<epam> Excellence in Software Engineering

BDO

JAVA ACADEMY

Design Patterns

Fejér Attila

attila@javadev.hu

Design Patterns

INTRODUCTION

Aim of this course

- **Provide overview of common design patterns**
  - Practical examples
- **Practice Object Oriented design**
- **Prerequisites**
  - Java programming language

Contents

- Overview
- Java
- UML basics
- Design patterns
  - ✓ Creational
  - ✓ Structural
  - ✓ Behavioural

What Makes A Good Object-Oriented model?

- **Elegance**
  - A simple solution to a complex problem
- **Adaptability**
  - Can easily be changed to reflect a change in the requirements
- **Non-redundancy**
  - Does not repeat itself
  - Each fact is “in one place”

Design patterns

- "Each pattern describes a **problem** which **occurs over and over again** in our environment, and then describes the core of the **solution** to that problem, in such a way that you can **use this solution a million times over**, without ever doing it the same way twice" (**Christopher Alexander:** A Pattern Language: Towns, Buildings, Construction)

Design patterns

- "descriptions of **communicating objects** and classes that are customized to solve a **general design problem in a particular context" (The Gang of Four:** Design Patterns)

Design patterns usage

- Common language for developers
- Capture good practices
- Speed up the development process
- Avoid common pitfalls
- Avoid solving problems from first principles
- Practice object-oriented design

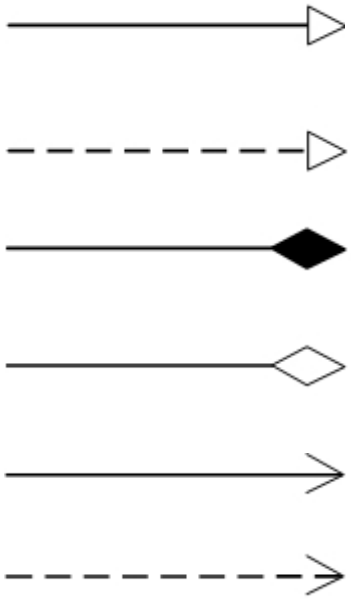
Reminder

- Design patterns usually introduce additional levels of indirection
- Inappropriate use of patterns may unnecessarily increase complexity
- Design patterns may just be a sign of some missing features of a given programming language

OBJECT ORIENTATED PROGRAMMING

OO programming

- Inheritance
- Implementation
- Composition
- Aggregation
- Association
- Dependency



OO programming

- Encapsulation
- Delegation
- Abstraction
  - interface, abstract class, class
  - class, object
- Polymorphism
  - function overloading
  - generic programming
  - polymorphism (function override)

OO programming

- Inheritance
  - white-box reuse
  - easy to use
  - compile time, static
  - relationships between classes
- Composition
  - black-box reuse
  - harder to understand
  - runtime, dynamic
  - relationships between objects

Favour composition over inheritance?

General goals

- Simplify, simplify, simplify
- Separated business logic
- A class should have only one reason to change (one responsibility)
- Identify the aspect of your application that vary and separate them from what stays the same
- Principle of Least Knowledge: talk only to your immediate friends (Demeter law)
- Depend upon abstractions. Do not depend on concrete classes.

General goals

- Classes should be open for extension, but closed for modification
- Strive for loosely coupled designs between objects that interact

Design Patterns

JAVA

Java

- Initialization blocks
- Double Brace Initialization
- Private constructor
- Parameter passing
- Immutable object
- Synchronized
- Keyword: **final**
- Keyword: **static**

Design Patterns

CREATIONAL PATTERNS

Creational Patterns

- Object creation mechanisms
- Two dominant ideas:
  - encapsulating knowledge about which concrete classes the system uses
  - hiding how instances of these concrete classes are created and combined
- Categories:
  - Object-creational patterns
  - Class-creational patterns

Motivation

- Depend upon abstractions
- Remove explicit references to the concrete classes
  - **new** keyword
- Flexibility
- Reuse
- Handle object composition
- Independent system of how objects
  - are created
  - composed
  - represented

Creational Patterns

- Abstract factory
- **Builder**
- **Factory method**
- Prototype
- **Singleton**

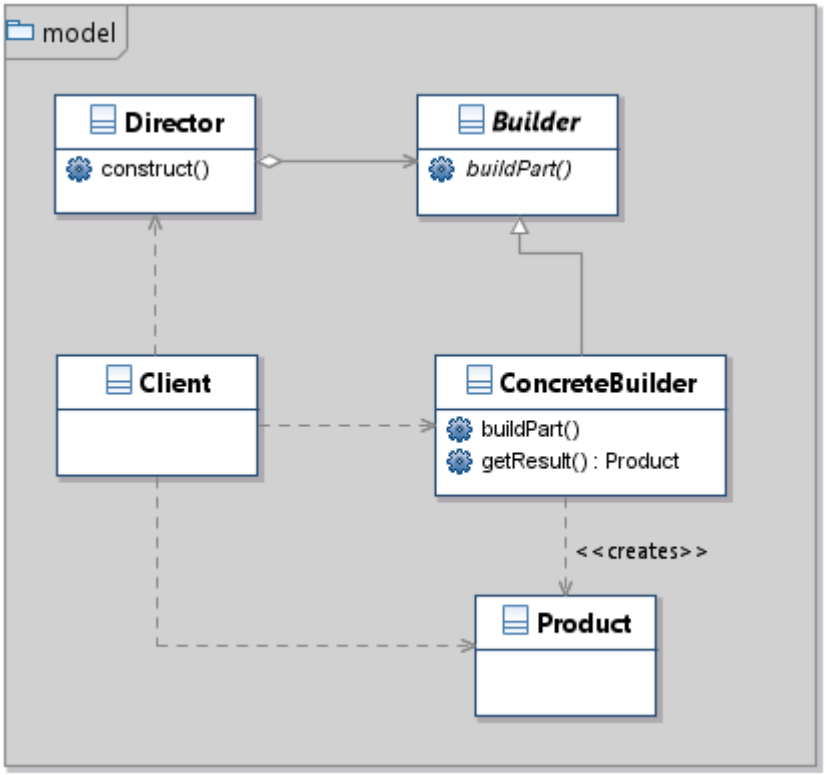
Design Patterns

BUILDER

Builder

- **Intent**
  Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- **Scope**
  - Object
- **Examples**
  - Converter
  - Processing inputs

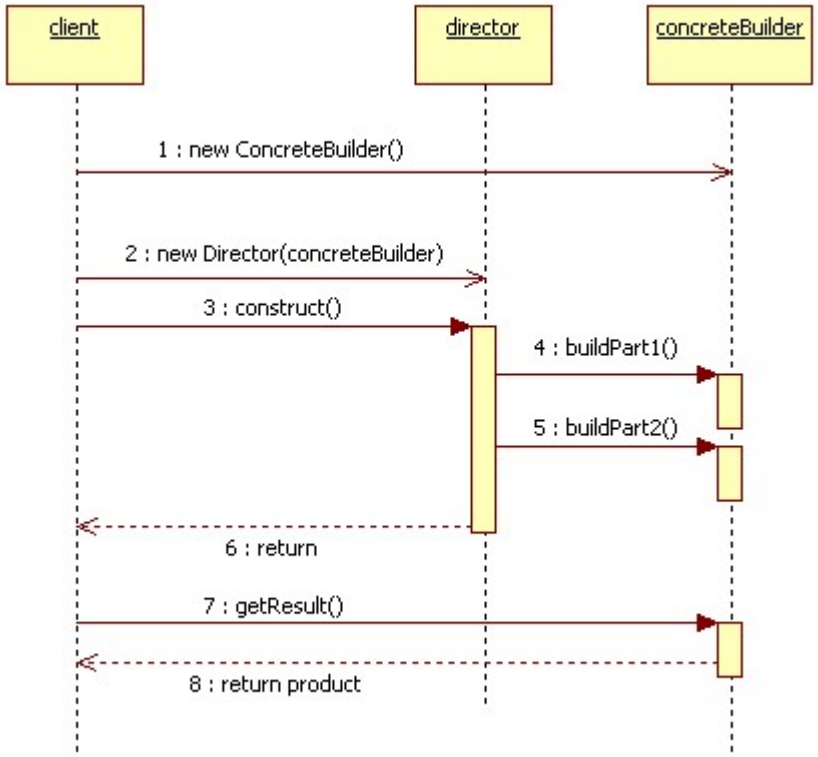
Static structure diagram



Participants

- **Builder**
  - Abstract interface for creating parts of a Product object
- **ConcreteBuilder**
  - Implementation for creating parts
  - Keeps track of the representation it creates
- **Director**
  - Constructs an object using the Builder interface
- **Product**
- **Client**

Behaviour



- Client configures Director with a Builder implementation
- Build procedure
- Result queried from a ConcreteBuilder instance

Implementation

- There is a high level construction process
  - Step-by-step
- Why no abstract class for products?
- Empty methods as default in Builder
  - Letting clients override only the operations they're interested in
- Chaining
  - train wreck
- Product can have a Builder type parameter in the constructor

When to use?

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
- The construction process must allow different representations for the object that's constructed

- **Other benefits**
  - Named parameters (pass-by-name)
  - Avoid long parameter lists
  - Parameter validation in separate methods

Exercise

- Using the Builder design pattern write an interface, implementations and tests for creating instances of the following classes. Read carefully the given Javadoc documentation.
  - YearCashFlow
  - DayCashFlow
- Operations for the builder interface
  - setRate(BigDecimal rate)
  - addExpense(int year, int month, int day, BigDecimal amount)
  - addIncome(int year, int month, int day, BigDecimal amount)
- Validate the input values!

FACTORY METHOD

Factory method

- **Intent**

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

- **Scope**

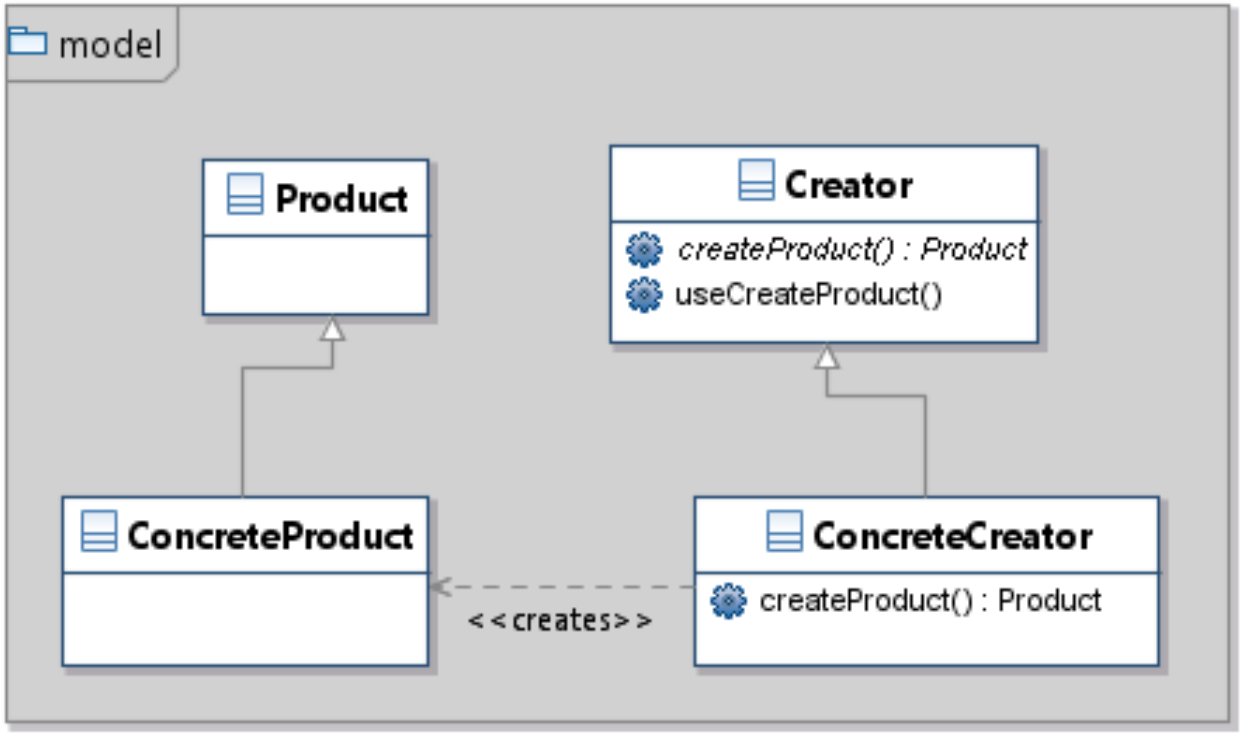
Class

- **Also Known As**

Virtual Constructor

- **Examples**
  - Extensible framework
  - Testing

Static structure diagram



Participants

- **Product**
  - Defines the interface of objects the factory method creates
- **ConcreteProduct**
  - Implementation for the Product interface
- **Creator**
  - Declares the factory method
  - May call the factory method
- **ConcreteCreator**
  - Overrides the factory method

Implementation

- **Subtypes**
  - Factory method is abstract
  - Factory method contains a default implementation
- **Parameterized factory methods**
- **Use suitable names for factory methods**

When to use?

- A class can't anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates

Variant

- **Static factory method**
  - Names
  - Not required to create a new object each time (*FlyWeight* design pattern)
  - Can return an object of any subtype of their return type
  - Creating parameterized type instances is easier (another possibility: diamond operator)
  - Client/Supplier-side factories
  - Limitations (private/protected constructor)

Exercise

- Create a one-time pad encryptor. The key and the enciphered message should be written in separate files.
- Let subclasses decide the next key to use.

Design Patterns

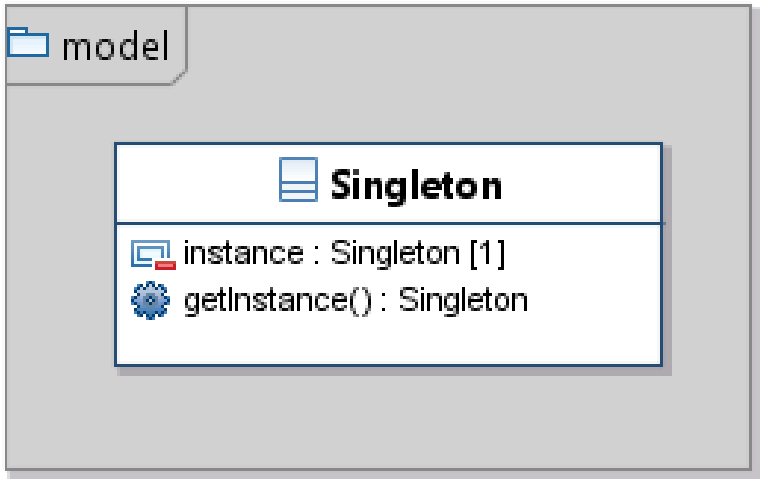
SINGLETON

Singleton

- **Intent**
  - Ensure a class only has one instance, and provide a global point of access to it.
- **Scope**
  - Object
- **Examples**
  - Application main window
  - Preferences
  - Statistics

Static structure diagram

- **private static field**
  - *instance*
- **public static method**
  - *getInstance()*
- **private constructor**



Participants and behaviour

- **Singleton**
  - Class with on single instance

- Eager loading
  - instance constructed when the class is loaded
- Lazy loading
  - instance constructed on first usage

Implementation

- **Concurrent access**
  - for *getInstance()*
  - for business methods
- **Is it possible to subclass a Singleton?**
- **Performance**

When to use?

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
- For a global state in the application

Remember:

- A singleton today is a multiple tomorrow

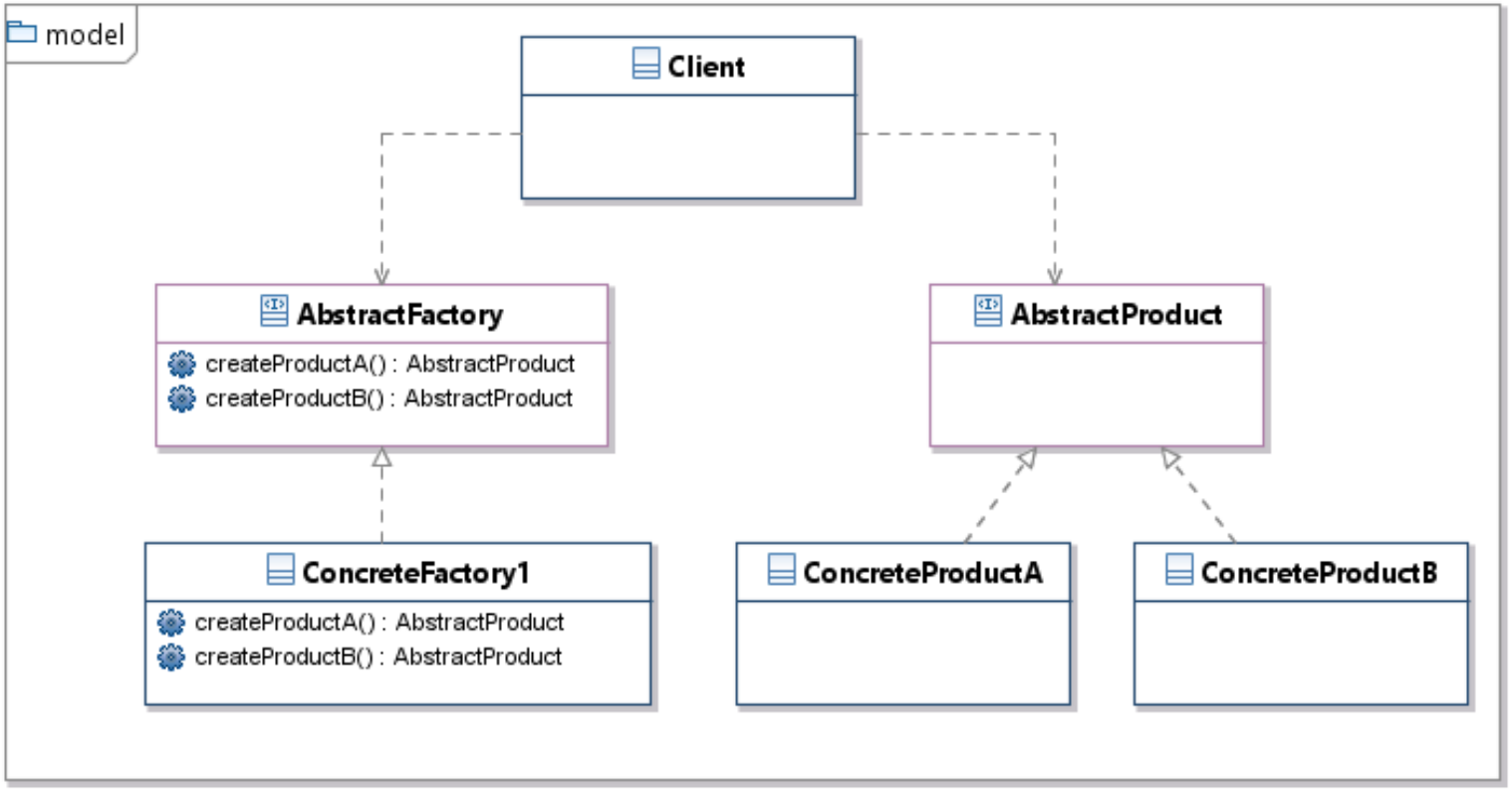
Variants

- **Eager initialization**
  - Declaration and initialization
  - Static initializer block
  - Enum
- **Lazy initialization**
  - Synchronized getter
  - Double-checked locking
  - Static inner class
- **Are they really singletons?**

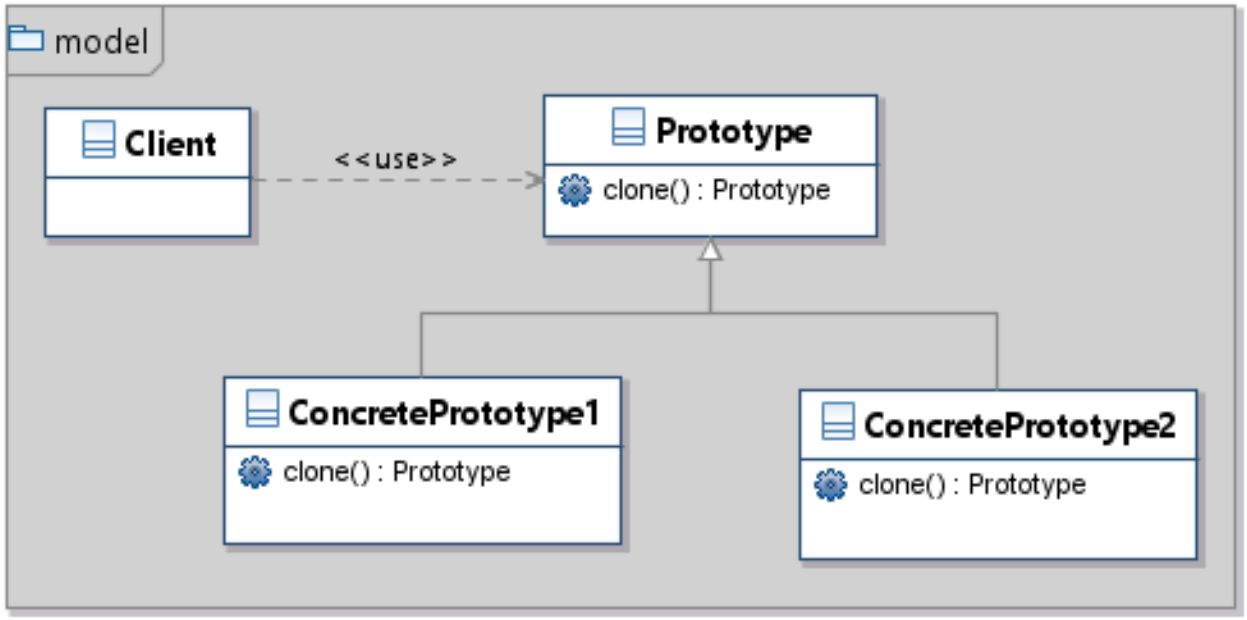
Design Patterns

OTHER PATTERNS

Abstract Factory



Prototype


model
Client
<<use>>
Prototype
clone() : Prototype
ConcretePrototype1
clone() : Prototype
ConcretePrototype2
clone() : Prototype

STRUCTURAL PATTERNS

Structural Patterns

- How classes and objects are composed to form larger structures
- Compose objects to realize new functionality
- Increase flexibility
- Same small set of language mechanisms for structuring objects

Structural Patterns

- **Adapter**
- **Bridge**
- Composite
- **Decorator**
- **Facade**
- Flyweight
- **Proxy**

Design Patterns

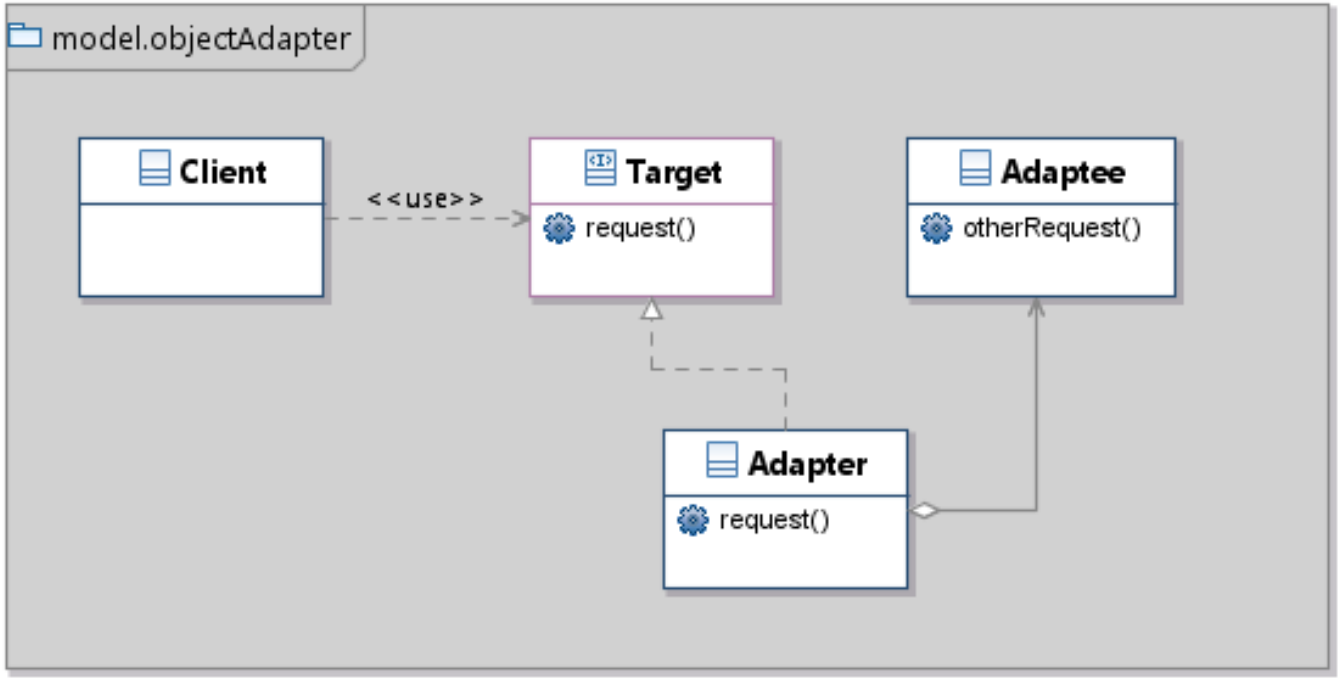
ADAPTER

Adapter

- **Intent**
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Scope**
  - Class, Object
- **Also Known As**
  - Wrapper
- **Examples**
  - Listeners as inner classes
  - Inherited listener implementations

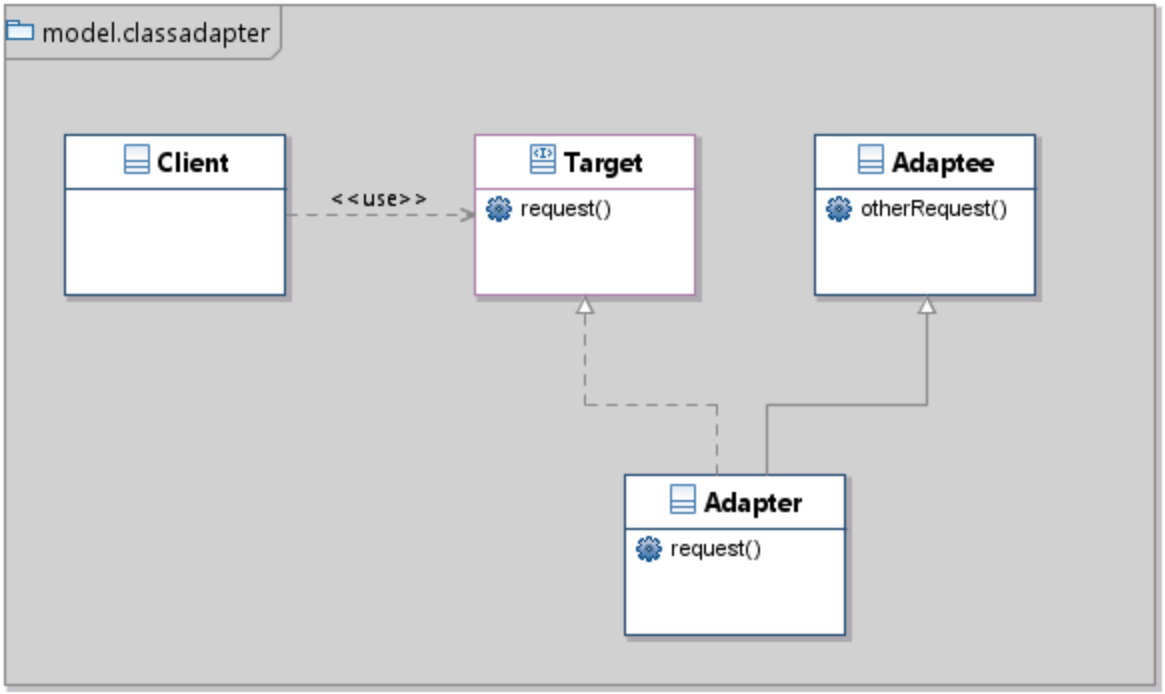
Static structure diagram

- Object Adapter



Static structure diagram

- Class Adapter



Participants

- **Target**
  - Interface used by the client
- **Client**
  - Collaborates with objects conforming to the Target interface
- **Adaptee**
  - Has an incompatible interface
- **Adapter**
  - Adapts the interface of Adaptee to the Target interface

Behaviour and implementation

- **Behaviour**
  - Clients call operations on an Adapter instance
  - Adapter calls Adaptee operations that carry out the request
- **Implementation**
  - Sometimes it’s hard to adapt interfaces created with different assumptions
  - Parameterized adapters

When to use?

- You want to use an existing class, and its interface does not match the one you need
- Use object adapter, when you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one

Exercise

- Create a FIFO implementation for the following interface. Adapt an existing Java class
  - Object adapter
  - Class adapter
- Test the adapters

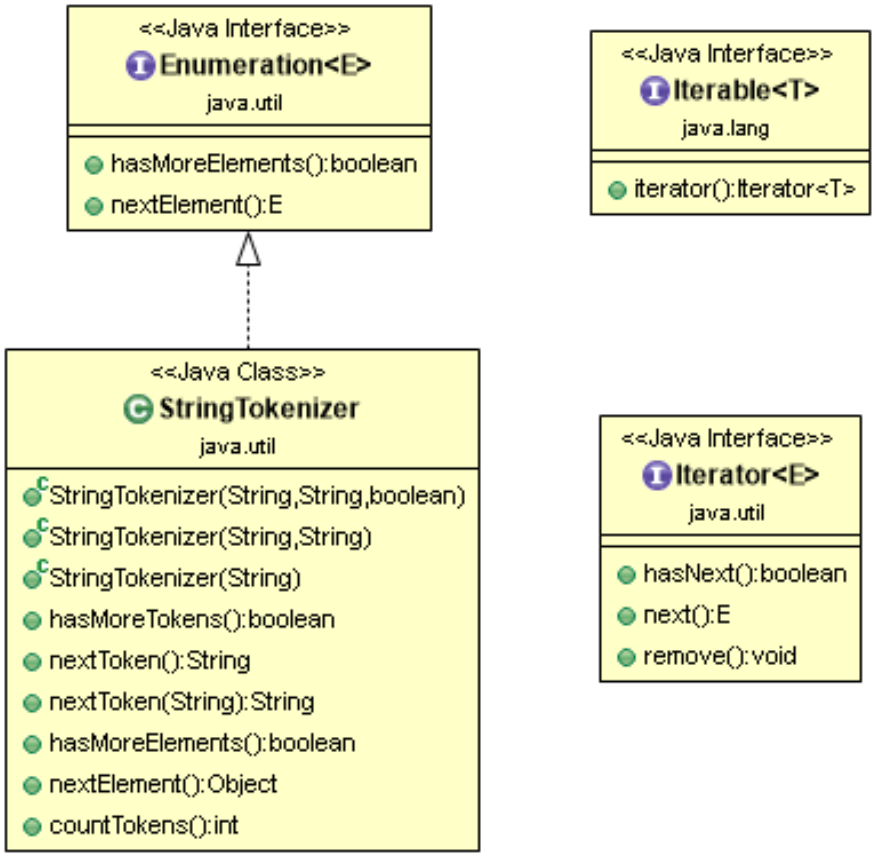
```
public interface Fifo<T> {

        void put(T t);

        T pop();
}
```

Exercise

- Adapt the StringTokenizer class, so we can use it with a for-each loop



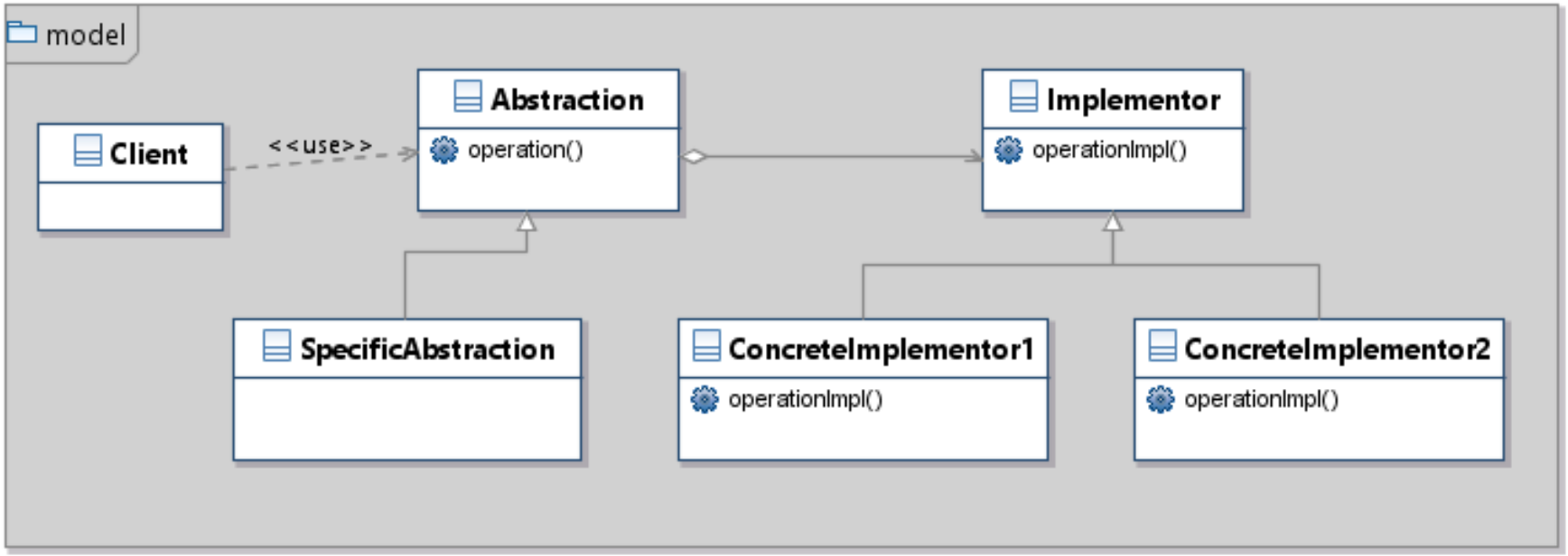
Design Patterns

BRIDGE

Bridge

- **Intent**
  - Decouple an abstraction from its implementation so that the two can vary independently.
- **Scope**
  - Object
- **Also Known As**
  - Handle/Body
- **Examples**
  - Most API implementations

Static structure diagram



Participants

- **Client**
  - Uses the implementation through the abstraction
- **Abstraction**
  - The way we would like to use the implementations
- **RefinedAbstraction**
  - Extends the interface defined by Abstraction
- **Implementor**
  - Defines the interface for implementation classes.
  - Typically the Implementor interface provides only primitive operations
- **ConcreteImplementor**
  - Defines the concrete implementation

Behaviour and implementation

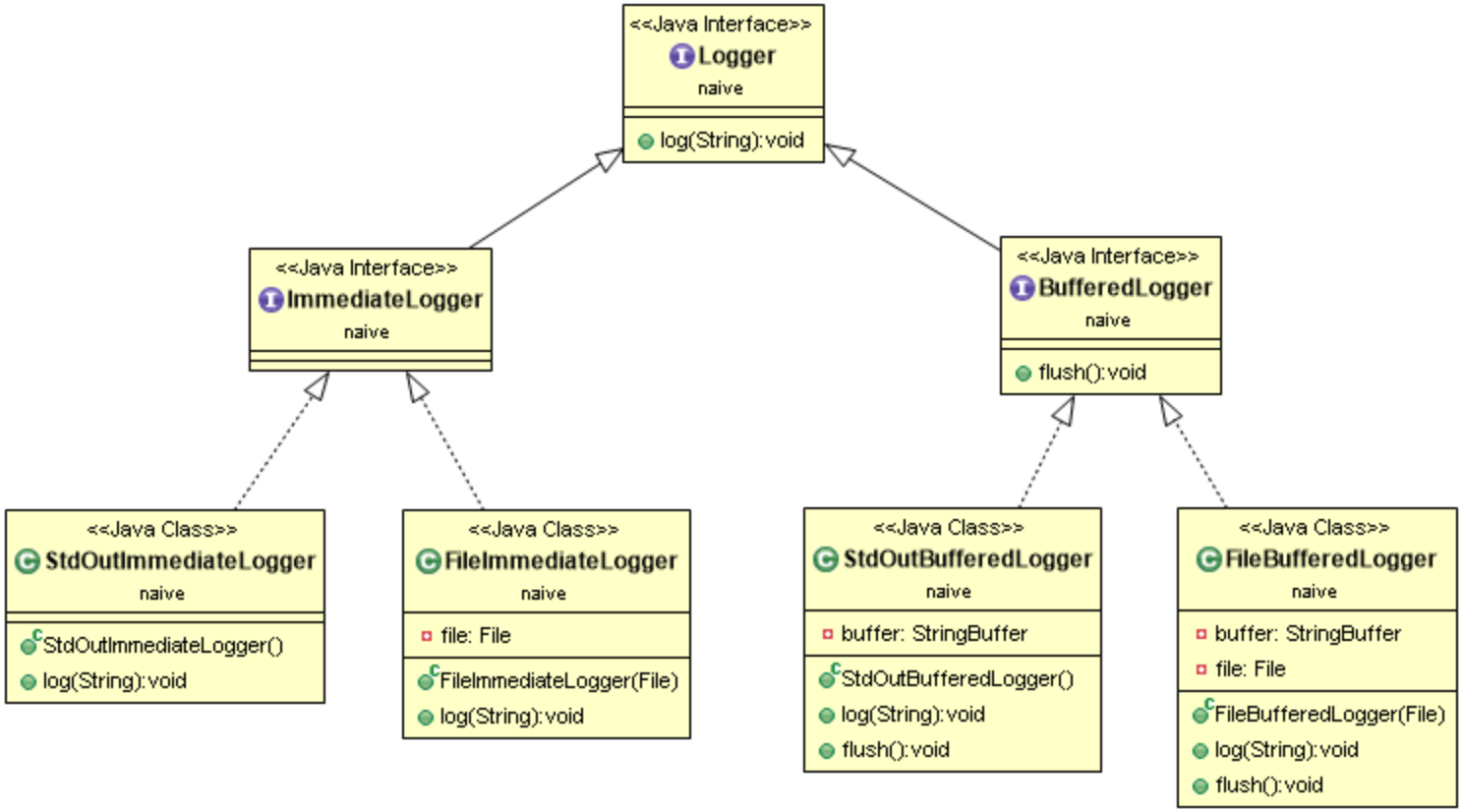
- **Behaviour**
  - Abstraction forwards client requests to its Implementor object
- **Implementation**
  - Only one Implementor
  - Using the right Implementor object
  - Sharing implementors

When to use?

- To avoid a permanent binding between an abstraction and its implementation
- If the implementation must be selected or switched at run-time
- Both the abstractions and their implementations should be extensible by subclassing
- To share an implementation among multiple objects

Exercise

- **Redesign and implement the given hierarchy**



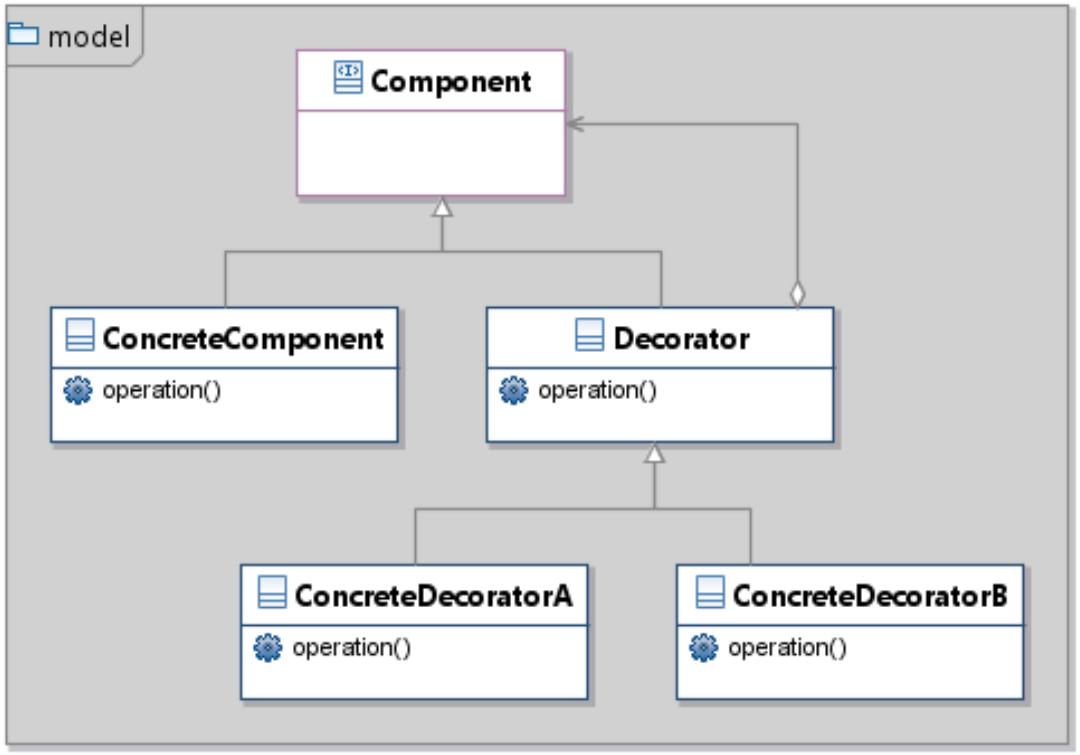
Design Patterns

DECORATOR

Decorator

- **Intent**
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **Scope**
  - Object
- **Also Known As**
  - Wrapper
- **Examples**
  - Scrollable panel
  - ObjectInputStream > GzipInputStream > BufferedInputStream > FileInputStream

Static structure diagram



Participants

- **Component**
  - Defines the interface for objects that can have responsibilities added to them dynamically
- **ConcreteComponent**
  - Defines an object to which additional responsibilities can be attached
- **Decorator**
  - Maintains a reference to a Component object and defines an interface that conforms to Component's interface
- **ConcreteDecoratorN**
  - Adds responsibilities to the component

Behaviour and implementation

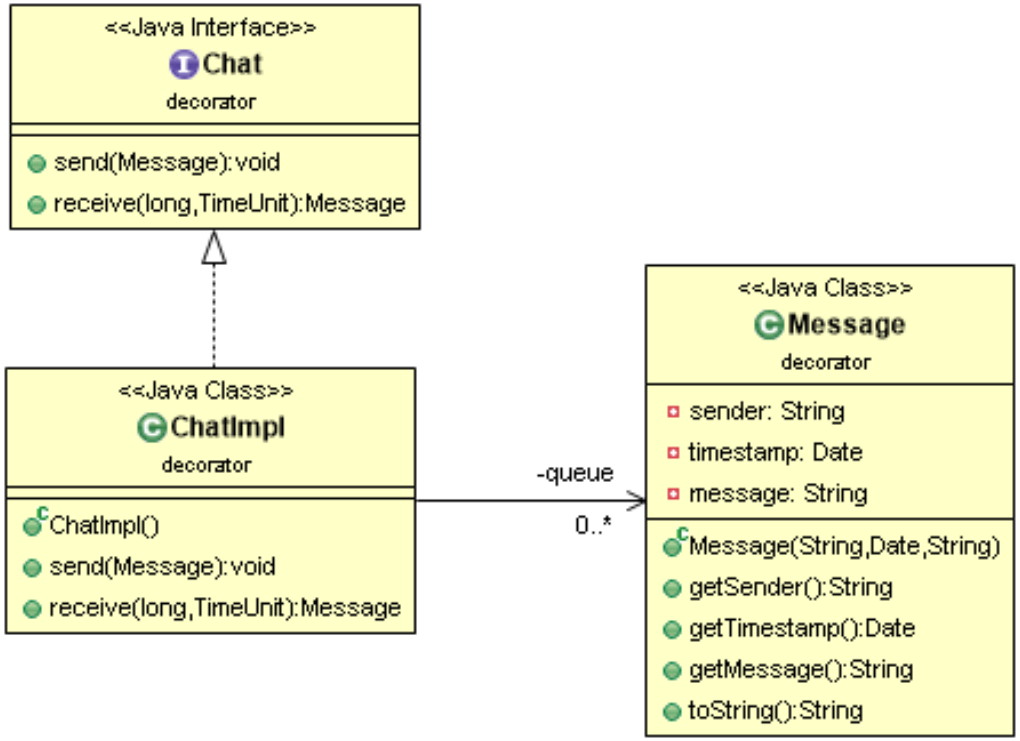
- **Behaviour**
  - Decorator forwards requests to its Component object
  - Decorator may optionally perform additional operations before and after forwarding the request
- **Implementation**
  - Interface conformance
  - Sometimes the common ancestor class for decorators can be omitted

When to use?

- Add responsibilities to individual objects dynamically and transparently
- For responsibilities that can be withdrawn
- When extension by subclassing is impractical

Exercise

- **Our chat application needs new abilities (these can be disabled individually):**
  - Remove obscene words from sent messages
  - Log all messages



What would be **without** the decorator pattern?

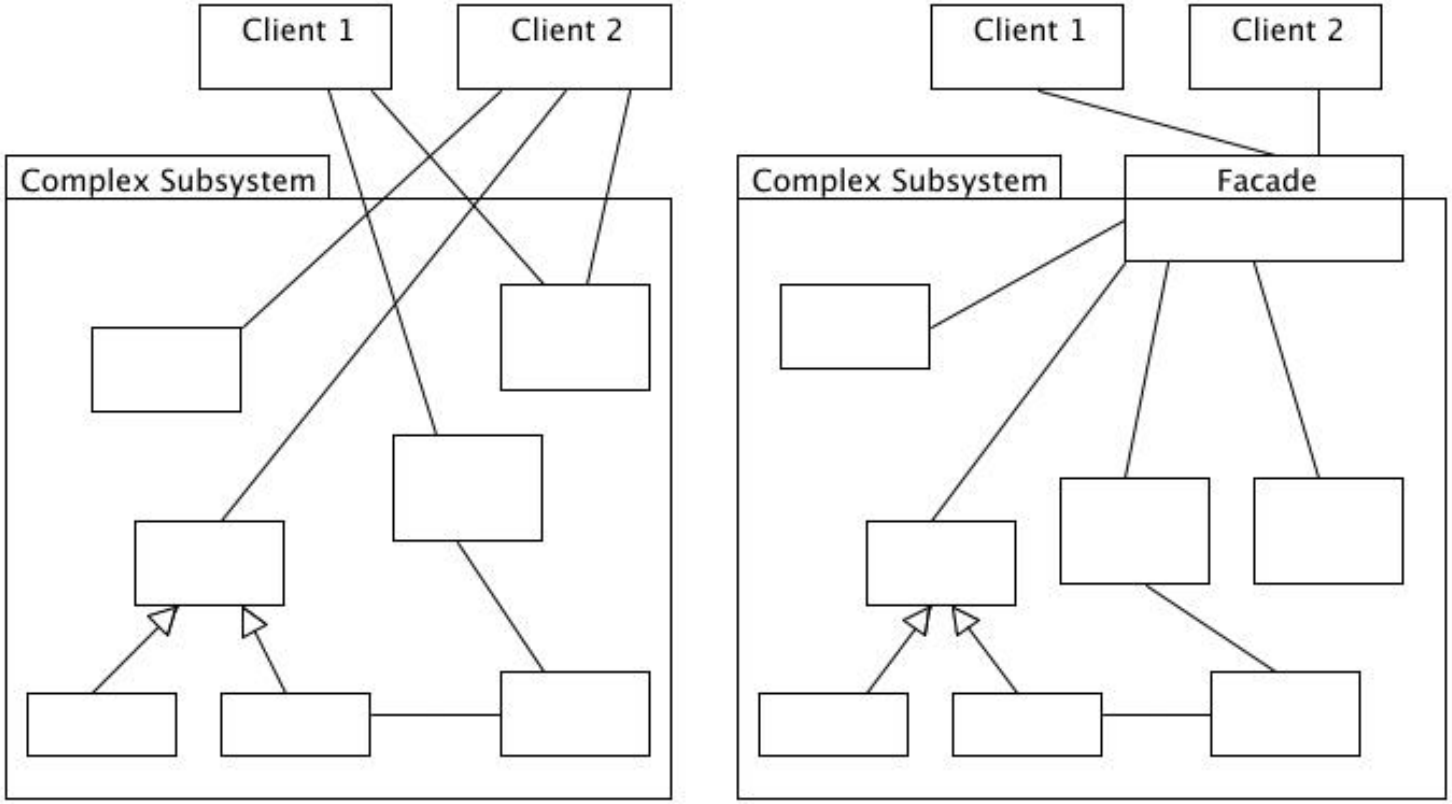
Design Patterns

FACADE

Facade

- **Intent**
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **Scope**
  - Object
- **Examples**
  - JPA API
  - Layered architecture

Facade


Client 1
Client 2
Complex Subsystem
Client 1
Client 2
Complex Subsystem
Facade

Participants

- **Facade**
  - Knows which subsystem classes are responsible for a request
  - Delegates client requests to appropriate subsystem objects
- **Subsystem classes**
  - Implement subsystem functionality
  - Have no knowledge of the facade

Behaviour and implementation

- **Behaviour**
  - Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem objects

- **Implementation**
  - Reducing client-subsystem coupling

When to use?

- You want to provide a simple interface to a complex subsystem
- There are many dependencies between clients and the implementation classes
- You want to layer your subsystems

Exercise

Write a Facade for our formatter package

- **Currency**
  - Dollar
  - Euro
  - Forint
- **Number**
  - Percent
  - Decimal
  - Integer
- **Date**
  - Short
  - Long

Design Patterns

PROXY

Proxy

- **Intent**

Provide a surrogate or placeholder for another object to control access to it.

- **Scope**

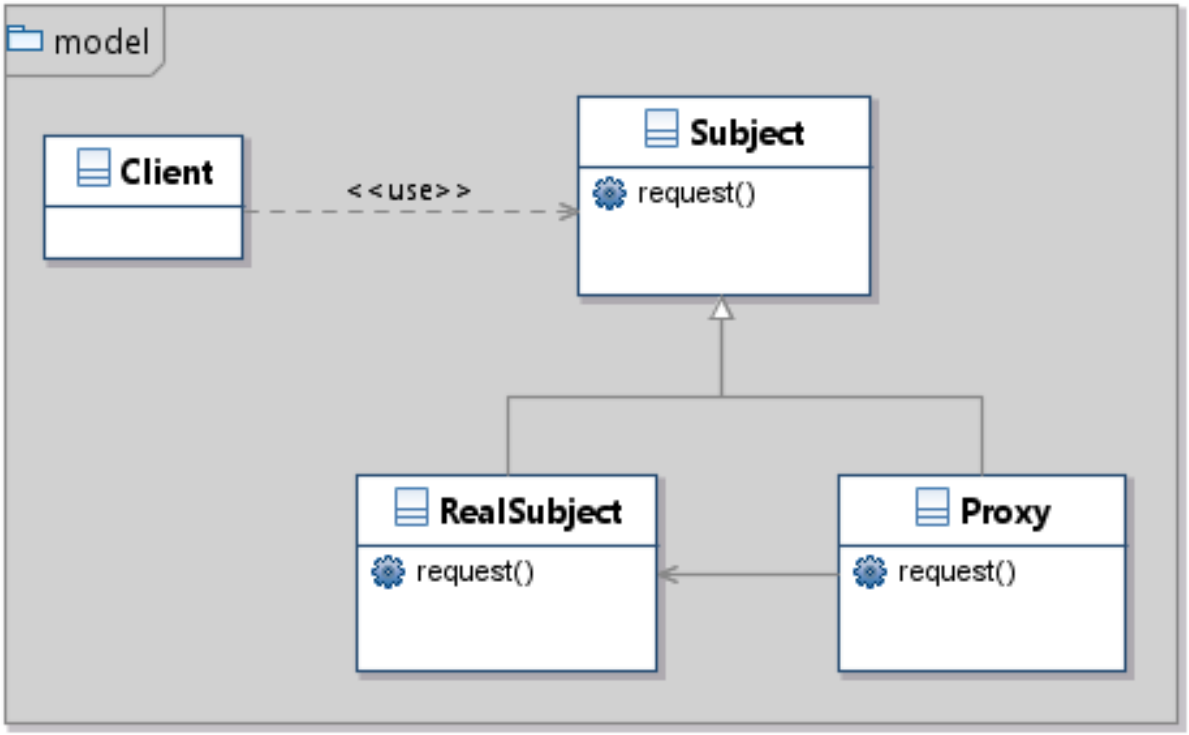
Object

- **Also Known As**

Wrapper

- **Examples**
  - Remote procedure call
  - Reference counting
  - Protection Proxy

Static structure diagram



Participants

- **Subject**
  - Defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected
- **RealSubject**
  - Defines the real object that the proxy represents
- **Proxy**
  - Maintains a reference that lets the proxy access the real subject

Behaviour and implementation

- **Behaviour**
  - Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy
- **Implementation**
  - Proxy doesn't always have to know the concrete type of real subject (ex. no instantiation)

When to use?

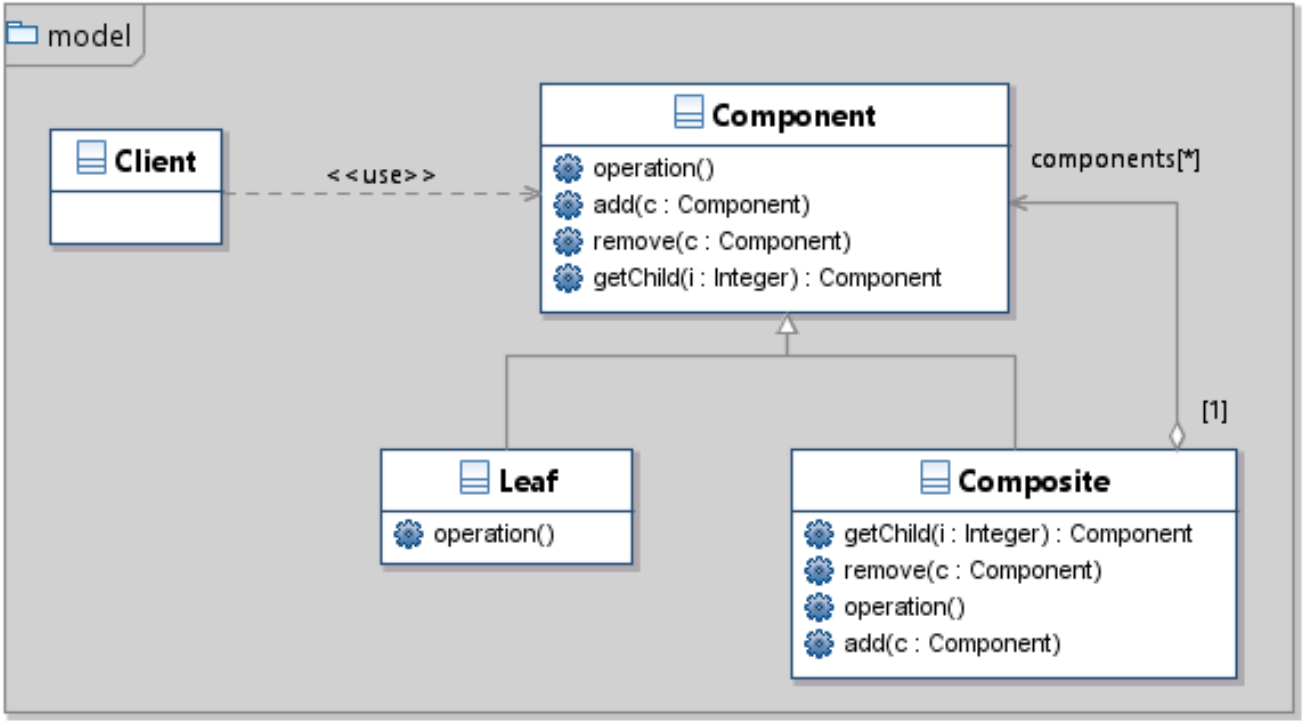
- **There is a need for a more sophisticated reference to an object than a simple pointer**
  - Remote proxy
  - Virtual proxy (expense objects on demand)
  - Protection proxy

Exercise

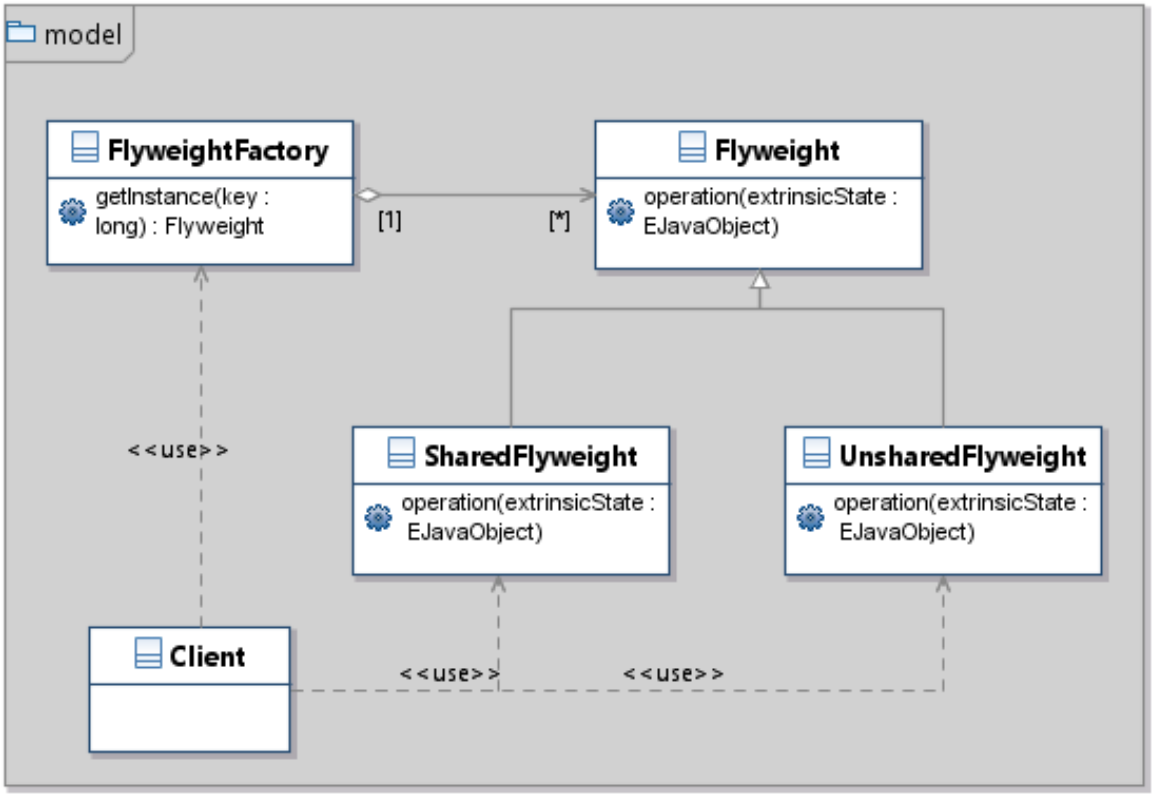
- We noticed that most of our Fifo instances are empty. This is a huge overhead, because they allocate space for 1000 items during construction and most of them is created during system start-up.
- Implement a virtual proxy to solve this problem.

OTHER PATTERNS

Composite


model
Client
<<use>>
Component
operation()
add(c : Component)
remove(c : Component)
getChild(i : Integer) : Component
components[*]
[1]
Leaf
operation()
Composite
getChild(i : Integer) : Component
remove(c : Component)
operation()
add(c : Component)

Flyweight


model
FlyweightFactory
getInstance(key : long) : Flyweight
[1]
[*]
Flyweight
operation(extrinsicState : EJavaObject)
<<use>>
SharedFlyweight
operation(extrinsicState : EJavaObject)
UnsharedFlyweight
operation(extrinsicState : EJavaObject)
Client
<<use>>
<<use>>

BEHAVIOURAL PATTERNS

Behavioural patterns

- Communication patterns between objects
- Assignment of responsibilities between objects
- Loose coupling
- Increased flexibility

Behavioural patterns

- Chain of responsibility
- **Command**
- Interpreter
- **Iterator**
- **Mediator**
- Memento
- **Observer**
- State
- **Strategy**
- **Template method**
- Visitor

Design Patterns

COMMAND

Command

- **Intent**

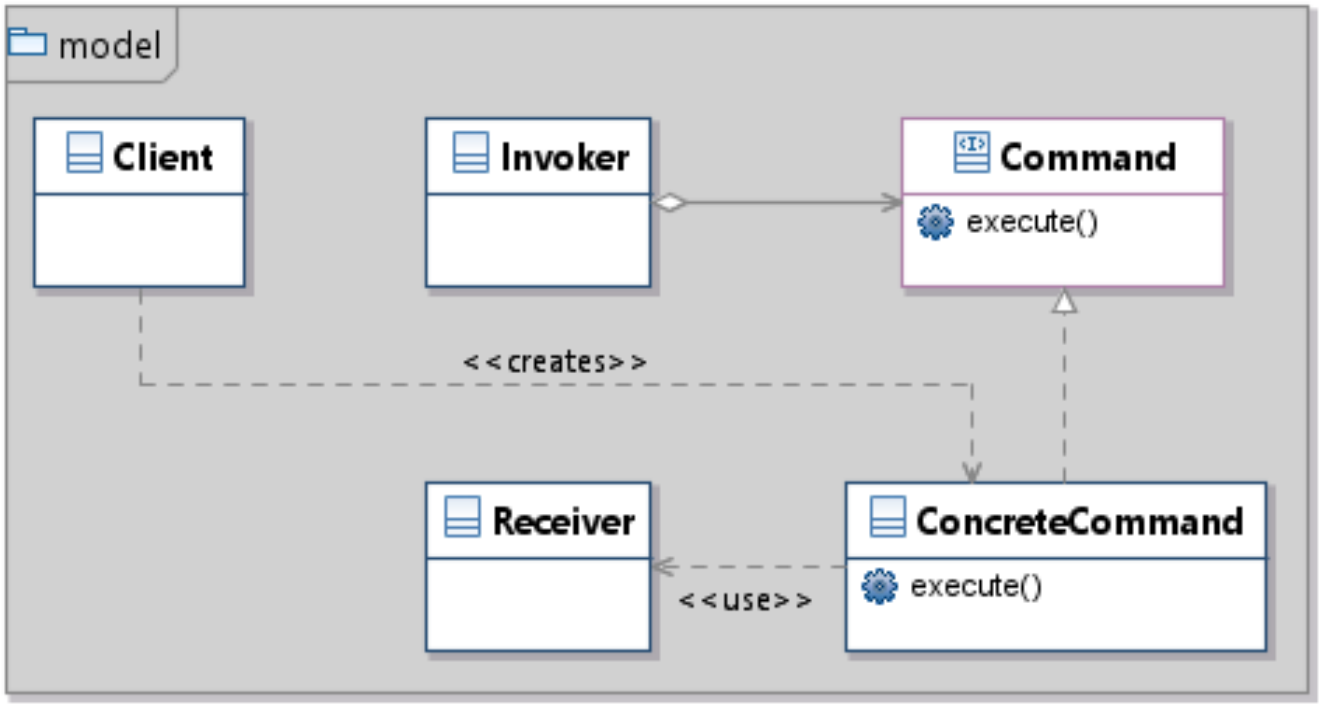
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

- **Scope**

Object

- **Examples**
  - Application with undoable operations
  - Batch jobs

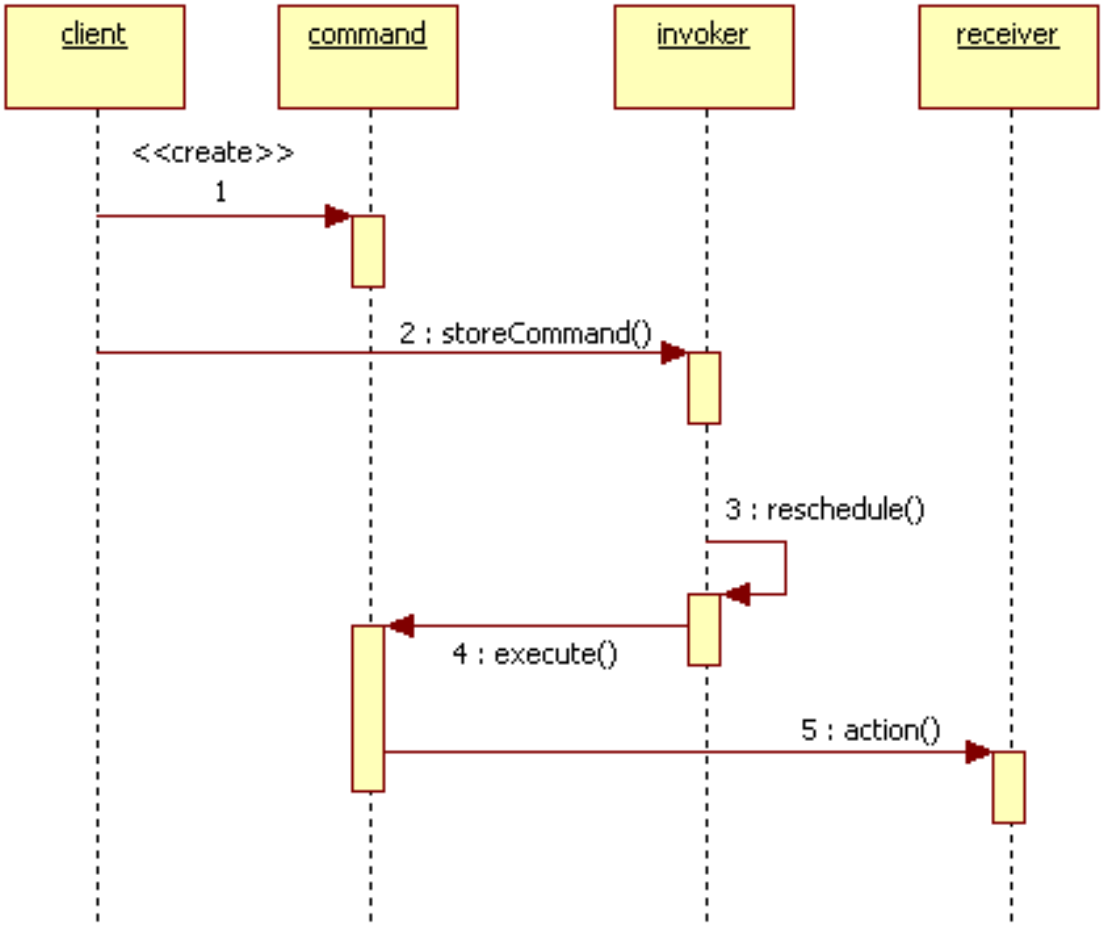
Static structure diagram



Participants

- **Client**
  - Creates a ConcreteCommand object and sets its receiver
- **Invoker**
  - Asks the command to carry out the request
- **Command**
  - Declares an interface for executing an operation
- **ConcreteCommand**
  - Defines a binding between a Receiver object and an action
  - Implements execute by invoking the corresponding operation(s) on Receiver
- **Receiver**
  - Knows how to perform the operations associated with carrying out a request

Behaviour


client
command
invoker
receiver
<<create>>
1
2 : storeCommand()
3 : reschedule()
4 : execute()
5 : action()

Behaviour and implementation

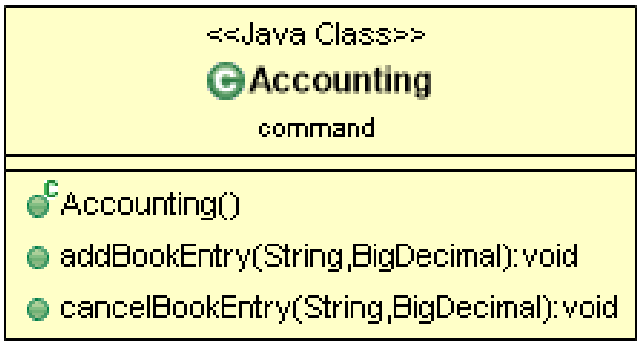
- **Behaviour**
  - Client creates a ConcreteCommand object and specifies its receiver
  - Invoker object stores the ConcreteCommand object
  - Invoker issues a request by calling Execute on the command
- **Implementation**
  - Responsibilities of a Command object
  - Error accumulation in the undo process

When to use?

- Specify, queue, and execute requests at different times
- Support undo
- Logging changes so that they can be reapplied in case of a system crash

Exercise

- **Our old accounting application is not thread-safe, but sometimes we receive hundreds of write requests per second. Live read operations are not needed. Due to regular technical problems we have to support revert operations for a specified time frame.**
  - Implement a queue for the specified class (Accounting)
  - Implement revert operations



Design Patterns

ITERATOR

Iterator

- **Intent**

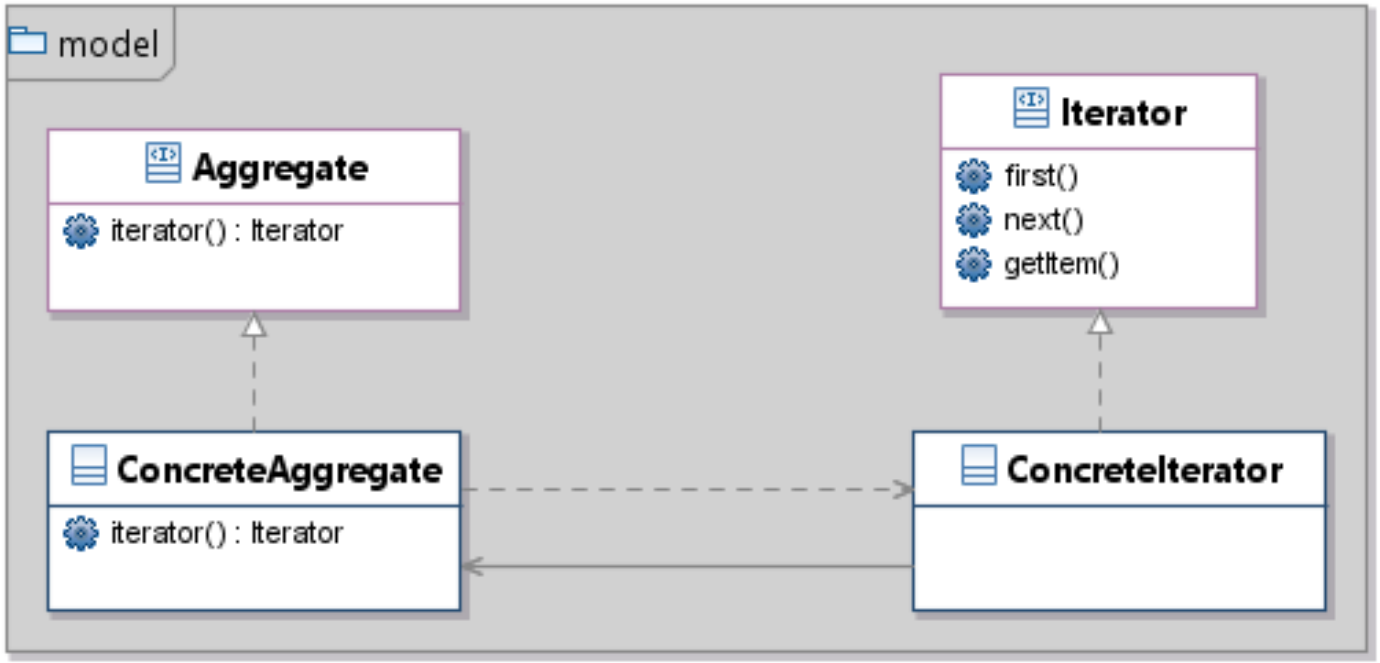
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- **Scope**

Object

- **Examples**
  - List, Set

Static structure diagram



Participants

- **Aggregate**
  - Defines an interface for creating an Iterator object
- **ConcreteAggregate**
  - Implements the Iterator creation interface to return an instance of the proper ConcreteIterator
- **Iterator**
  - Defines an interface for accessing and traversing elements
- **ConcreteIterator**
  - Implements the Iterator interface

Behaviour and implementation

- **Behaviour**
  - A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal
- **Implementation**
  - External / Internal iterator
  - Concurrent access
  - Iterators may have privileged access

When to use?

- To access an aggregate object's contents without exposing its internal representation
- To provide a uniform interface for traversing different aggregate structures

Exercise

- **Implement a Fifo based on an array (or linked list) and make it iterable**
  - Support the remove operation

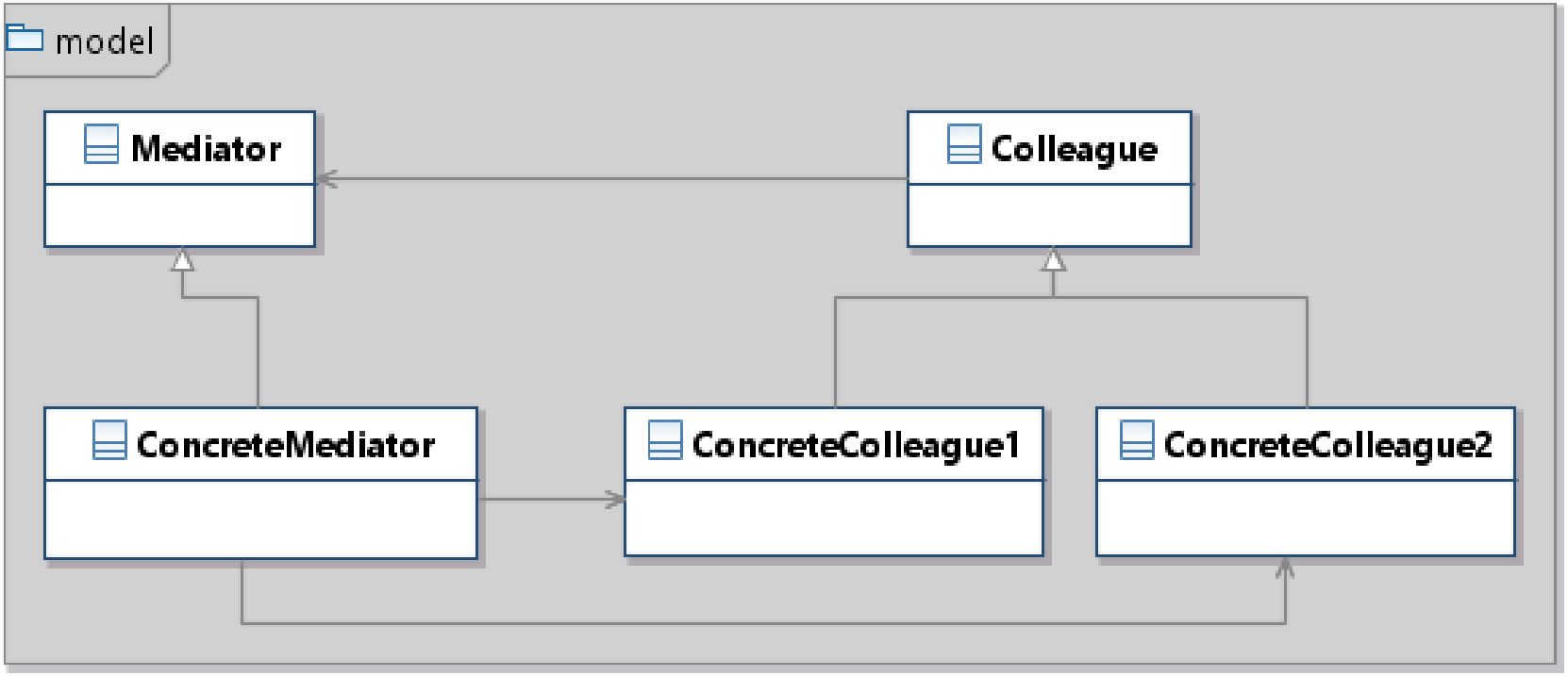
Design Patterns

MEDIATOR

Mediator

- **Intent**
  - 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.
- **Scope**
  - Object
- **Examples**
  - GUI, JMS

Static structure diagram



Participants

- **Mediator**
  - Defines an interface for communicating with Colleague objects
- **ConcreteMediator**
  - Implements cooperative behaviour
- **Colleague**
  - Interface for Collegue implementations
- **ConcreteCollegueN**
  - Each colleague communicates with its mediator

Behaviour and implementation

- **Behaviour**
  - Colleagues send and receive requests from a Mediator object
- **Implementation**
  - Mediator abstract class can be omitted
  - Responsibilities of a Mediator object

When to use?

- A set of objects communicate in well-defined but complex ways
- Reusing an object is difficult because it refers to and communicates with many other objects
- A behavior that's distributed between several classes should be customizable without a lot of subclassing

Exercise

- **Simulate a Smart Home!**
- **Elements:**
  - Clock
  - Lamp
  - Coffee maker
  - Radio
  - Movement sensor
- **You can define your own rules.**

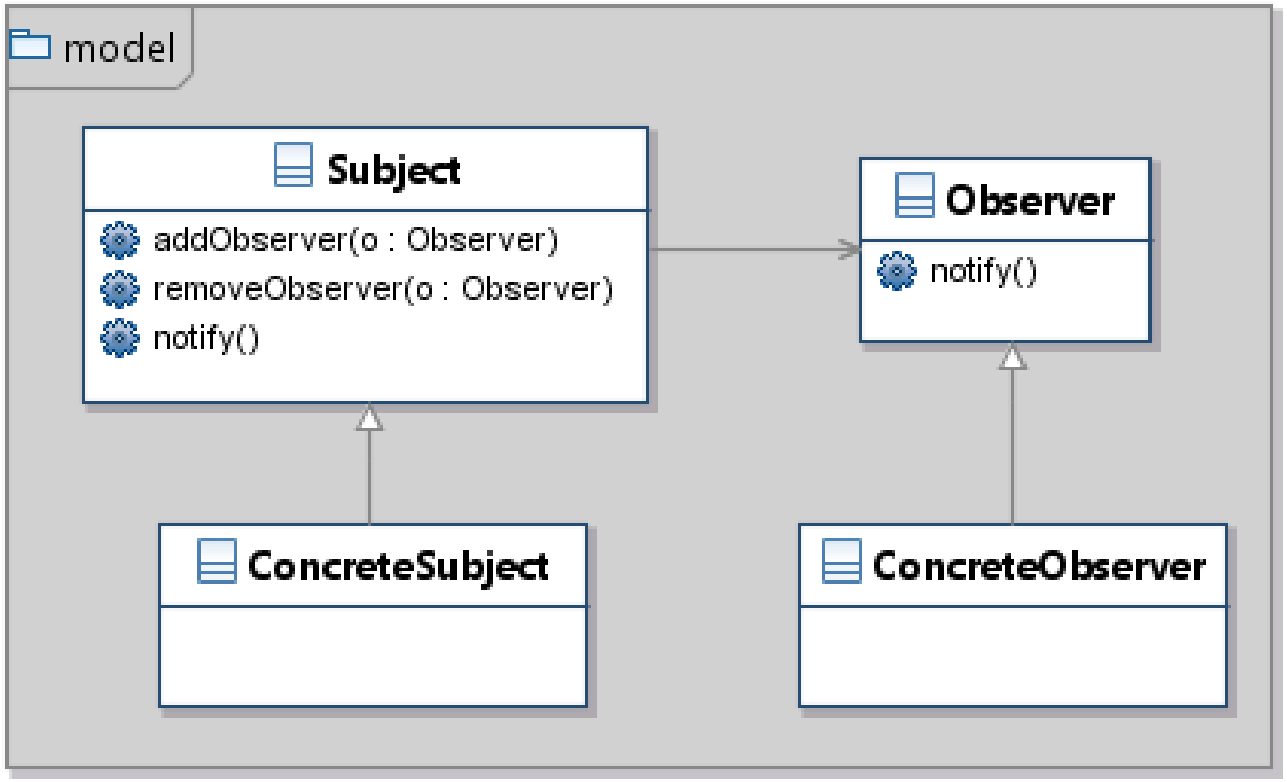
Design Patterns

OBSERVER

Observer

- **Intent**
  - Define a many-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Scope**
  - Object
- **Examples**
  - GUI

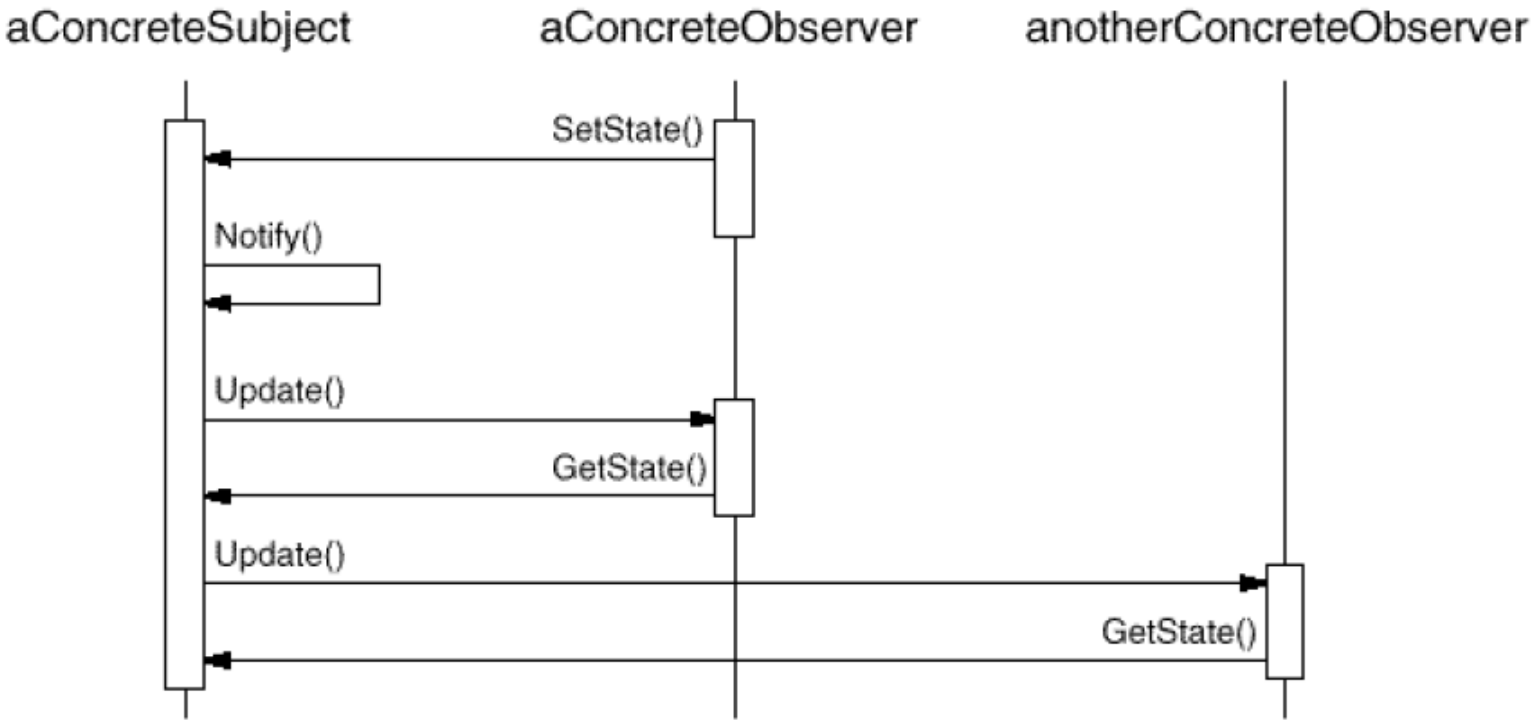
Static structure diagram



Participants

- **Subject**
  - Knows its observers
  - Provides an interface for attaching and detaching Observer objects
- **Observer**
  - Defines an updating interface for objects that should be notified of changes in a subject
- **ConcreteSubject**
  - Sends a notification to its observers when its state changes
- **ConcreteObserver**
  - Implements the Observer updating interface

Behaviour


aConcreteSubject
aConcreteObserver
anotherConcreteObserver
SetState()
Notify()
Update()
GetState()
Update()
GetState()

Implementation

- **Observing more than one subject**
- **Triggering the update**
  - Other observer
  - State change
- **Subject state should be self-consistent before notification**

When to use?

- When an object should be able to notify other objects without making assumptions about who these objects are
- A change to one object requires changing others
- Alternative for Mediator

Exercise

- **Based on the Observer design pattern create the following components and connect them:**
  - Counter (period: 2 seconds)
  - Counter (period: 3 seconds)
  - Standard output logger
  - File logger

Design Patterns

STRATEGY

Strategy

- **Intent**

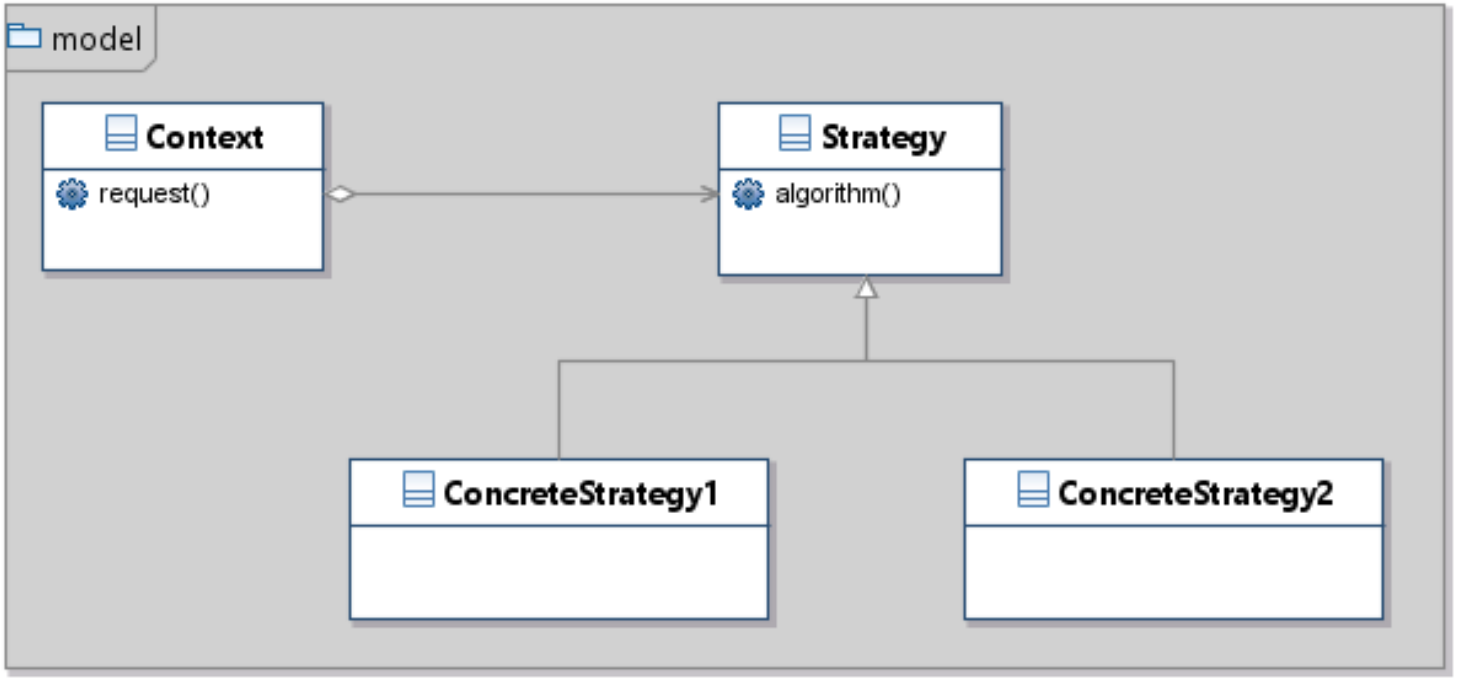
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- **Scope**

Object

- **Examples**
  - Layout manager

Static structure diagram



Participants

- **Strategy**
  - Declares an interface common to all supported algorithms
- **ConcreteStrategyN**
  - Implements the algorithm using the Strategy interface
- **Context**
  - Is configured with a ConcreteStrategy object
  - May define an interface that lets Strategy access its data

Behaviour and implementation

- **Behaviour**
  - Strategy and Context interact to implement the chosen algorithm
  - A context forwards requests from its clients to its strategy
- **Implementation**
  - Default strategy
  - Hierarchy for strategies

When to use?

- Many related classes differ only in their behavior
- Different variants of an algorithm are needed
- An algorithm uses data that clients shouldn't know about
- Class defines many behaviors, and these appear as multiple conditional statements in its operations

Exercise

- **Write a number guessing game with two strategies!**
- **The user chooses a number between 0-1023 and answers the questions (smaller, greater or equal)**
- **Algorithms**
  - Binary searching
  - Random

Design Patterns

TEMPLATE METHOD

Template method

- **Intent**

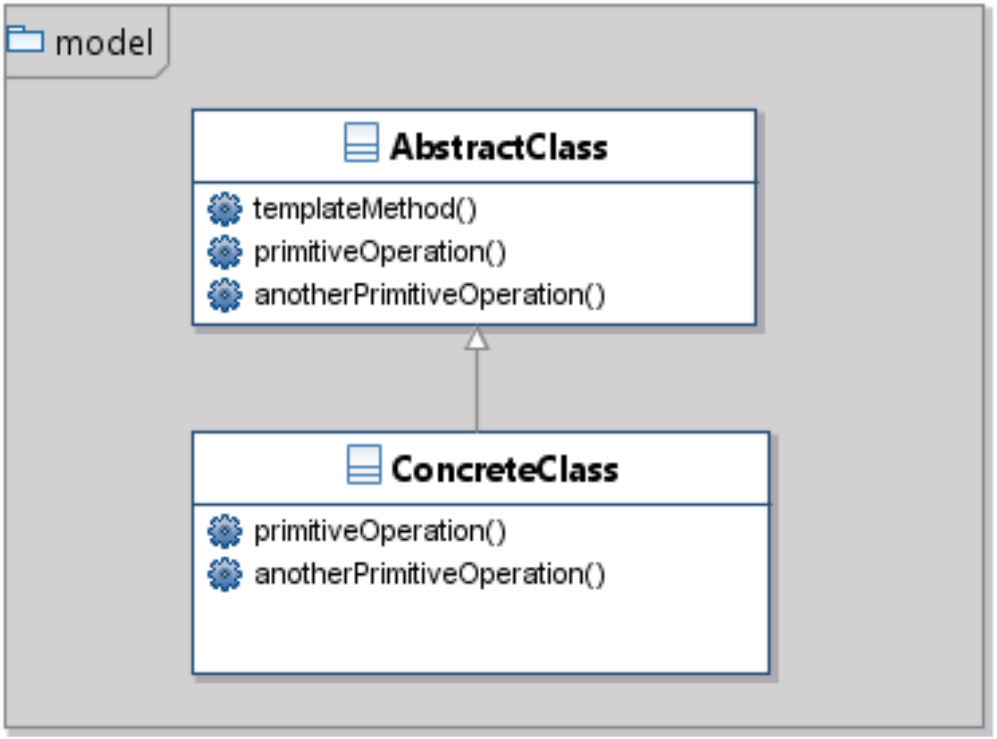
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

- **Scope**

Class

- **Examples**
  - Sorting, Game strategy

Static structure diagram



Participants

- **AbstractClass**
  - Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm
  - Implements a template method defining the skeleton of an algorithm
- **ConcreteClass**
  - Implements the primitive operations
  - Subclass-specific steps

Behaviour and implementation

- **Behaviour**
  - ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm
- **Implementation**
  - Naming conventions
    - Final methods
    - Abstract methods

When to use?

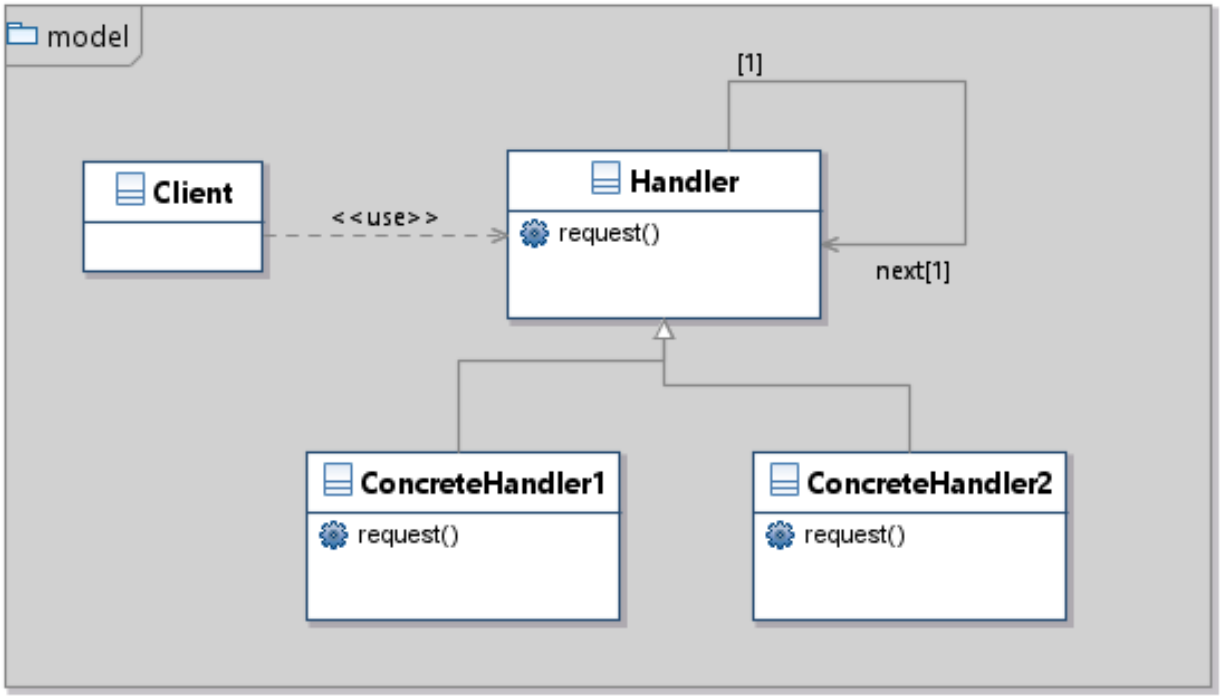
- Implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication
- To control subclasses extensions with hooks

Exercise

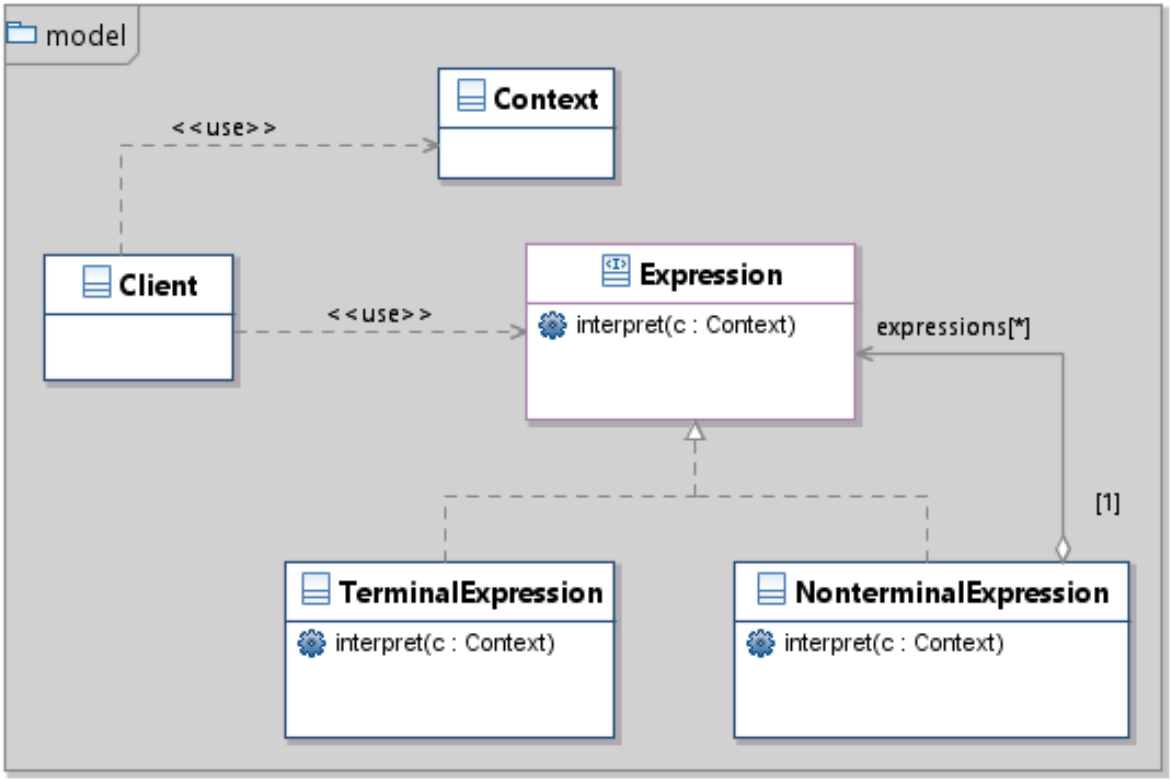
- **Write a personalised window class, which leaves the following responsibilities for the subclasses:**
  - Assemble view
  - Determine the size of the window
  - Determine the position of the window
  - Closing (hook method)

OTHER PATTERNS

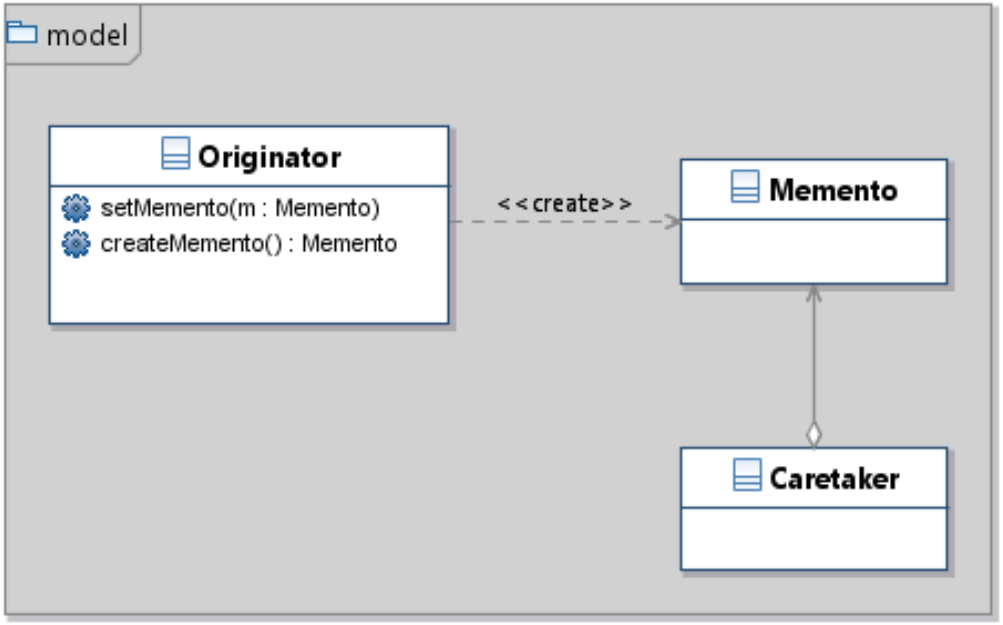
Chain of responsibility



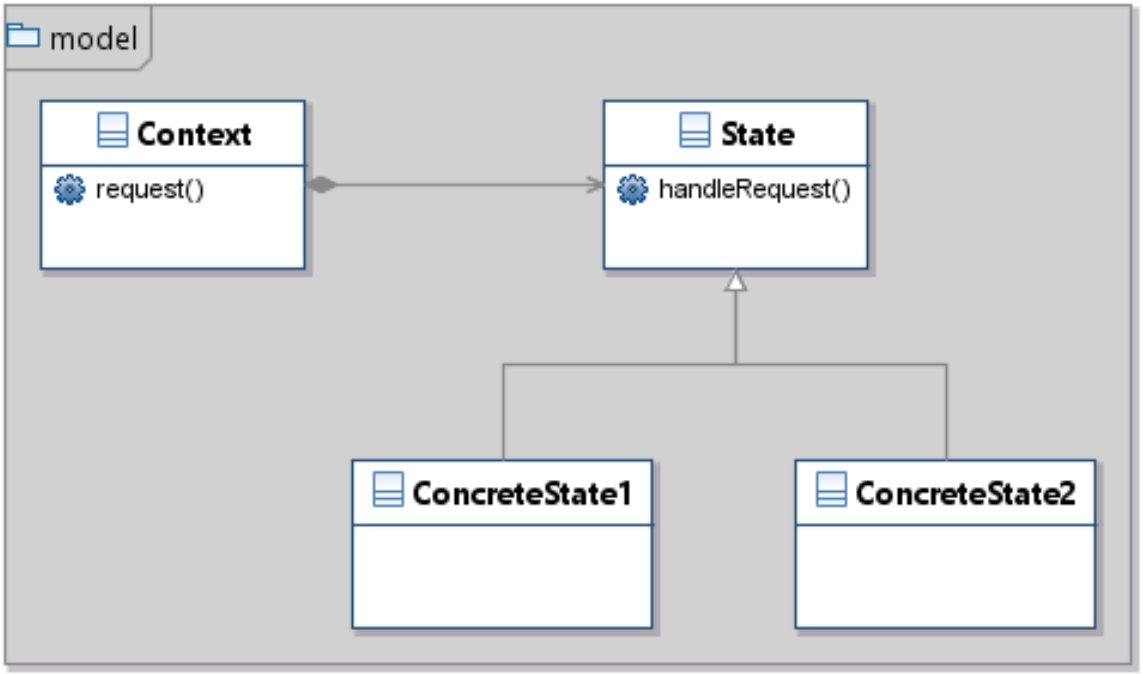
Interpreter



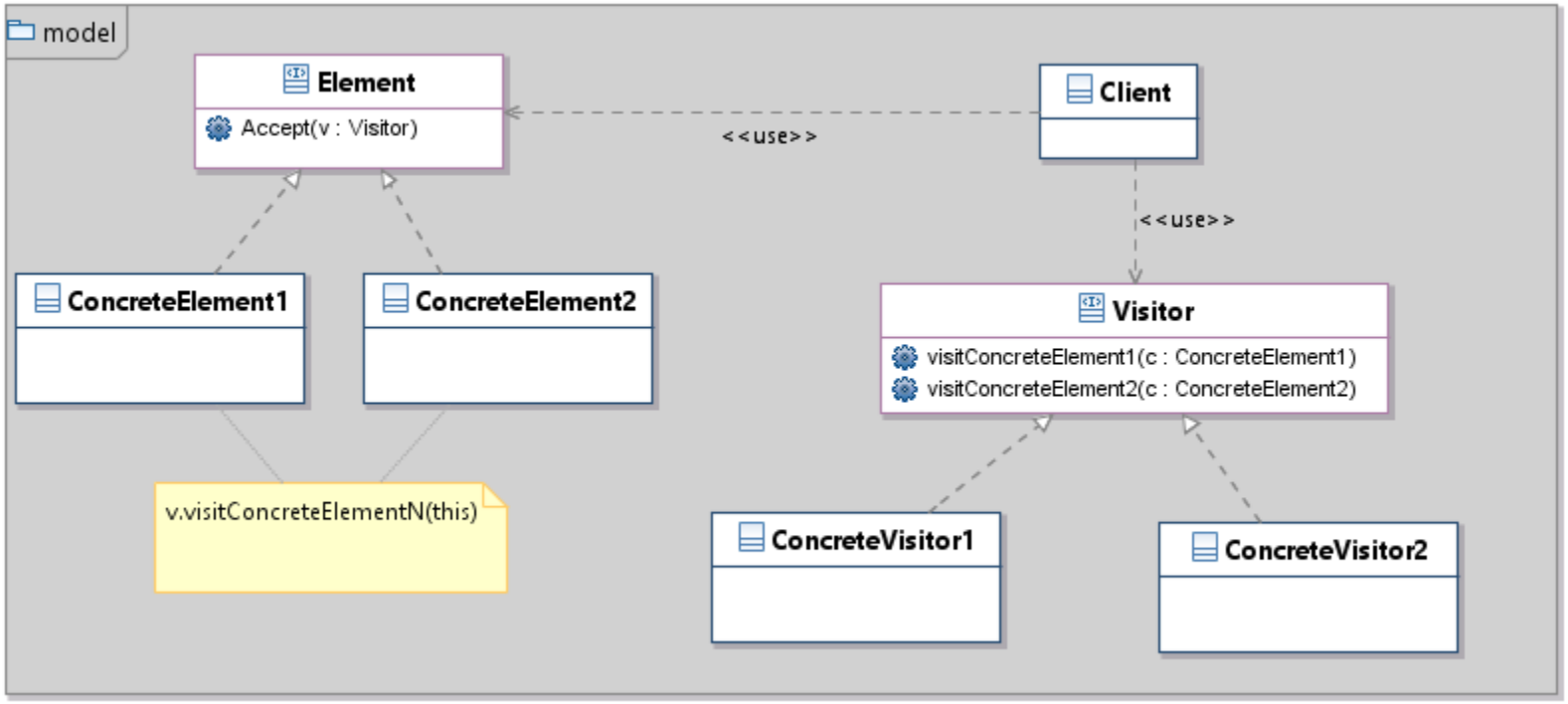
Memento


model
Originator
setMemento(m : Memento)
createMemento() : Memento
<<create>>
Memento
Caretaker

State


model
Context
request()
State
handleRequest()
ConcreteState1
ConcreteState2

Visitor


model
Element
Accept(v : Visitor)
Client
<<use>>
<<use>>
ConcreteElement1
ConcreteElement2
Visitor
visitConcreteElement1(c : ConcreteElement1)
visitConcreteElement2(c : ConcreteElement2)
v.visitConcreteElementN(this)
ConcreteVisitor1
ConcreteVisitor2

Design Patterns

HOMEWORK

Just Another Fast Food Restaurant

- Simulate the workflow of a restaurant
- **Workflow**
  1. Client orders a product
  2. The orders are prepared by a robot one after another (FIFO)
  3. The client receives and consumes the products

Just Another Fast Food Restaurant

- **Products**
  - The restaurant sells hot dogs and chips
  - Extras for the products: ketchup, mustard
  - Hot dog increases client happiness by 2
  - Chips increases client happiness by 5%
  - Ketchup doubles the effect of a product
  - Mustard increases client happiness by 1 and removes the effect of the product
  - We will introduce new products and extras (with different effects) in the near future

Just Another Fast Food Restaurant

- **Other informations**
  - We don’t expect other changes in the future
- **Hand in (email)**
  - Source code
  - Tests
  - For each design pattern
    - Where did you use it and why

<epam> Excellence in Software Engineering

BDO

THANK YOU FOR YOUR KIND ATTENTION!