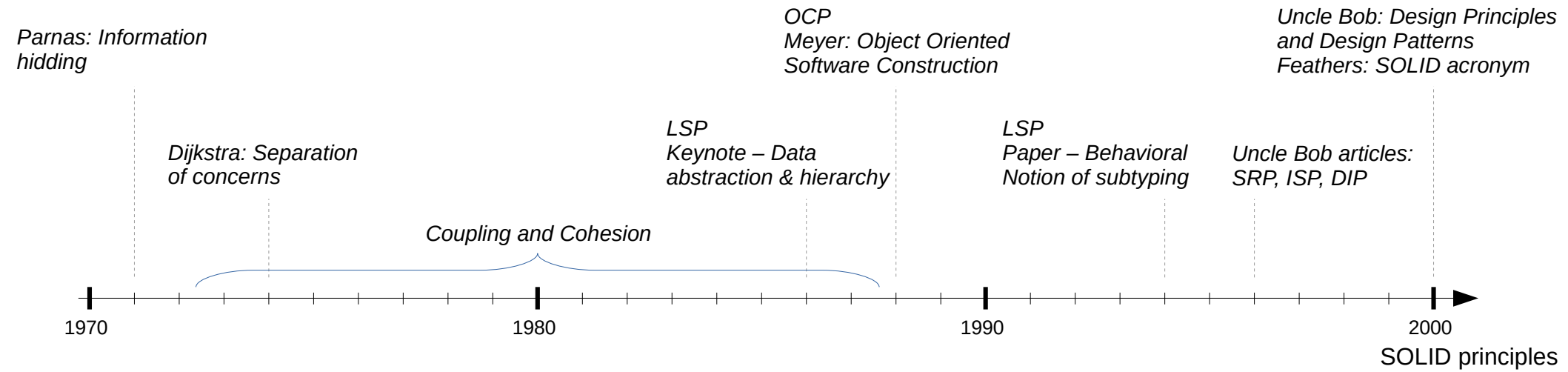
Design patterns are SOLID

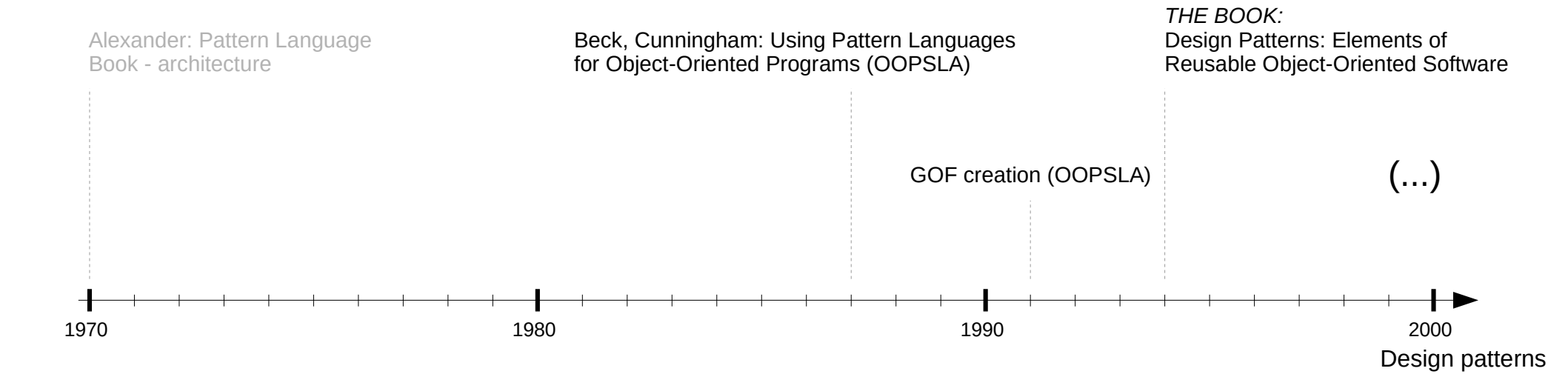
Jean-Luc Delarbre

Introduction

- **In previous lecture, we studied the SOLID principles**
  - Make software design understandable, flexible and maintainable
- **Our today topic: Design patterns**
  - Well designed solutions of recurring problems
  - Objectives
    - Not a long description of each one, but
    - How patterns fulfill SOLID principles
    - How patterns usage allows to be compliant with SOLID principles

Chronology of patterns and SOLID principles





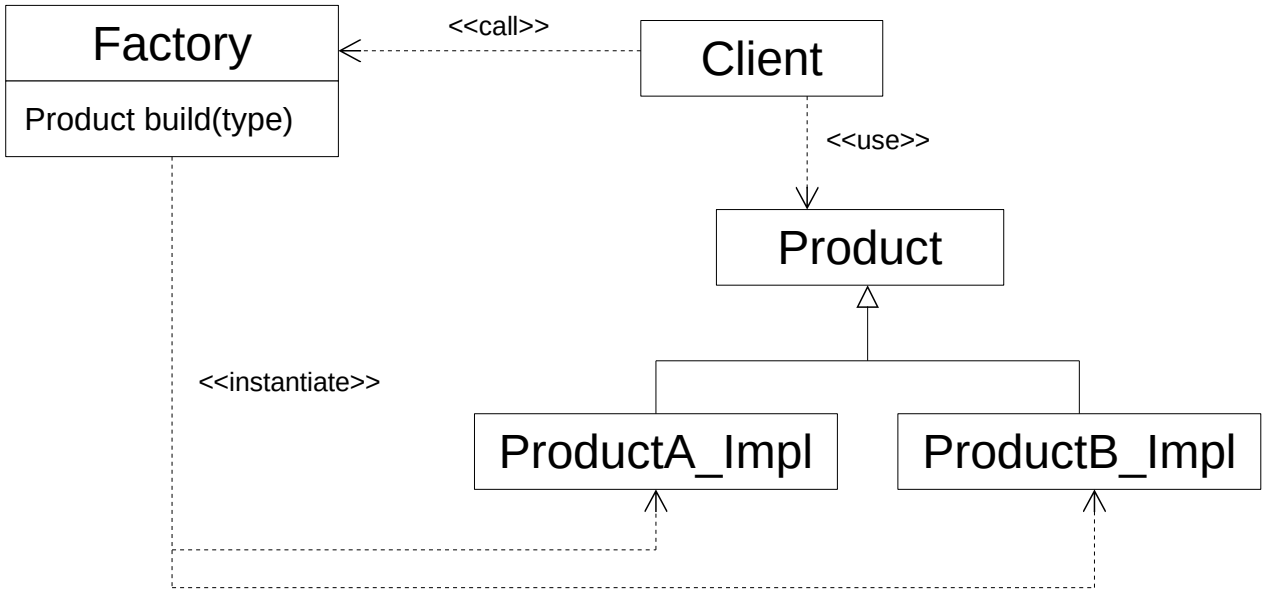
Creational Patterns

- **Abstract factory, builder, factory method, static factory… allow**
  - SRP
    - Wire the application is a full fledged responsibility
      - Avoid object to fetch other objects
      - Avoid to unrelated things into object just to allow easy access on them
  - OCP / DIP
    - Since code shall be built upon abstractions, you need to link it with concrete things
    - Lower coupling in the application

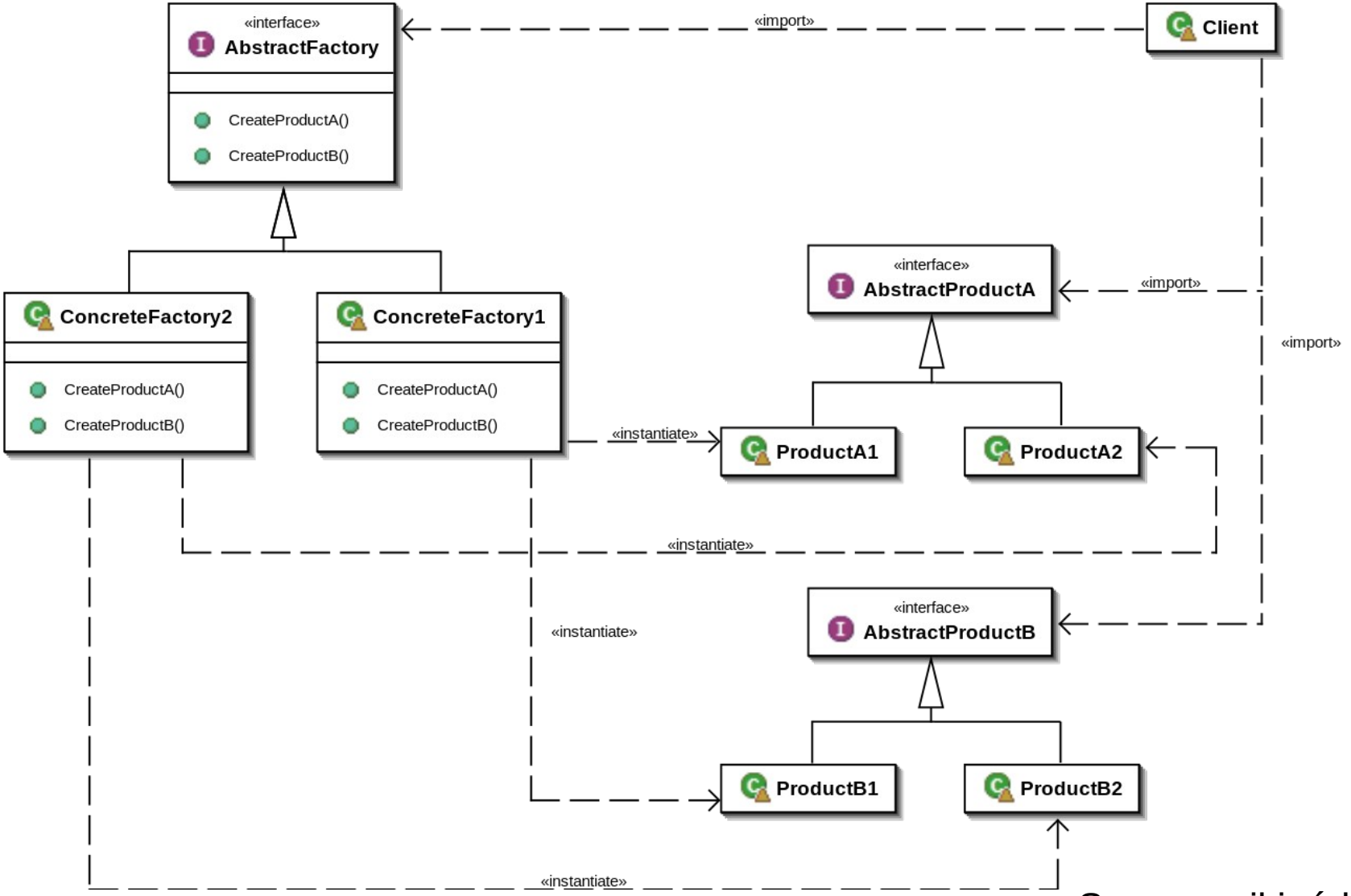
Roles of factories / builders

- **Build concrete instance**
  - Initialize data
  - Inject collaborators (wires dependencies)
    - Build or retrieve them
    - Select concrete implementation
  - Call constructor (factory privilege for object construction only)
    - Constructor only set class fields
  - Initialize object with init method (almost never)
- **Select (and hide) concrete class to build for client**
  - Return object of a given interface: select implementation
- **Select instance to return**
  - Create a new one or return an unique instance (immutable object)
- **Ease construction of complex objects (builder)**
- **Return ready to use object or fail**
  - Check network connection established or file opening

Factory


Factory
Product build(type)
<<call>>
Client
<<use>>
Product
<<instantiate>>
ProductA_Impl
ProductB_Impl

Abstract factory

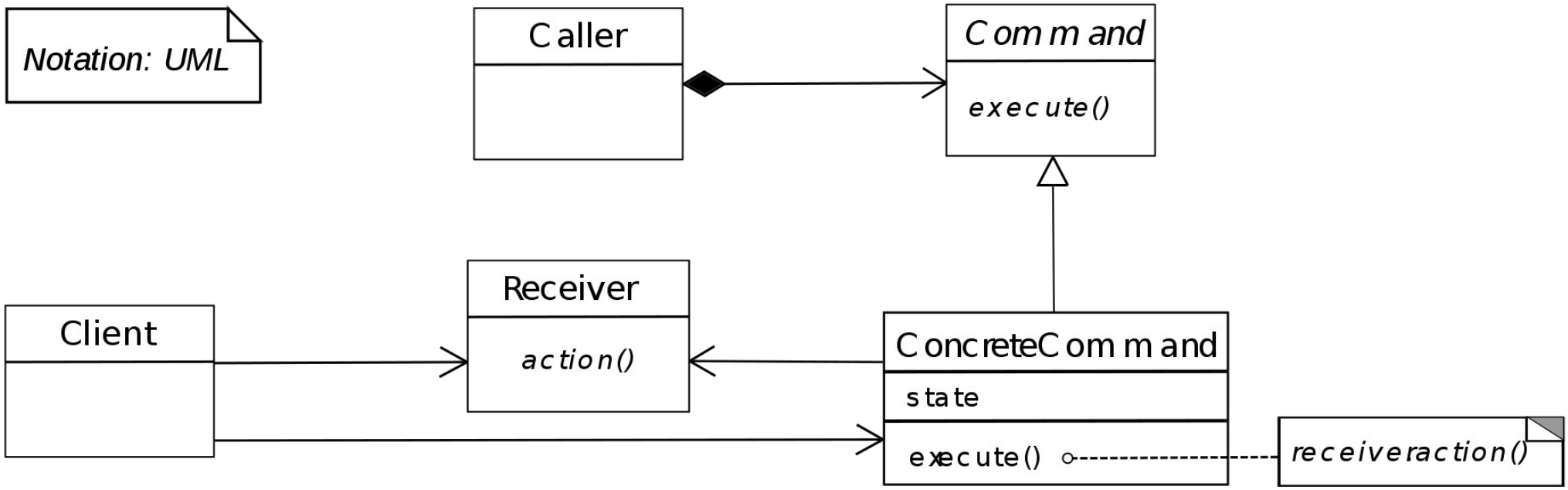


Source: wikipédia

Command

- **Behavioral pattern**
- **Simplest pattern ever**
  - Highly focused on 1 responsibility (SRP)
  - Highly handy and reusable
    - Components that use a “command” object fulfill OCP and DIP
- **Often work with factory**

Command

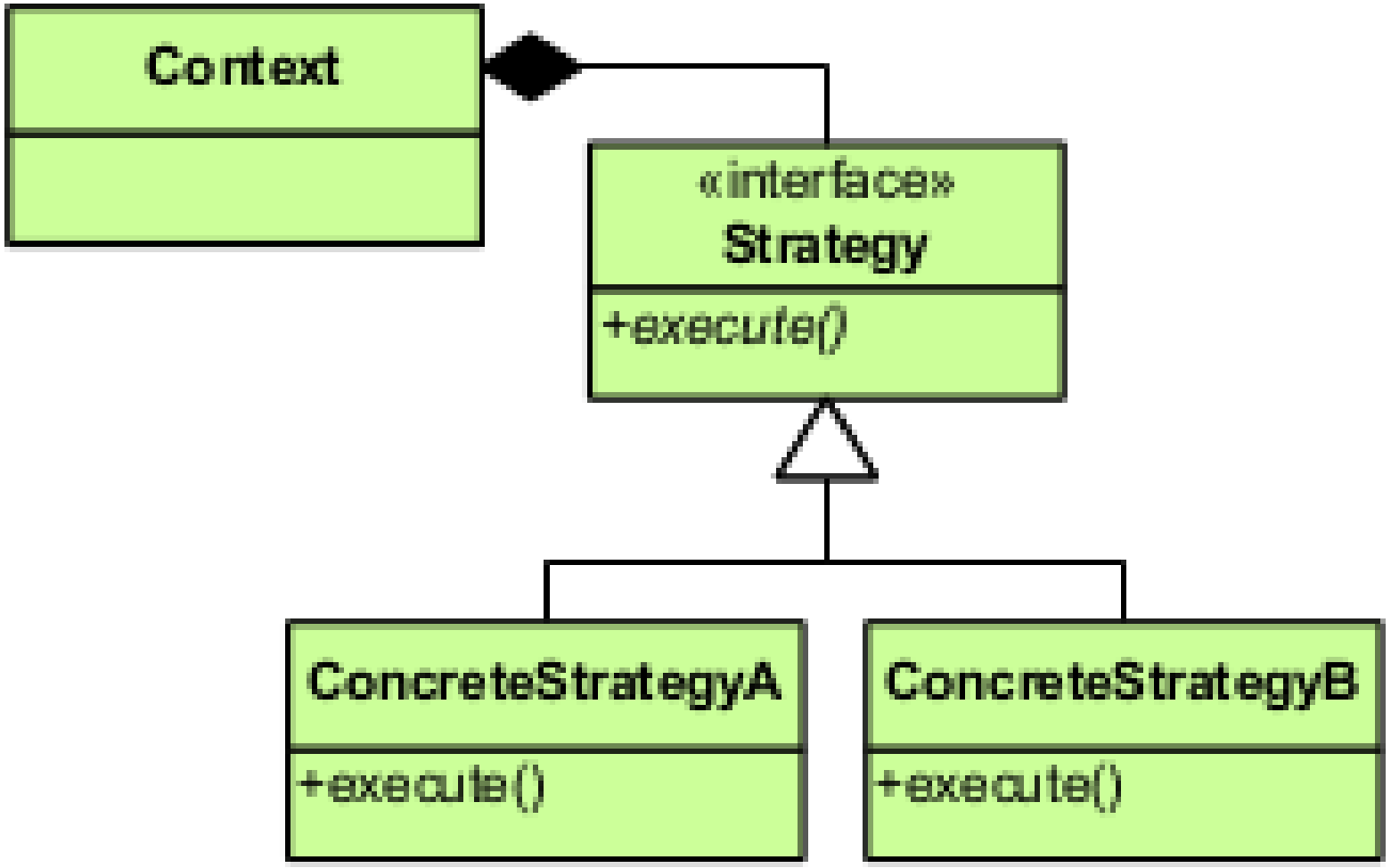


Source: wikipédia

Strategy

- **Behavioral pattern**
- **Delegate a part of behavior to dedicated object (SRP)**
  - Implemented by a family of algorithm
  - Algos and clients in this pattern vary independently (OCP / DIP)
  - Changeable at runtime
- **Like a command pattern with rich interface**

Strategy

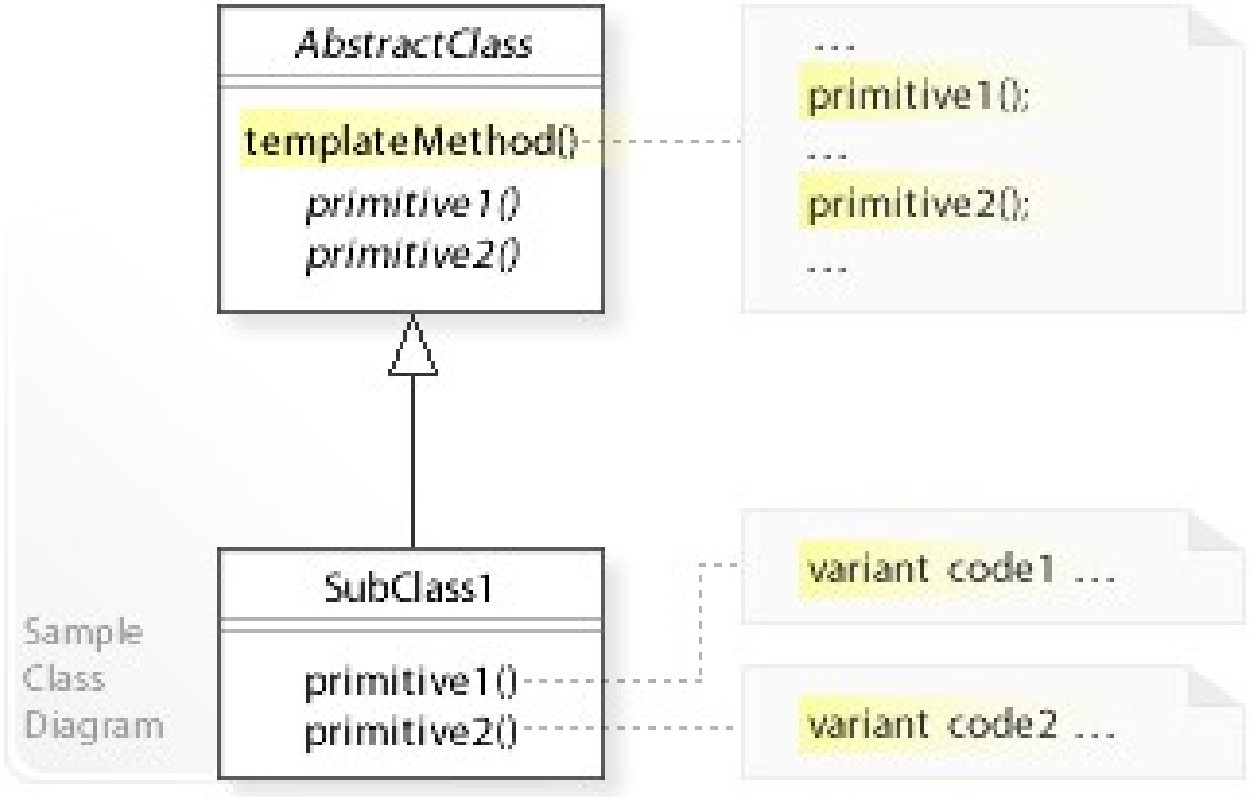


Source: wikipédia

Template method

- **Behavioral pattern**
- **Equivalent to strategy but with inheritance**
  - Stronger coupling
  - Strategy not updatable

Template method

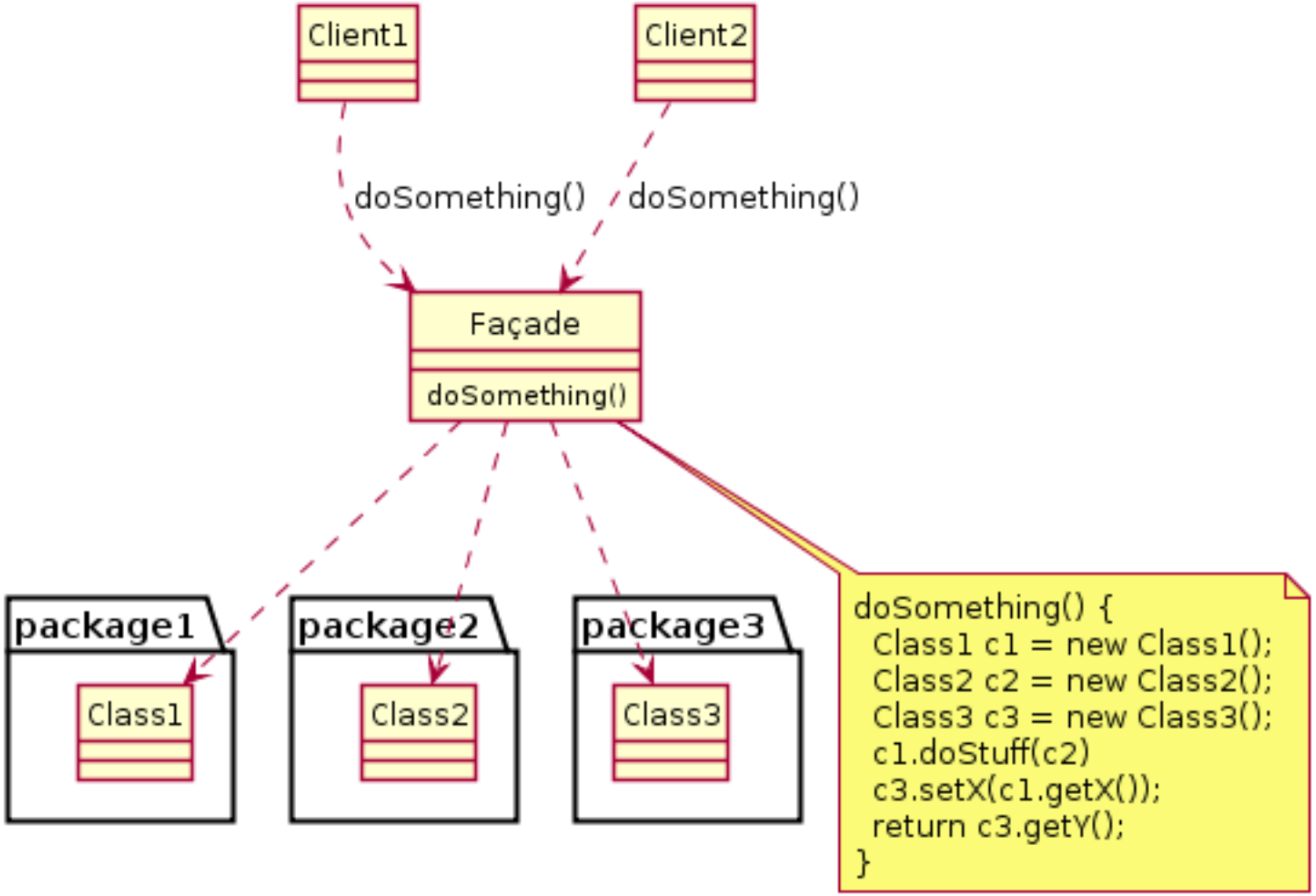


Source: wikipédia

Facade / Mediator

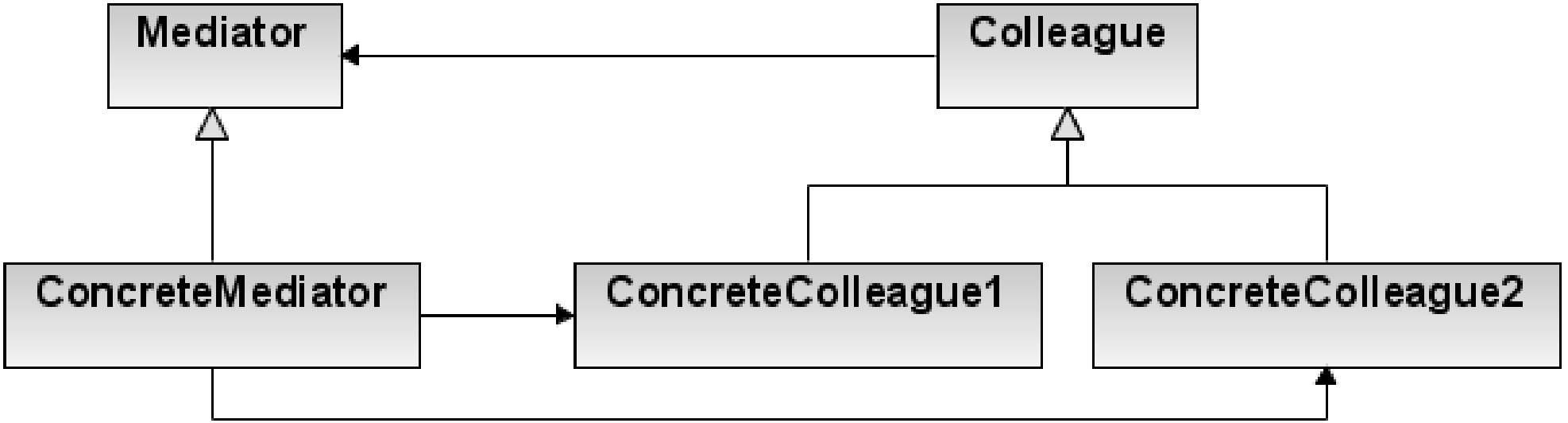
- **Impose a policy on a group of objects**
  - Ensure coherence or application policy is a full fledged responsibility (SRP)
- **Facade**
  - Structural pattern
  - Hide complexity, give better readability and usability of complex components
  - Impose policy (often with business value) through an API which impose a way to use it
- **Mediator**
  - Behavioral pattern
  - Encapsulate interaction between objects
  - Allow a given policy behind the scene

Facade

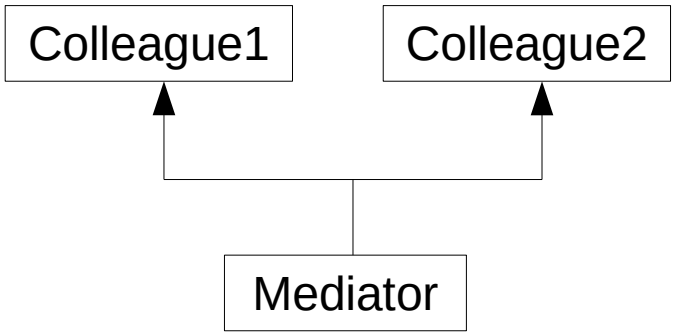


Source: wikipédia

Mediator



Source: wikipédia



Simpler diagram

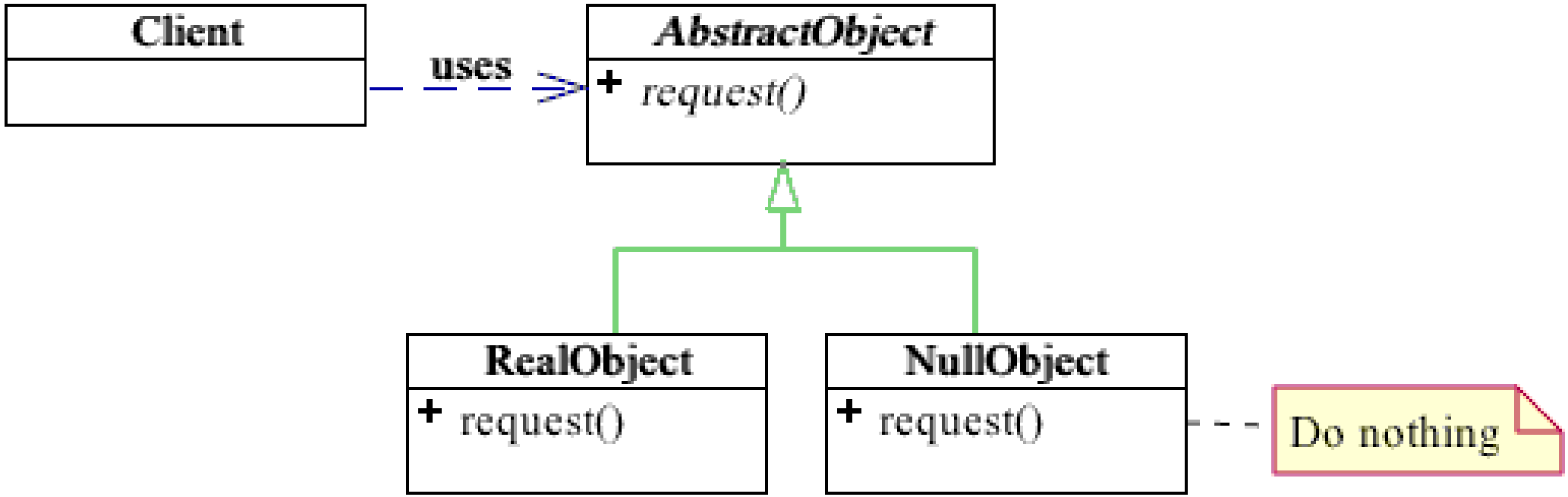
Null Object / Optional

- **Avoid use of null pointer (or exception handling)**
- **Null Object**
  - Not in GOF (Fowler)
  - Allow to easily do nothing on special cases (SRP)
    - Simplify code architecture (avoid if / else)
    - Main algo does not have to bothered with special cases
    - Ex: File processing in a directory tree, use null object with empty directory
  - Special case of strategy or state pattern
  - Degenerated case of mediator which deals with nothing

Null Object / Optional

- **Avoid use of null pointer (or exception handling)**
- **Optional**
  - Explicitly manage special cases
    - Kind of facade: impose policy of special case management
  - Simplest monad (functional programming)

Null Object

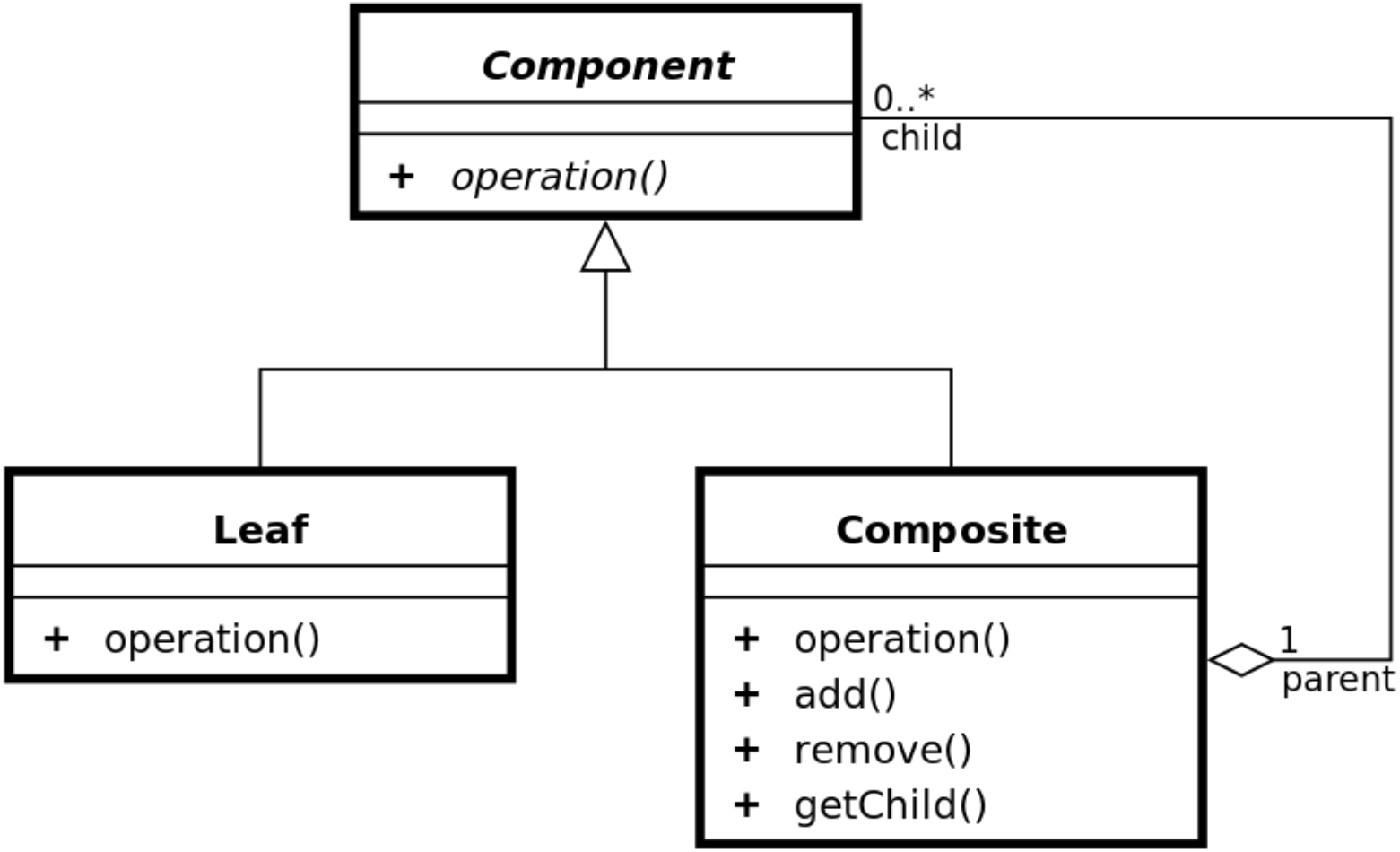


Source: Google image / cs.oberlin.edu

Composite

- **Structural pattern**
- **Manage in the same way individual or composite elements**
  - One to many
  - Avoid client to manage cardinality (SRP, DIP)

Composite

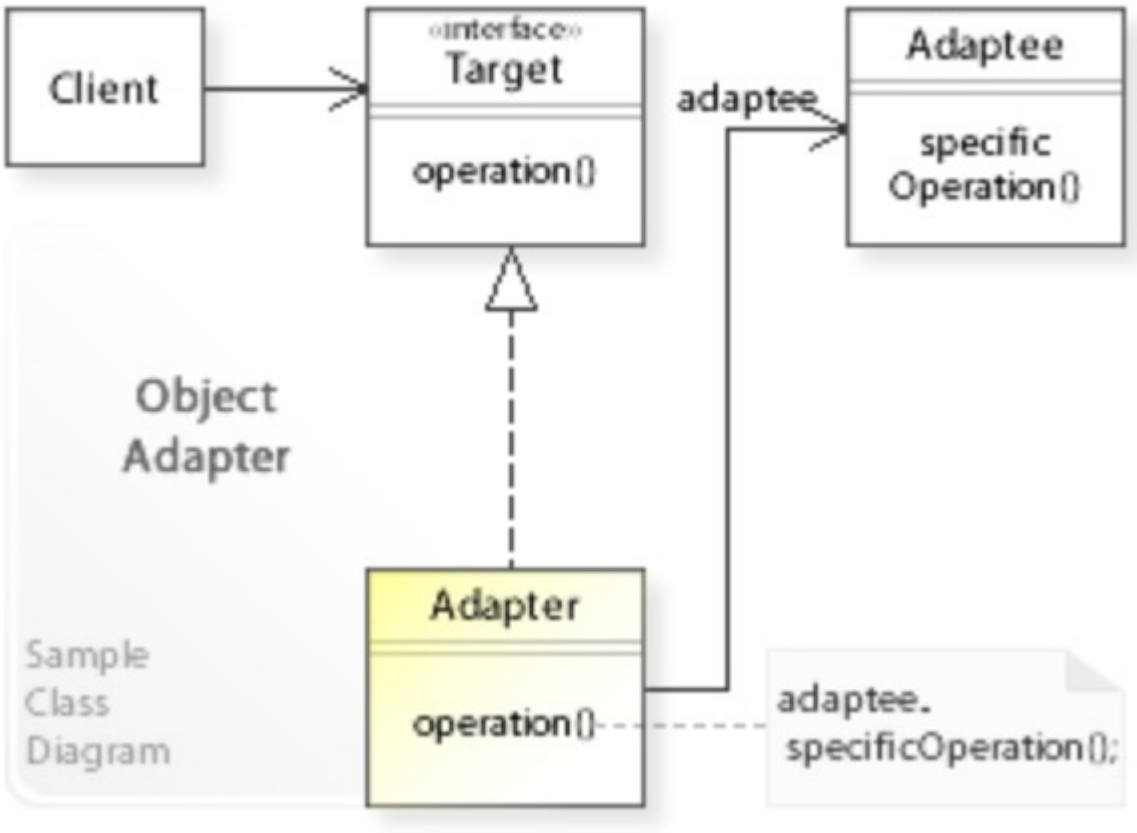


Source: wikipédia

Adapter

- **Structural pattern**
- **Convert an interface of foreign element**
  - Let other client access through original original interface
  - Allow us to be independent from external library
    - Creation of our own interface to use external library (not used it directly)
    - For already existing interface: adapter is needed

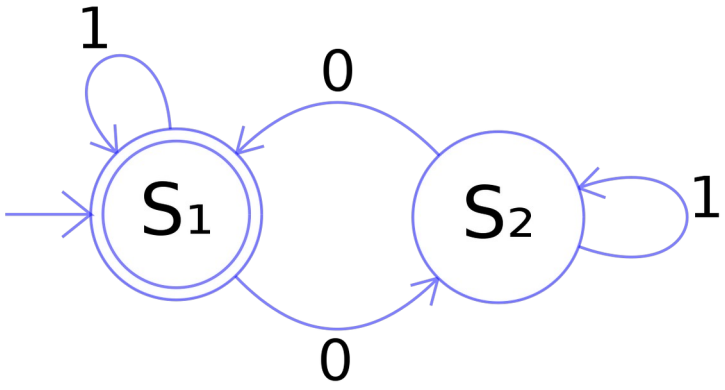
Adapter



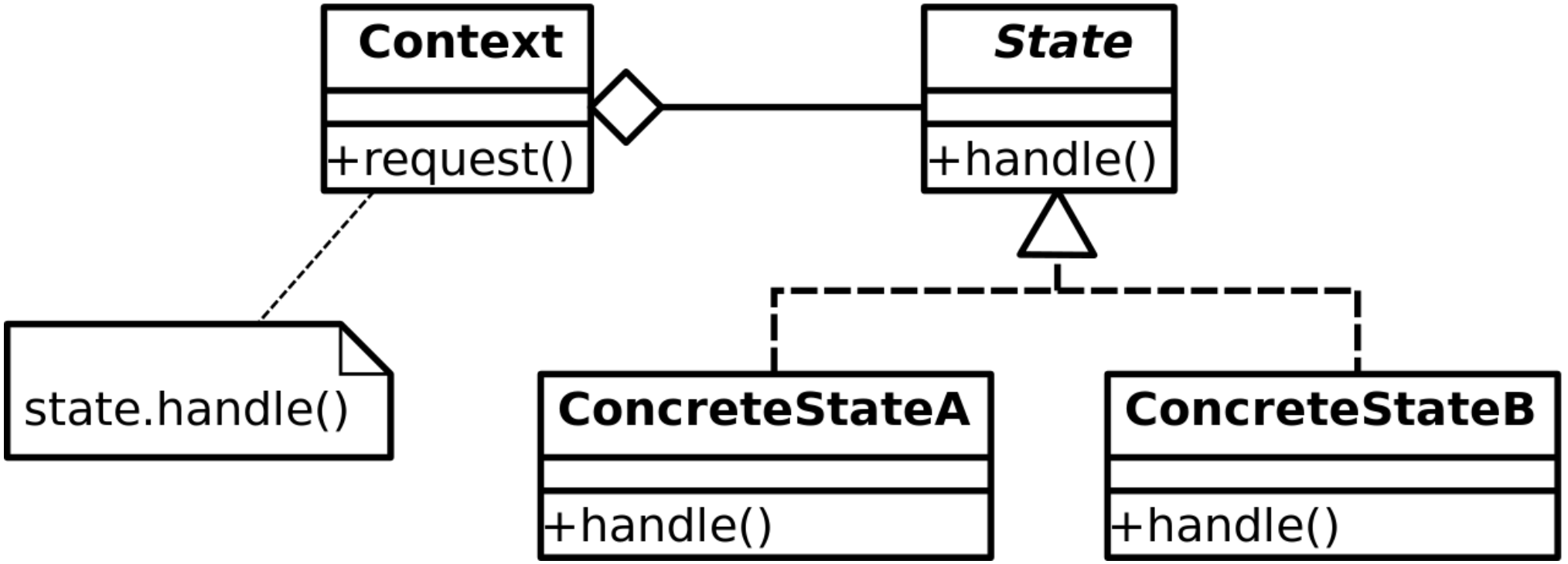
Source: wikipédia

State

- **Behavioral pattern**
- **Close to Finite State Machine**
- **State pattern**
  - State specific behaviors are delegated to dedicated state objects
    - State objects can vary independently
    - Implement a common interface
  - Main (context) class
    - Receive requests and transfer it to active state for execution
    - Memorize active state



State



Source: wikipédia

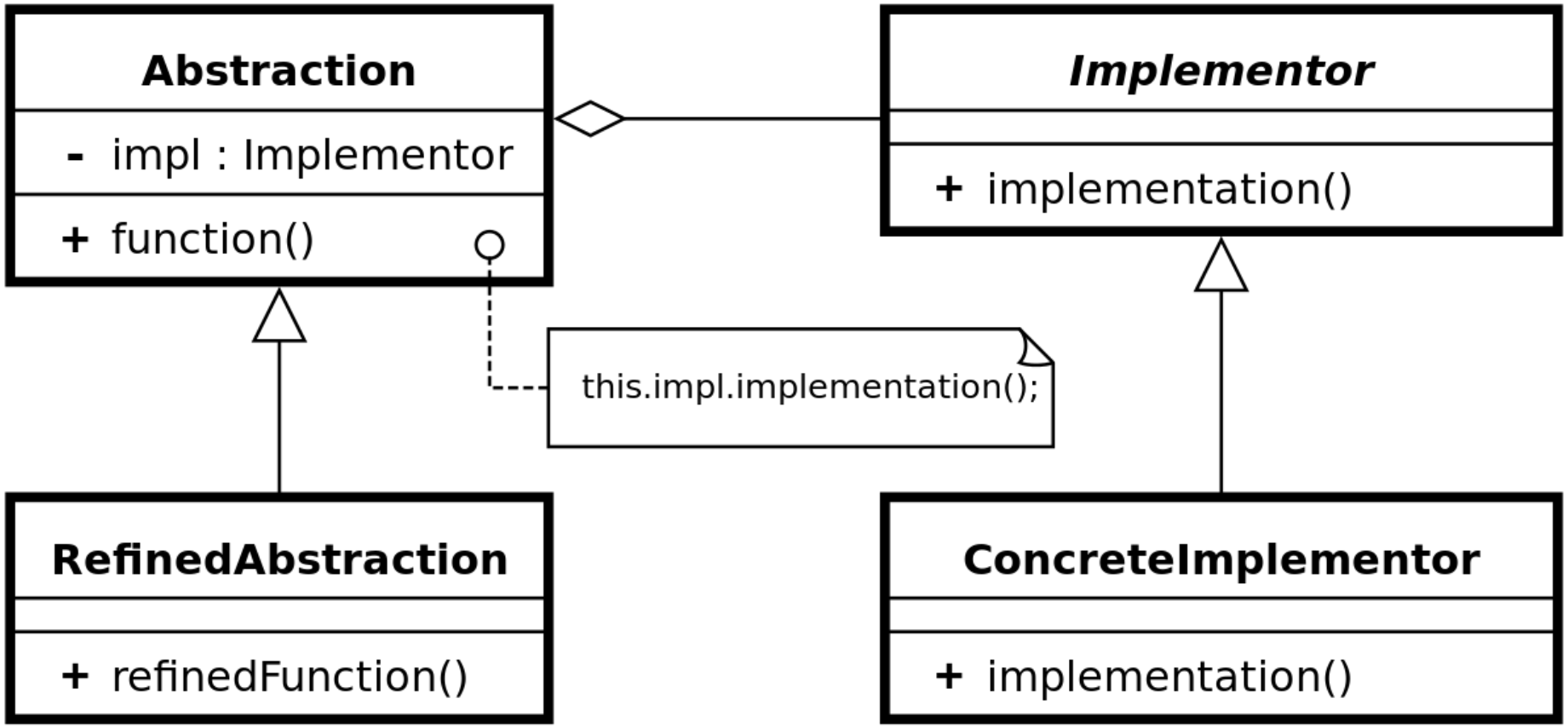
State

- **Very helpful for GUI**
- **Ease State/transition of FSM decoupling**
- **State versus Strategy**
  - State is also a Strategy pattern (see UML diagram)
    - Both strategy and state delegate behavior to derivative classes
  - Strategy instances are not always state
    - In state, derivative classes have the logic to change the state of the context class

Bridge

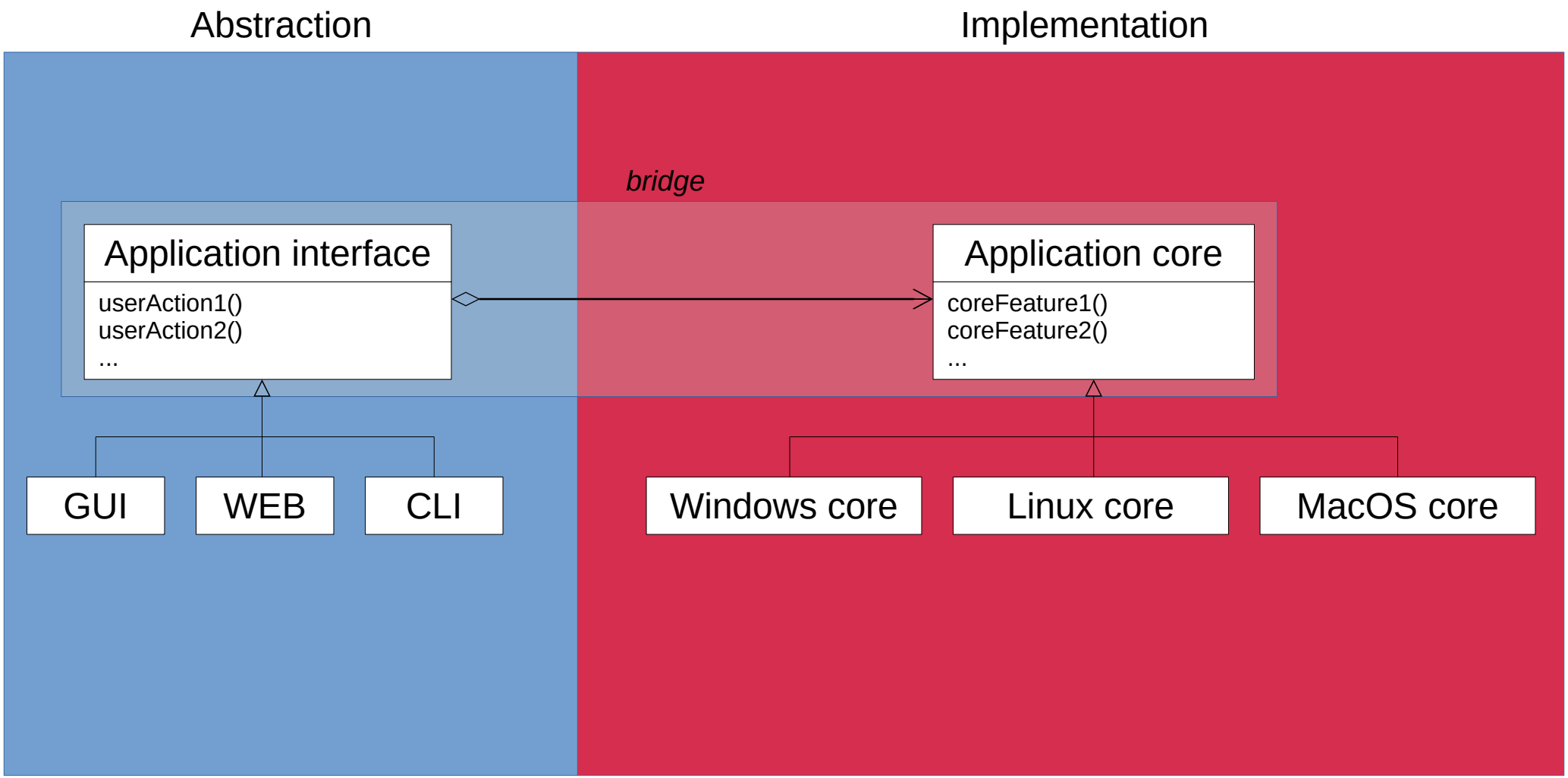
- **Structural pattern**
- **Decouple interface from its implementation**
  - Let them vary independently
  - Use both aggregation and inheritance
- **Simplified in other languages by traits or mixins (Scala)**

Bridge



Source: wikipédia

Bridge (frequent use case)



Next objective

- Tools for today programmers
- One more pattern

Reference

- **Design Patterns: Elements of Reusable Object-Oriented Software - GOF - Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides**
- **Agile Principles, Patterns, and Practices in C# - Martin C. Robert, Martin Micah**
- **Clean Code, A Handbook of Agile Software Craftsmanship - Robert C. Martin**
- **Refactoring: Improving the Design of Existing Code - Martin Fowler, Kent Beck (Contributor), John Brant (Contributor), William Opdyke, don Roberts**