

# Design Patterns

---



# Who am I?

---

- ◆ Name: Maurizio Martignano
- ◆ Experience: working with C/C++ since 1983, with Java since 1996.
- ◆ Application domains: system software, embedded systems, satellites data management systems, radiation software, control automation.
- ◆ Currently he's Technopoint – EXOR International Corporate Technology Co-ordinator, responsible for co-ordinating the development efforts in all technical areas, among which relevant to this presentation are:
  - Embedded systems / control automation
  - Communication protocols
  - (Web based) Distributed Applications
- ◆ Where: Verona, ITALY (+39-045-8750404, Maurizio.Martignano@acm.org)

# Definitions

---

- ◆ A pattern is a generalised solution to a particular type of problem.
- ◆ Everywhere that we face that same type of problem, we can use the same solution.
- ◆ In addition, a pattern typically provides a flexible and robust solution.
- ◆ Using patterns enables designers to arrive at a solution much quicker than if they had to devise the solution themselves.
- ◆ Design patterns also enable inexperienced designers to develop solutions that are just as flexible and powerful as those from expert designers with many years of experience.

# History

---

- ◆ Patterns are a recent software engineering problem-solving discipline that emerged from the object-oriented community. Patterns have roots in many disciplines, including literate programming, and most notably in Alexander's work on urban planning and building architecture (Alexander, 1977).
- ◆ The goal of the pattern community is to build a body of literature to support design and development in general. There is less focus on technology than on a culture to document and support sound design. Software patterns first became popular with the object-oriented Design Patterns book (Gamma et al., 1995).

- ◆ But patterns have been used for domains as diverse as development organization and process, exposition and teaching, and software architecture. At this writing, the software community is using patterns largely for software architecture and design ...
- ◆ Today, the pattern discipline is supported by several small conferences, by a broad spectrum of activities at established software engineering conferences, and by a rapidly growing body of literature.

# The GoF Book

---

- ◆ Strongly suggested reading:

Gamma, Helm, Johnson and Vlissides “Design Patterns (Elements of Reusable Object-Oriented Software), Addison Wesley, 1995, ISBN 0-201-63361-2.

# Describing Design Patterns

---

- ◆ Pattern Name – the name of the pattern
- ◆ Intent – what does the pattern do
- ◆ A.k.a – also known as
- ◆ Motivation – an illustrating scenario
- ◆ Applicability – when, where the pattern can be applied
- ◆ Structure – a graphic representations of the patterns
- ◆ Participants – the classes / objects participating in the pattern
- ◆ Collaborations – how the participants collaborate to carry out their responsibility

# Describing Design Patterns

(cont)

- ◆ Consequences – how does the pattern supports its objectives? What are the trade-offs and result
- ◆ Implementation – what pitfalls, hints, techniques, should you be aware of
- ◆ Sample code
- ◆ Known uses
- ◆ Related Patterns

# Design Patterns Classification

---

## ◆ Creational Patterns

- Abstract Factory – Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Builder – Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Factory method – Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
- Prototype – Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Singleton – Ensure a class only has one instance, and provide a global point of access to it.

# Design Patterns Classification

(cont)

## ◆ Structural patterns

- Adapter – Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Bridge – Decouple an abstraction from its implementation so that the two can vary independently.
- Composite – Compose objects into three structures to represent the part-whole hierarchies. Composite lets clients treat individual objects and composition of objects uniformly.
- Decorator – Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

# Design Patterns Classification

(cont)

- Facade – 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.
- Flyweight – Use sharing to support large numbers of fine-grained objects efficiently.
- Proxy - Provide a surrogate or a placeholder for another object to control access to it.

## ◆ Behavioural Patterns

- Chain of Responsibility – Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Command – Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations.
- Interpreter – Given a language, define a representation for its grammar along with an interpreter that uses that representation to interpret sentences in the language.
- Iterator – Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

# Design Patterns Classification

(cont)

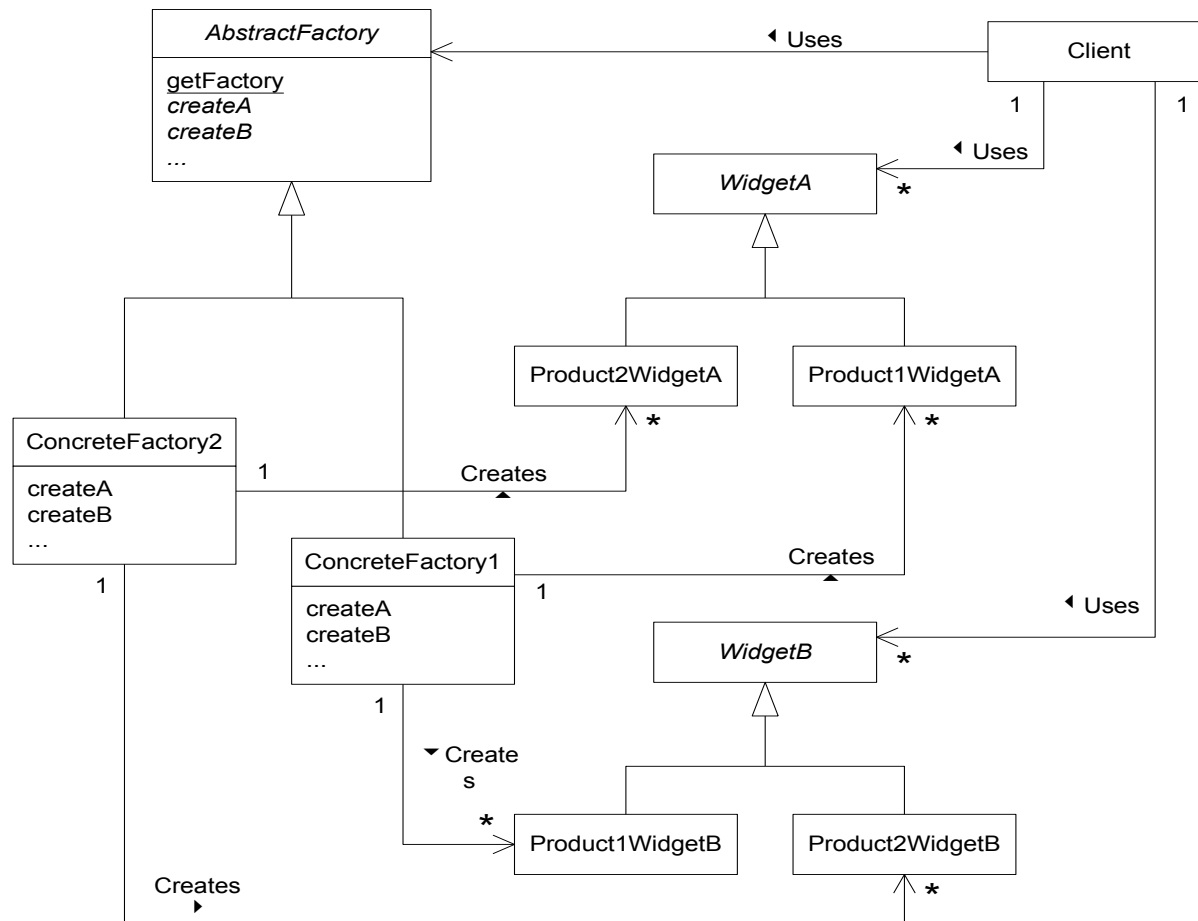
- Mediator – Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Memento – Without violating encapsulation, capture and externalise an object's internal state so that the object can be restored to this state later.
- Observer – Define a one-to-many dependency between objects so that when one object changes state, all its dependent are notified and updated automatically.
- State – Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

# Design Patterns Classification

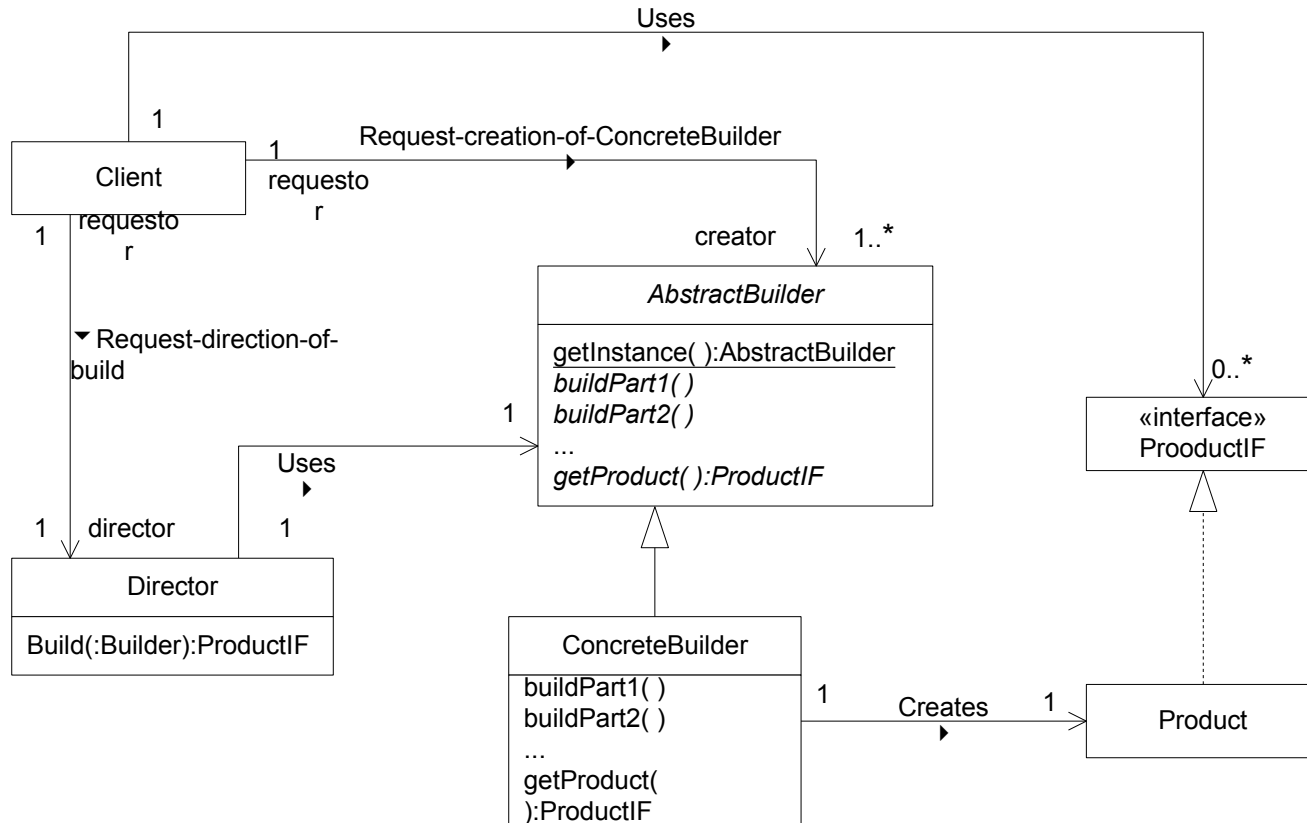
(cont)

- Strategy – Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Template Method – 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 structure.
- Visitor – Represent the 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.

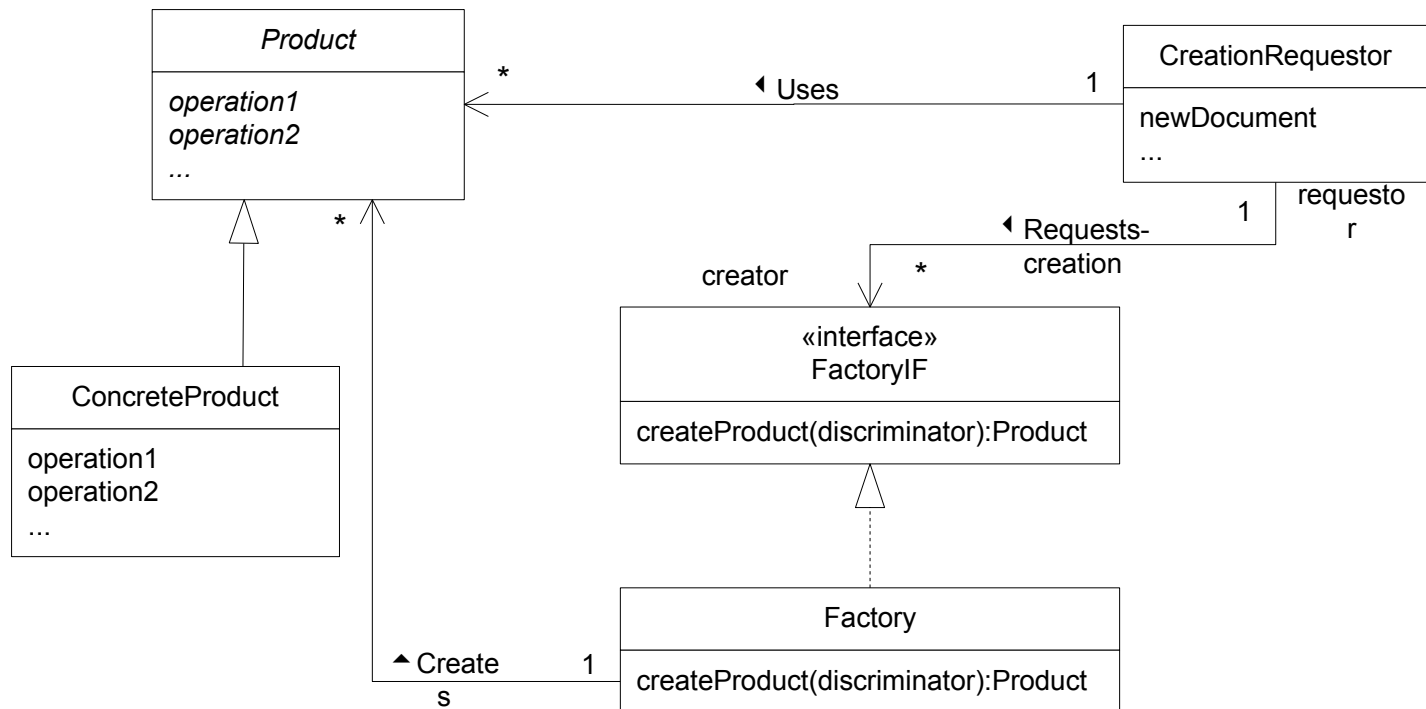
# Creational - Abstract Factory



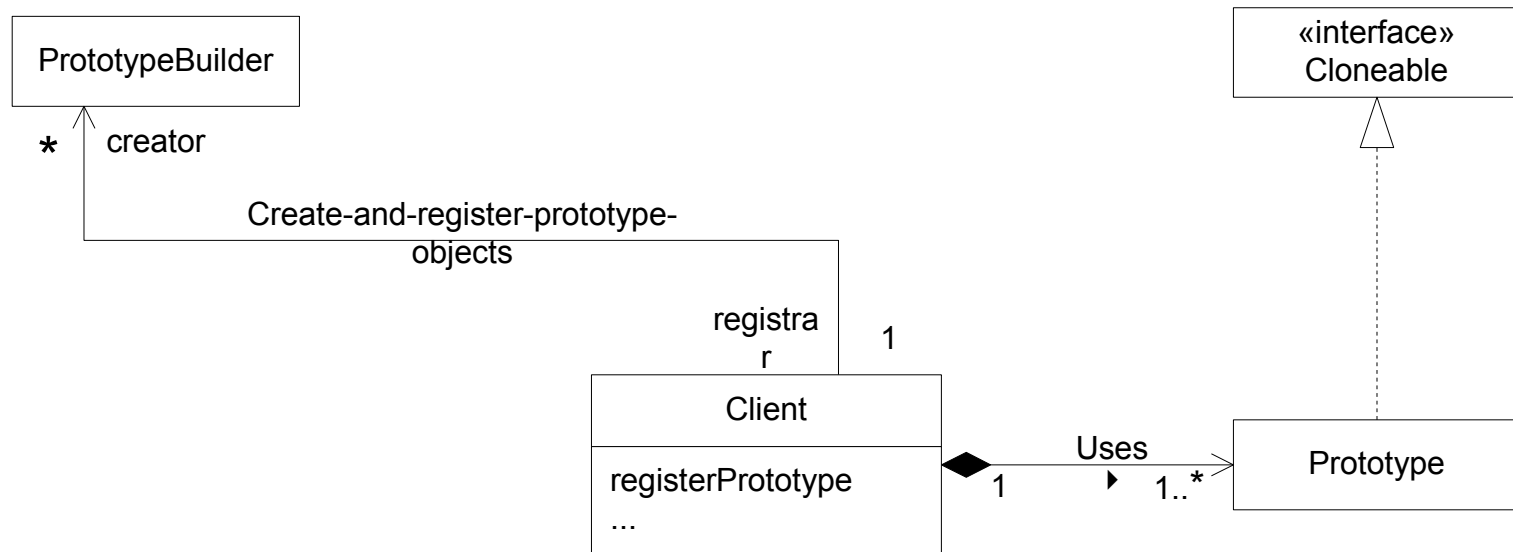
# Creational - Builder



# Creational - Factory Method



# Creational - Prototype

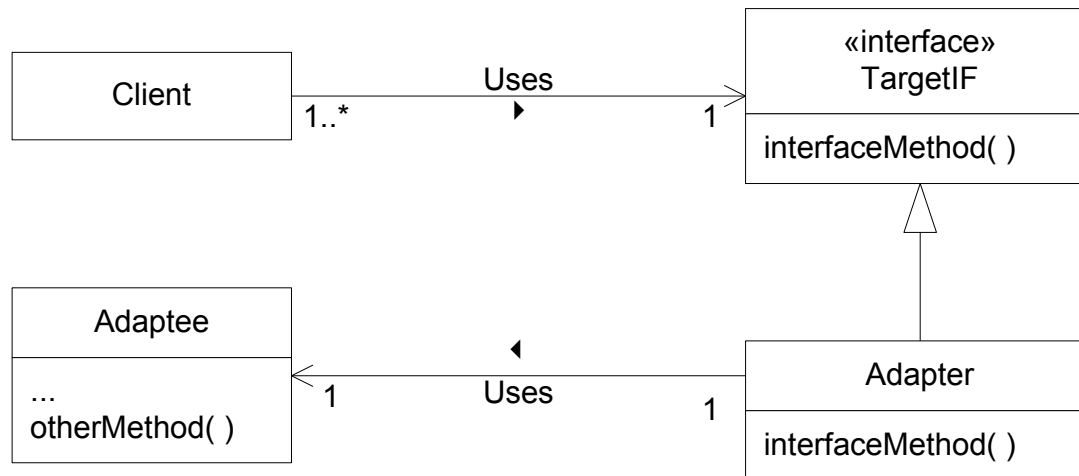


# Creational - Singleton

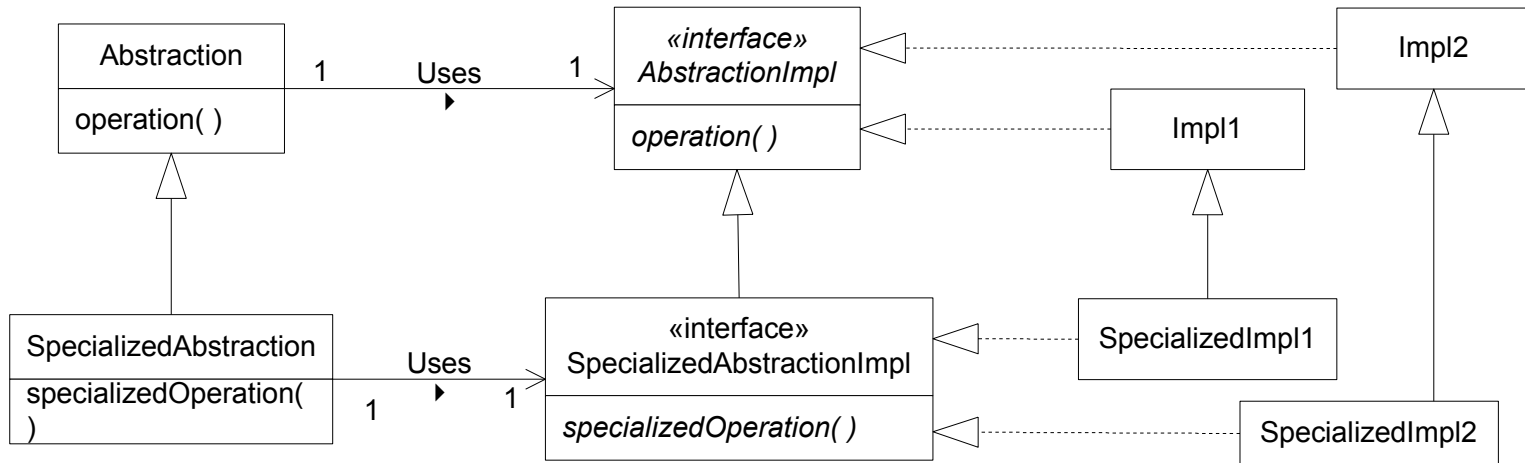
---

Singleton
<u>-singletonInstance</u> ...
«constructor» -Singleton( ) «misc» <u>+getInstance( )</u> ...

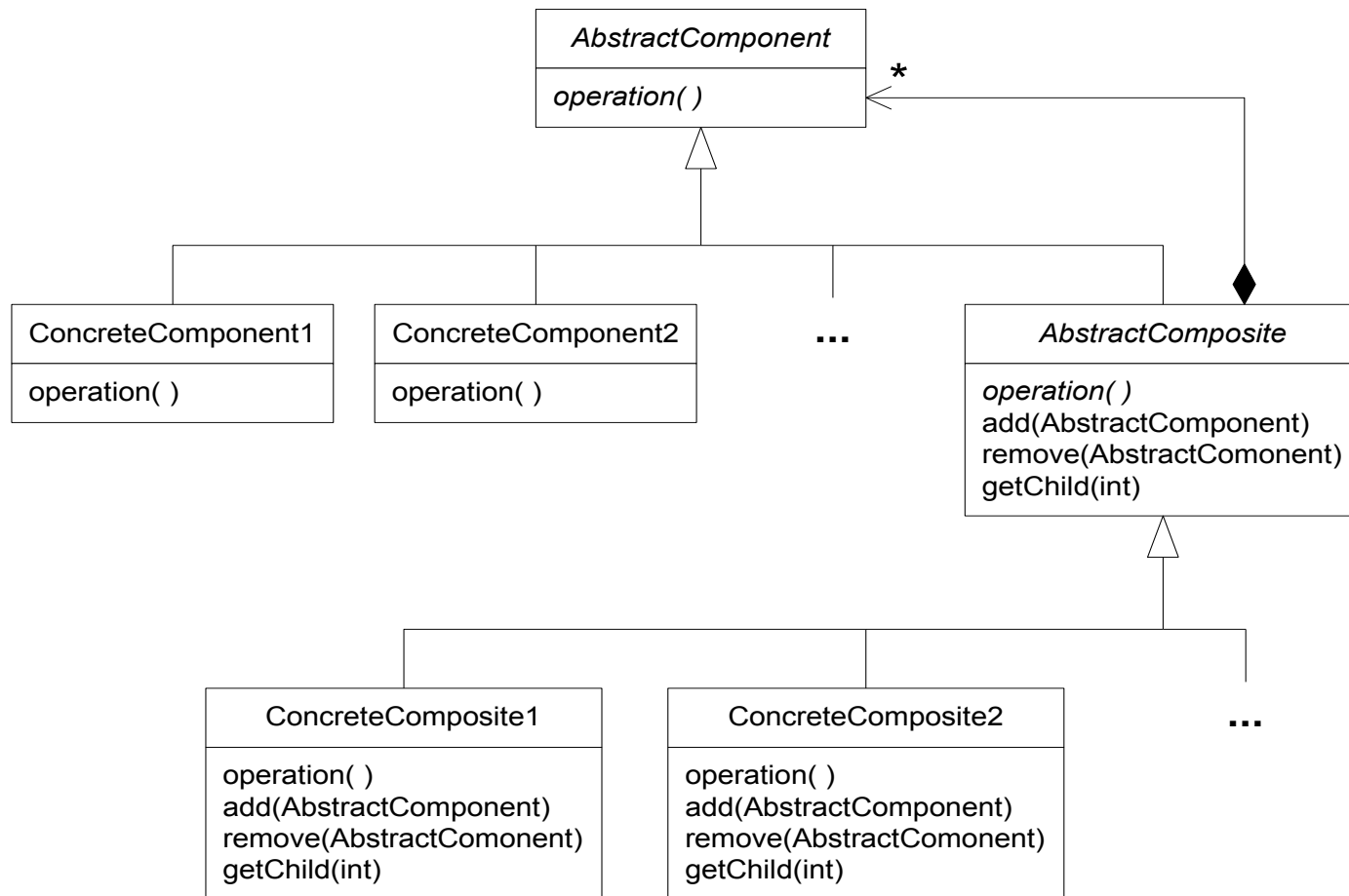
# Structural - Adapter



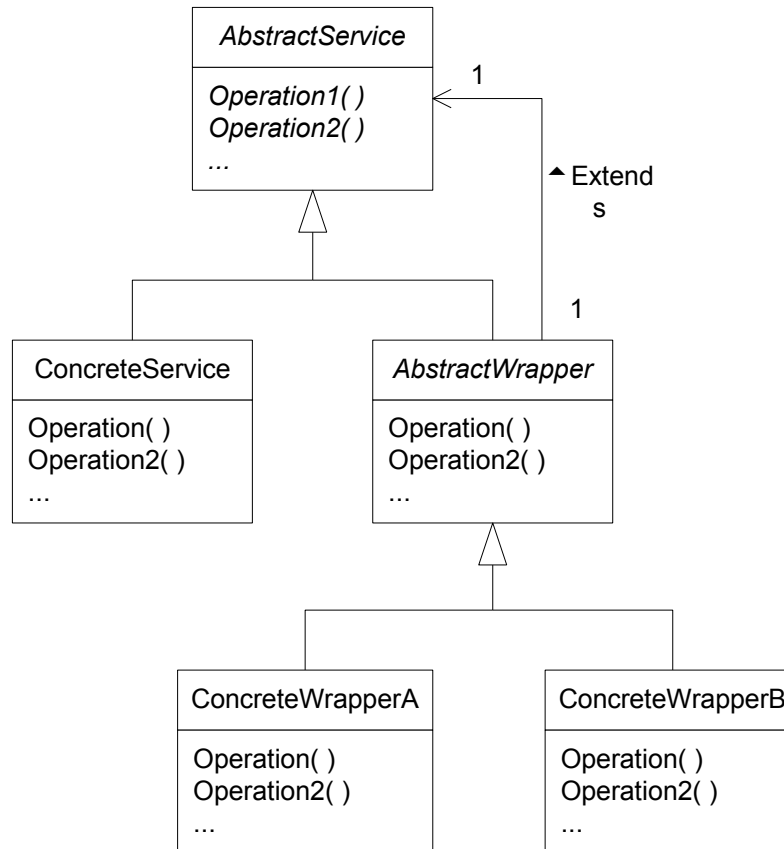
# Structural - Bridge



# Structural - Composite

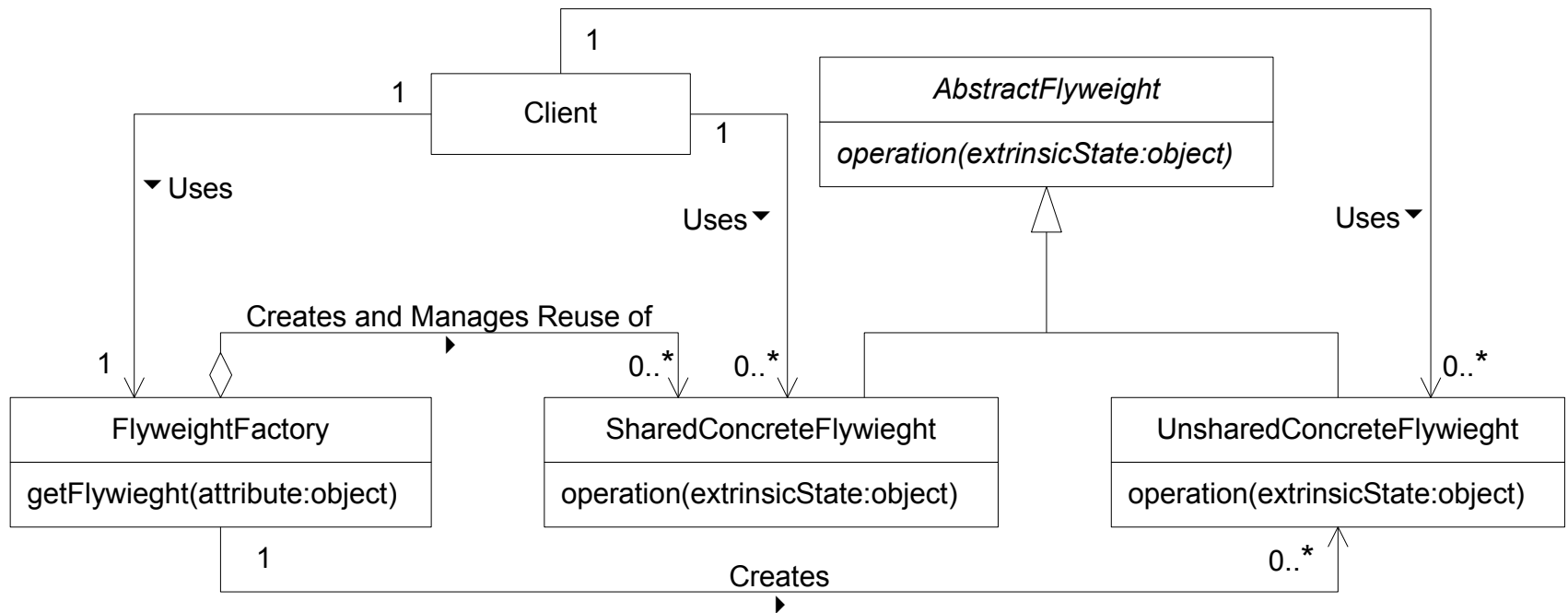


# Structural - Decorator

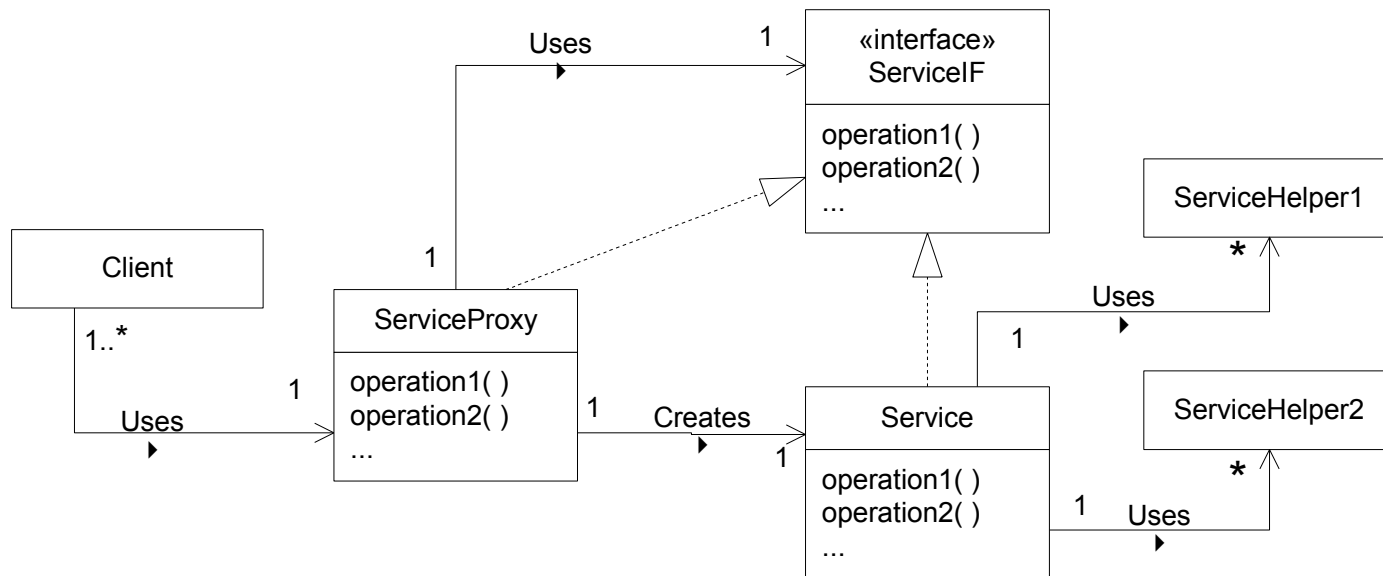




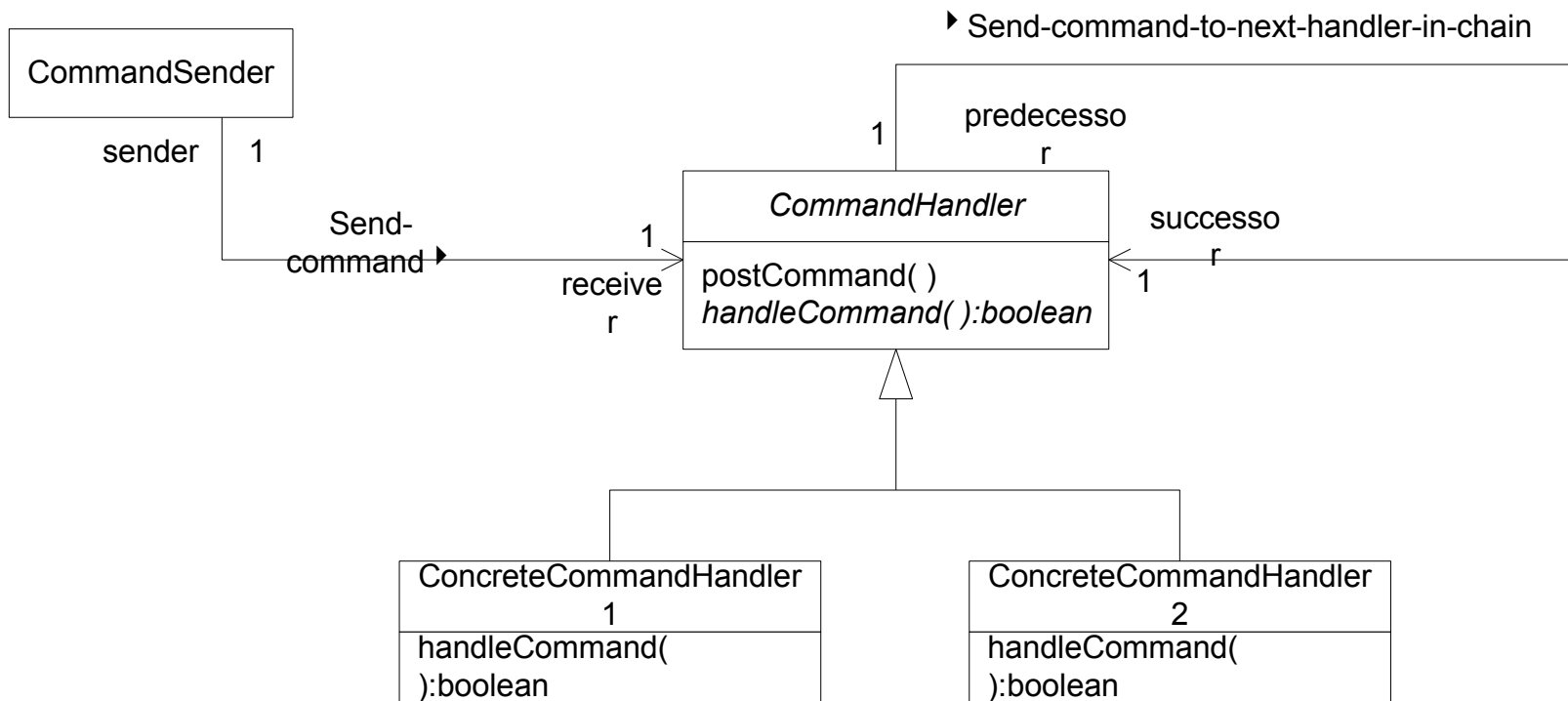
# Structural - Flyweight



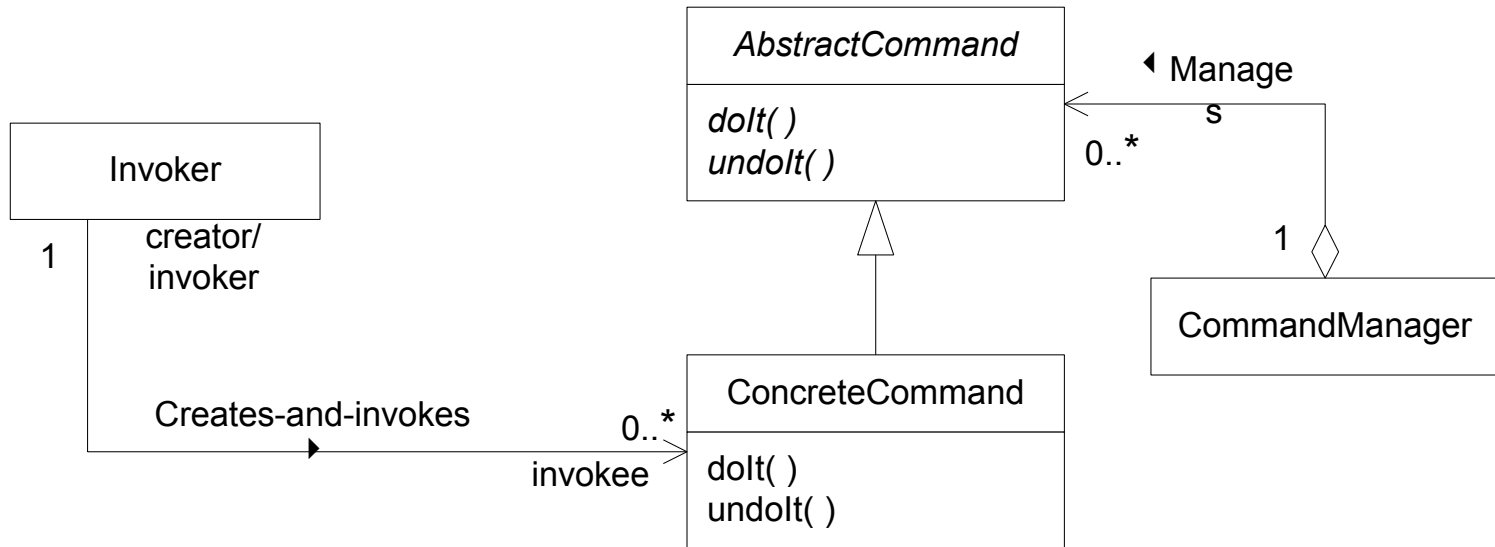
# Structural - Proxy



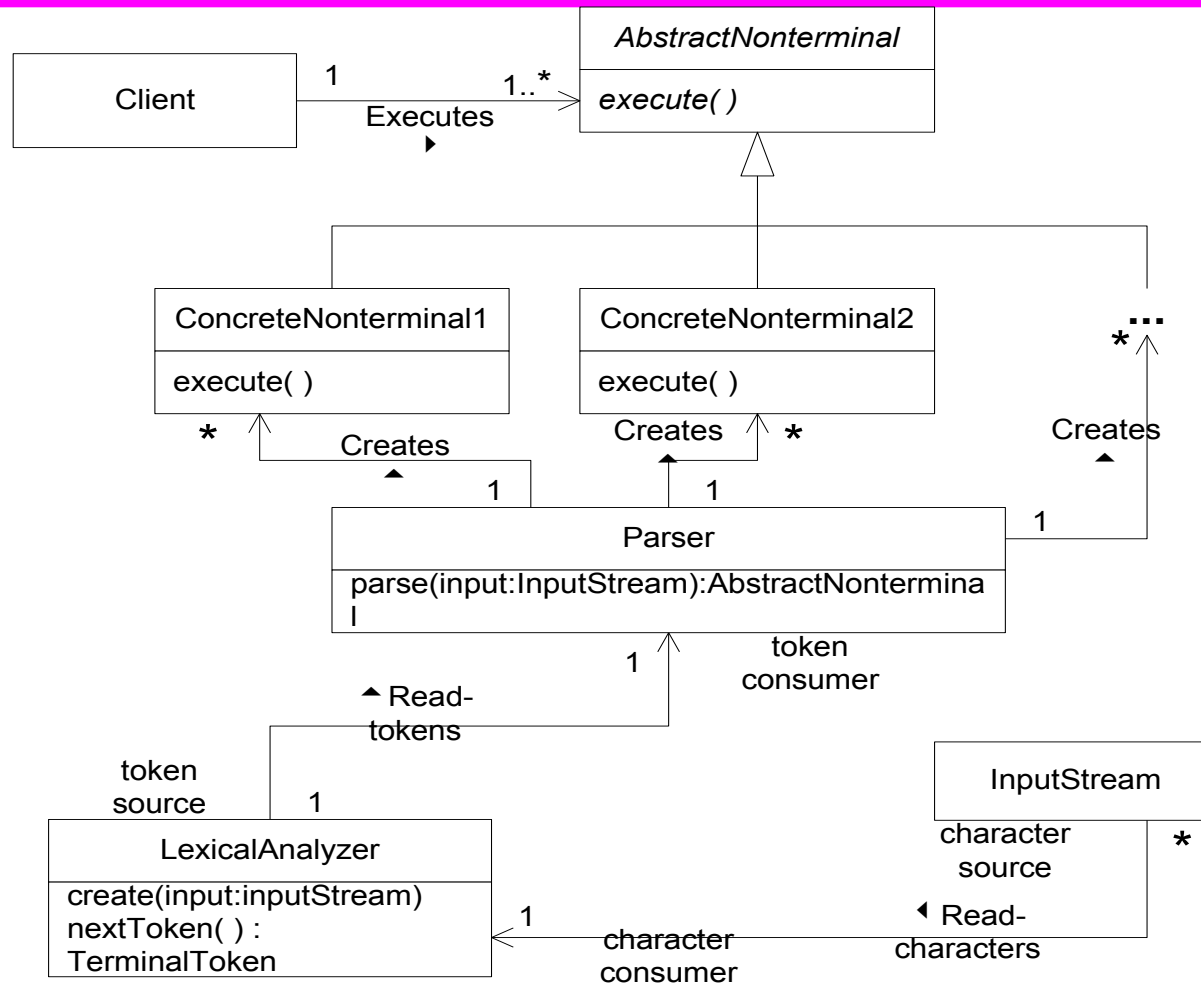
# Behavioural - Chain of Responsibility



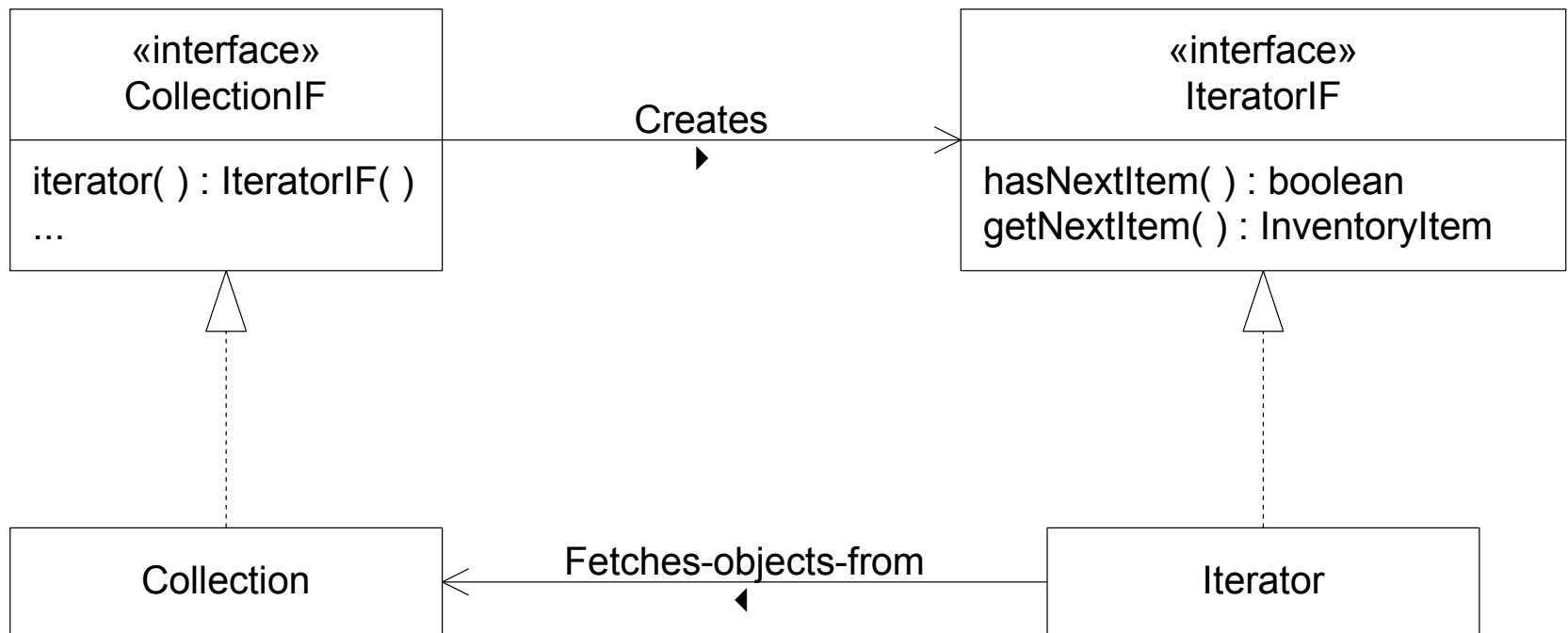
# Behavioural - Command



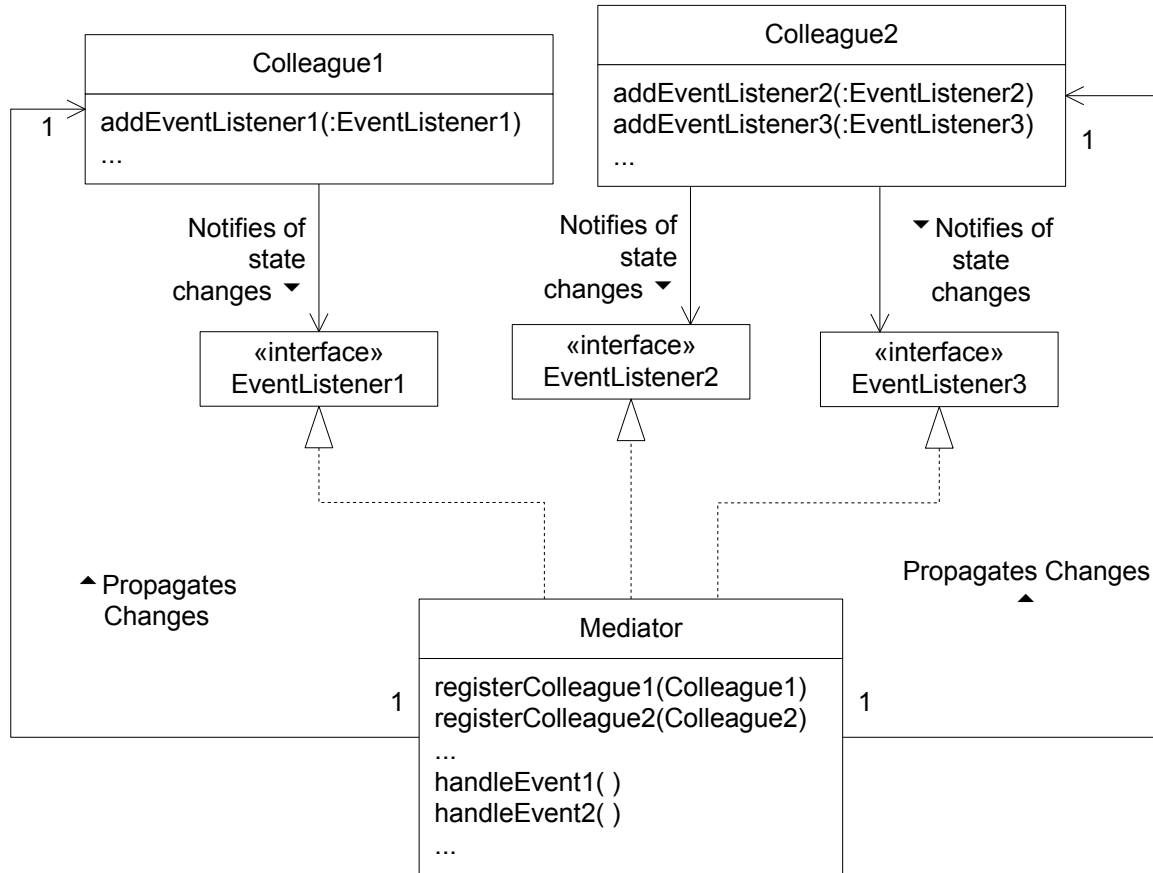
# Behavioural - Interpreter



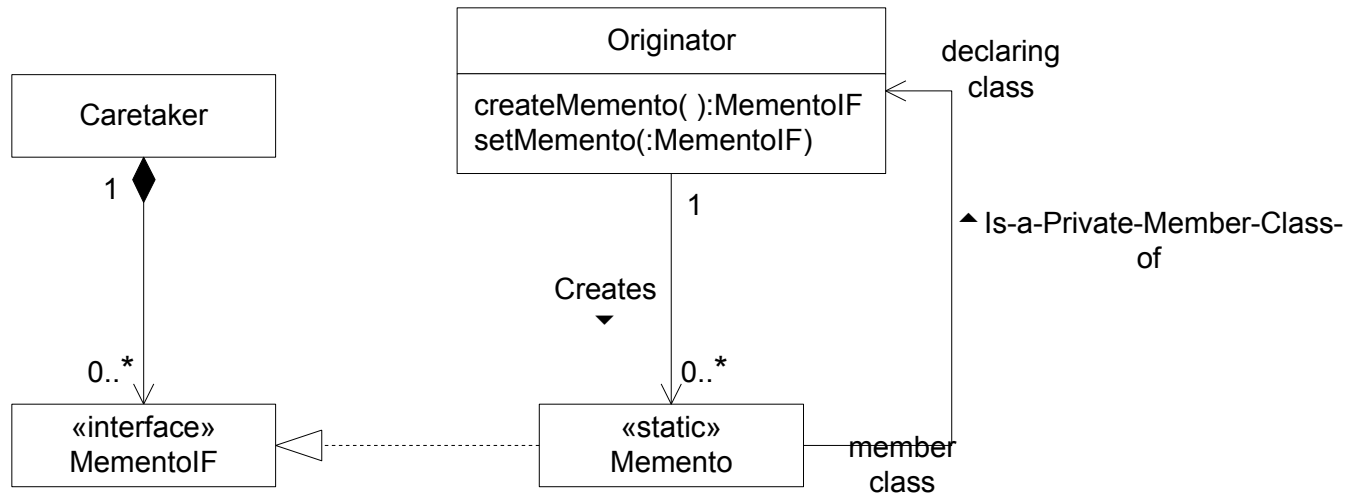
# Behavioural - Iterator



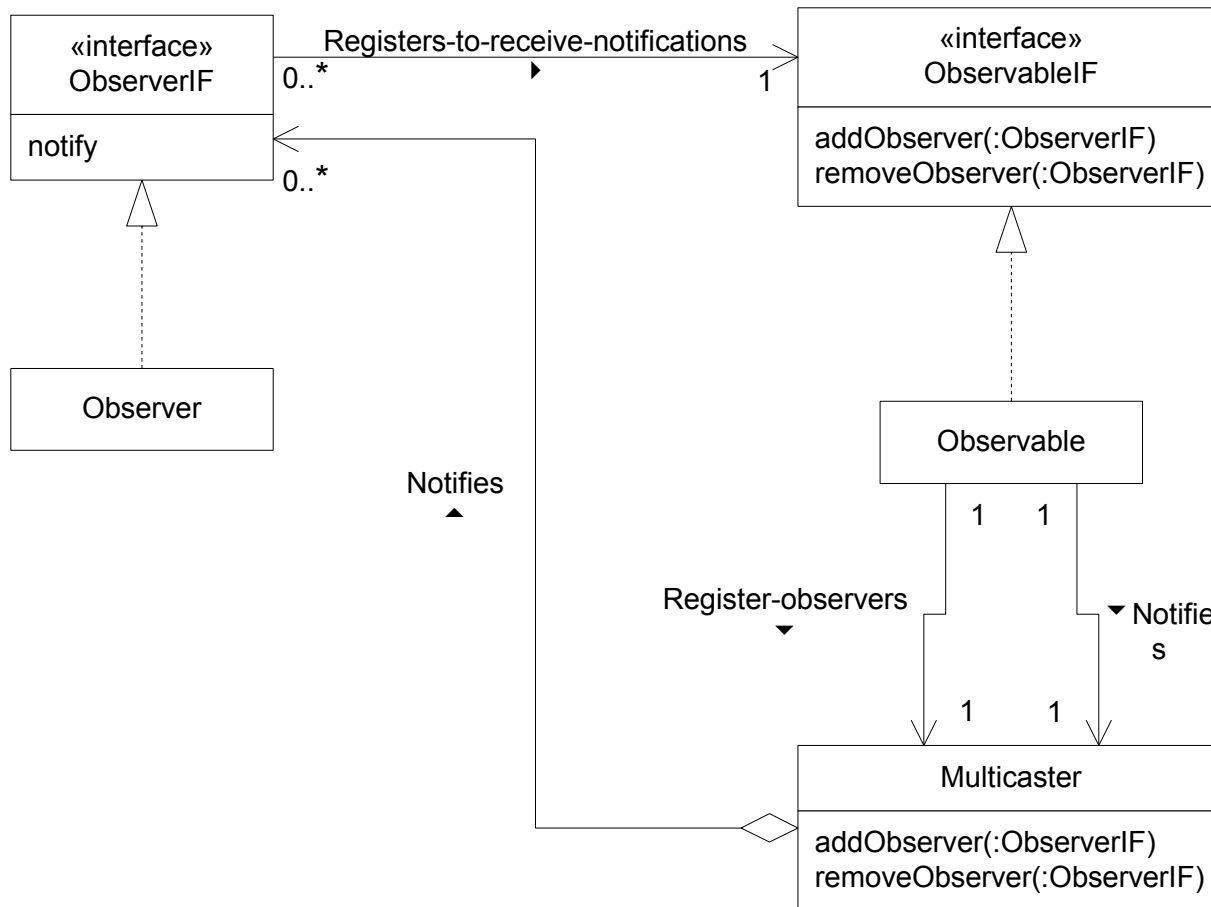
# Behavioural - Mediator



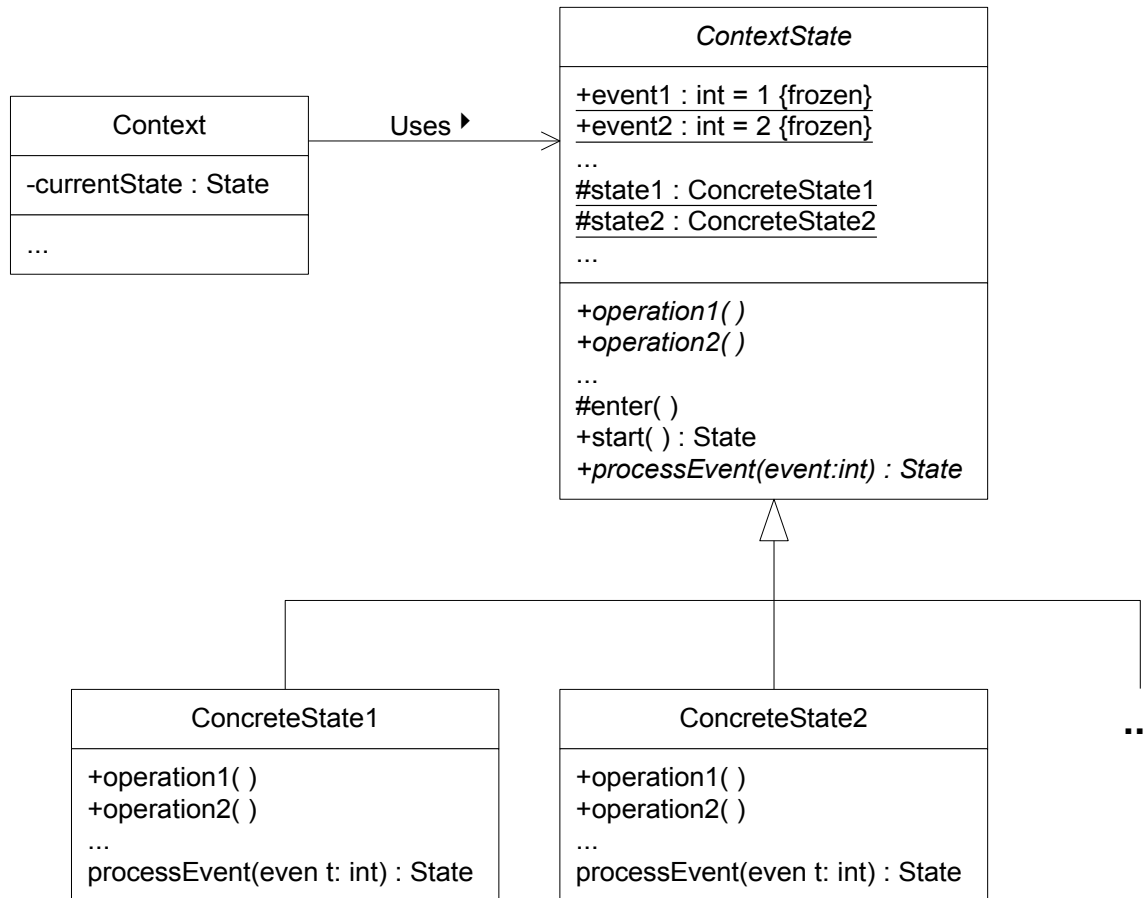
# Behavioural - Memento



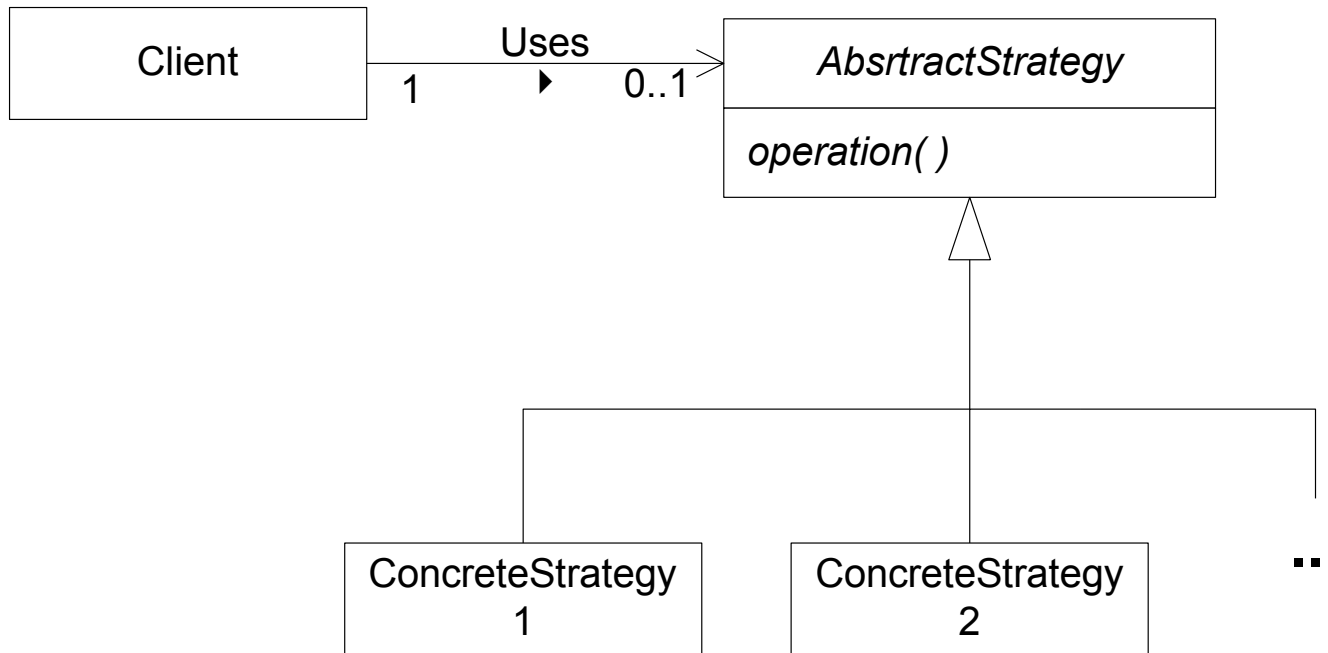
# Behavioural - Observer



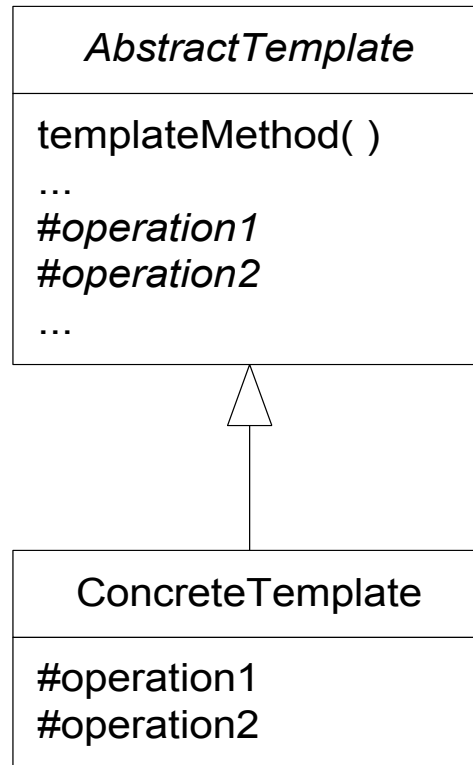
# Behavioural - State



# Behavioural - Strategy



# Behavioural - Template Method



# Behavioural - Visitor

