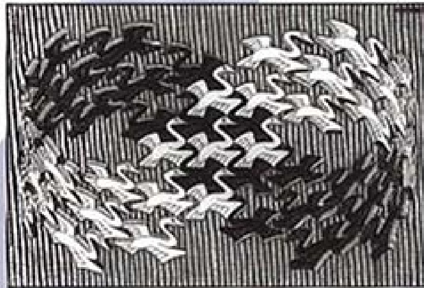
Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Thinking in Patterns with Java
Bruce Eckel
http://www.mindview.net/Books/TIPatterns/

Design Patterns Classification

- Creational Design Patterns
- Structural Design Patterns
- Behavioral Design Patterns

Creational Design Patterns

- Singleton
- Factory Method
- Factory Pattern
- Abstract Factory
- Builder
- Reusable Pool
- Prototype

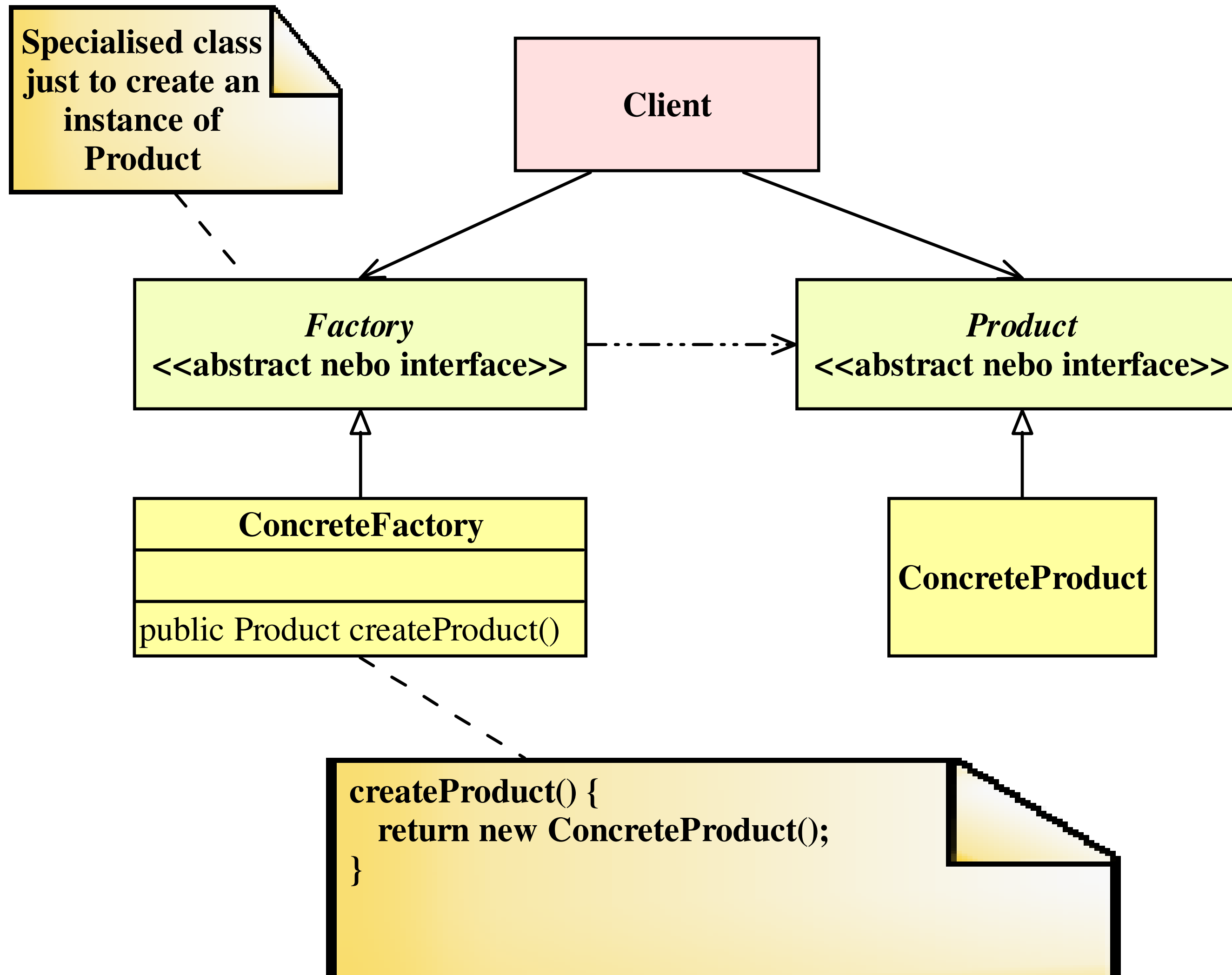
Singleton (creational)

```
Class Singleton {
    private static Singleton instance = null;

    private Singleton() {}; //private constructor

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

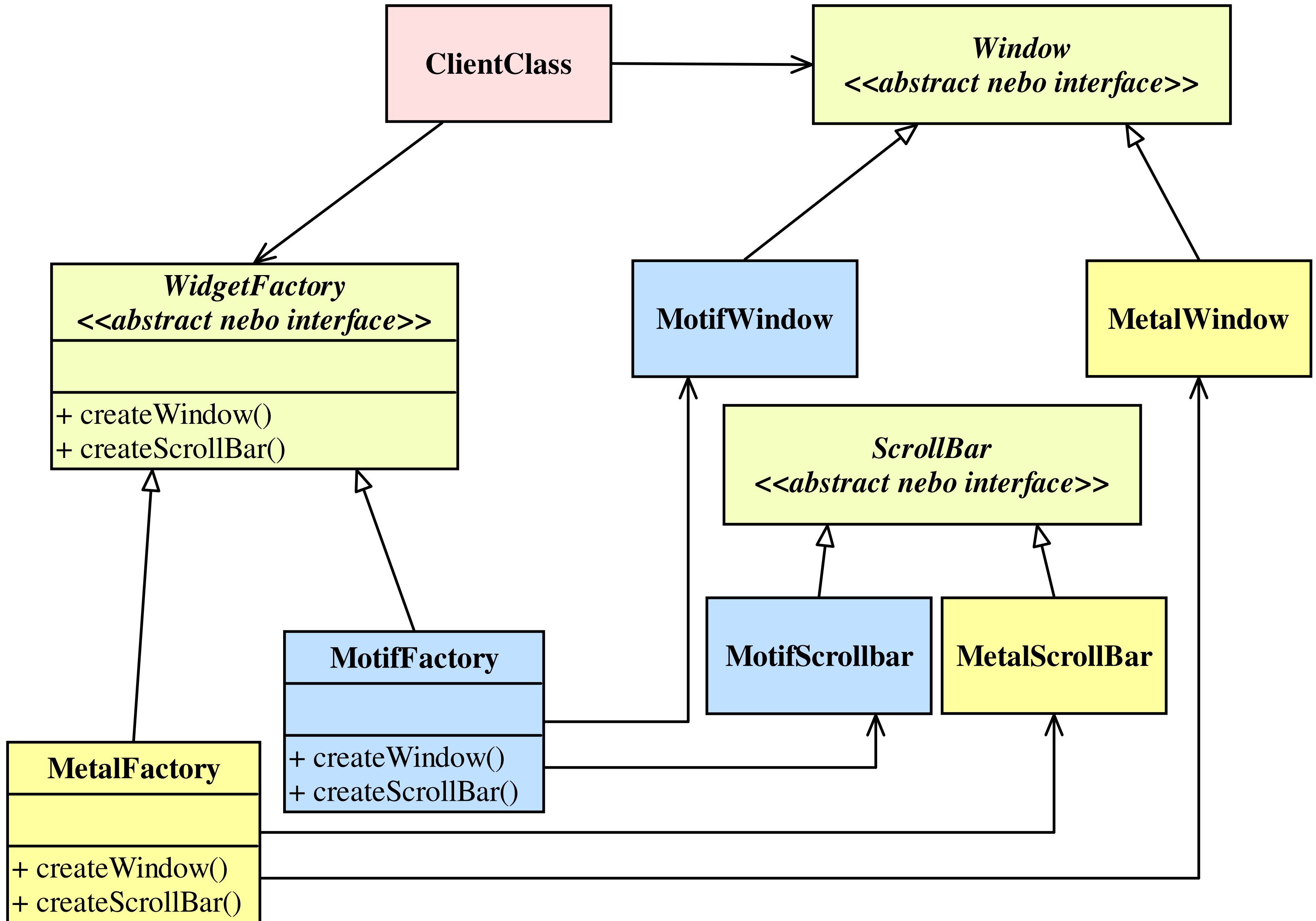
Factory method (creational)

```
Class Singleton {
    private static Singleton instance = null;

    private Singleton() {}; //private constructor

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

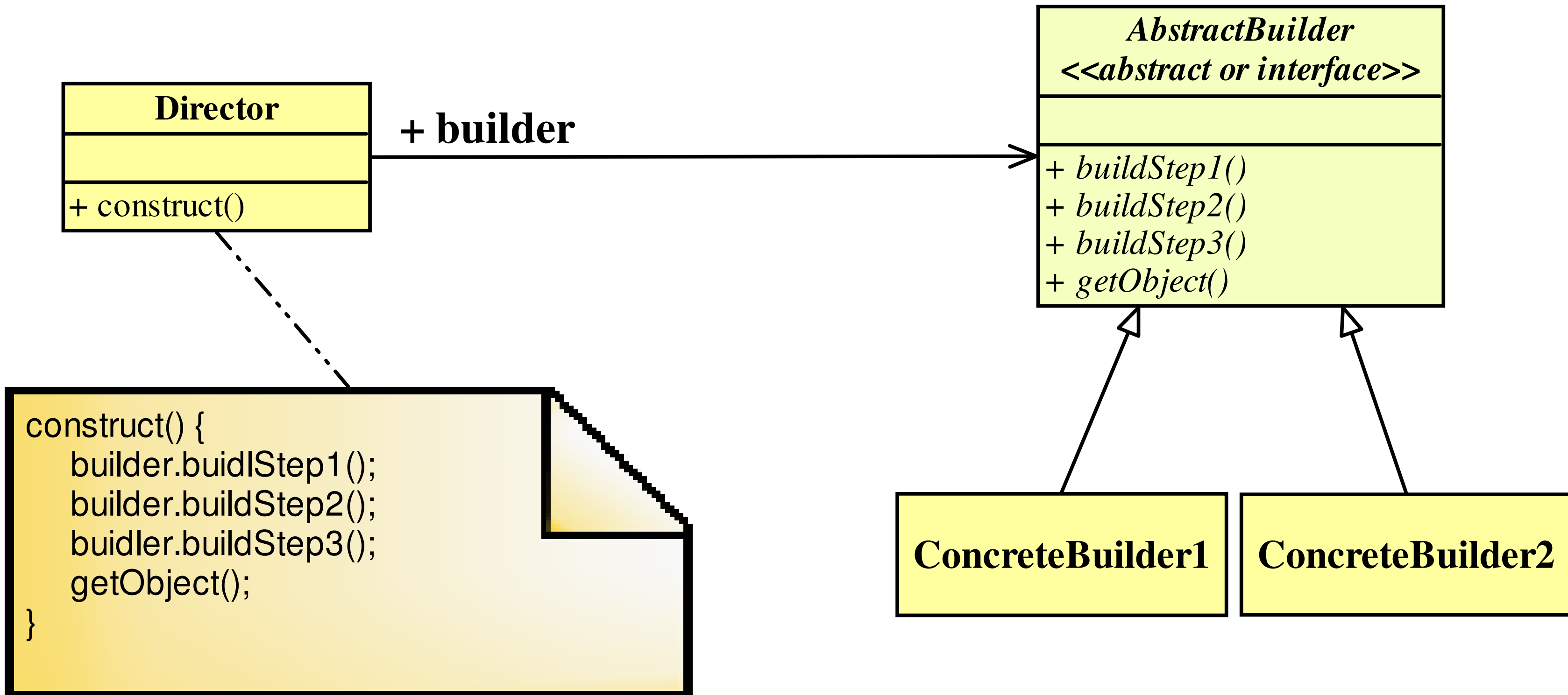
Abstract Factory Pattern (creational)



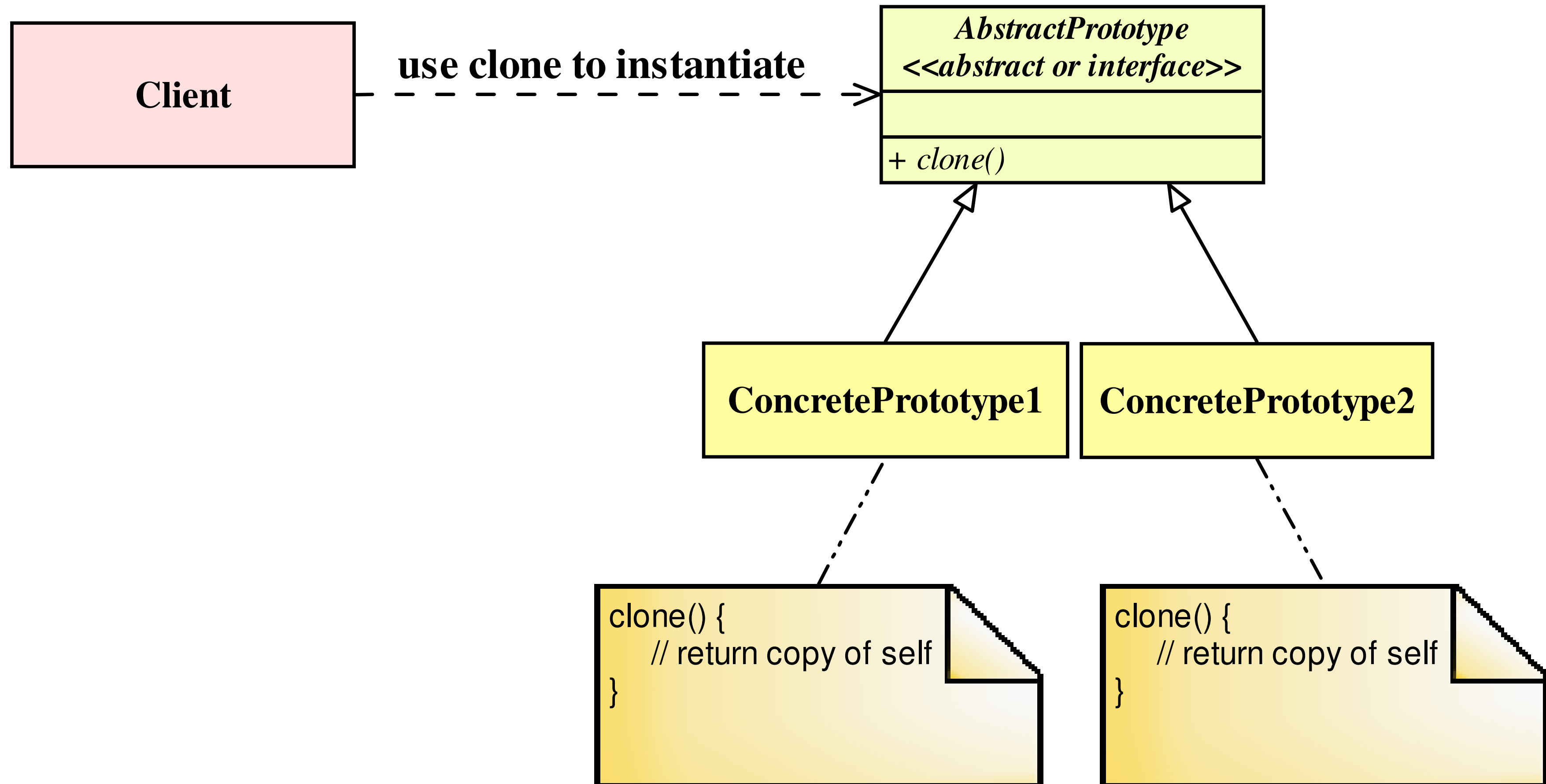
Abstract Factory Pattern (creational)



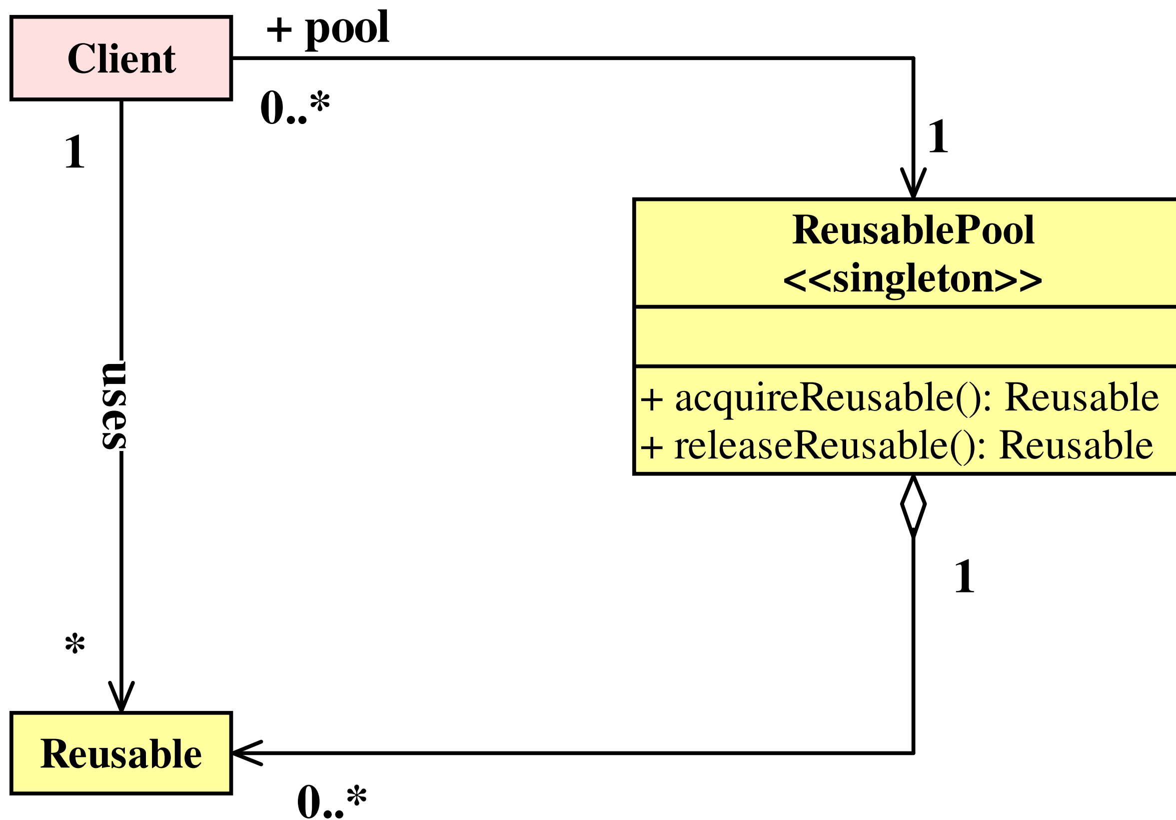
Builder (creational)



Prototype (creational)



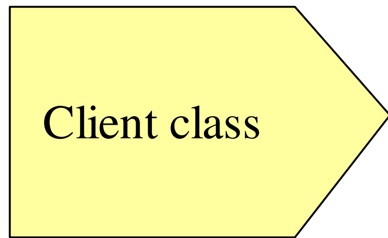
Reusable Pool (creational)

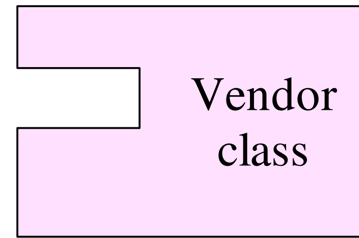


Structural Design Patterns

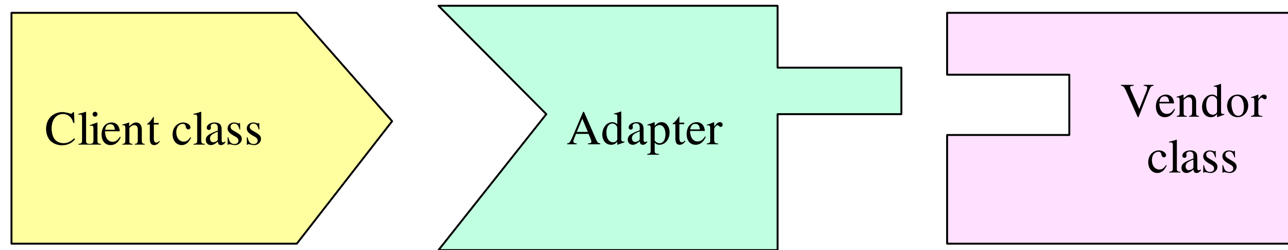
- Adapter
- Bridge (sometimes considered to be a behavioral DP)
- Decorator
- Composite
- Facade
- Proxy
- and more

Adapter (structural)

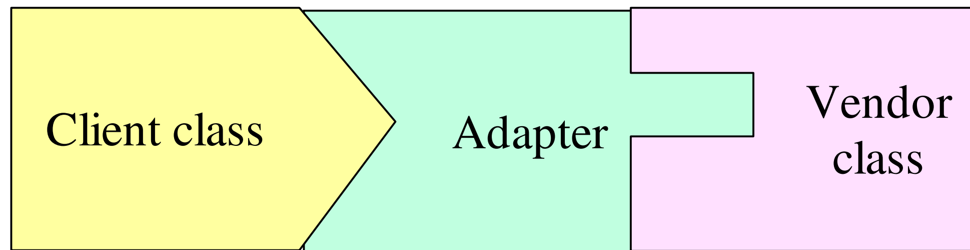




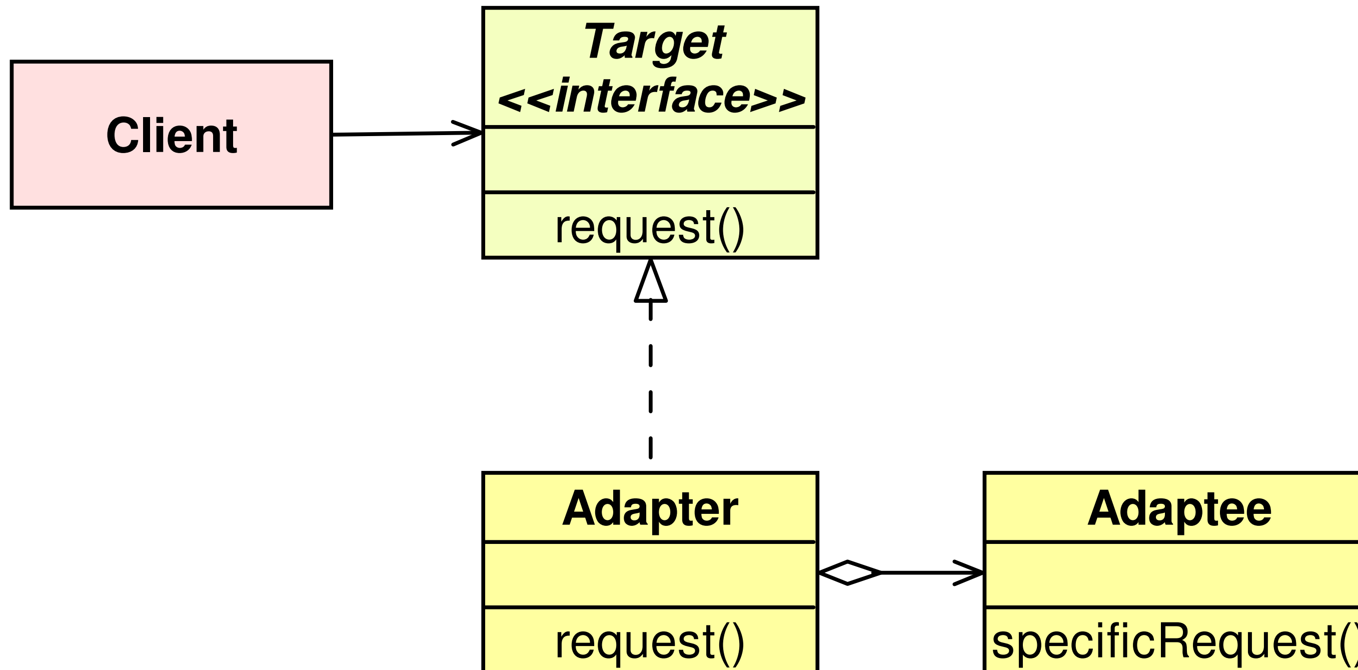
Adapter (structural)



Adapter (structural)

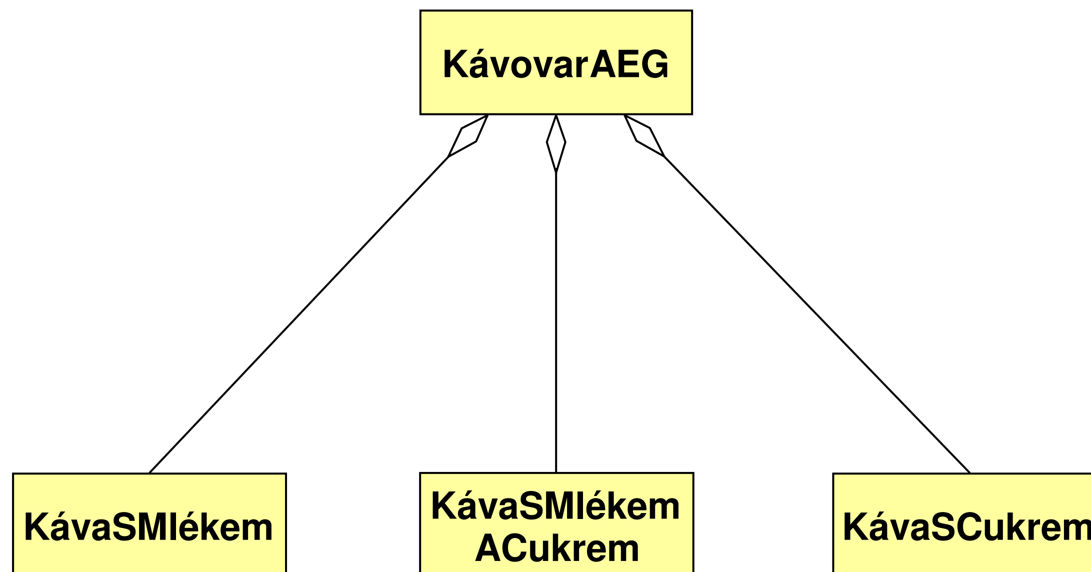


Adapter (structural)



The Dependency Inversion Principle

- High level modules should not depend upon low level modules; both should depend on abstractions.
- Abstractions should not depend upon details; details should depend upon abstractions.



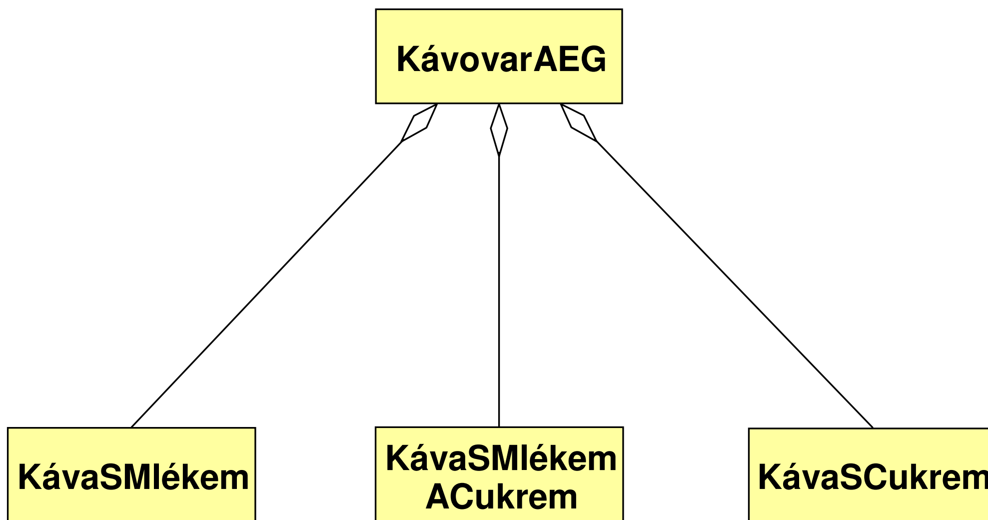
Dependency inversion principle

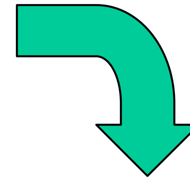
specific form of decoupling where conventional dependency relationships established from high-level are inverted (e.g. reversed) for the purpose of rendering high-level modules independent of the low-level module implementation details. The principle states:

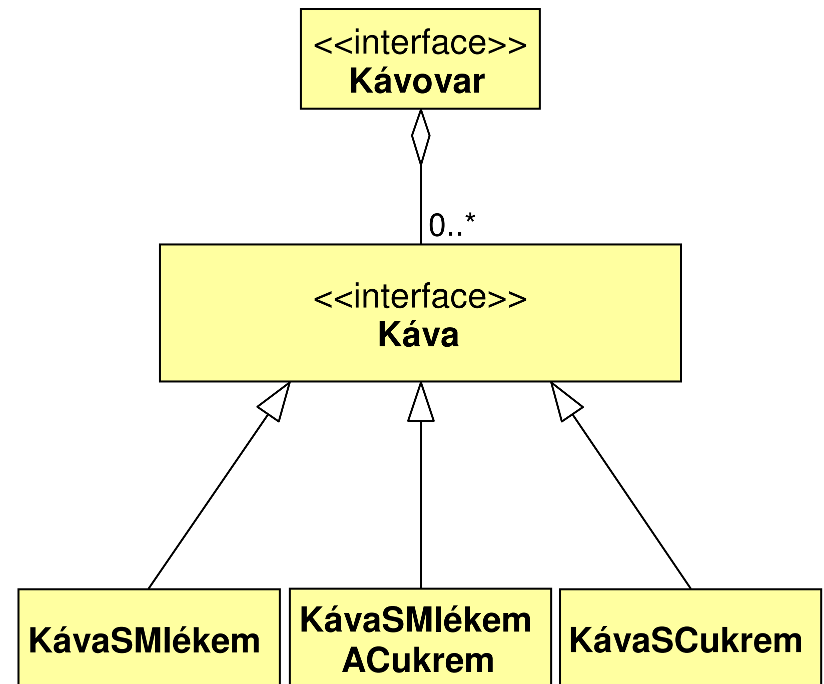
- *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
- *Abstractions should not depend upon details. Details should depend upon abstractions.*

(From Wikipedia, the free encyclopedia)

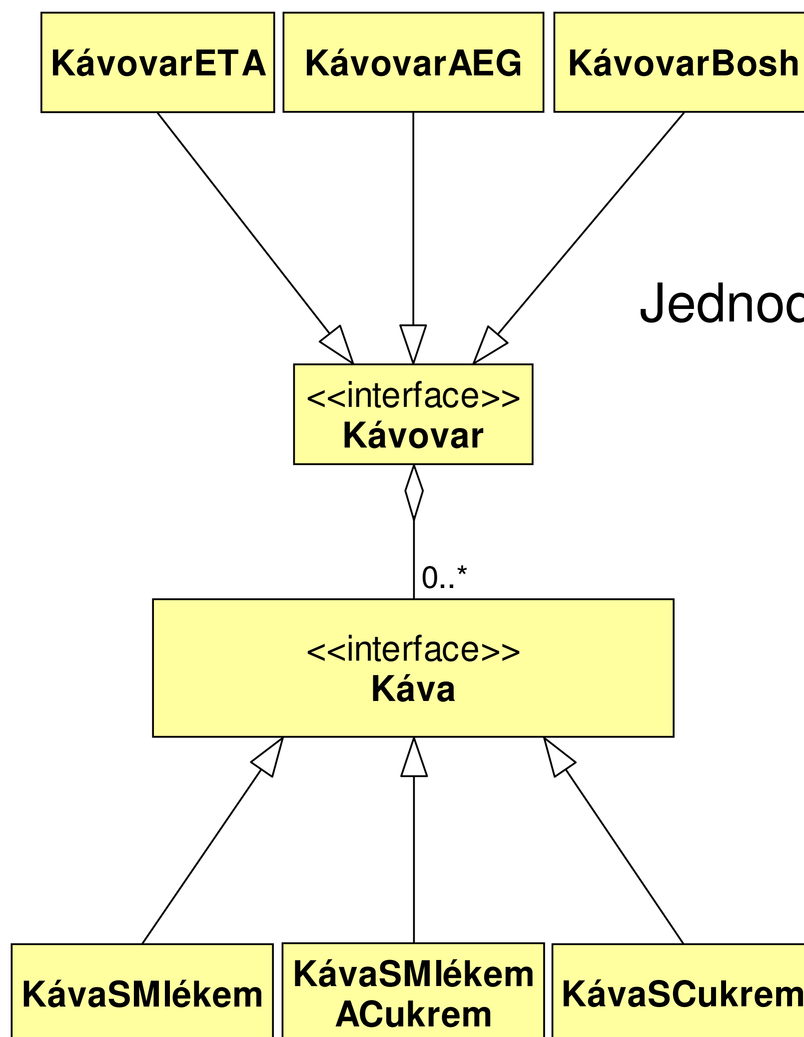
The Dependency Inversion Principle







The Dependency Inversion Principle

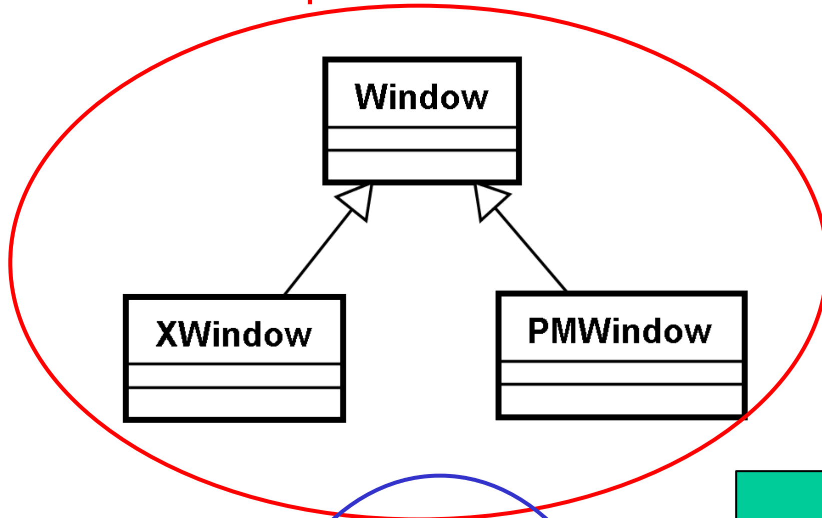


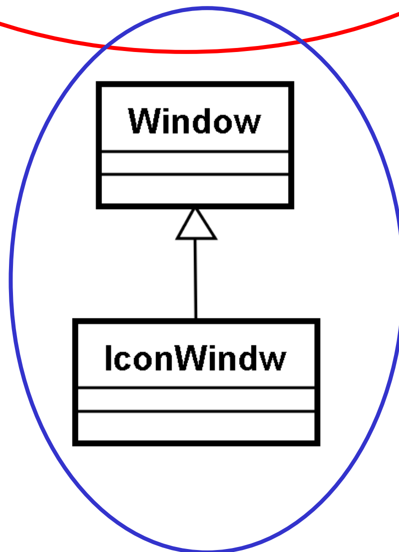
Jednoduché rozšíření předešlého diagramu

- design Pattern Bridge

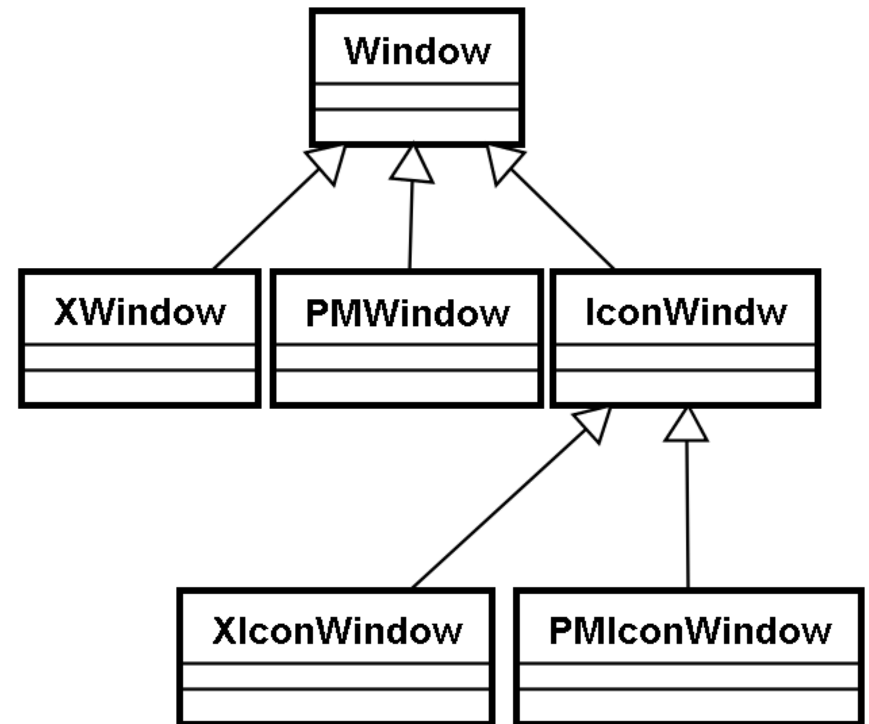
Bridge

Implementation



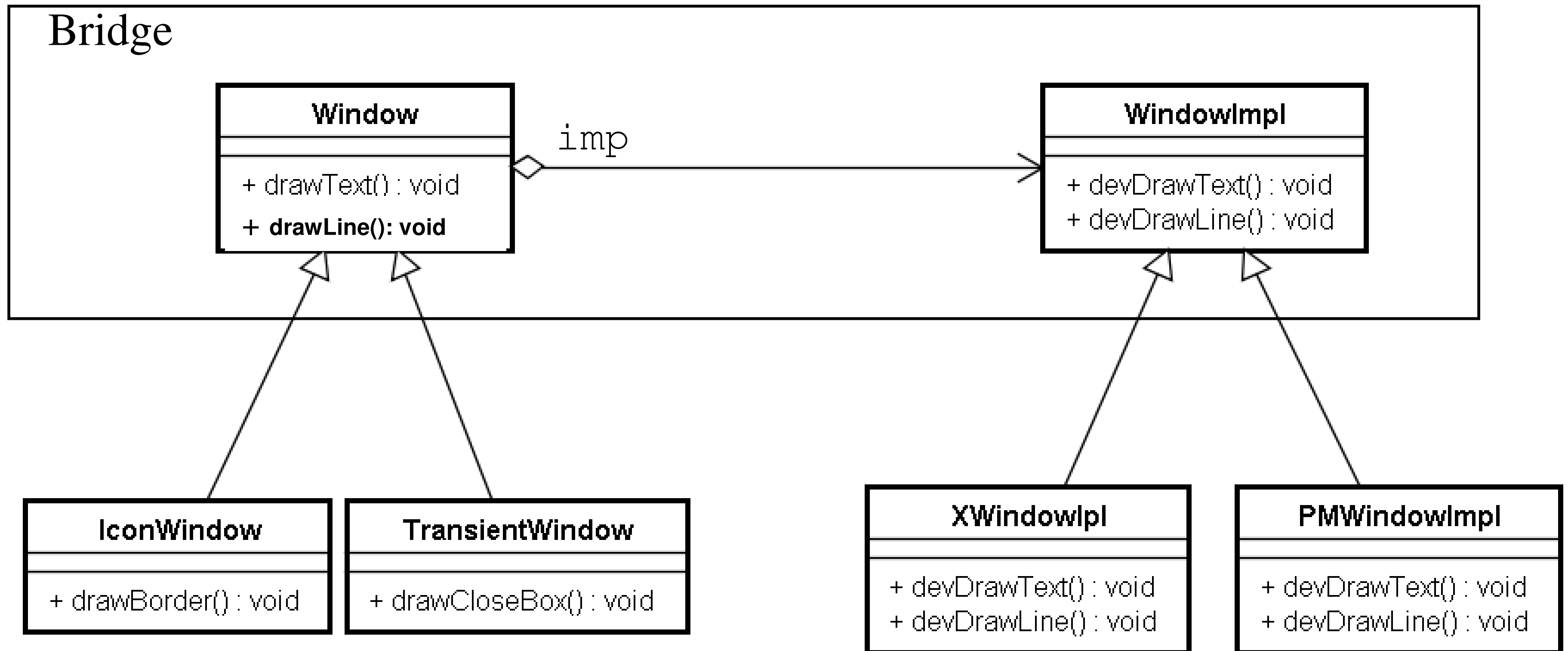


Abstraction refinement

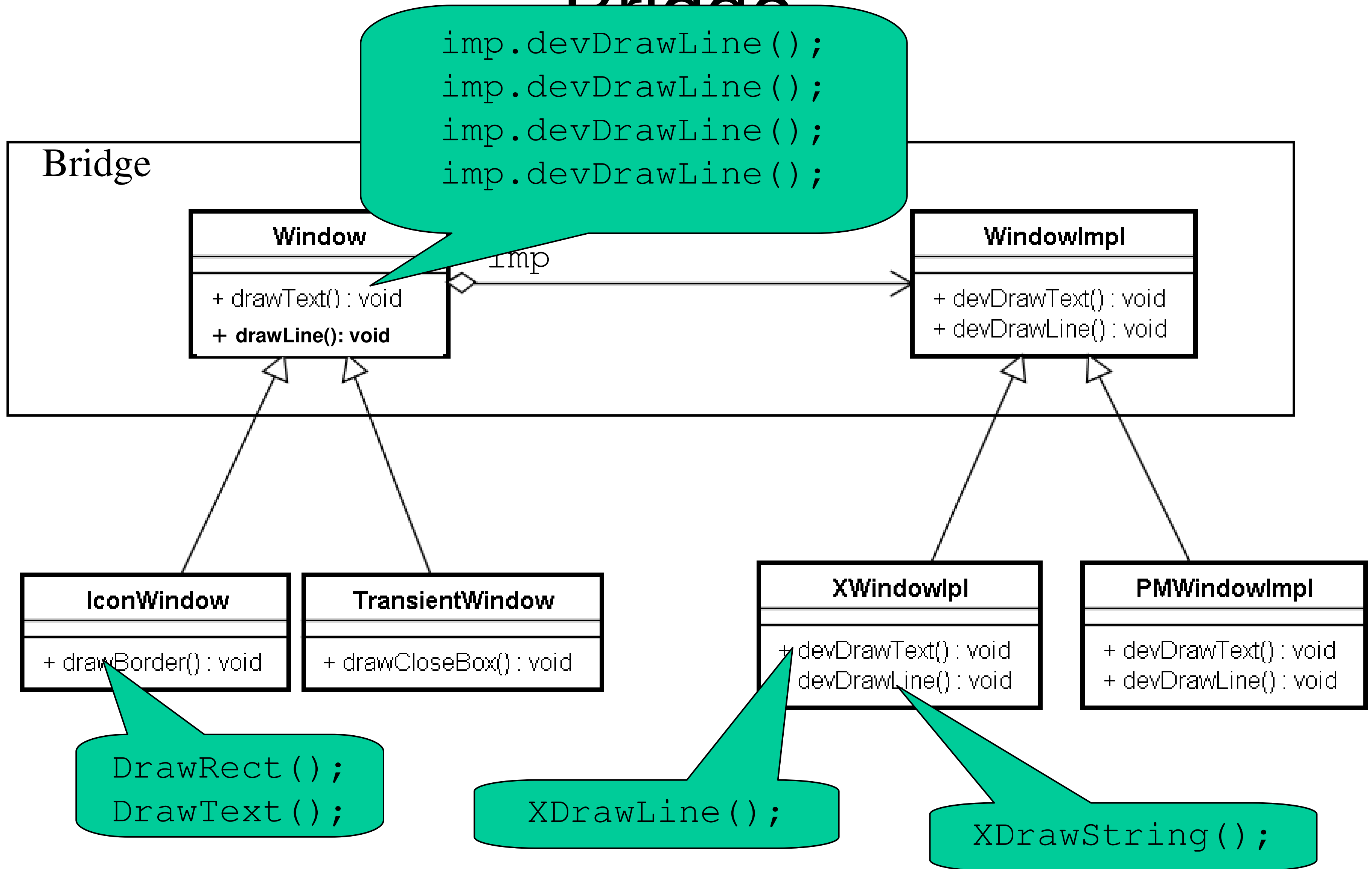


- Máme více hledisek klasifikace.
- Pokud bychom je chtěli zohlednit v jedné inheritanční hierarchii, došlo by k explozi počtu tříd (pro různé kombinace).
- Řešení – oddělit klasifikační hlediska.

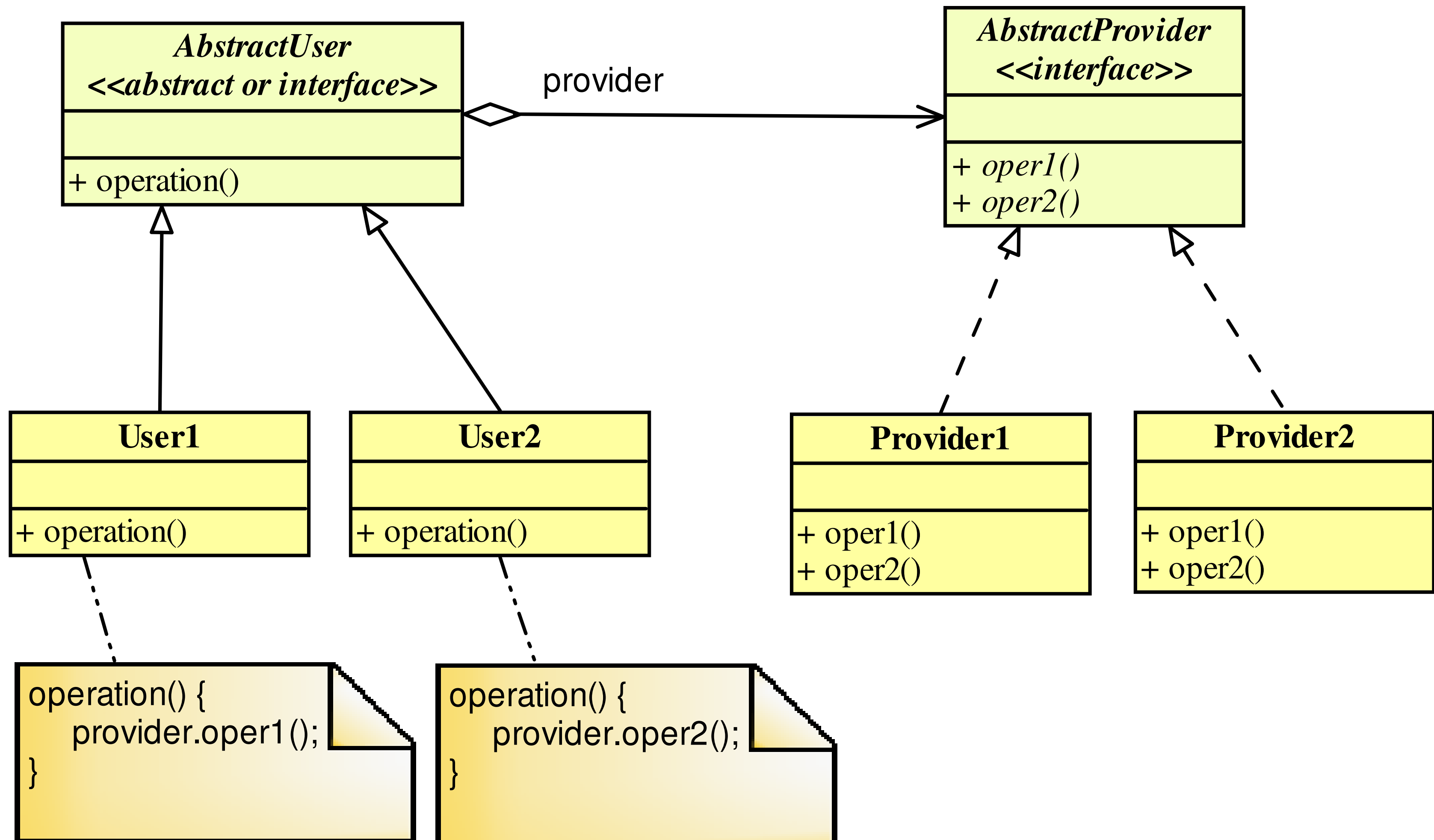
Bridge



Bridge



Bridge (structural)



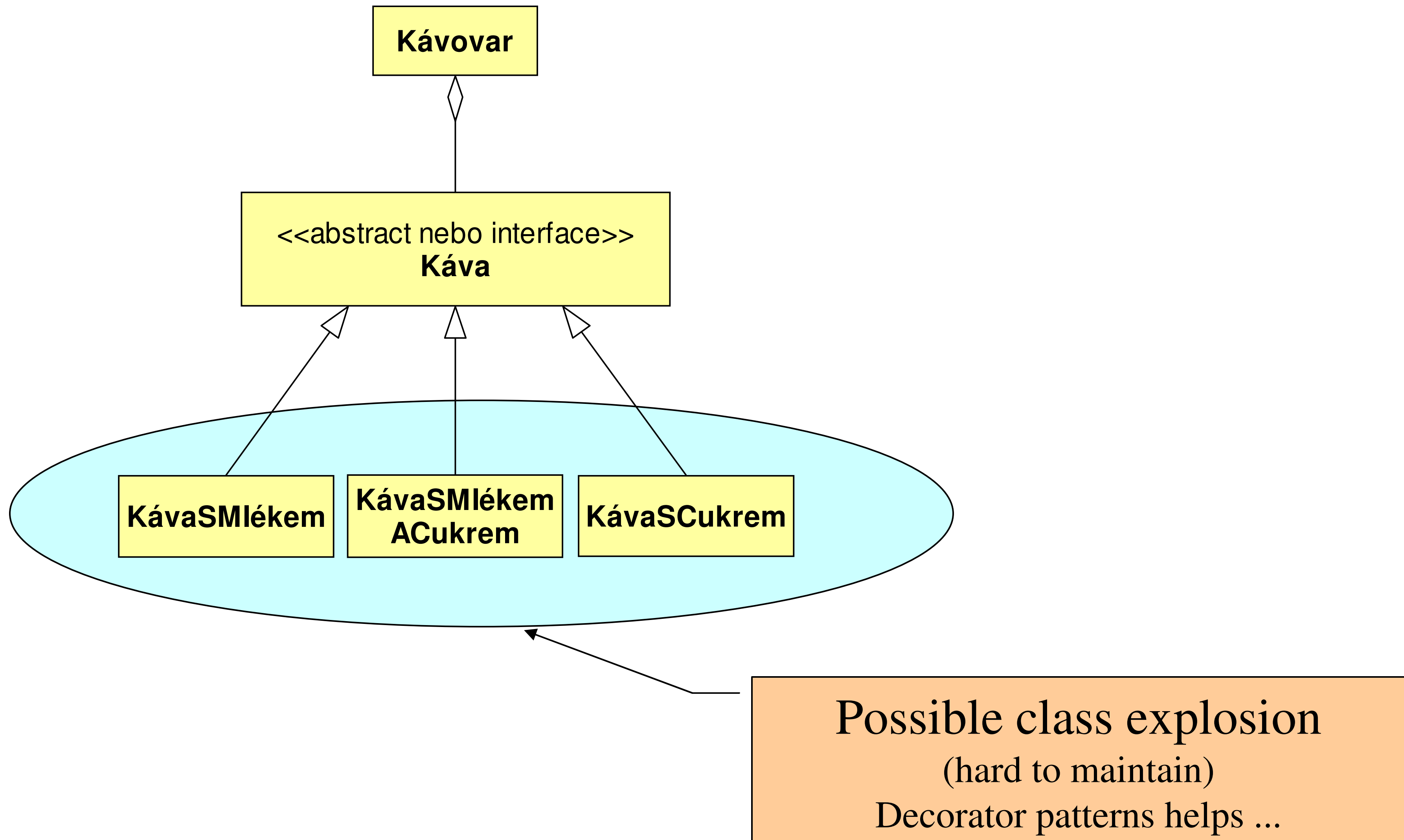
Bridge (structural)

1. The Bridge pattern is intended to keep the interface to your client class constant while allowing you to change the actual kind of class you use.

2. You can extend the implementation class and the bridge class separately, and usually without much interaction with each other (avoiding permanent binding between abstraction and its implementation).

3. You can hide implementation details from the client class much more easily.

Decorator



The Open-Close Design Principle

Software entities like classes, modules and functions should be open for extension but closed for modifications.

Adding new functionality should involve minimal changes to existing code.

=> adding new functionality will not break the old functionality

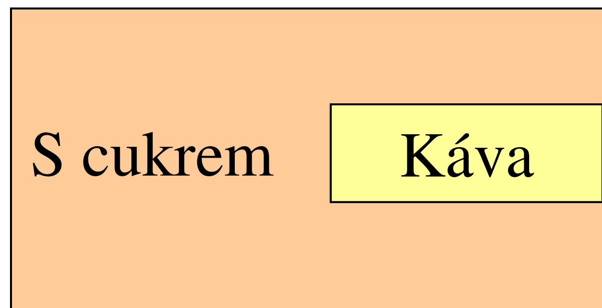
=> old functionality need not be re-tested

Most changes will be handled as new methods and new classes.
Designs following this principle would result in resilient code which does not break on addition of new functionality.

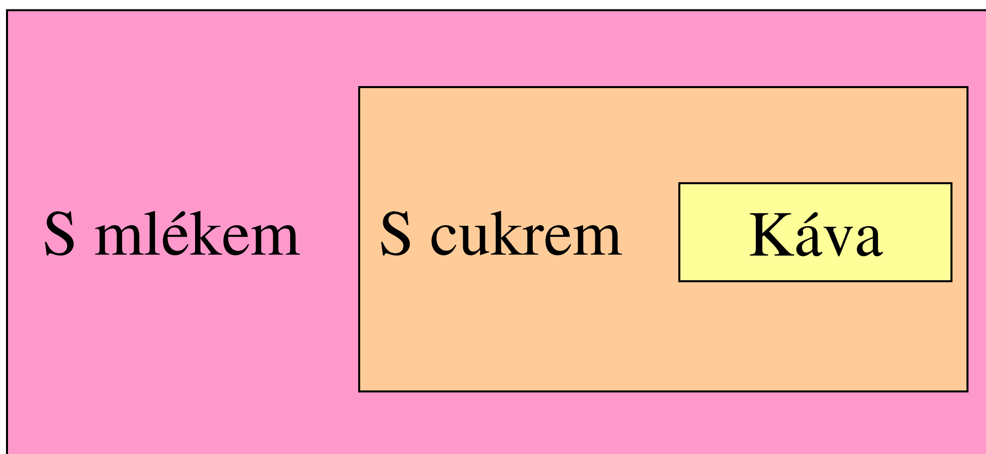
Decorator (structural)

Káva

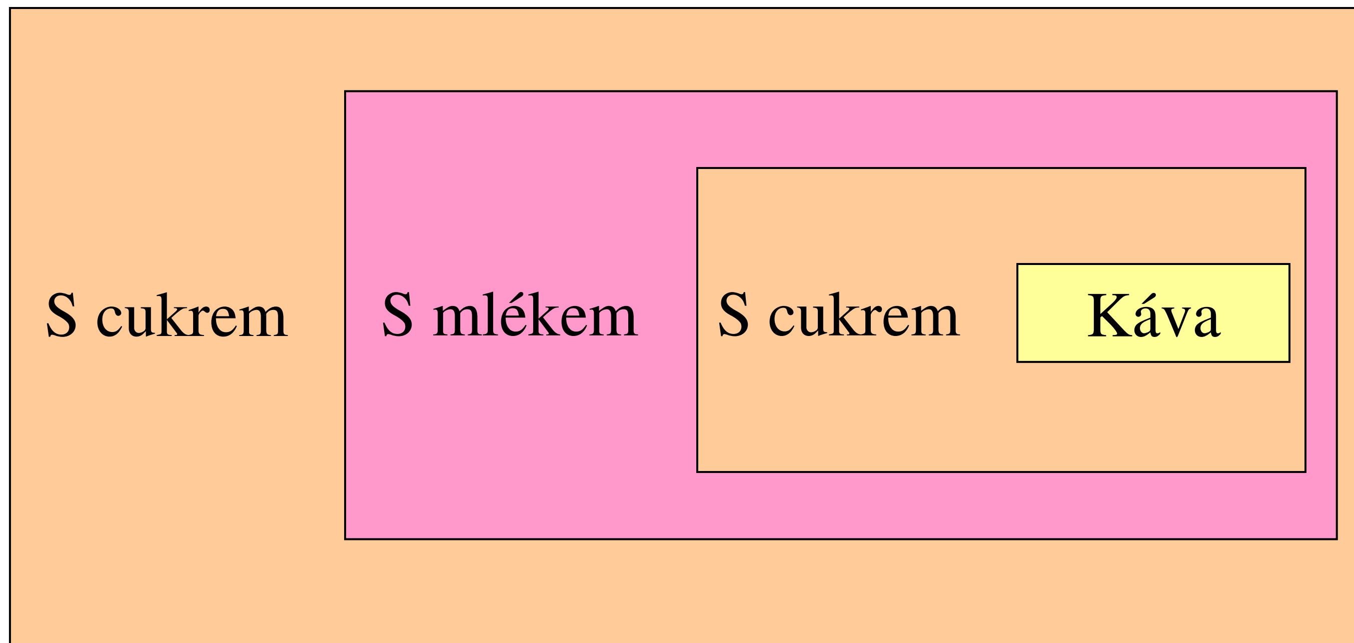
Decorator (structural)



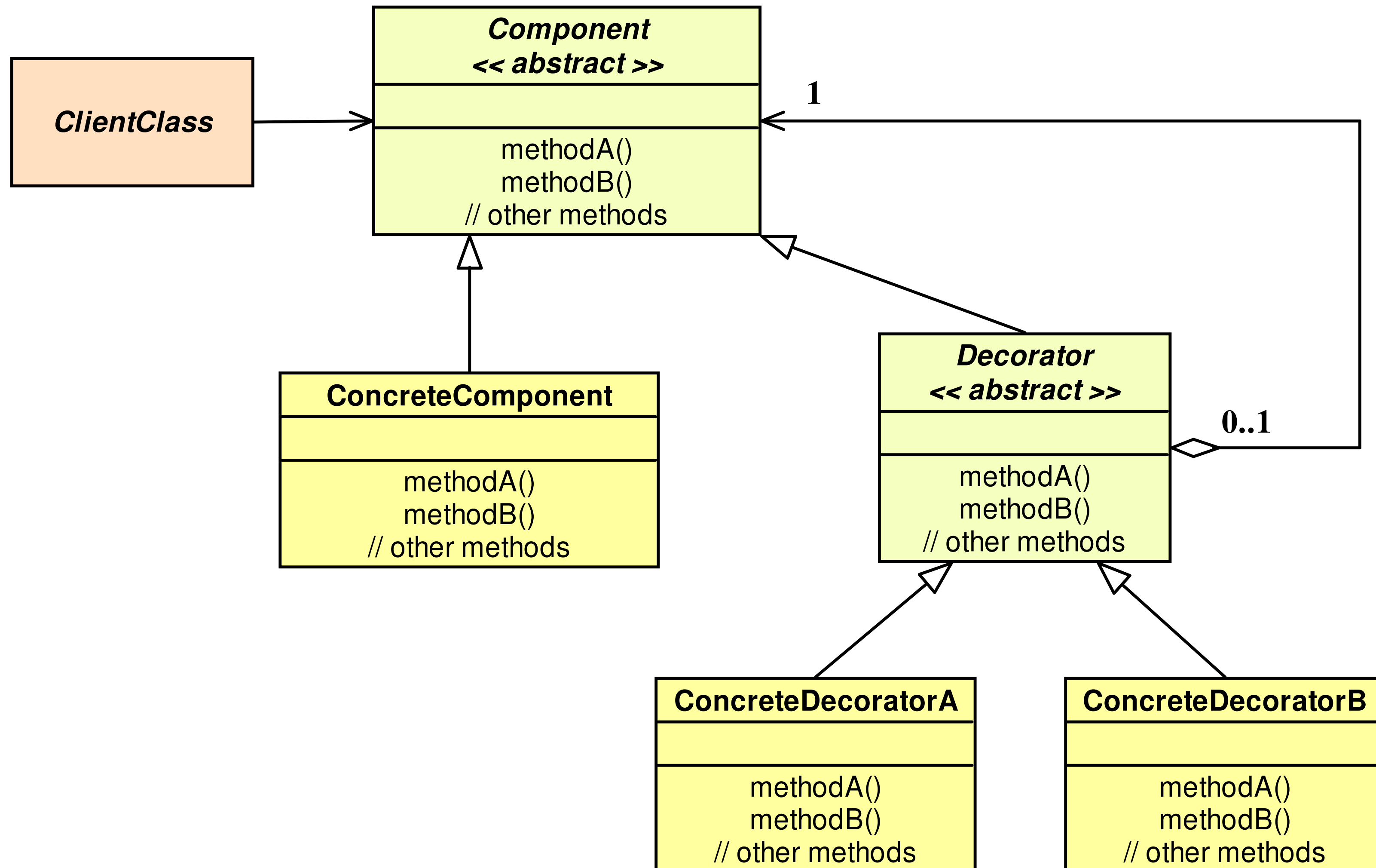
Decorator (structural)



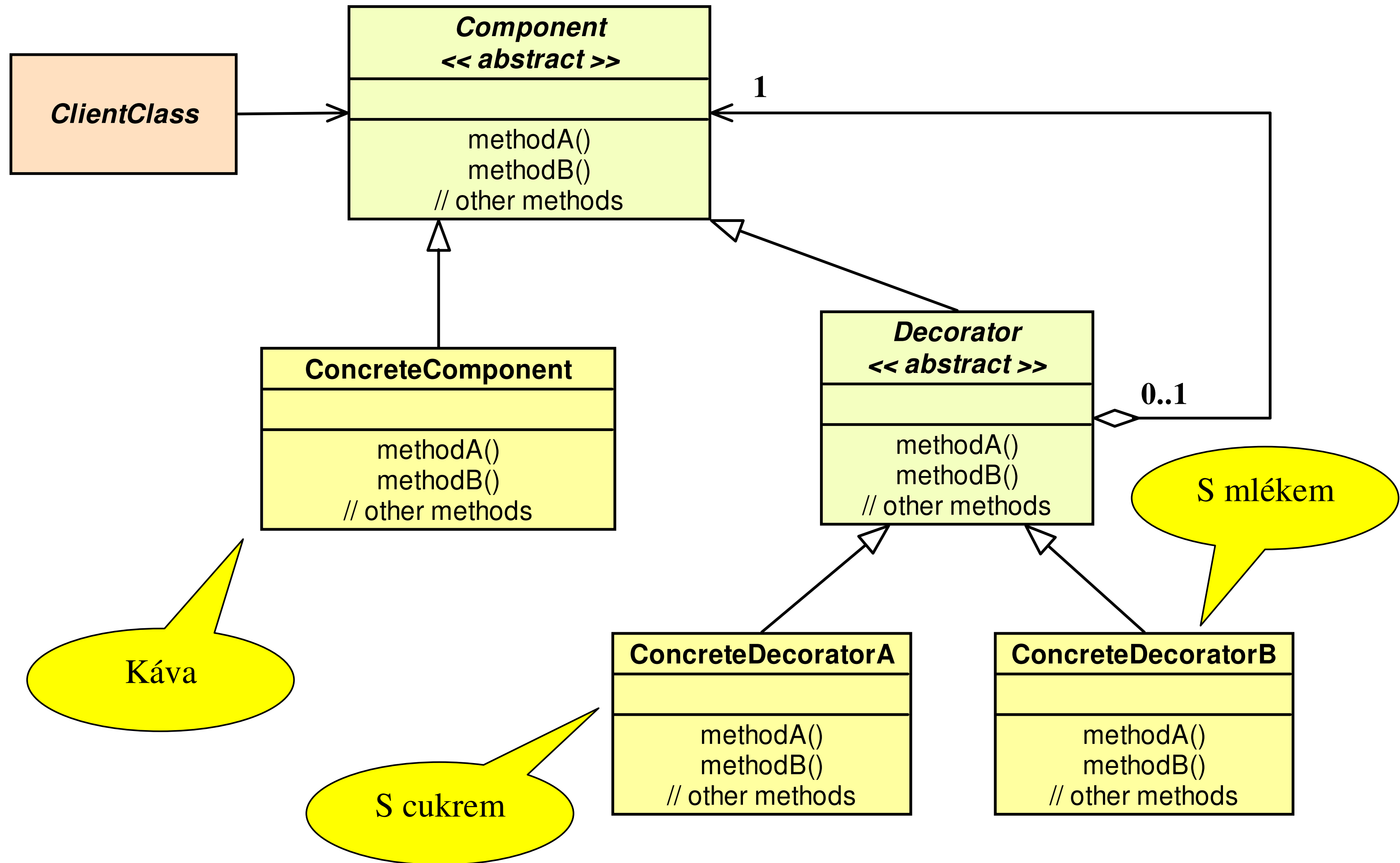
Decorator (structural)



Decorator (structural)



Decorator (structural)



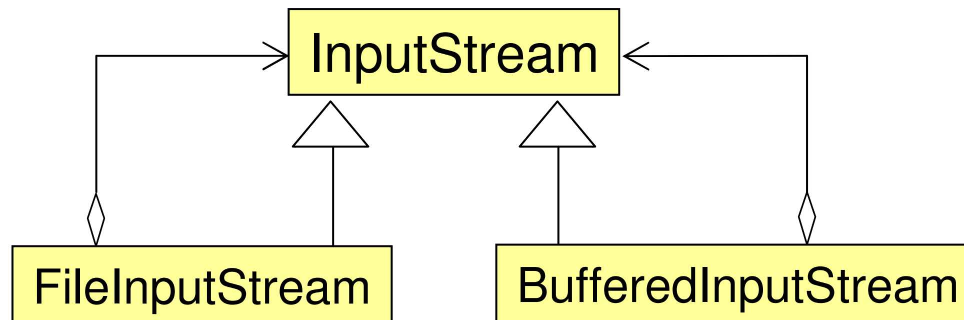
Decorator (structural)

```
Kava kava = new Kava();

Kava kavaSMlekem = new SMlekem(kava);

Kava kavaSCukremSMlekem = new SCukrem(kavaSMlekem);
```

```
BufferedInputStream bis = new BufferedInputStream(new FileInputStream("fff"));
```

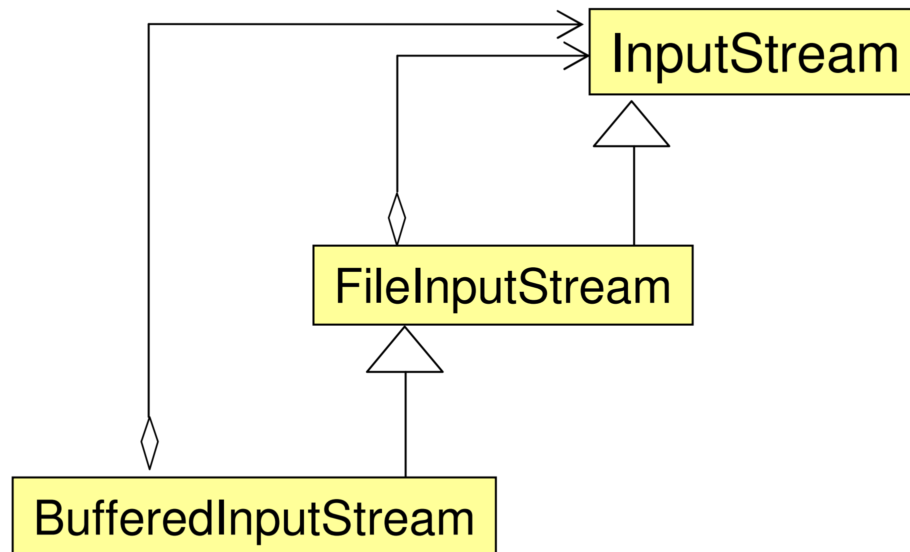
Decorator (structural)

```
BufferedInputStream bis = new BufferedInputStream(new FileInputStream("fff"));
```



Decorator (structural)

```
BufferedInputStream bis = new BufferedInputStream(new FileInputStream("fff"));
```



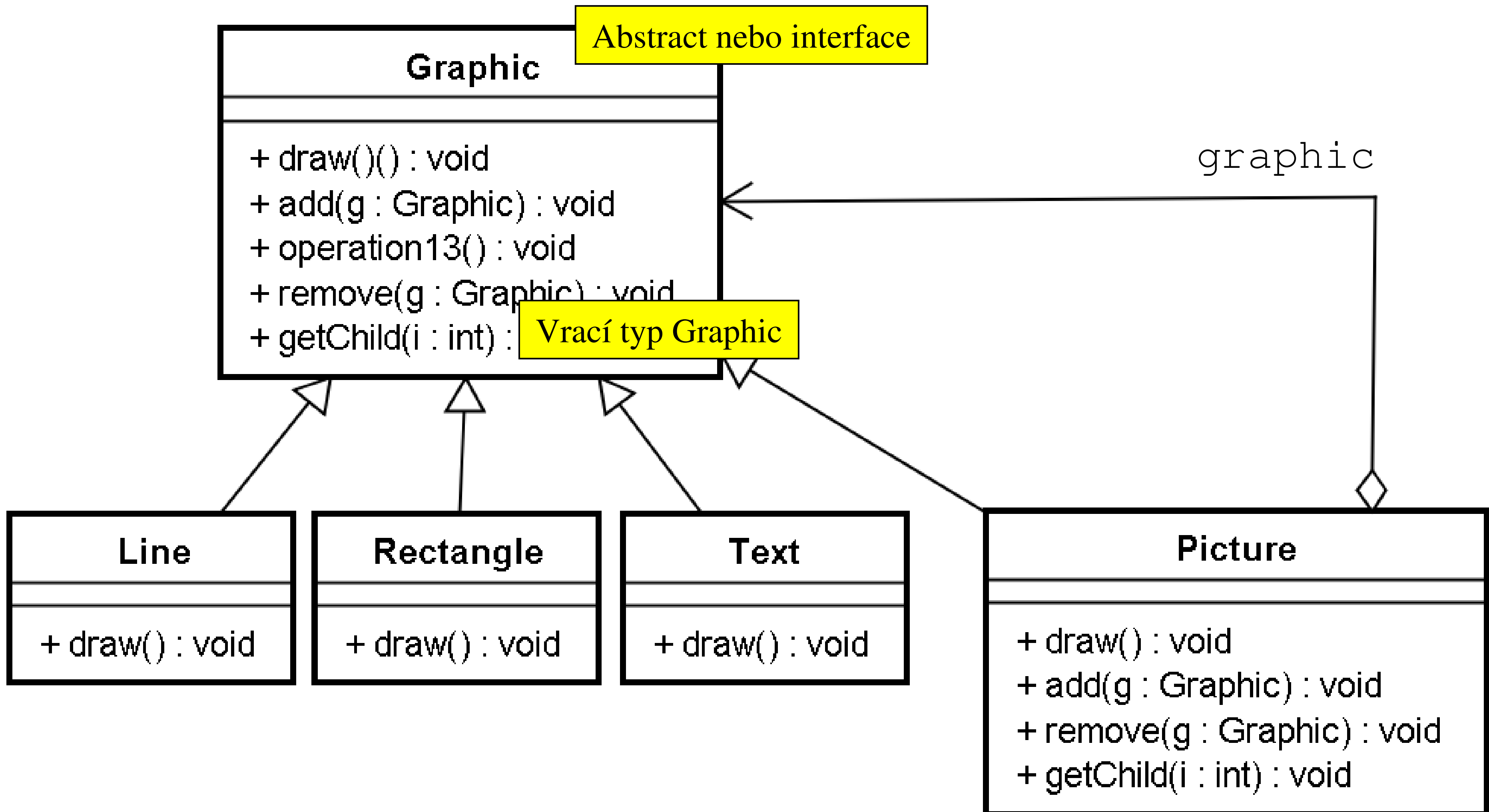
Decorator (structural)

The Decorator Pattern attaches additional functionalities to an object dynamically.

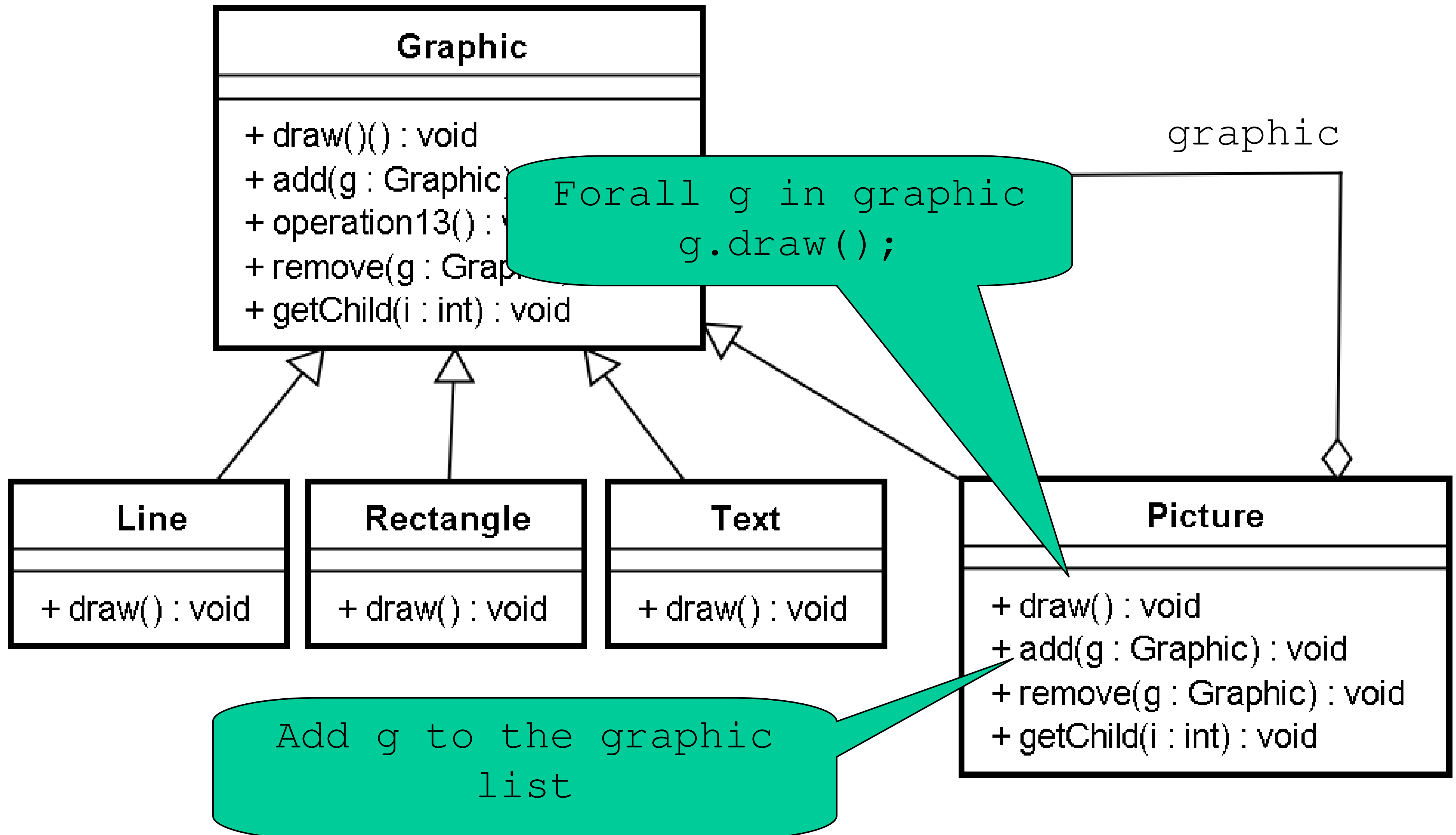
Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator prevents explosion of (sub)classes.

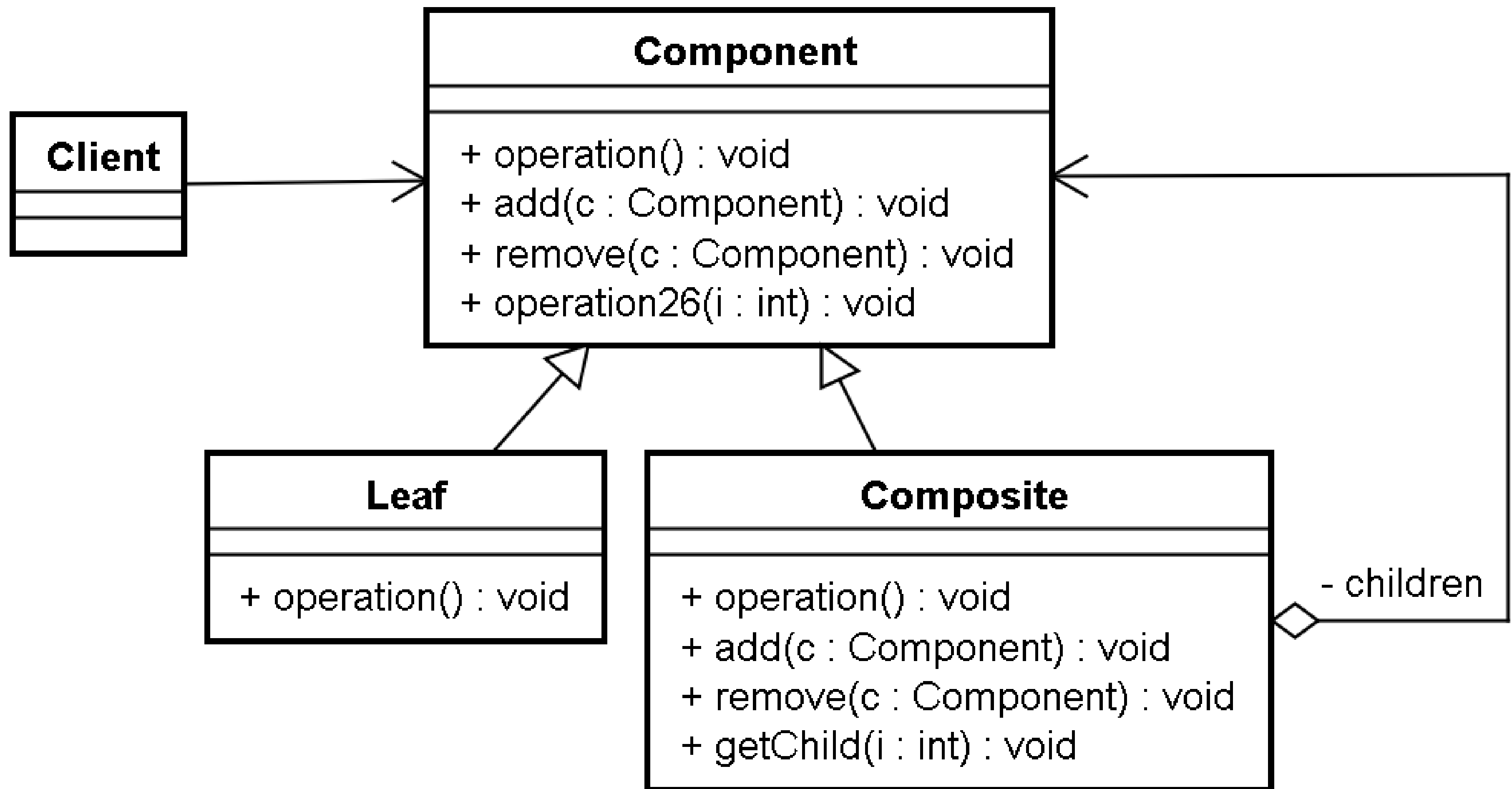
Composite (structural)



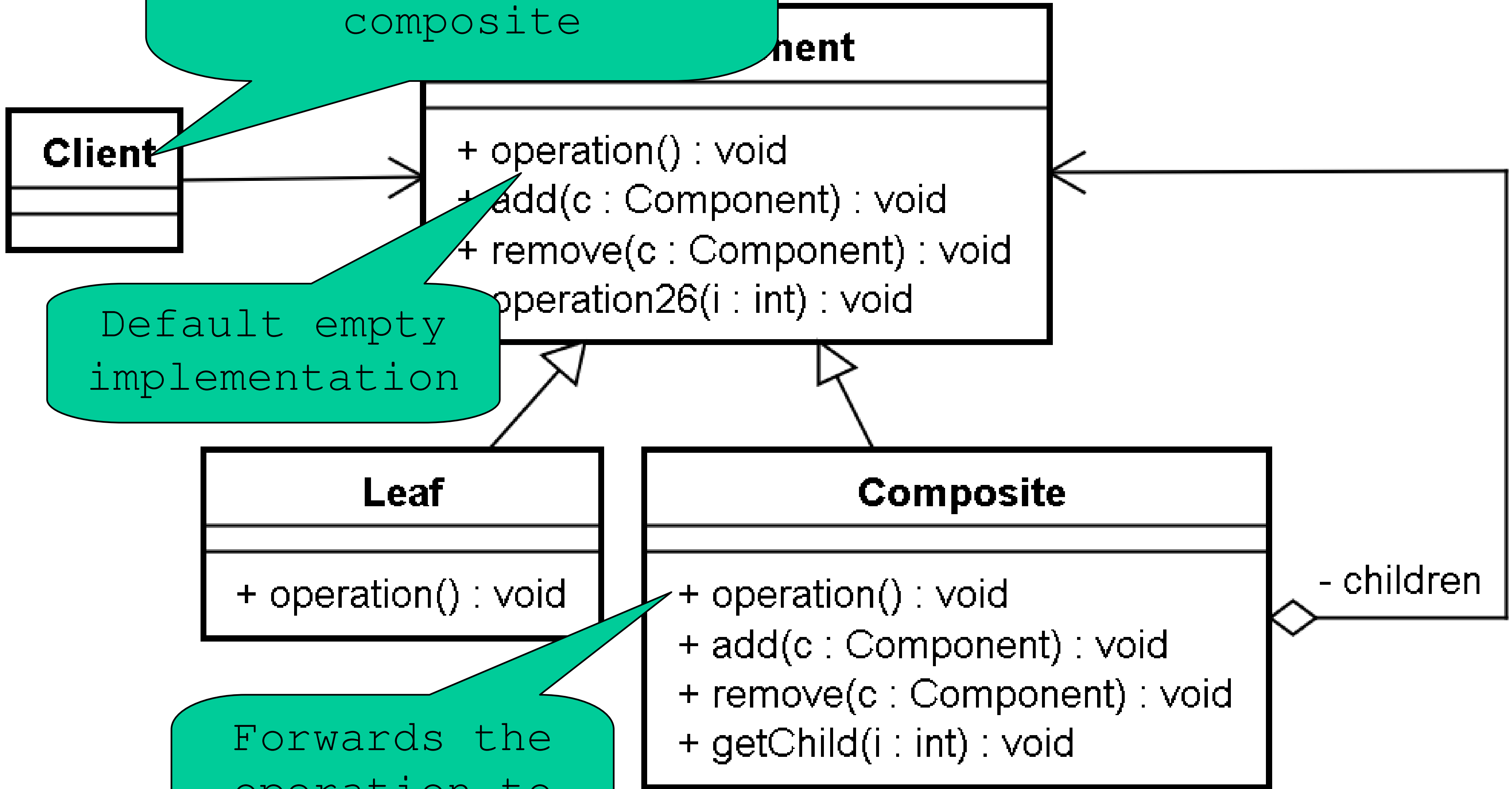
Composite (structural)



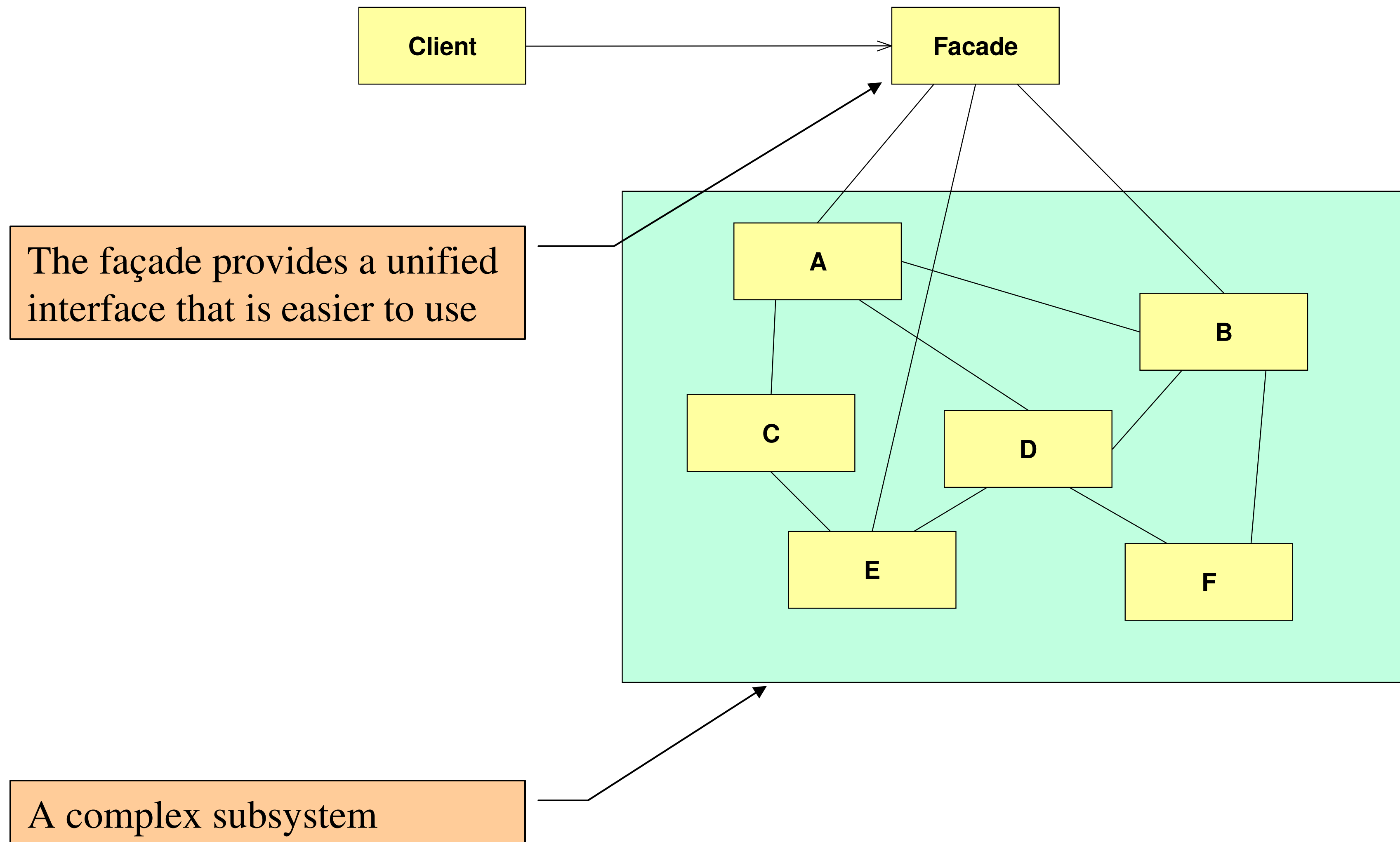
Composite (structural)



te (structural)



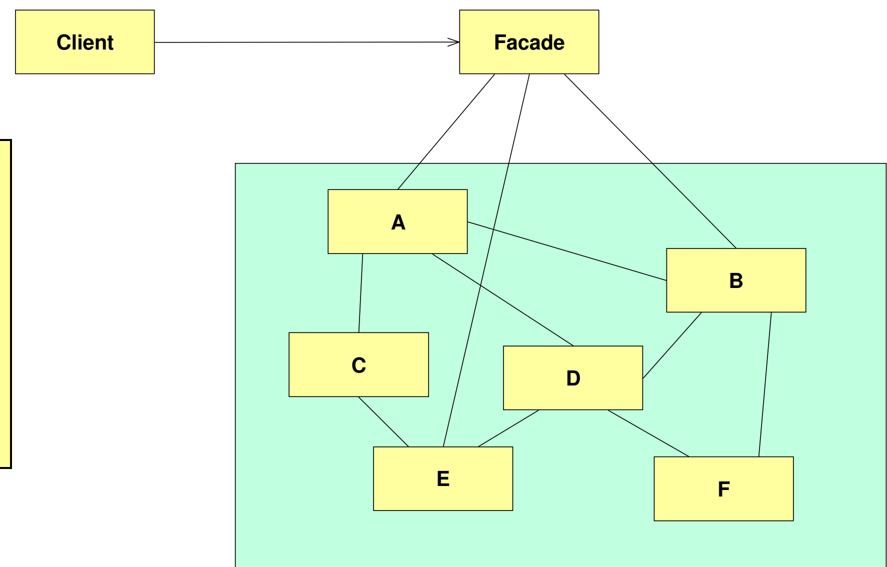
Façade (structural)



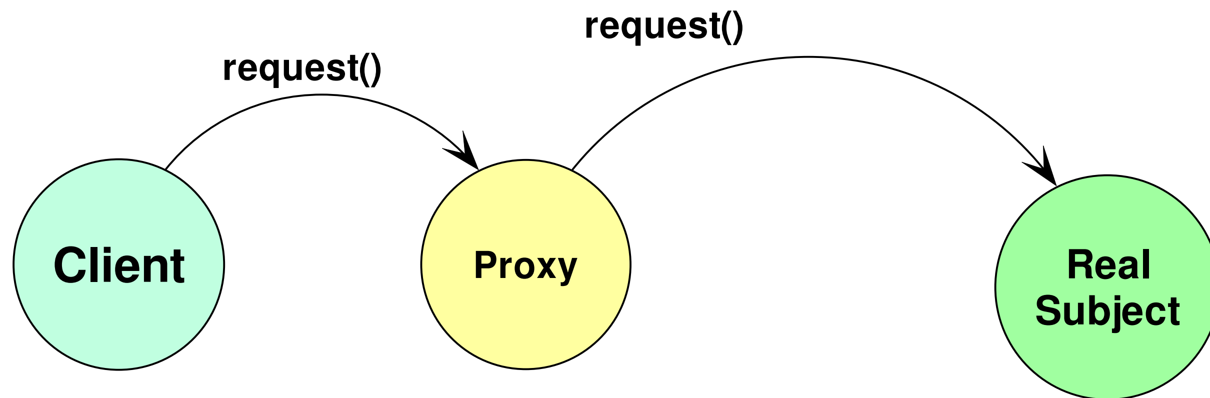
Façade (structural)

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

Design Principle
Principle of least knowledge - talk only to your immediate friends.

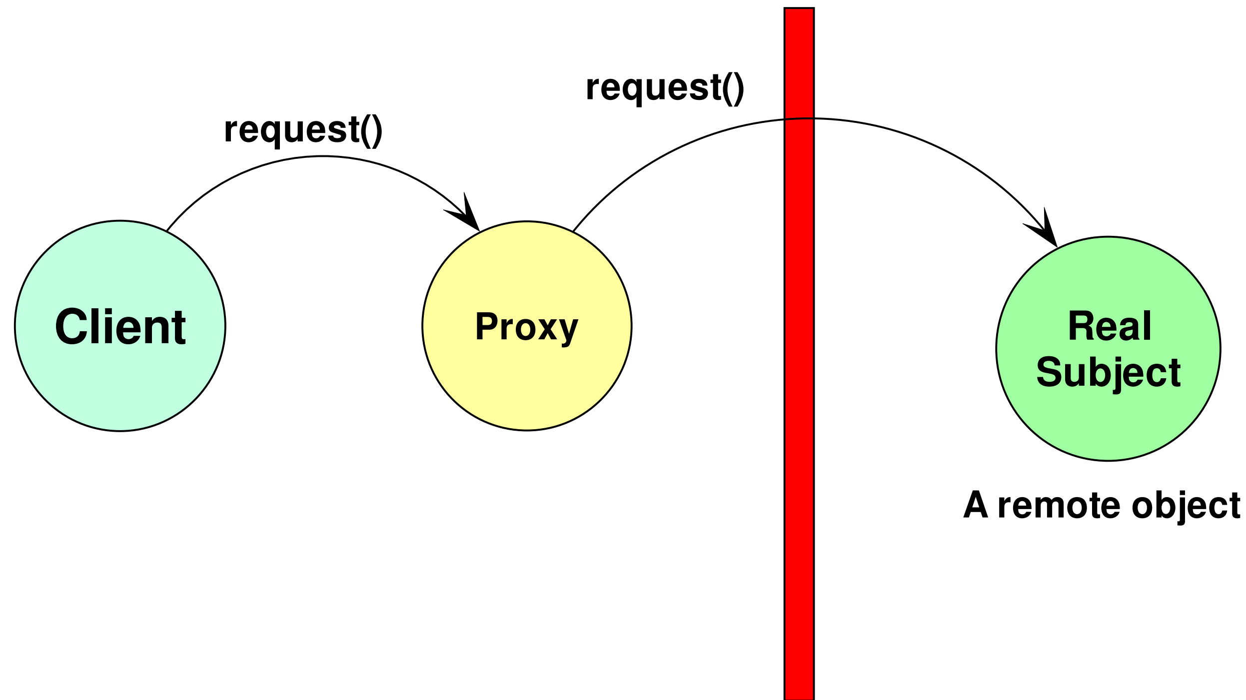


Proxy (structural)



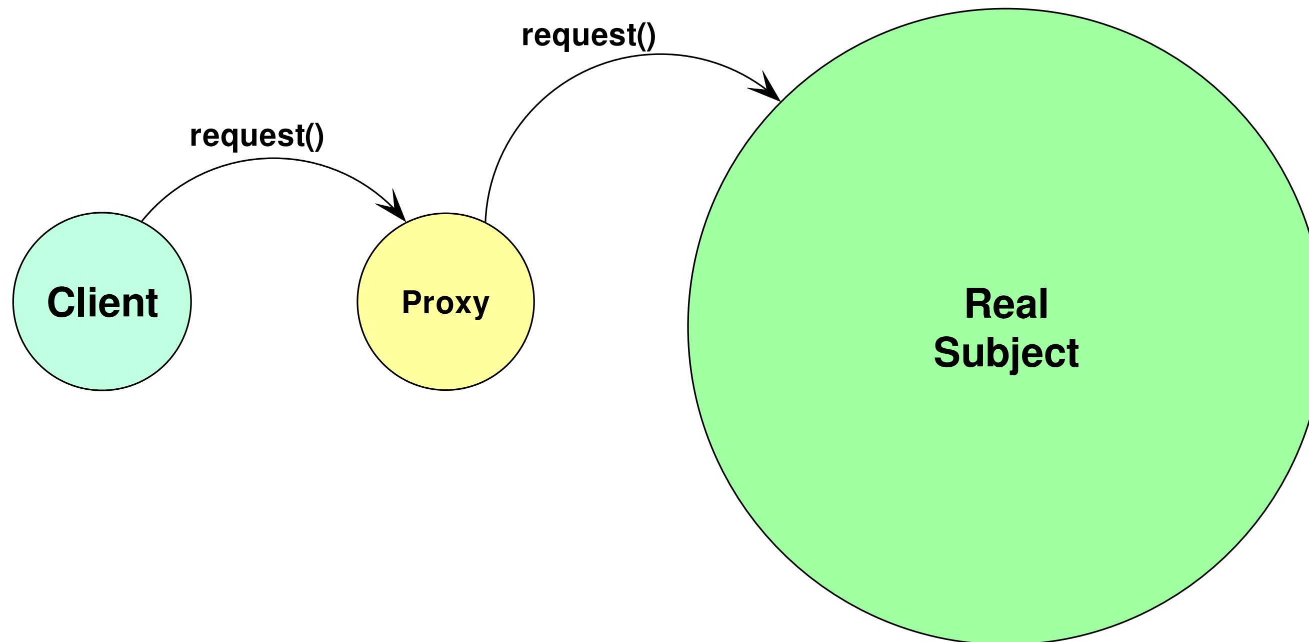
The Proxy Pattern provides a surrogate or placeholder for another object to access control to it.

Proxy (structural)



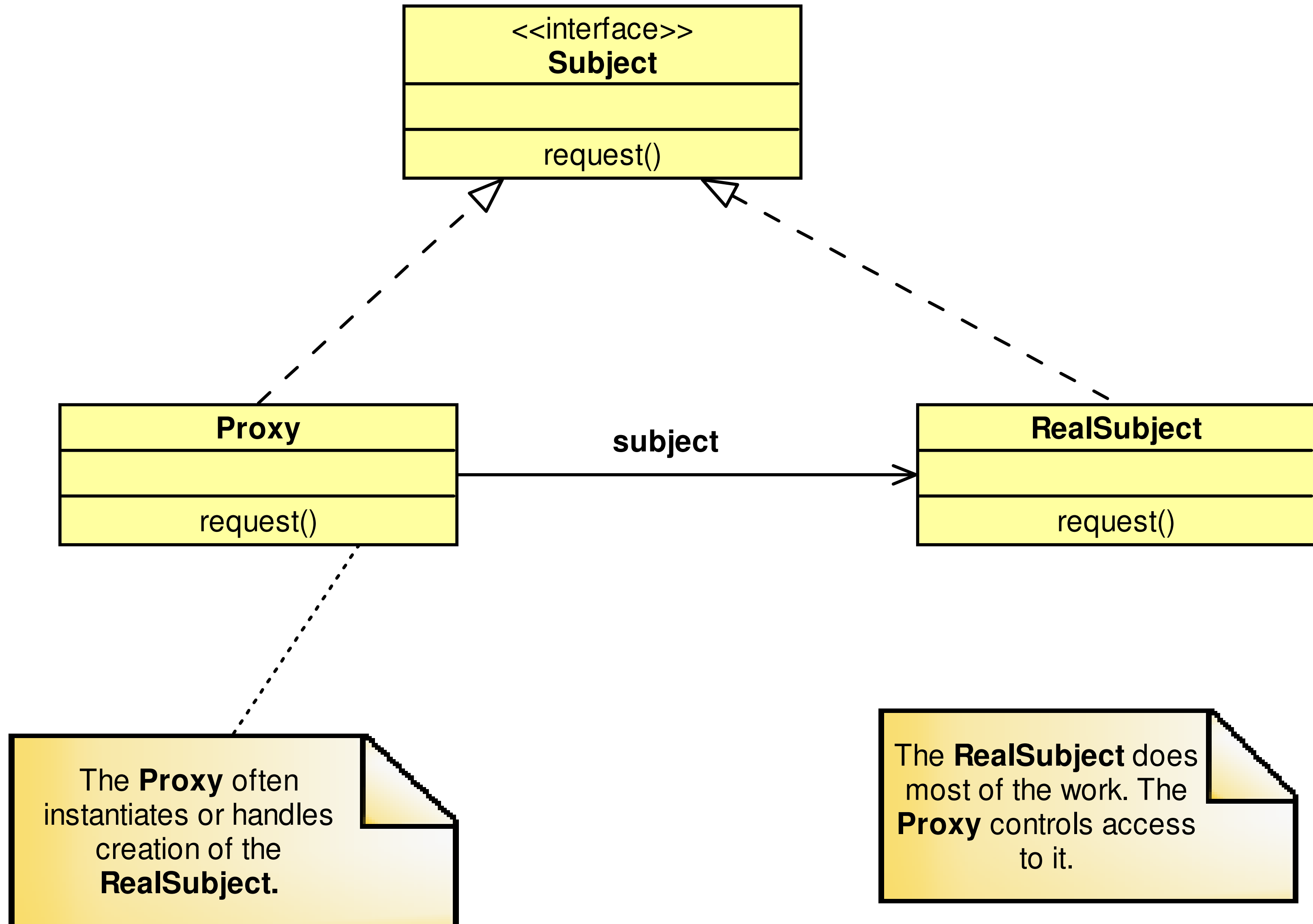
With **Remote Proxy**, the proxy act as a local representative for an object that lives in a different JVM.

Proxy (structural)



Virtual Proxy acts as a representative for an object that may be expensive to create. The virtual proxy often defers creation of the object until it is needed. After that, the virtual proxy delegates requests to the RealSubject.

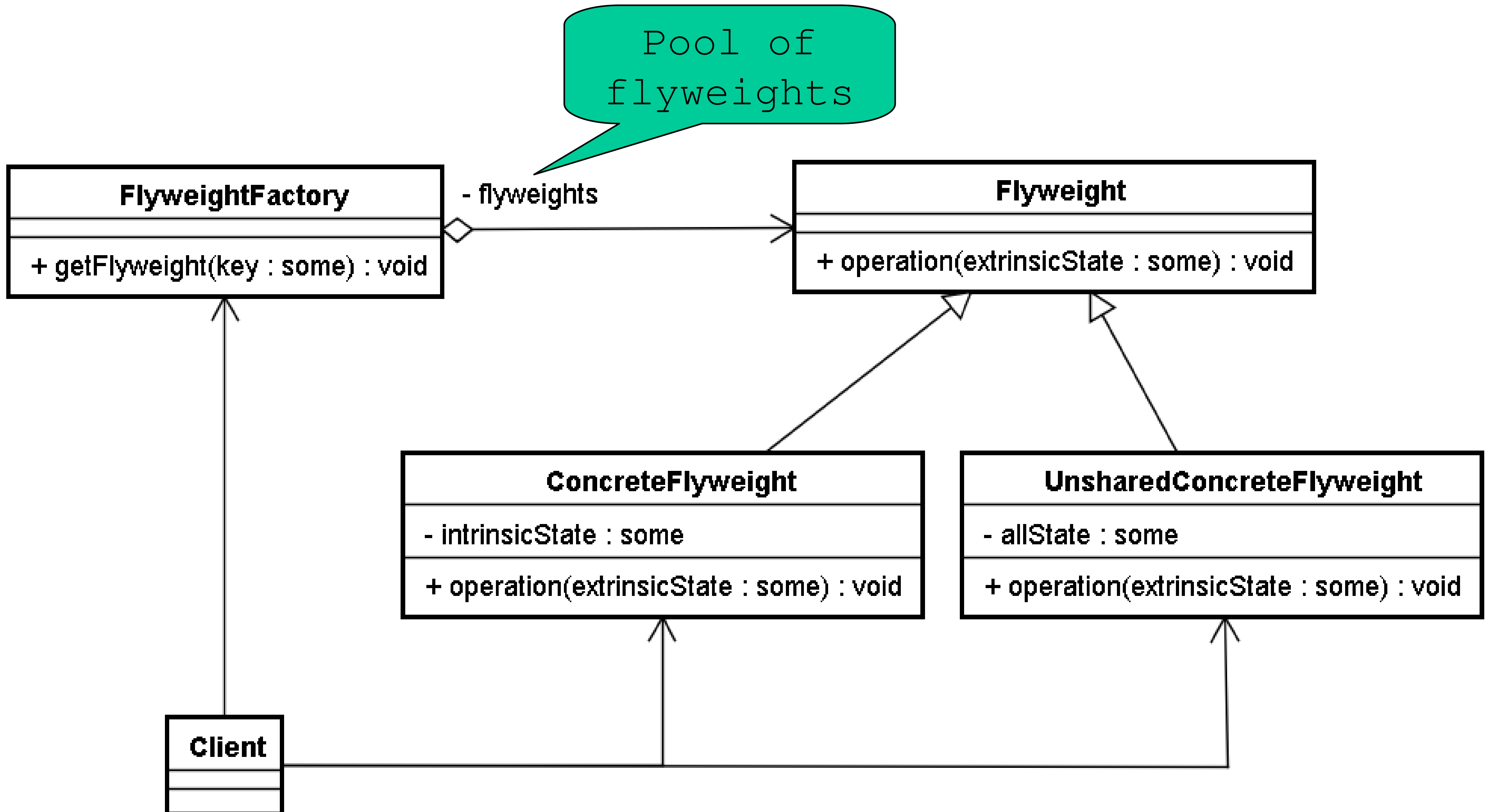
Proxy (structural)



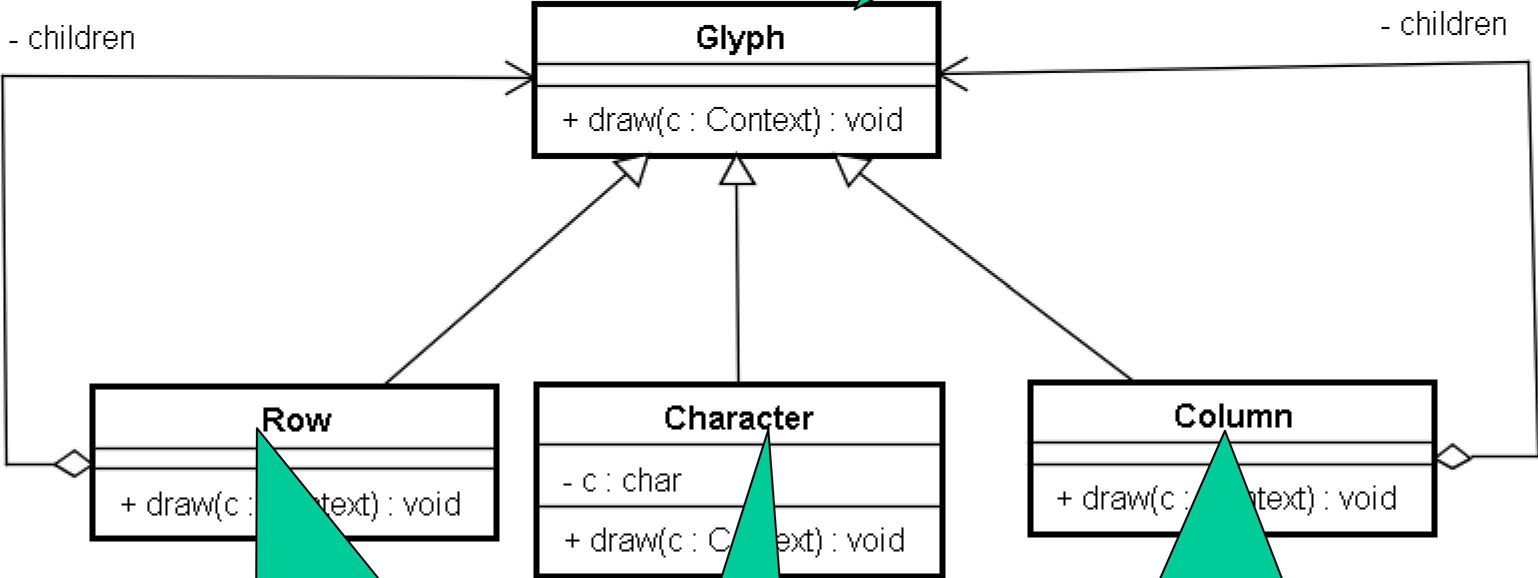
Flyweight (structural)

- Intrinsic state is stored in the `ConcreteFlyweight` object
- Extrinsic state is stored or computed by `Client` objects. Clients pass this state to the flyweight when they invoke they operations
- Clients should not instantiate `ConcreteFlyweights` directly. Clients must obtain `ConcreteFlyweights` from the `FlyweightFactory` to ensure they are shared properly.
- not all Flyweight objects need to be shared. It is common for `UnsharedConcreteFlyweight` objects to have `ConcreteFlyweight` objects as children.

Flyweight (structural)



Flyweight (structural)



Behavioral Design Patterns

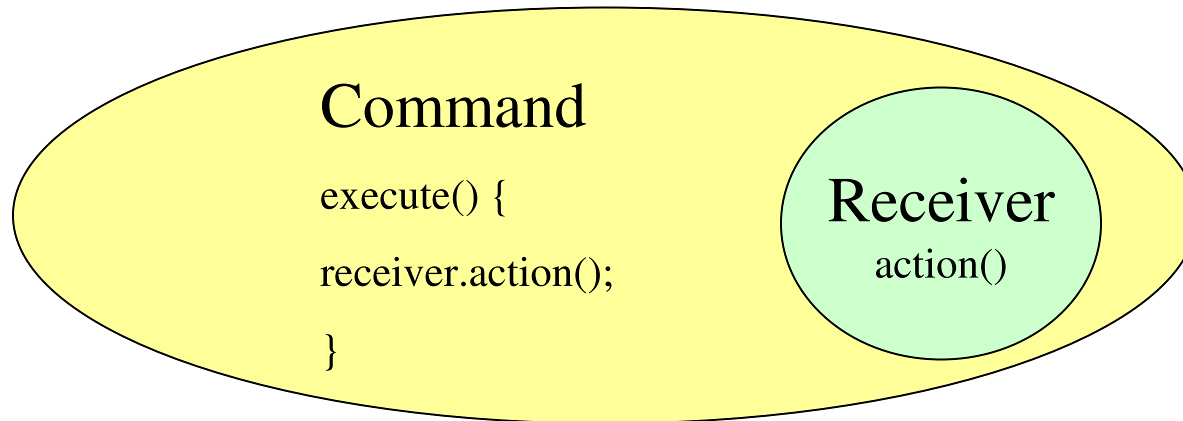
- Command
- Interpreter
- Iterator
- Mediator
- Observer
- State
- Strategy
- Visitor

Command (behavioral)

Command represents an abstract algorithm independent on:

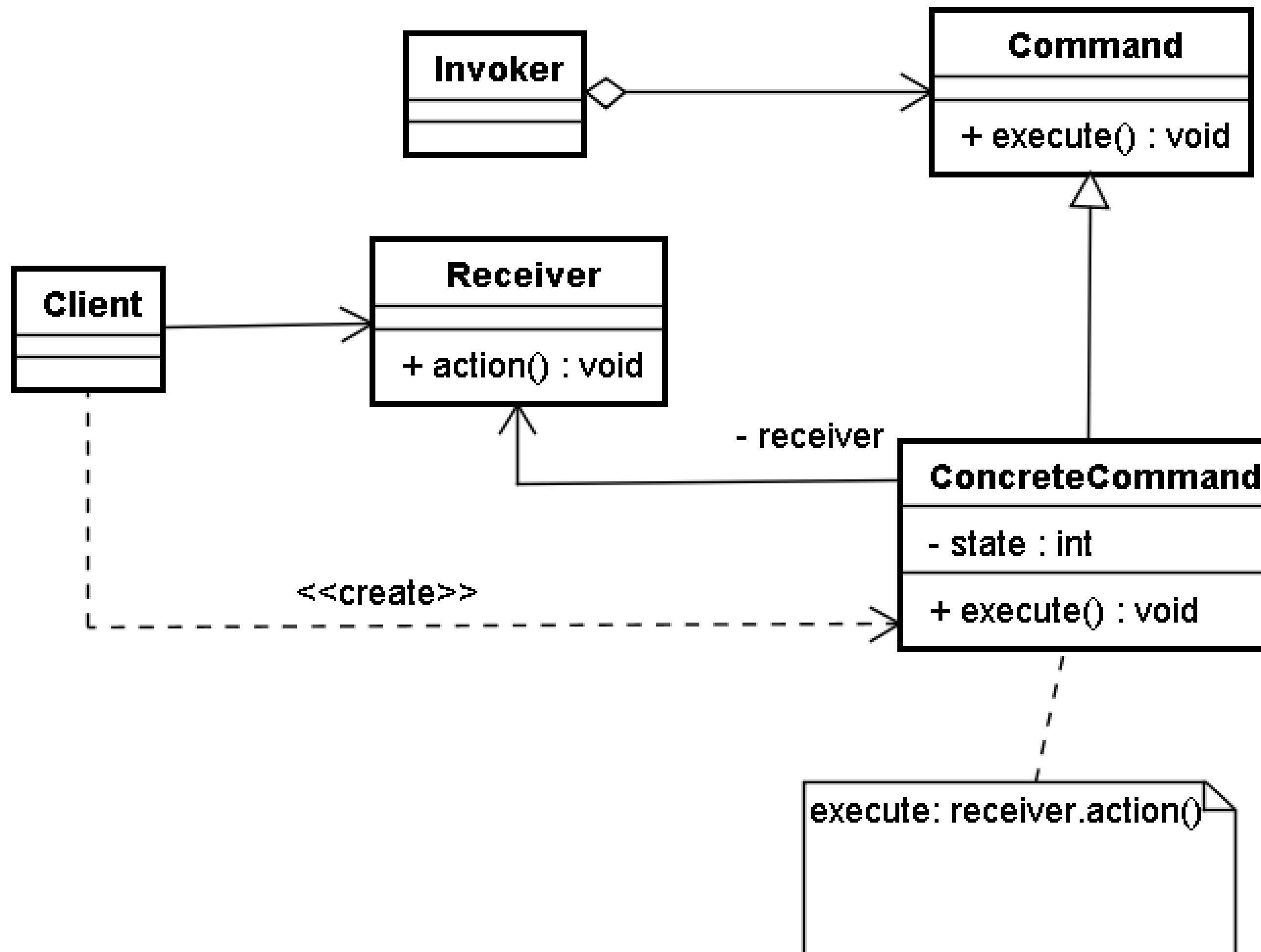
1) The application of the command (Client)
2) The particular implementation of the command (Receiver)

Command (behavioral)

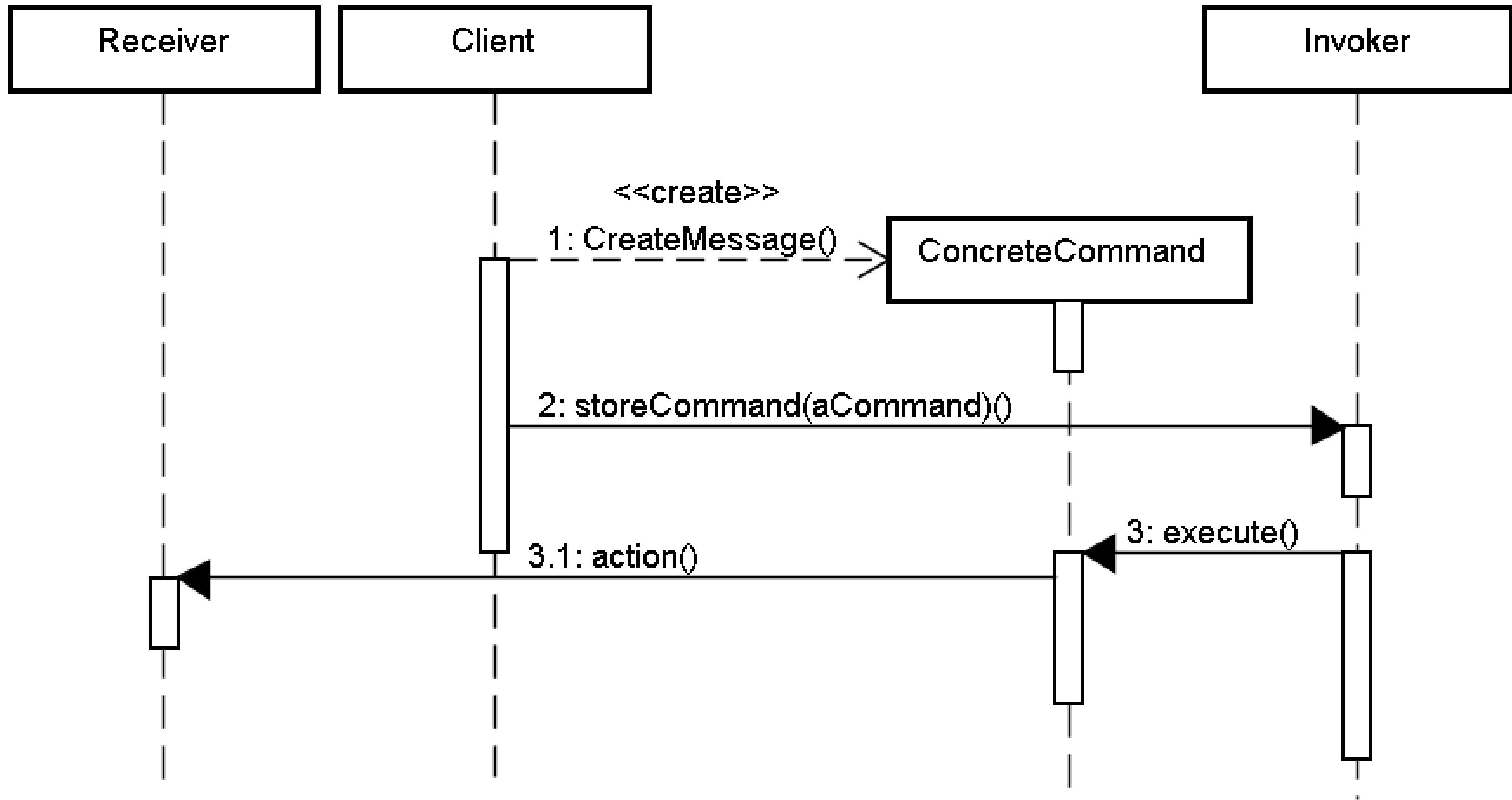


Command encapsulates a **request** as an object, thereby letting you parametrize other objects with different requests, queue or log requests and support undoable operations.

Command (behavioral)



Command (behavioral)



Command (behavioral)

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- You can assemble commands into a composite command (composite commands are instances of Composite pattern)
- Easy to add new commands.

Behavioral Design Patterns

Behavioral **Class** Patterns

Use inheritance to distribute behaviour between classes

- Template Method
- Interpreter

Behavioral **Object** Patterns

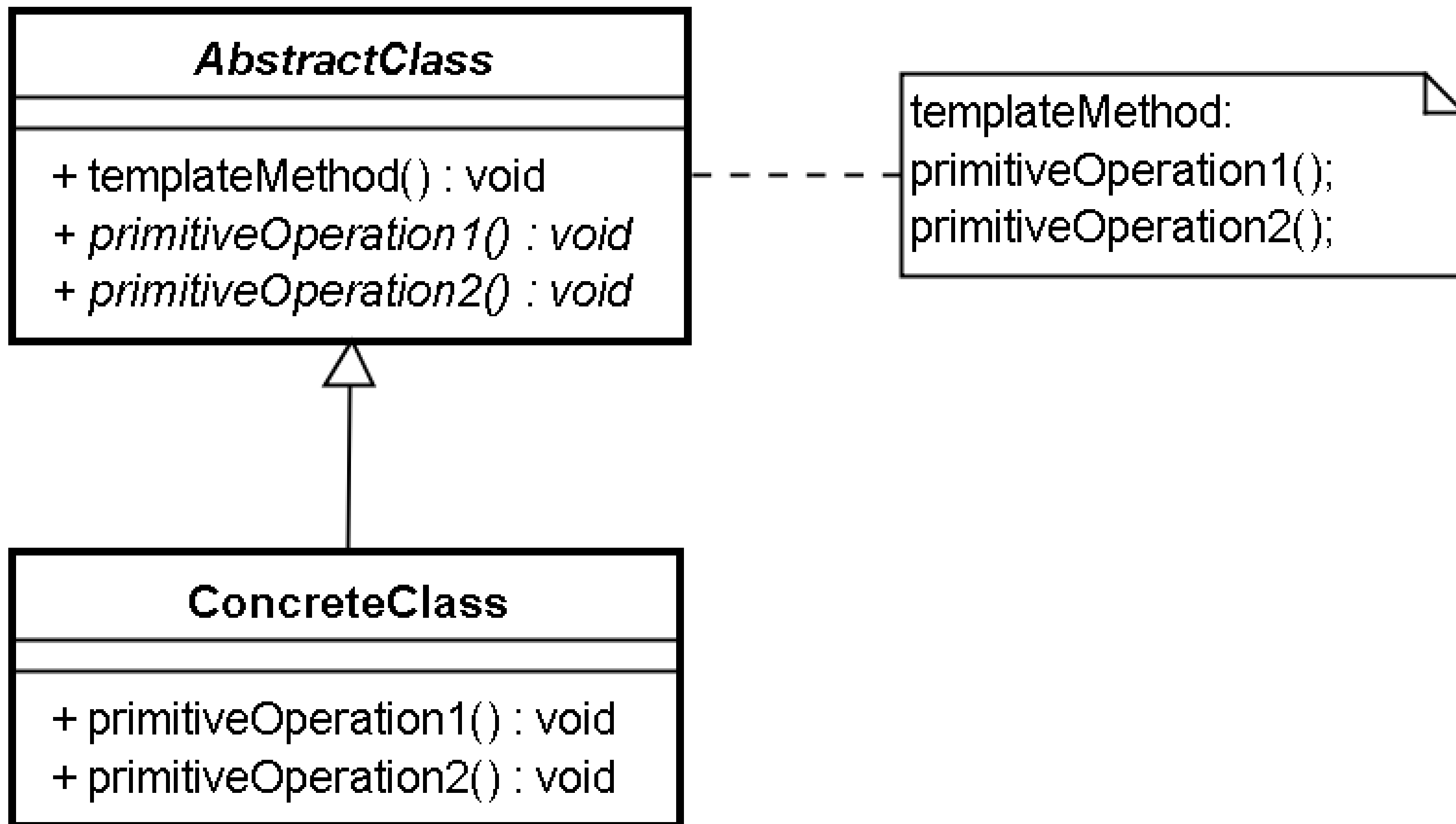
Use composition rather than inheritance to distribute behaviour between objects

- Mediator
- Chain of Responsibility
- Observer

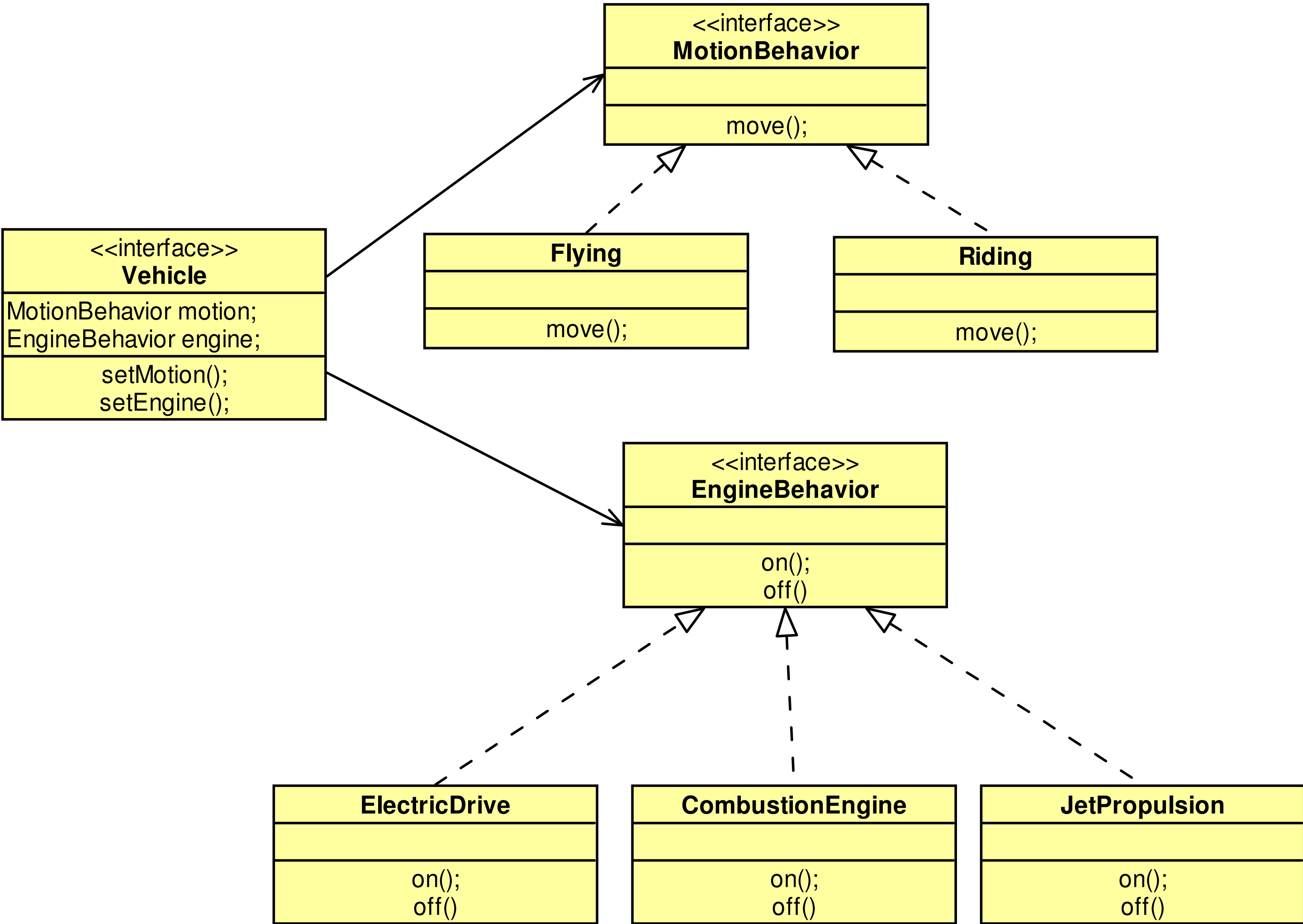
Other Behavioral Patterns

- Strategy
- Command
- Visitor
- Iterator

Template Method



Strategy (behavioral)



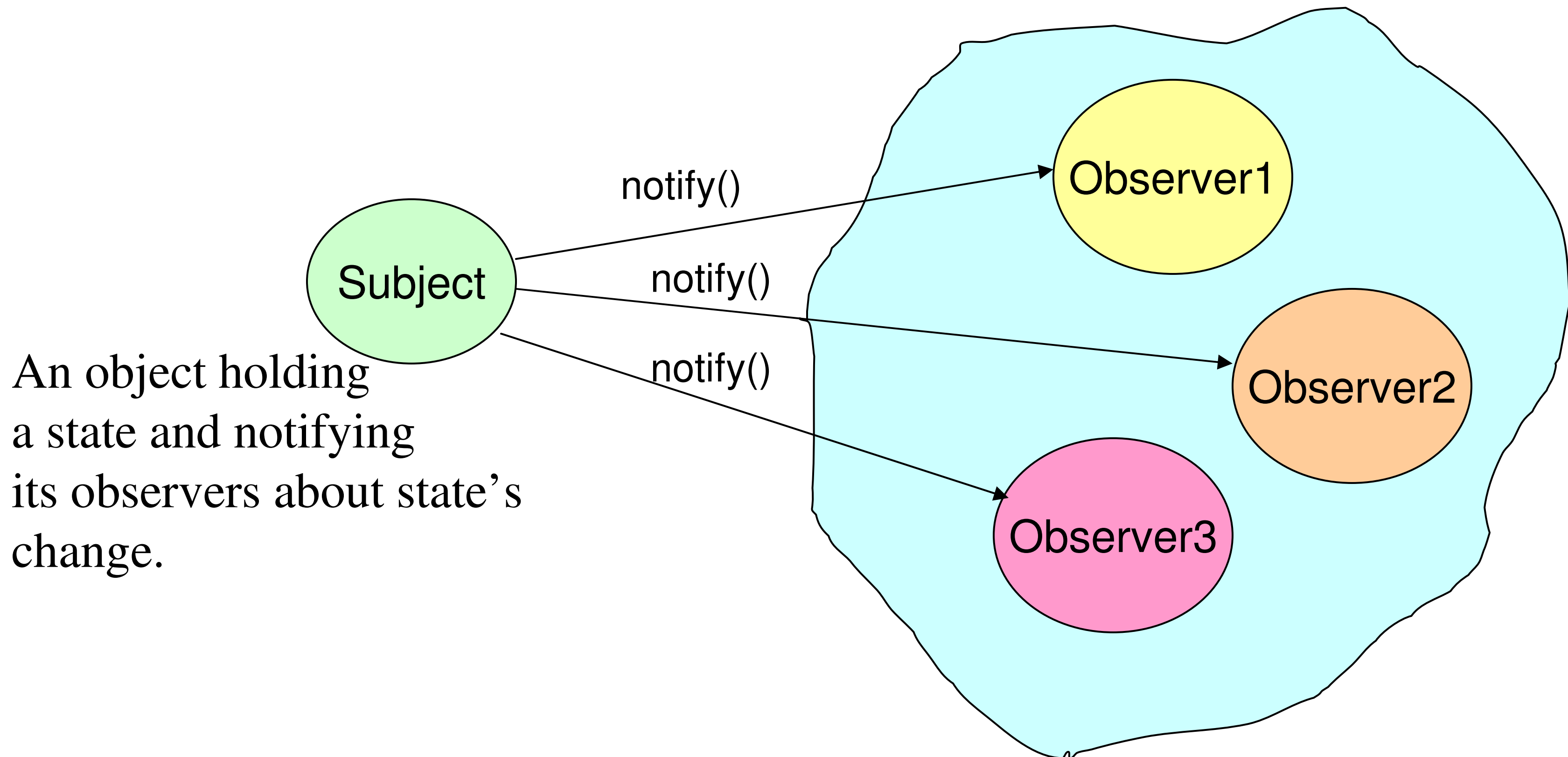
Strategy (behavioral)

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

Desing Principal

Favor composition over inheritance.

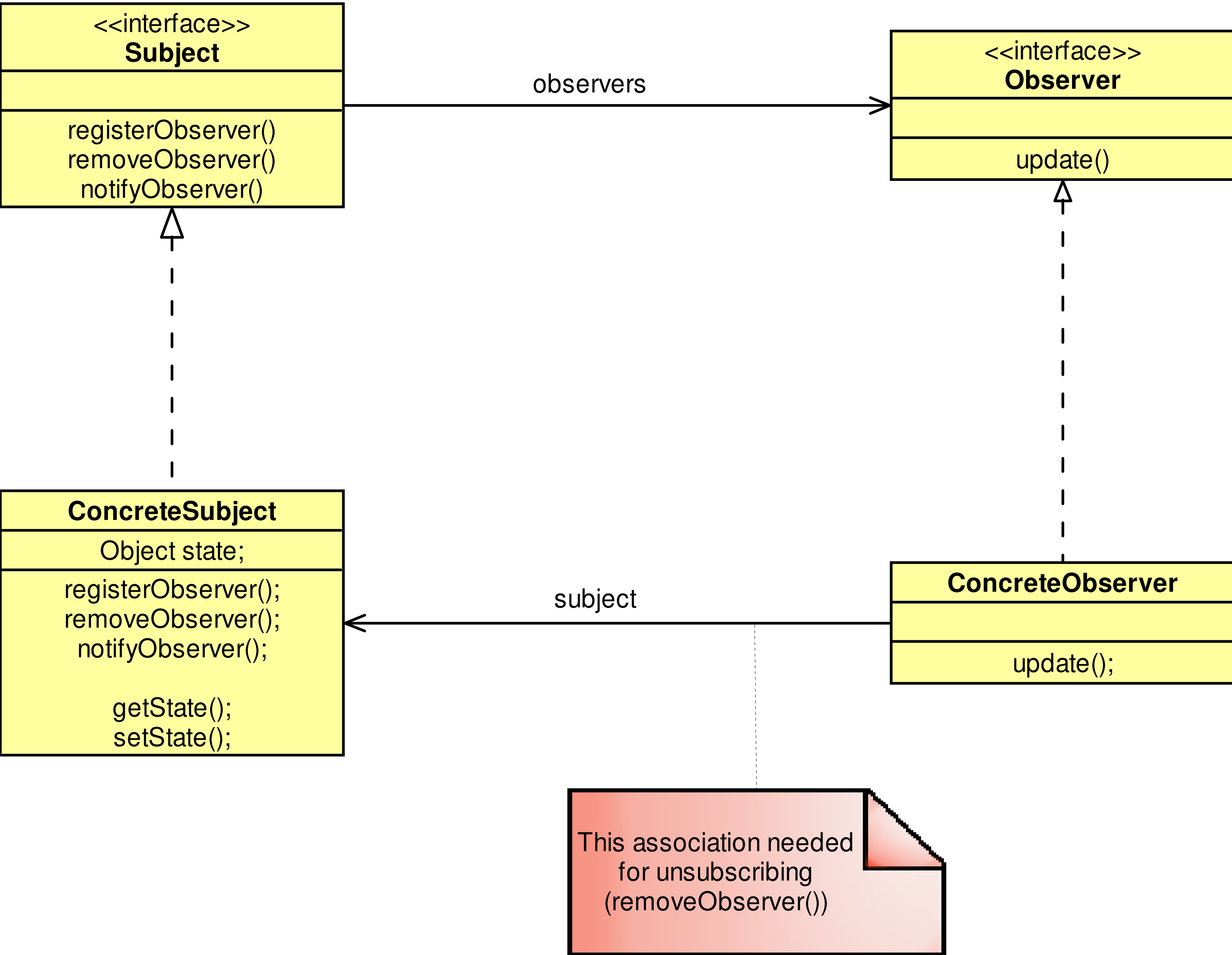
Observer (behavioral)



An object holding a state and notifying its observers about state's change.

The aim is to make subject independent on observers - loose coupling.

Observer (behavioral)



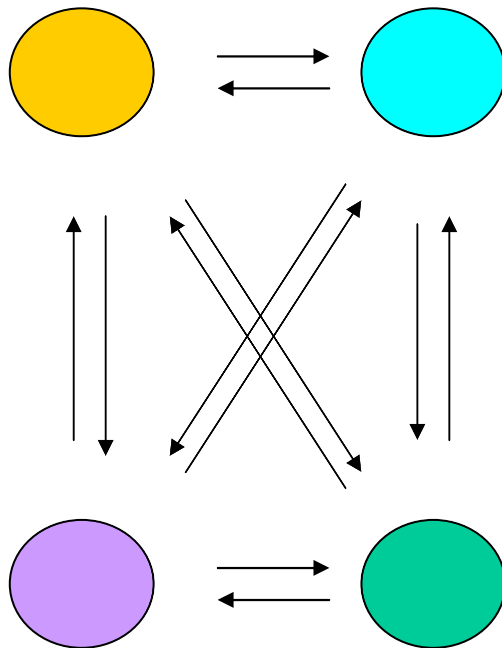
Observer (behavioral)

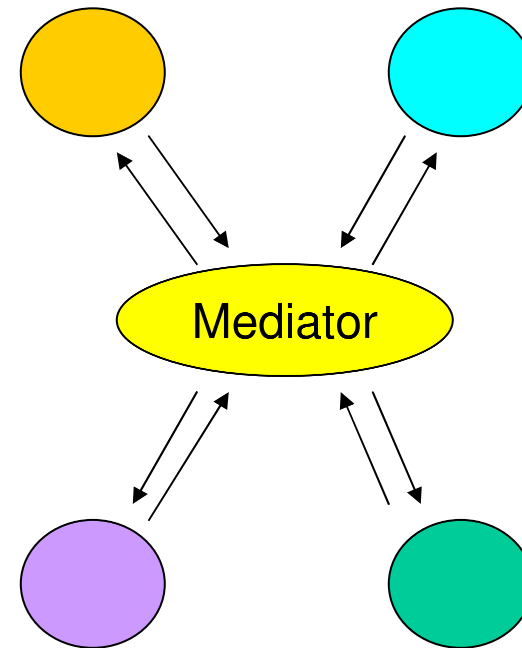
Desing Principle

Loosely coupled designs allows us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

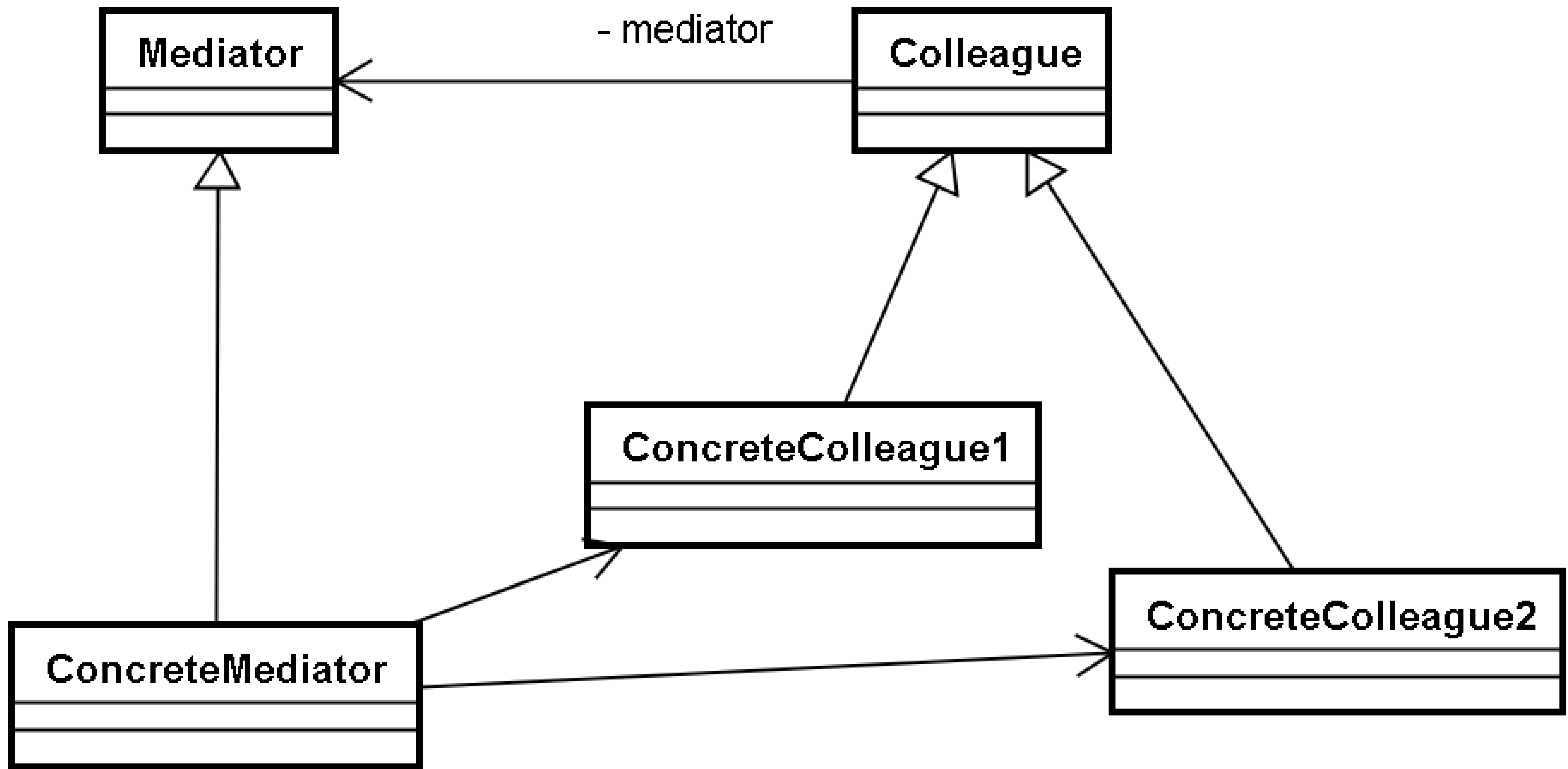
Mediator (behavioral)

- define an object that encapsulates how a set of objects interacts
- mediator promotes loose coupling by keeping objects from refering to each other explicitly
- it lets you vary their interactions independently

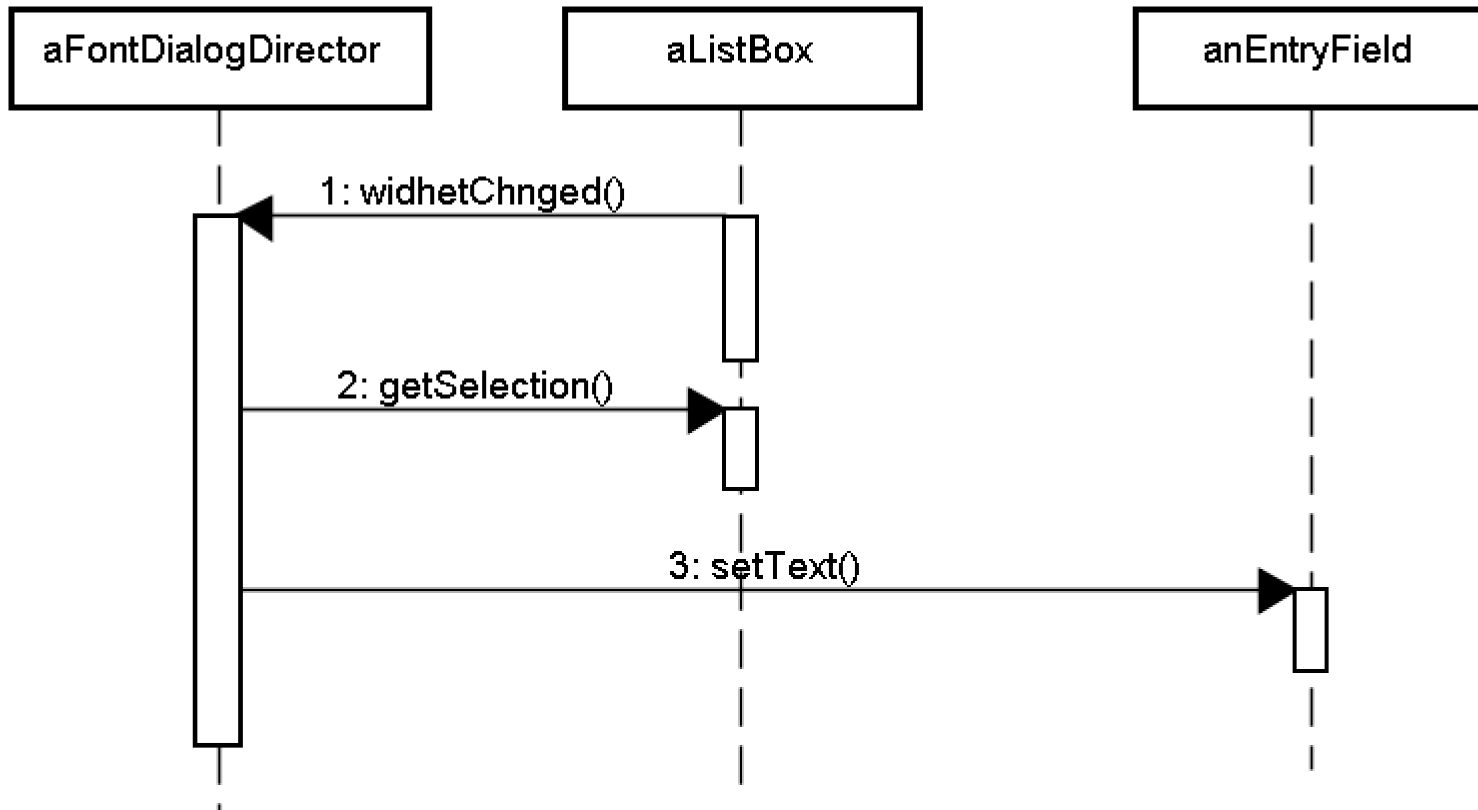




Mediator (behavioral)

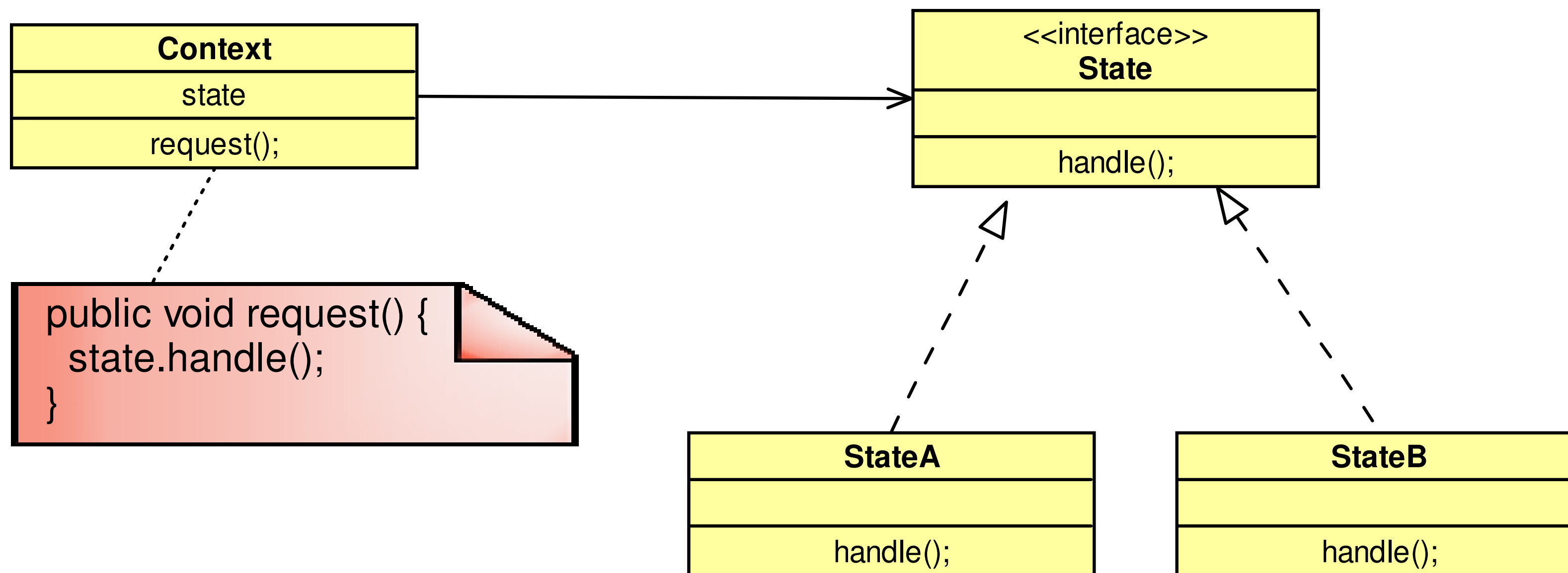


Mediator (behavioral)

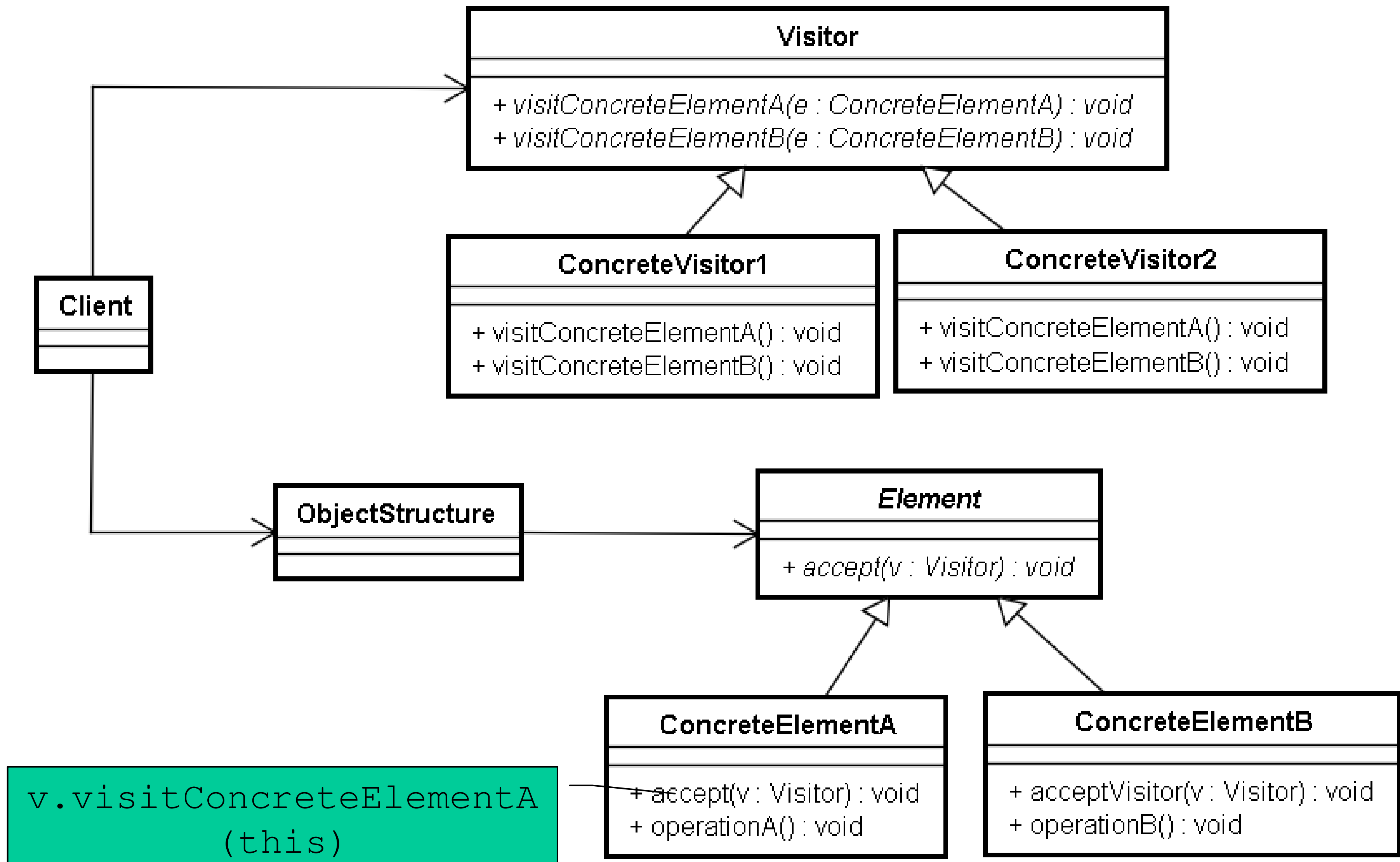


State (behavioral)

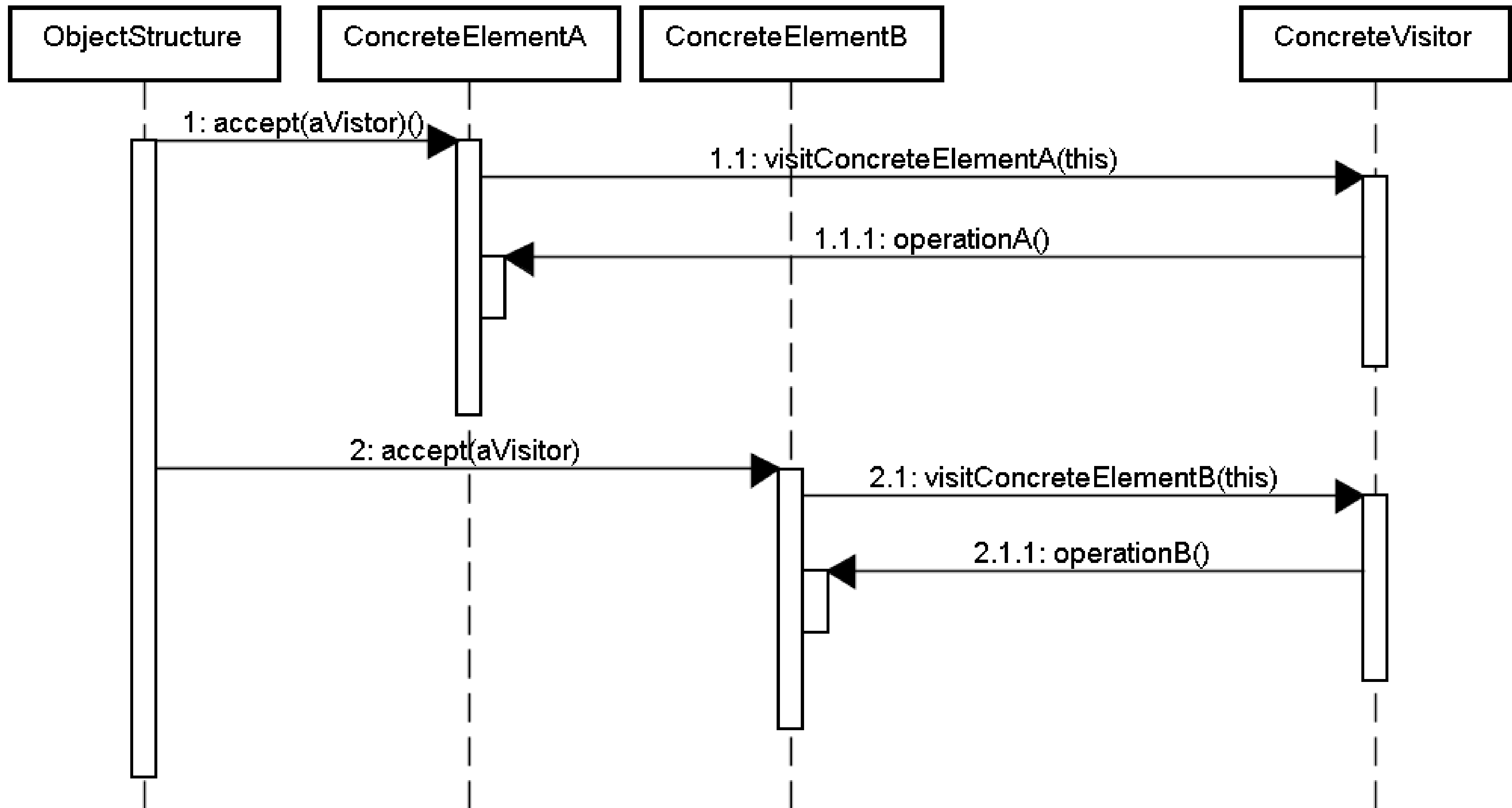
State Pattern allows an object to alter its behavior when its internal state changes.



Visitor (behavioral)



Visitor (behavioral)



Visitor (behavioral)

- Visitor makes adding new operations **easy**
  simply by adding a new visitor
- A visitor gathers related operations and separates unrelated ones
- Adding new ConcreteElement classes is **hard**
  Is mostly likely to change the algorithm or the classes of objects that make up the structure?
- Visitor can accumulate state as they visit each element