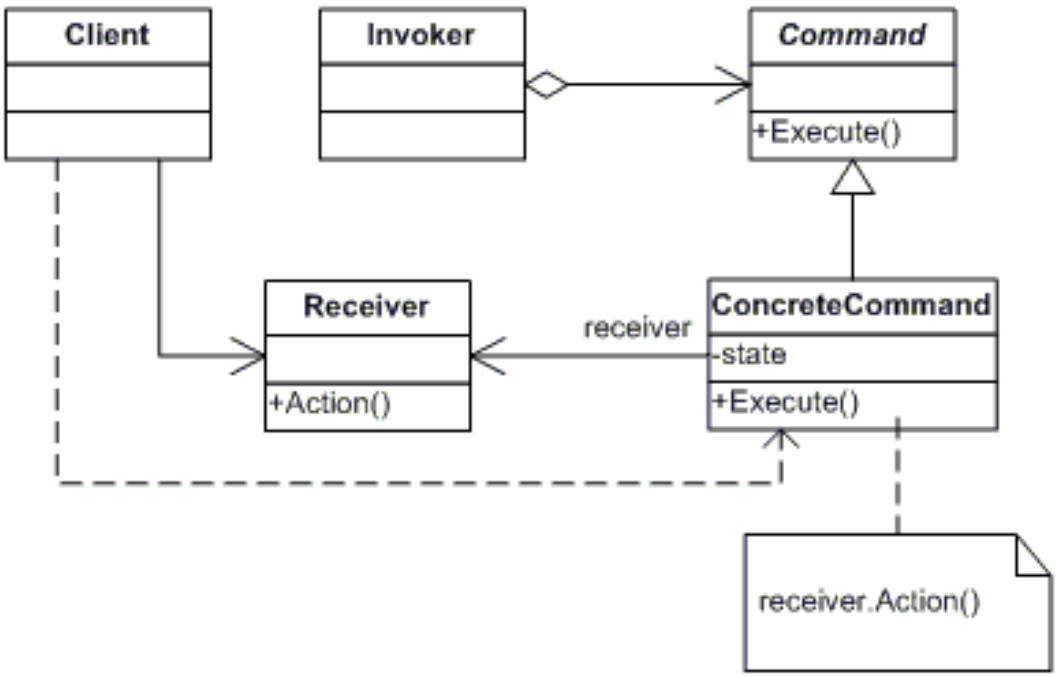
- **Client** (ChainApp):
  - initiates the request to a ConcreteHandler object on the chain

Code example
C# **Chain of Responsibility** Design Pattern (dofactory.com)

Command

Definition
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Medium-high frequency of use.

UML Class Diagram



- **Command** (Command):
  - declares an interface for executing an operation

- **ConcreteCommand** (CalculatorCommand):
  - defines a binding between a Receiver object and an action
  - implements Execute by invoking the corresponding operation(s) on Receiver

- **Client** (CommandApp):
  - creates a ConcreteCommand object and sets its receiver

- **Invoker** (User):
    - asks the command to carry out the request

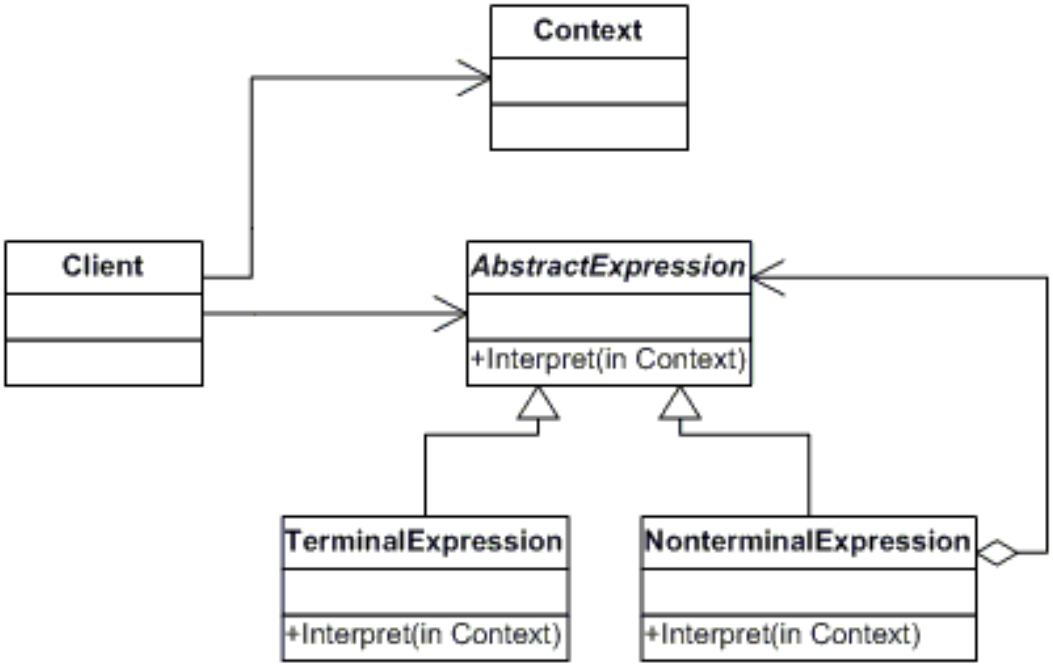
- **Receiver** (Calculator):
    - knows how to perform the operations associated with carrying out the request

Code example
C# **Command** Design Pattern (dofactory.com)

Interpreter

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

UML Class Diagram



- **AbstractExpression** (Expression)
    - declares an interface for executing an operation

- **TerminalExpression** (ThousandExpression, HundredExpression, TenExpression, OneExpression)
    - implements an Interpret operation associated with terminal symbols in the grammar
    - an instance is required for every terminal symbol in the sentence

- **NonterminalExpression** (not used)
  - one such class is required for every rule R ::= R_1 , R_2 . . . . . R_n in the grammar
  - maintains instance variables of type AbstractExpression for each of the symbols R_1 through R_n
  - implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R_1 through R_n

- **Context** (Context)
  - contains information that is global to the interpreter

- **Client** (InterpreterApp)
  - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes
  - invokes the Interpret operation

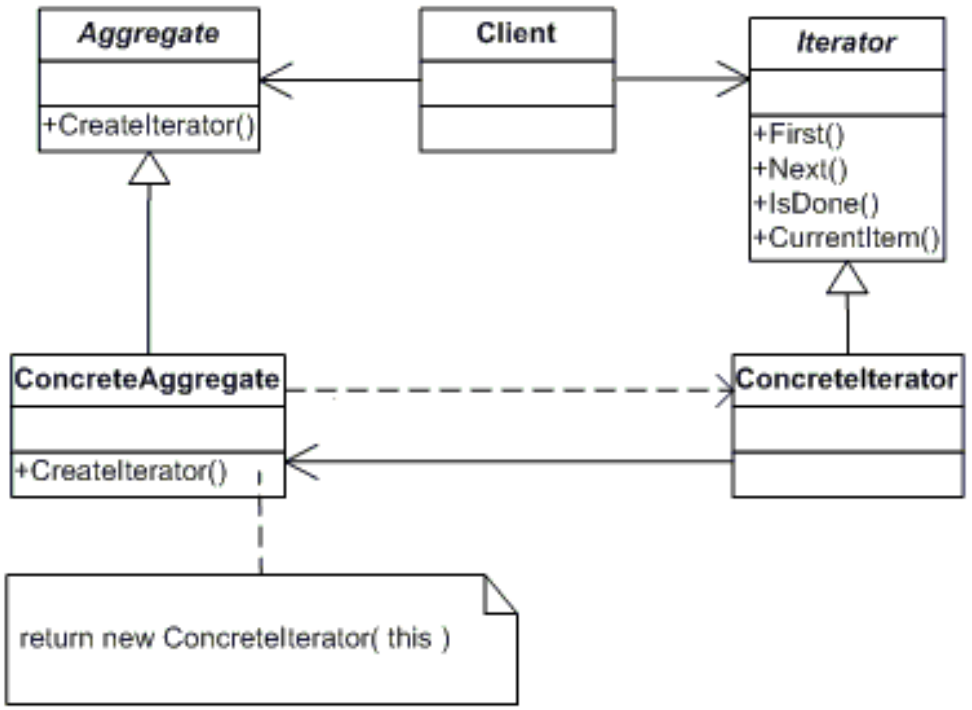
Code example

C# **Interpreter** Design Pattern (dofactory.com)

Iterator

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

UML Class Diagram



- **Iterator** (AbstractIterator)
  - defines an interface for accessing and traversing elements

- **ConcreteIterator** (Iterator)
  - implements the Iterator interface
  - keeps track of the current position in the traversal of the aggregate

- **Aggregate** (AbstractCollection)
  - defines an interface for creating an Iterator object

- **ConcreteAggregate** (Collection)
    - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

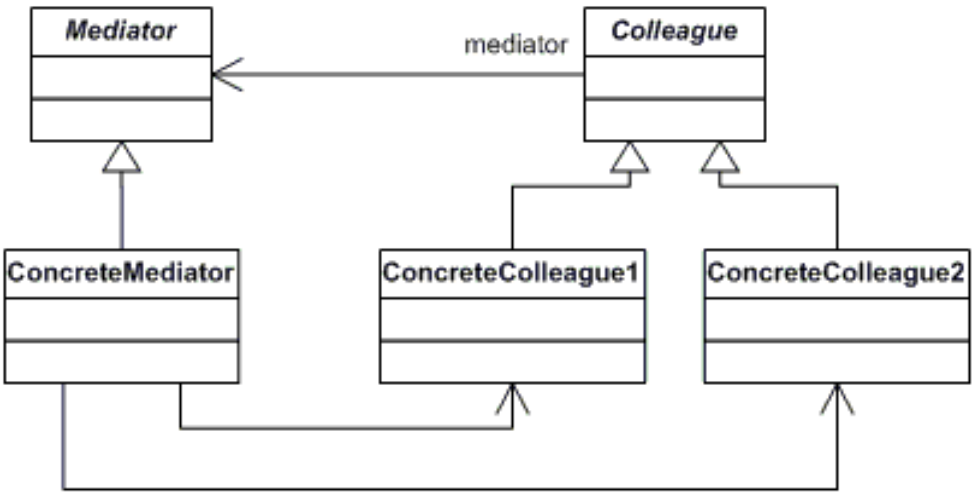
Code example

C# **Iterator** Design Pattern (dofactory.com)

Mediator

Definition
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.
Medium-low frequency of use.

UML Class Diagram



- **Mediator** (IChatroom)
  - defines an interface for communicating with Colleague objects

- **ConcreteMediator** (Chatroom)
  - implements cooperative behavior by coordinating Colleague objects
  - knows and maintains its colleagues

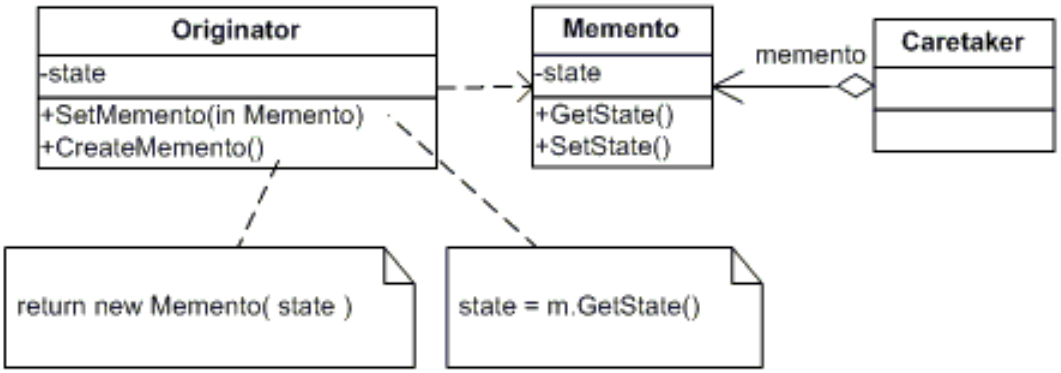
- **Colleague classes** (Participant)
  - each Colleague class knows its Mediator object
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

Code example
C# **Mediator** Design Pattern (dofactory.com)

Memento

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

UML Class Diagram



- **Memento** (Memento)
  - stores the internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion
  - protect against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento; it can only pass the memento to the other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produces the memento would be permitted to access the memento's internal state

- **Originator** (SalesProspect)
  - creates a memento containing a snapshot of its current internal state
  - uses the memento to restore its internal state

- **Caretaker** (Caretaker)
    - is responsible for the memento's safekeeping
    - never operates on or examines the contents of a memento

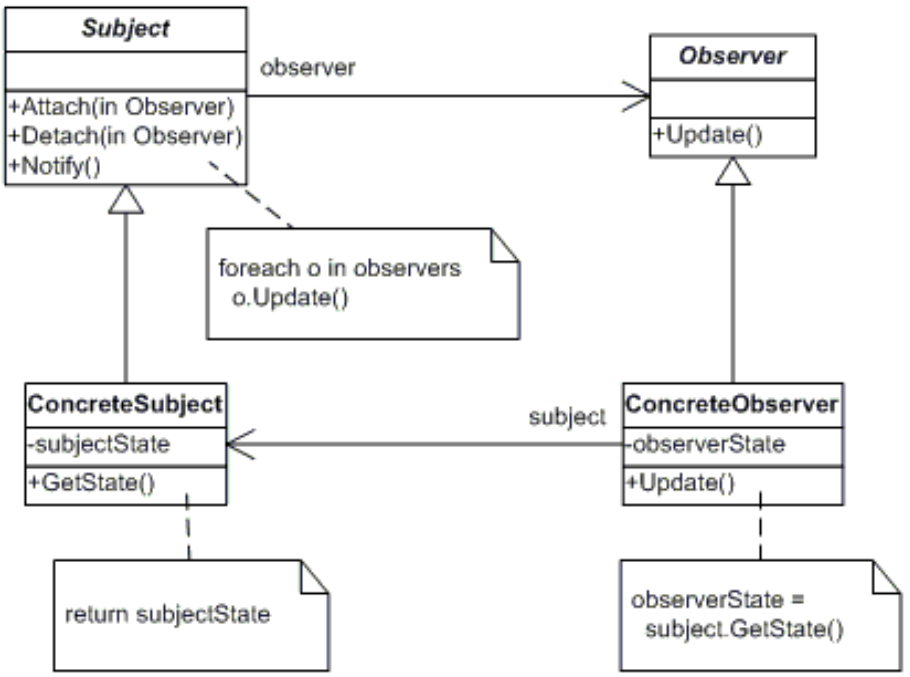
Code example

C# **Memento** Design Pattern (dofactory.com)

Observer

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

UML Class Diagram



- **Subject** (Stock)
  - knows its observers. Any number of Observer objects may observe a subject
  - provides an interface for attaching and detaching Observer objects

- **ConcreteSubject** (IBM)
  - stores state of interest to ConcreteObserver
  - sends a notification to its observers when its state changes

- **Observer** (IInvestor)
    - defines an updating interface for objects that should be notified of changes in a subject

- **ConcreteObserver** (Investor)
    - maintains a reference to a ConcreteSubject object
    - stores state that should stay consistent with the subject's
    - implements the Observer updating interface to keep its state consistent with the subject's

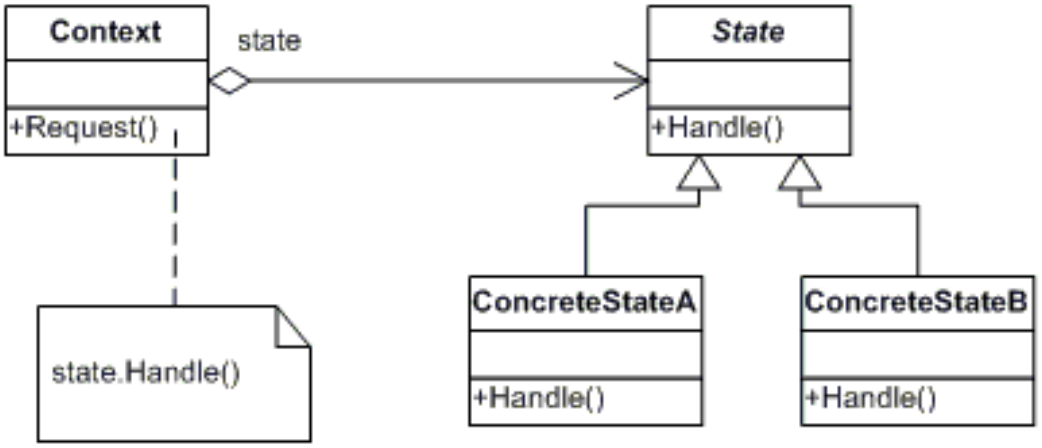
Code example

C# **Observer** Design Pattern (dofactory.com)

State

Definition
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Medium frequency of use.

UML Class Diagram



- **Context** (Account)
  - defines the interface of interest to clients
  - maintains an instance of a ConcreteState subclass that defines the current state

- **State** (State)
  - defines an interface for encapsulating the behavior associated with a particular state of the Context

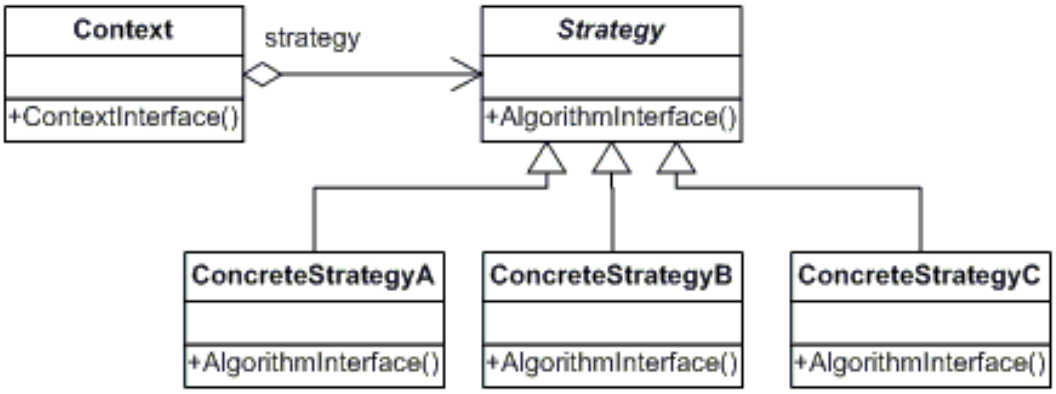
- **Concrete State** (RedState, SilverState, GoldState)
  - each subclass implements a behavior associated with a state of Context

Code example
C# **State** Design Pattern (dofactory.com)

Strategy

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

UML Class Diagram



- **Strategy** (SortStrategy)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy

- **ConcreteStrategy** (QuickSort, ShellSort, MergeSort)
  - implements the algorithm using the Strategy interface

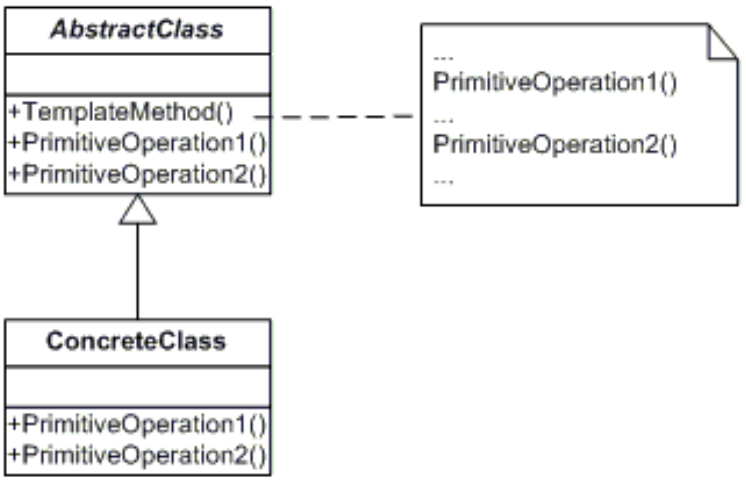
- **Context** (SortedList)
  - is configured with a ConcreteStrategy object
  - maintains a reference to a Strategy object
  - may define an interface that lets Strategy access its data

Code example
C# **Strategy** Design Pattern (dofactory.com)

Template Method

Definition
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.
Medium frequency of use.

UML Class Diagram



- **AbstractClass** (DataObject)
  - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects

- **ConcreteClass** (CustomerDataObject)
  - implements the primitive operations to carry out subclass-specific steps of the algorithm

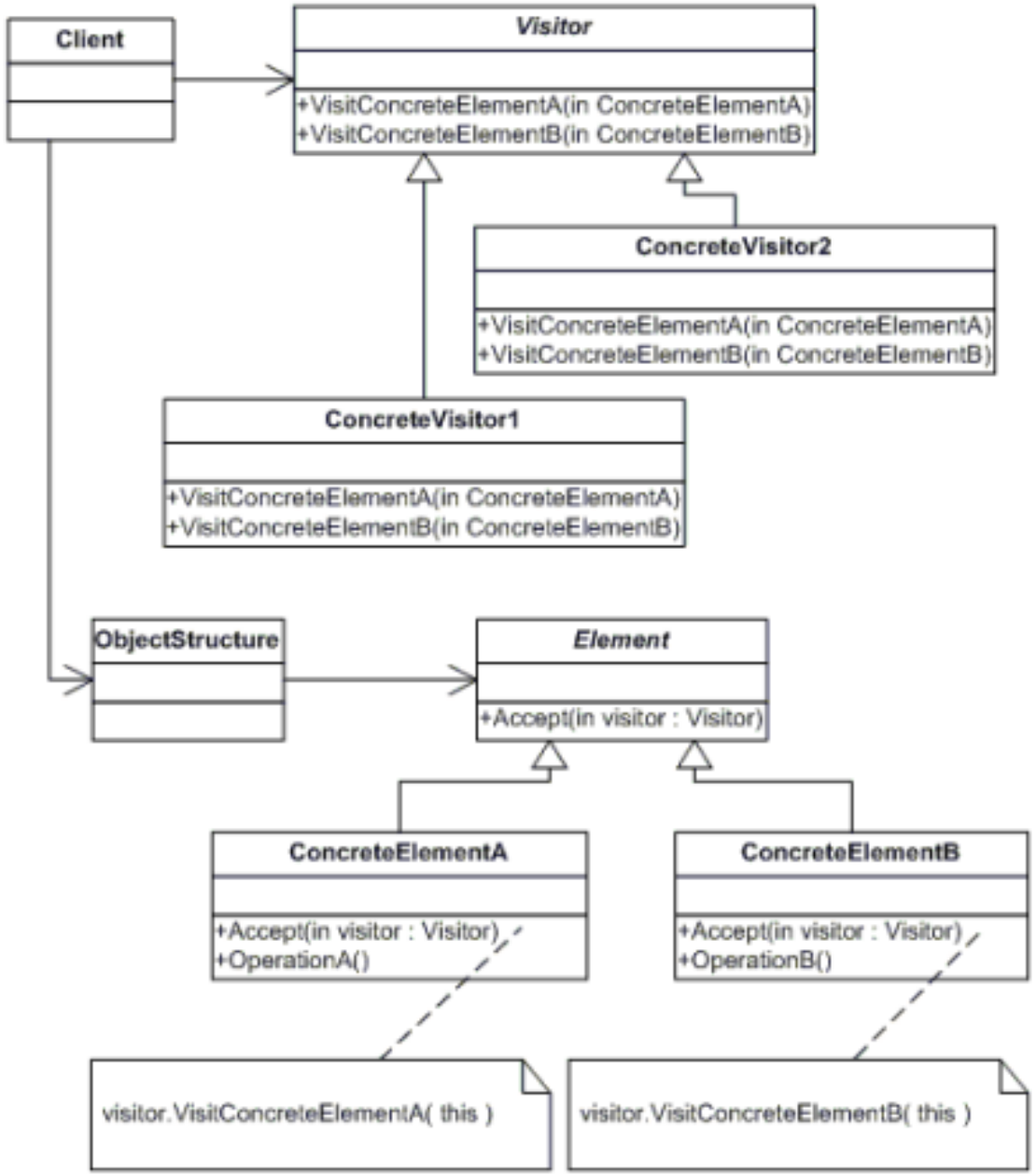
Code example
C# **Template Method** Design Pattern (dofactory.com)

Visitor

Definition

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.
Low frequency of use.

UML Class Diagram



- **Visitor** (Visitor)
  - declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the elements directly through its particular interface

- **ConcreteVisitor** (IncomeVisitor, VacationVisitor)
  - implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure

- **Element** (Element)
  - defines an Accept operation that takes a visitor as an argument

- **ConcreteElement** (Employee)
  - implements an Accept operation that takes a visitor as an argument

- **ObjectStructure** (Employees)
  - can enumerate its elements
  - may provide a high-level interface to allow the visitor to visit its elements
  - may either be a Composite (pattern) or a collection such as a list or a set

Code example

C# **Visitor** Design Pattern (dofactory.com)