



JAVA 达摩班

设计模式

The Problem with “Best Practices”

DOGBERT CONSULTS

I'LL TEACH YOU
THE BEST PRACTICES
OF COMPANIES THAT
HAVE NOTHING IN
COMMON WITH
YOURS.

Dilbert.com DilbertCartoonist@gmail.com

THOSE PRACTICES
WILL FIT YOUR
COMPANY LIKE A
FOOT IN A GLOVE.

52-31-13 © 2013 Scott Adams, Inc. /Dilbert by Universal Uclick

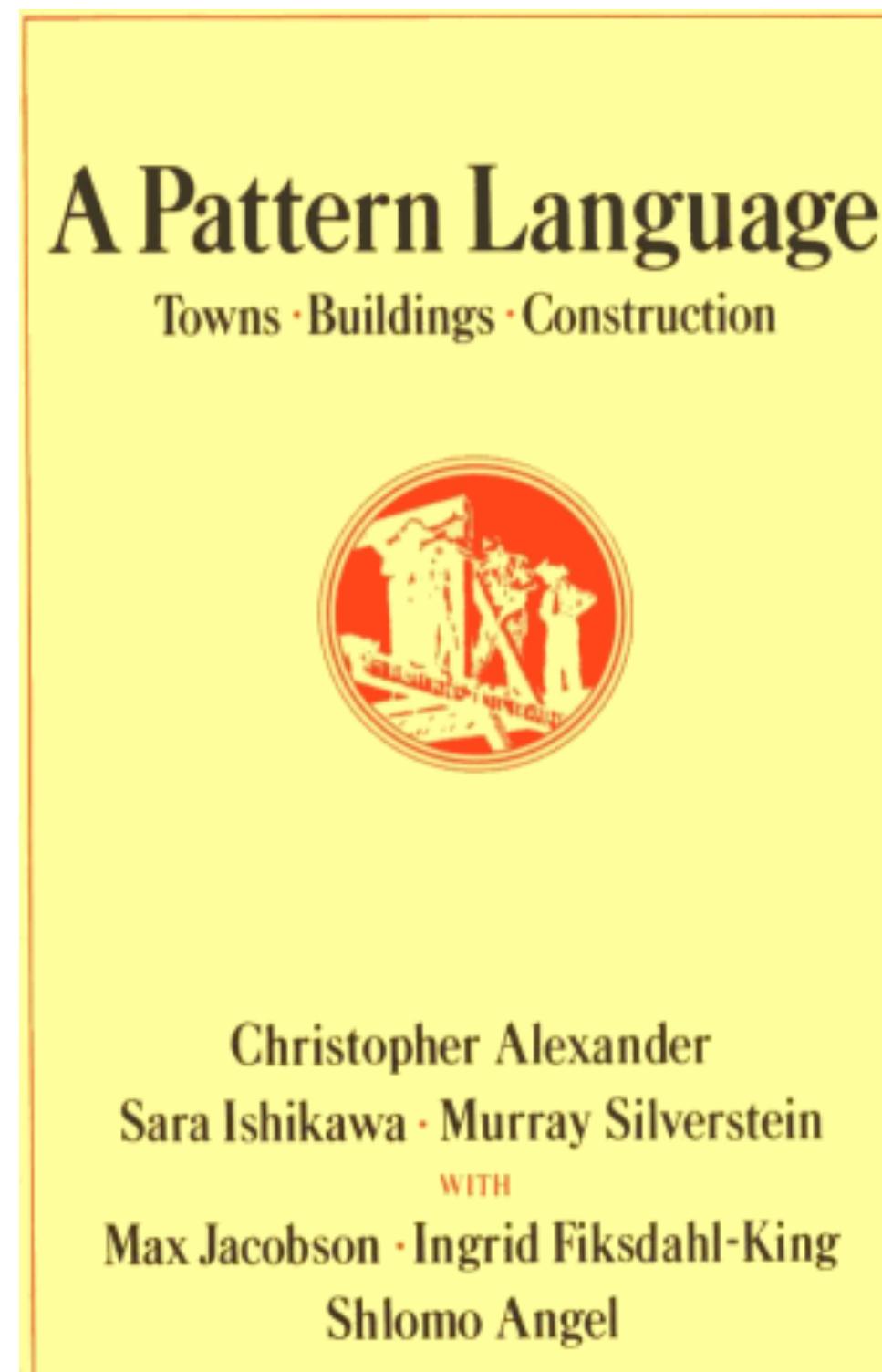
CLOSE
ENOUGH.



1

从建筑模式 与软件模式

1977 模式语言



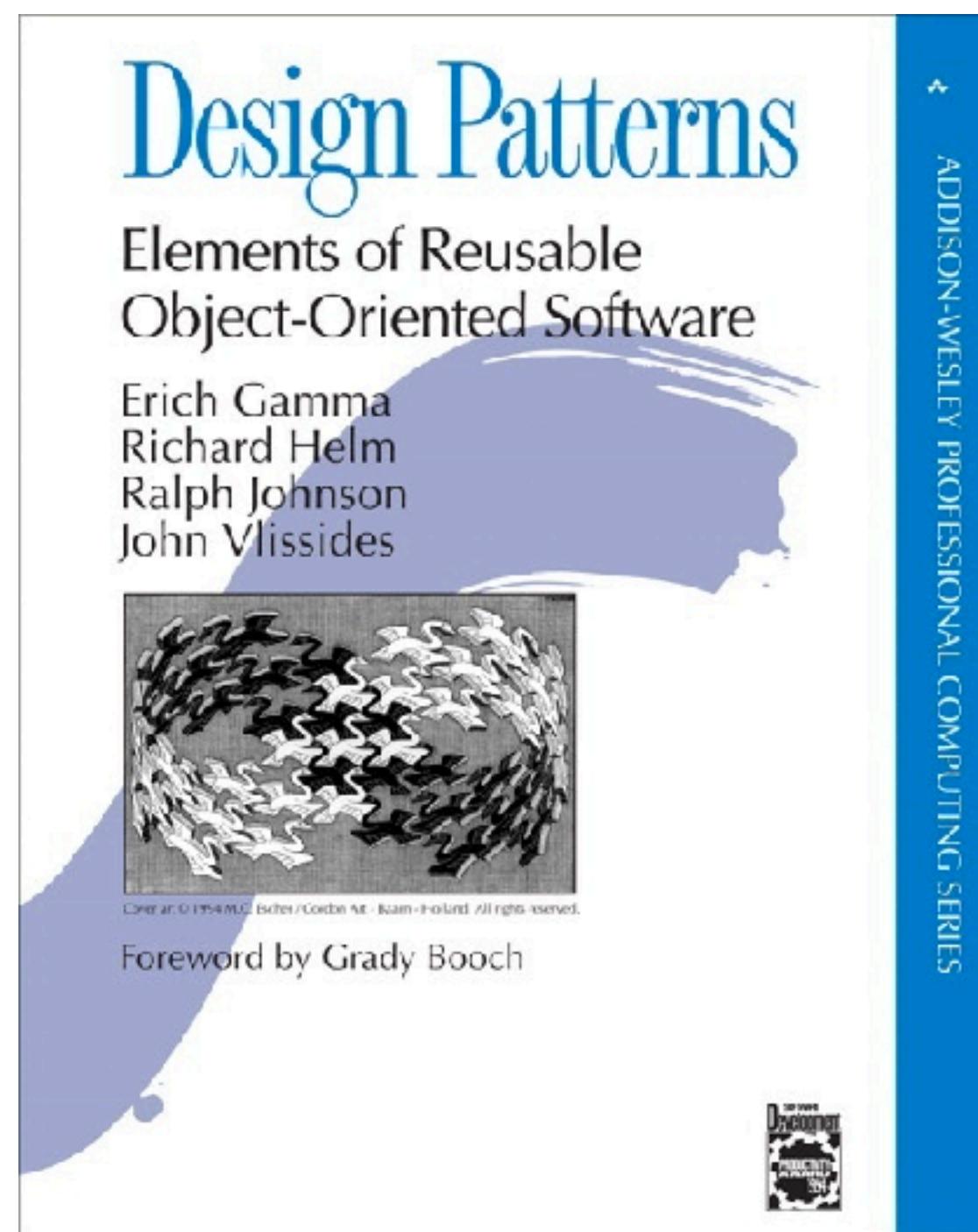
Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice

- Christopher Alexander

模式用于描述一种反复发生在当前环境下的问题，然后描述该问题的核心解决方案。

遇到同样的问题可以套用同样的解决方式，而不是重新做一遍

1994 设计模式



设计模式是通用的，可重复的软件设计方法或解决方案。
设计模式是最佳实践的合集
设计模式并不是可以直接转换为代码的设计，而是解决方案的描述或者模版，可以用在多种不同的情况下

GoF (Gang of Four) 为面向对象归纳常用的设计模式
三个大类，二十三个小类

编程接口，而不是编程实现
使用对象组合，而不是对象继承

不重复造车轮

2000 SOLID



Single Responsibility Principle

单一职责原则

Open/Closed Principle

开闭原则

Liskov Substitution Principle

里氏替换原则

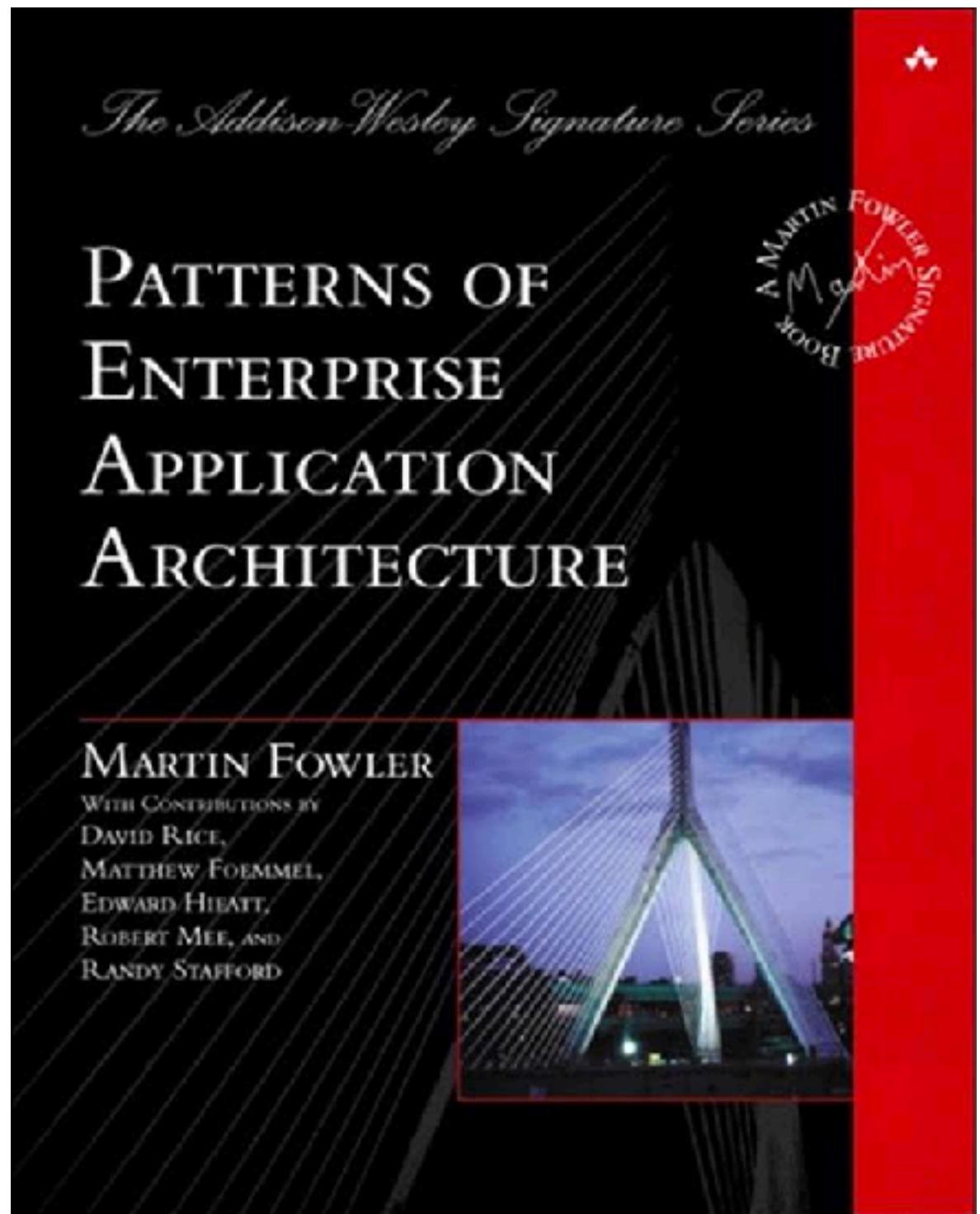
Interface Segregation Principle

接口分离原则

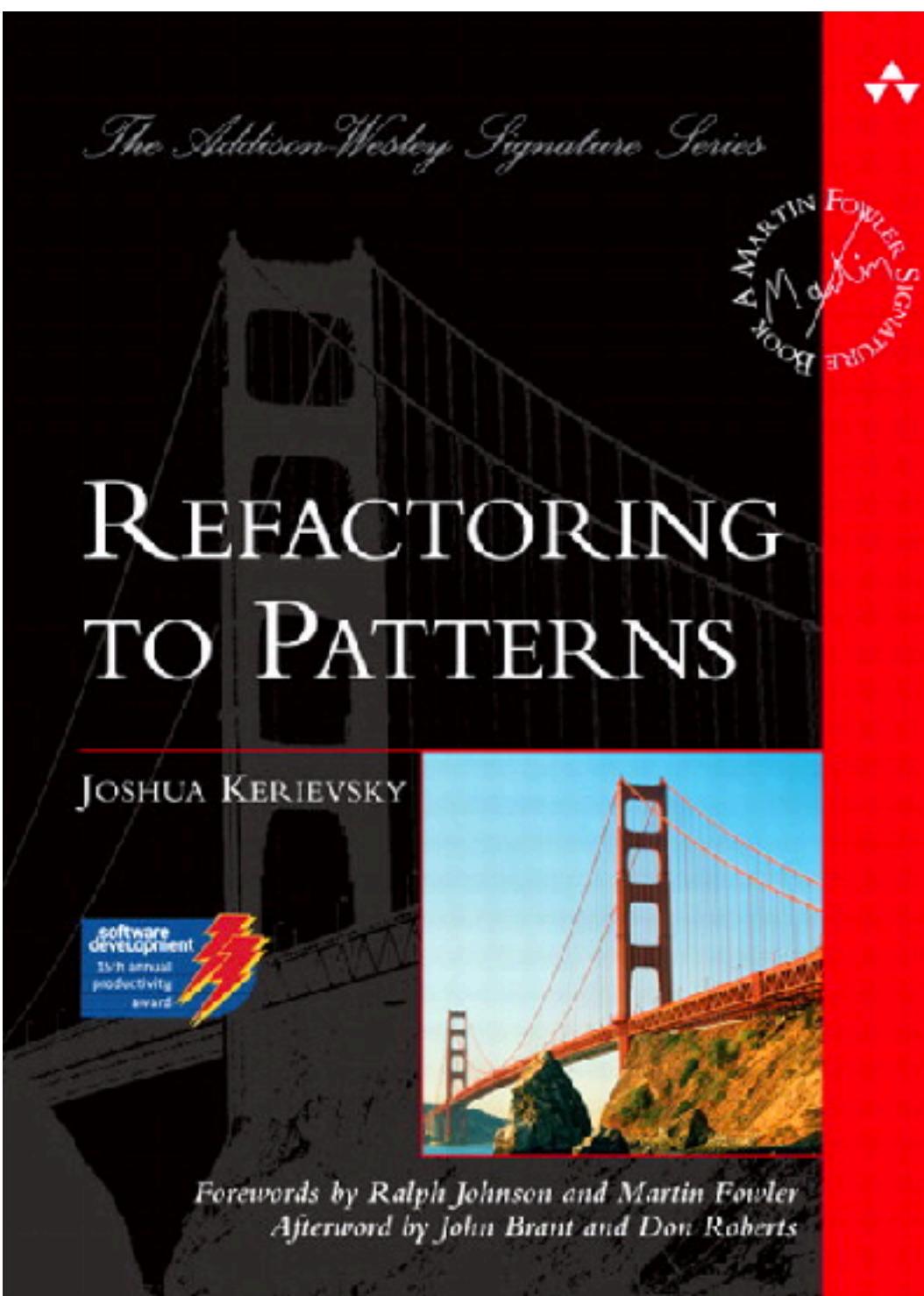
Dependency Inversion Principle

依赖反转原则

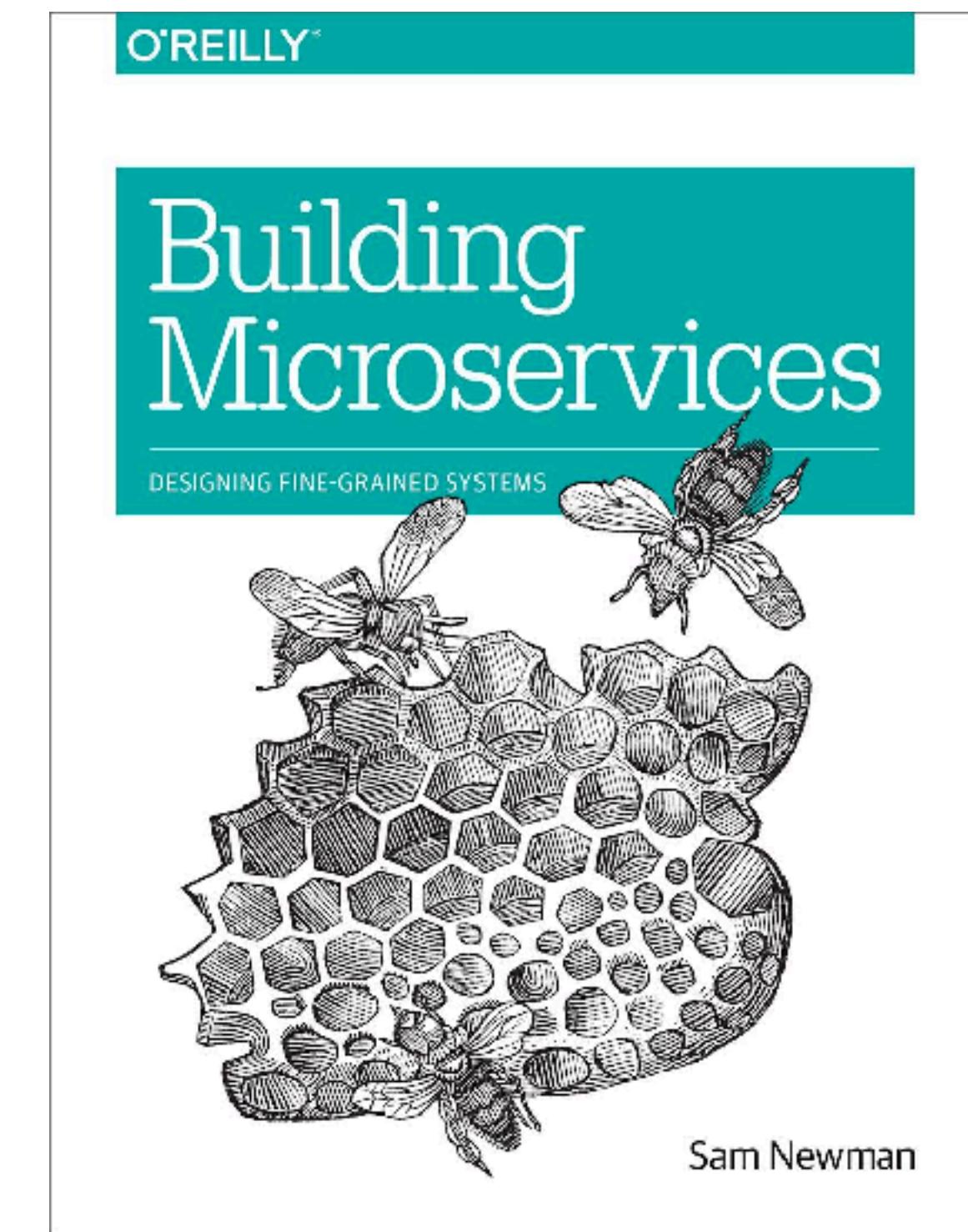
2002 企业应用架构



2005 从重构到模式



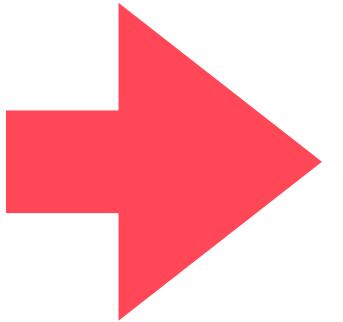
2014 微服务





SOLID 2

高内聚 低耦合



单一职责原则 SRP

每个类应该只有一个职责，并且该职责应该被完整的封装在类中。

职责可以看作“改变的原因”，那么类或者模块有且仅有一个“改变的原因”

可维护性：一个改变仅能发生在一个模块或者类中

例子：

假设存在一个打印机模块，且有两个原因导致它‘改变’：打印的内容变化和报告的格式变化。那么从SRP原则，那么应该创建两个单独的类分别实现。



开闭原则 OCP

“软件实体（类，模块，函数等）应该对扩展开放，但对修改闭合”

-Bertrand Meyer

通过最小化现有代码的变化，达到提高维护行和稳定性的目的

一个类应该是闭合的，因为它可能被编译和存储在类库中，可能客户正在使用
一个类应该是开放的，因为新类可以该类为父类添加新特性，而不破坏原始类



里氏替换原则 LSP

可替换性

程序中的对象可以被替换为该对象的子对象，而不会改变程序的正确性。

例如

C类是P类的子类型（继承/实现），那么P类的对象可以被替换为C类的对象



接口分离原则 ISP

客户端不应该强行依赖于它用不到的方法

多个具有特殊意义的接口好过通用接口

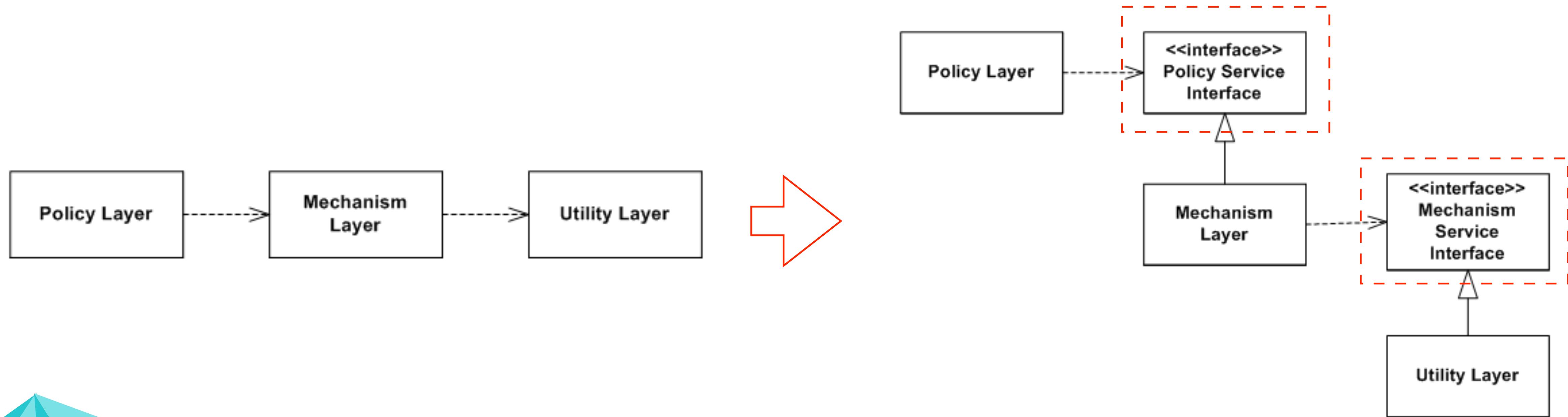
将“胖”接口分成多个小而具有特定含义的接口

如果一个类实现了一个不用的接口，那么调用者应该是（不得不）知道该方法不会被调用



依赖反转原则 DIP

高层模块不应该依赖于底层模块，他们都应该依赖于抽象
抽象不应该依赖于细节，细节应该依赖于抽象

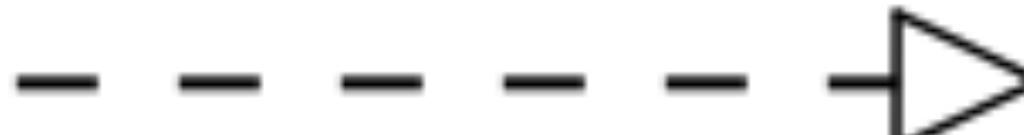




GoF 软件模式

3

UML常用关系

	Association	A关联B，例如学生和老师有关系（使用，连接），无拥有和关联
	Inheritance	A继承于B，泛化关系，例如人继承于哺乳
	Realization / Implementation	A实现了B接口，例如狗实现了生物接口
	Dependency	A依赖于/引用B，例如汽车（弱）依赖于轮胎
	Aggregation	A聚合B，A（弱）拥有B，但B可以独立于A存在，例如学生有文具
	Composition	A由B组成，A（强）拥有B，B的存在依赖于A，例如大学由系组成

弱→强：association → aggregation → composition

创建型 (creation)

单例(Singleton Pattern) 工厂(Factory Pattern) 抽象工厂(Abstract Factory Pattern)
构建(Builder Pattern) 原型(Prototype Pattern)

结构型 (structural)

适配器(Adapter Pattern) 组合(Composite Pattern) 代理(Proxy Pattern)
享元(Flyweight Pattern) 门面(Facade Pattern) 桥接(Bridge Pattern) 装饰器(Decorator Pattern)

行为型 (behavioral)

模板方法(Template Method Pattern) 中介者(Mediator Pattern) 链式责任(Chain of Responsibility Pattern)
观察者(Observer Pattern) 策略(Strategy Pattern) 命令(Command Pattern) 状态(State Pattern)
访问者(Visitor Pattern) 解释器(Interpreter Pattern) 迭代器(Iterator Pattern) 备忘录
(Memento Pattern)



创建型模式



单例模式

保证每个类只有一个实例， 提供一个全局访问点

日志， 驱动， 缓存和线程池等

`java.lang.Runtime, java.awt.Desktop`

实现方法：

- Eager initialization
- Static block initialization
- Lazy Initialization
- Bill Pugh Singleton Implementation



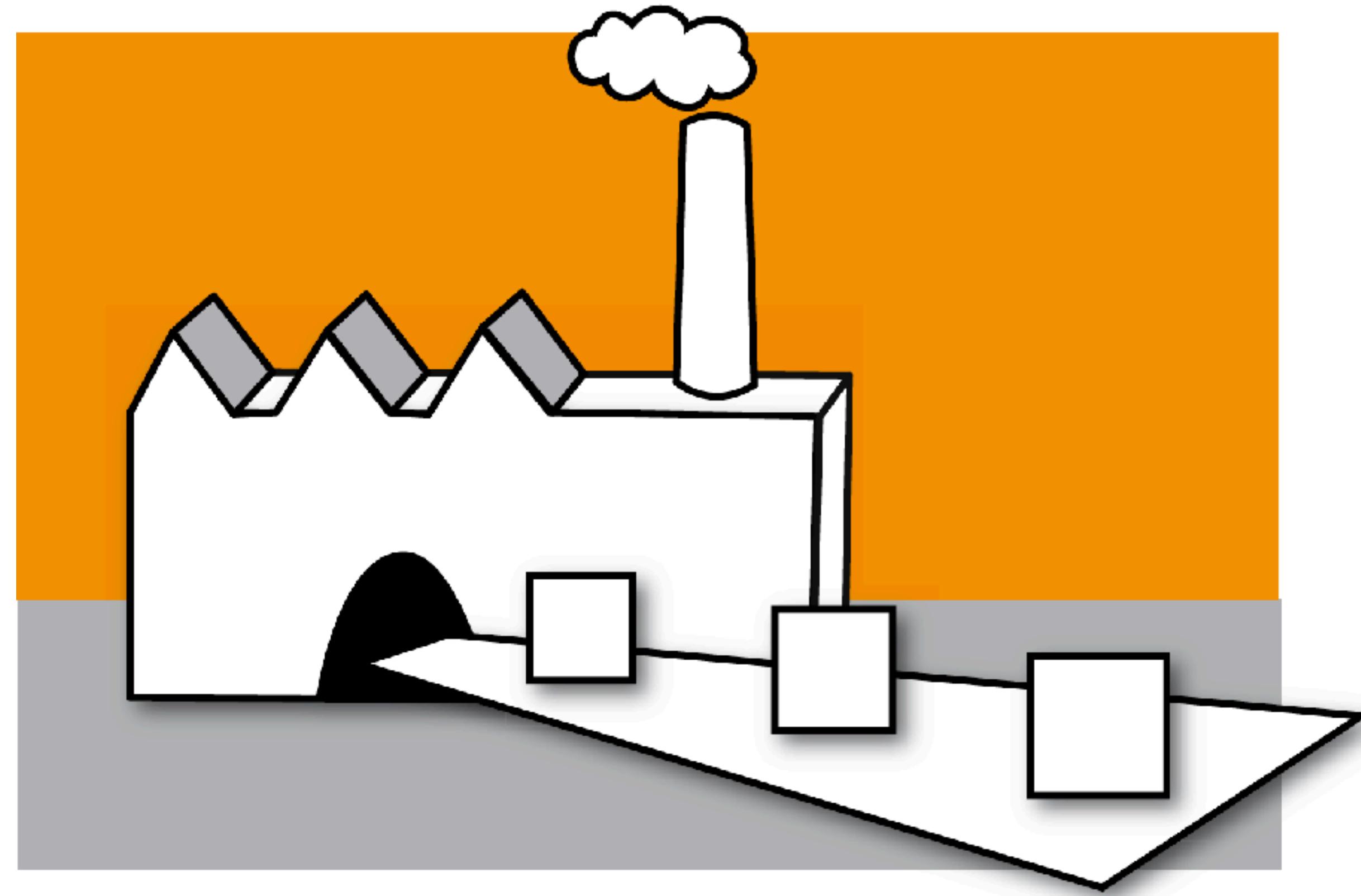
工厂方法

提供专用接口，用于创建对象

允许子类决定实例化哪个类

JDK (valueOf), Spring, Struts...

- 编程接口而不是实现
- 代码鲁棒性更佳，低耦合和易于扩展
- 实现类的实例化，可以放在客户端外面
- 通过继承实现类和客户端之间的抽象



工厂方法

超类规定标准和通用的行为，业务细节由子类实现

超类可以是接口，抽象类或者普通类

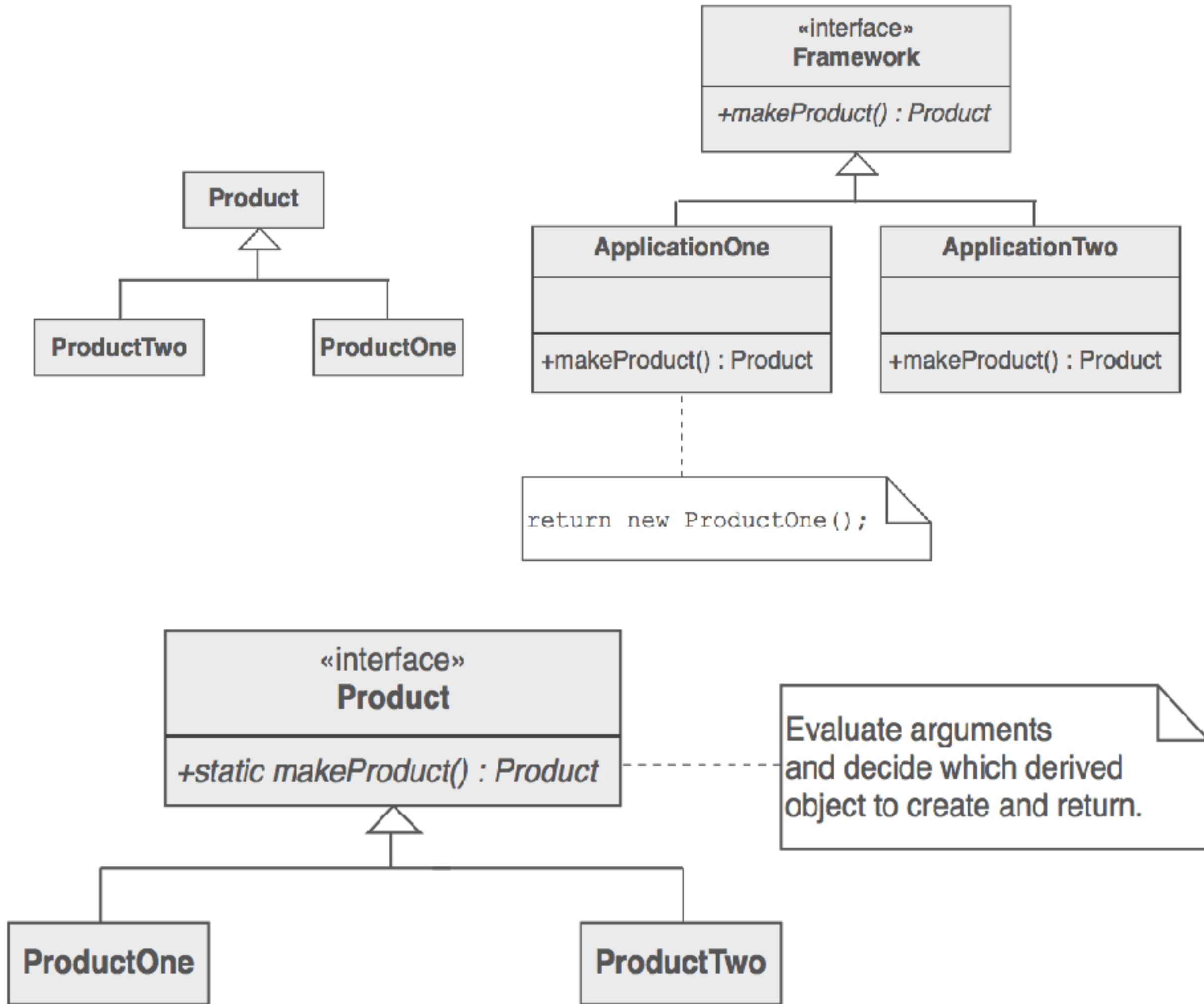
架构框架定义一个创建对象的接口，框架用户决定具体实例化哪个类

“虚拟”构造器，接口编程，

场景：当超类有多个子类实现的时候，根据输入生成相应的子类实例

缺点：框架必须标准化架构模式

应用：java.util.Calendar, NumberFormat的getInstance方法
Boolean, Integer的valueOf()方法

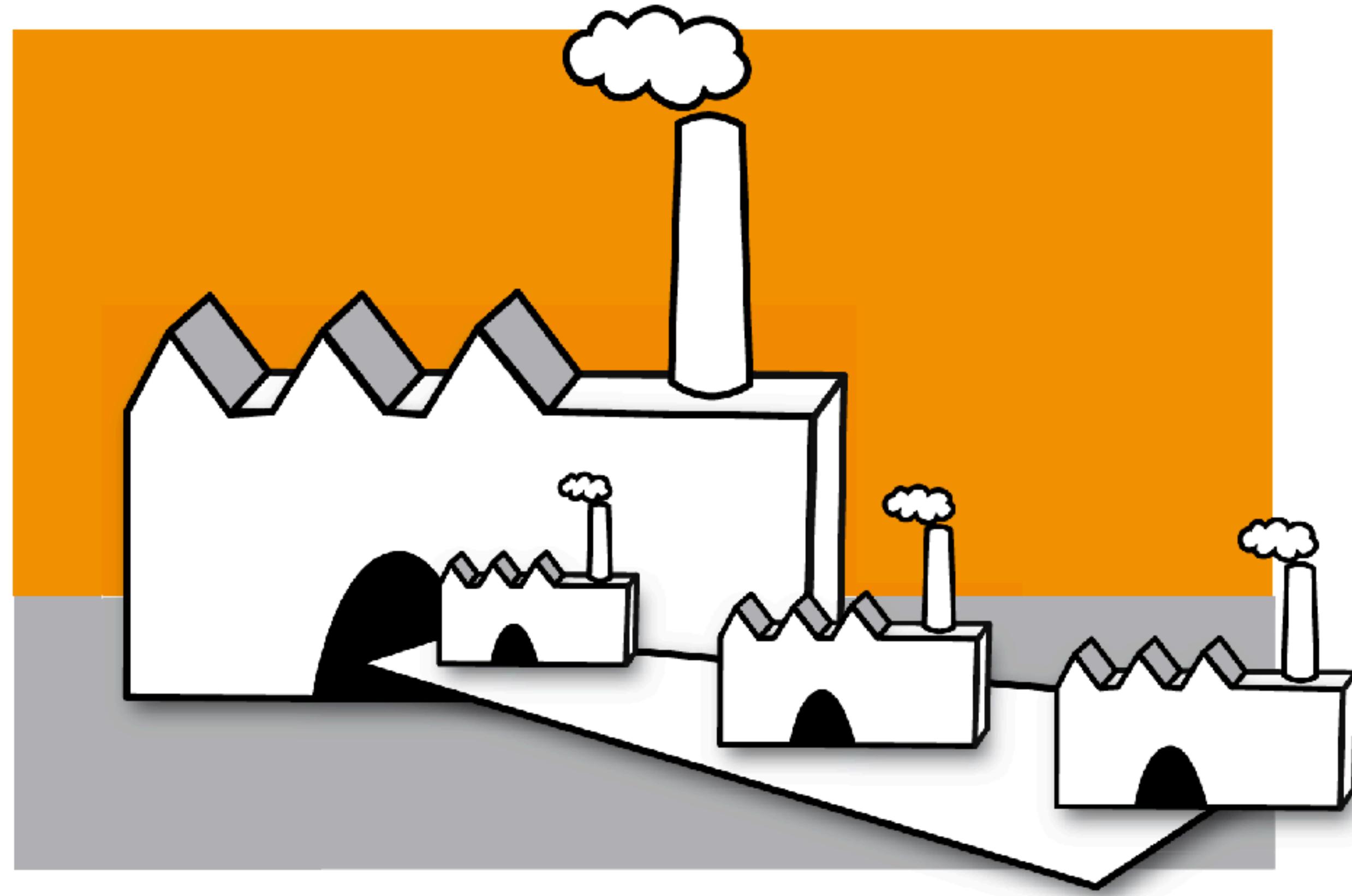


抽象工厂模式

通过一个接口创建一系列相关的对象，而不需要指定具体的类（由工厂完成）

抽象工厂没有if-else，是根据输入工厂类返回子类

- “工厂中的工厂”，可以看作是一个层级结构的“平台”，很多“商家”，提供很多“商品”
- 间接抽象类家族，而不是直接创建
- 抽象工厂干的单例的事情
- 扩展性好于工厂方法，避免条件判断，但复杂度也增加

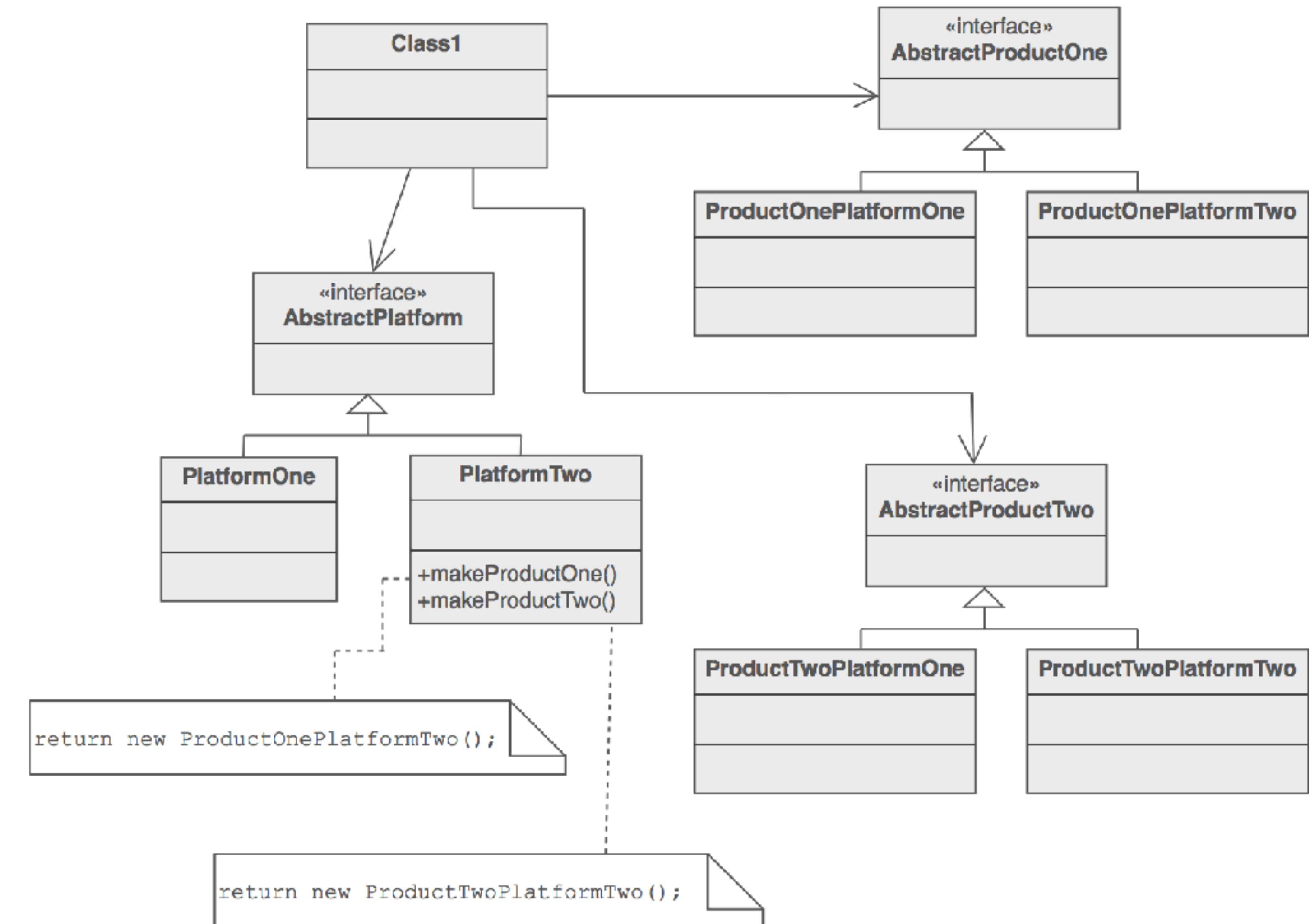
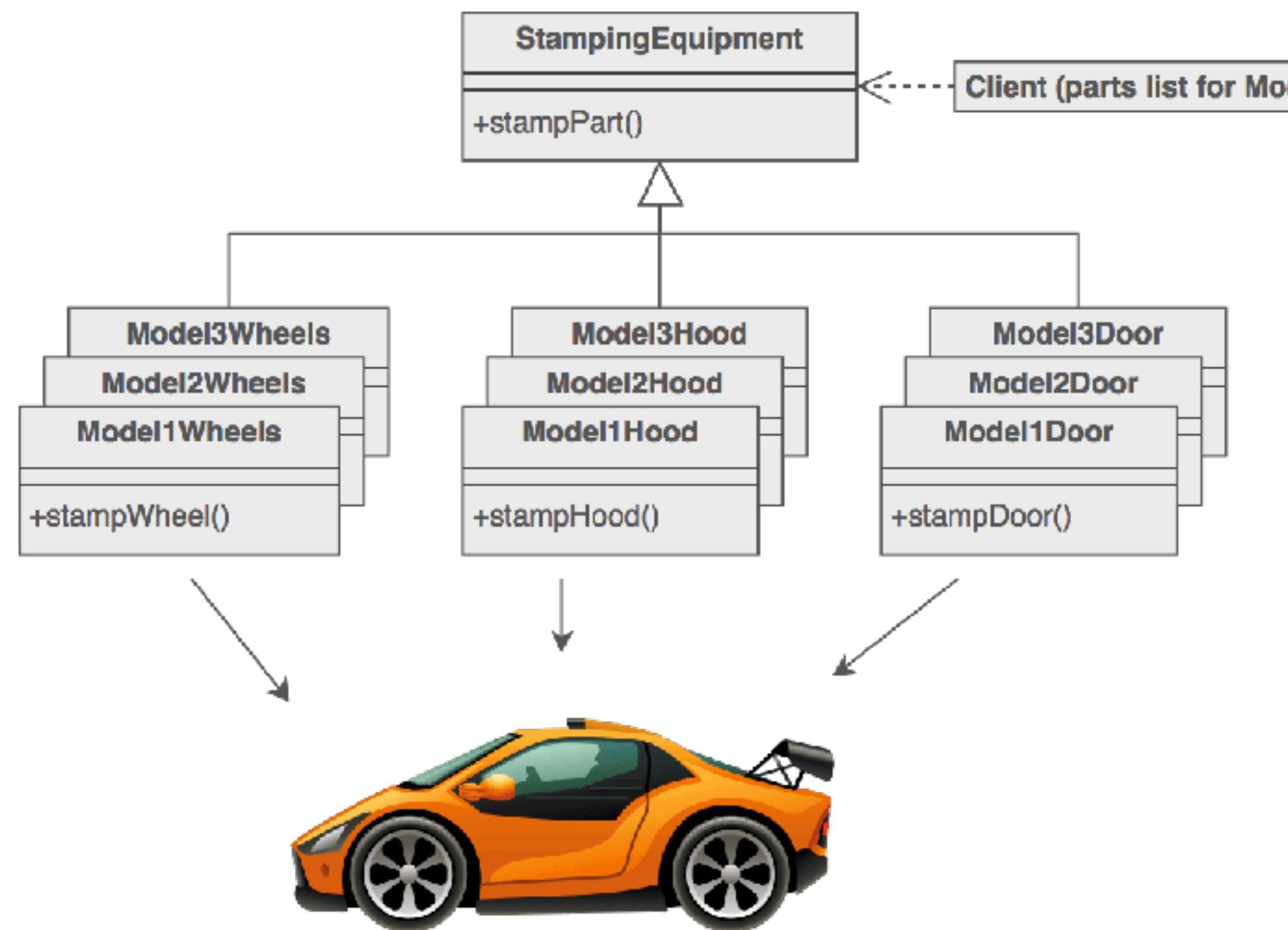


抽象工厂模式

JDK实例：

javax.xml.parsers.DocumentBuilderFactory,
javax.xml.transform.TransformerFactory,
javax.xml.xpath.XPathFactory的newInstance方法

案例：汽车冲压设备



构建模式

从具体的表现（what）中分离出复杂对象的构建过程（how）

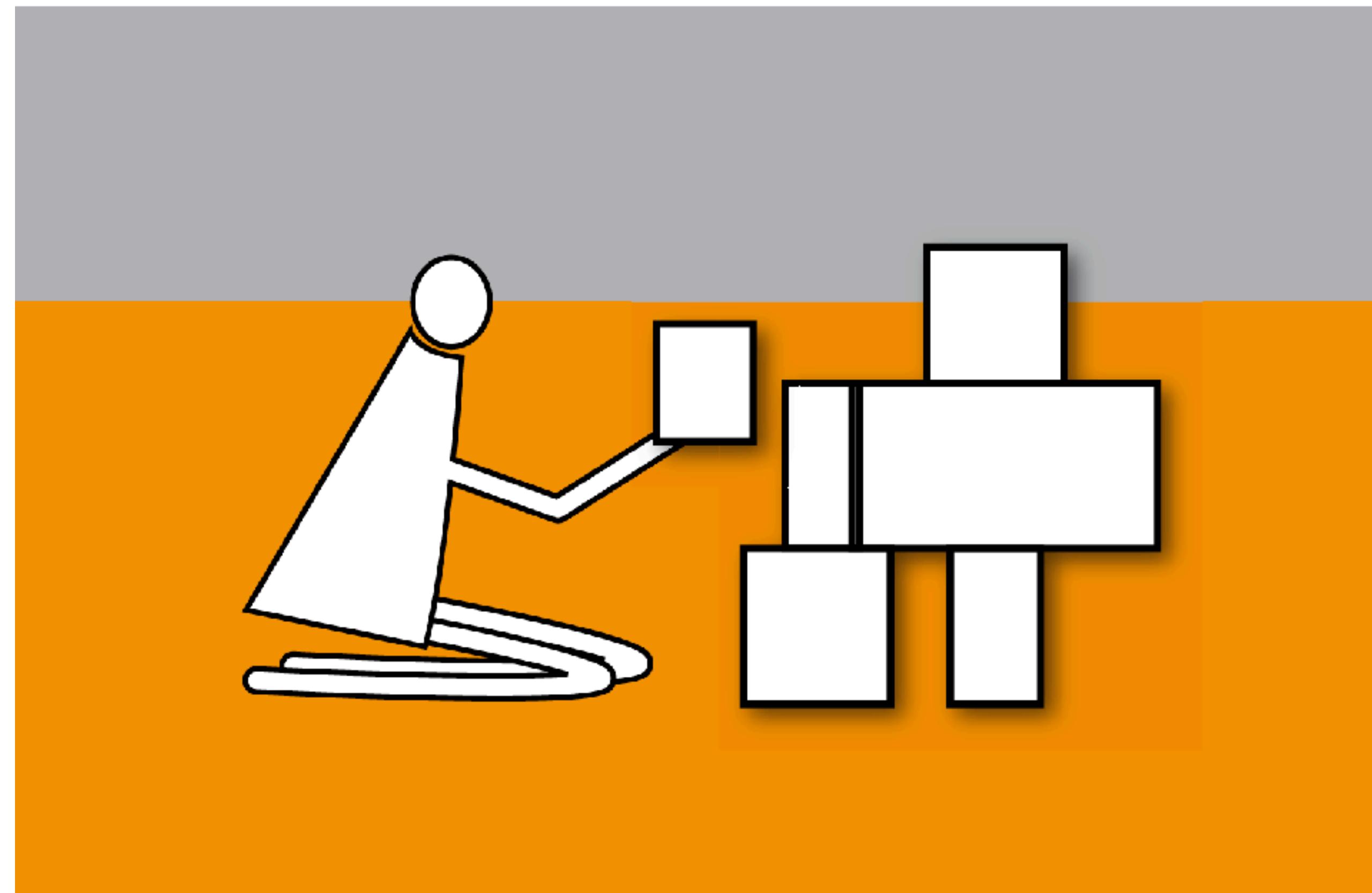
解决（抽象）工厂下当对象传递大量参数的问题

使用相同的构建过程，通过指定类型和内容创建不同的表现

从表现层中分离复杂对象的构建过程，重用对象构建过程

场景：一个输入，多个可能的输出

- 可选参数
- 按步骤创建
- 解耦表现层



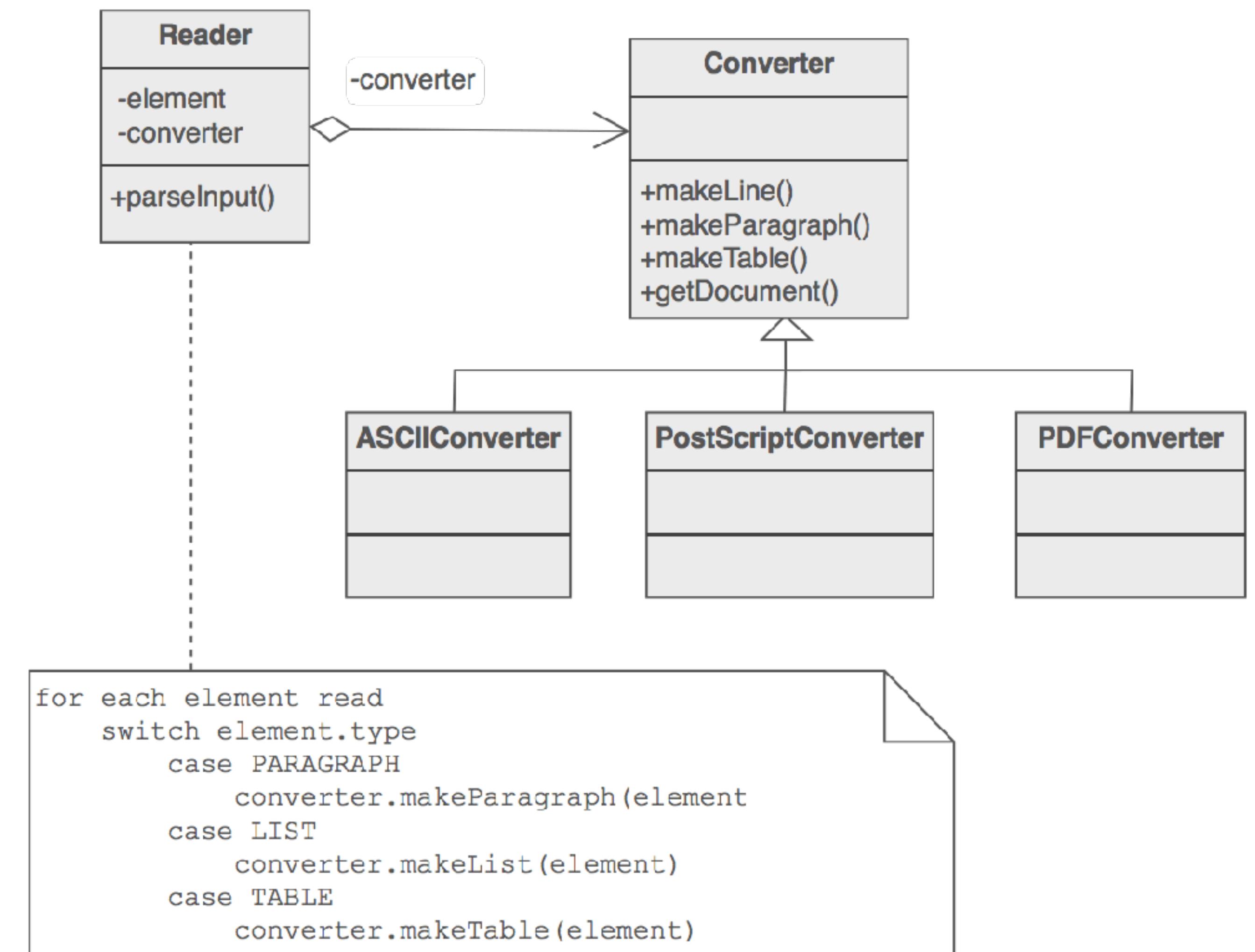
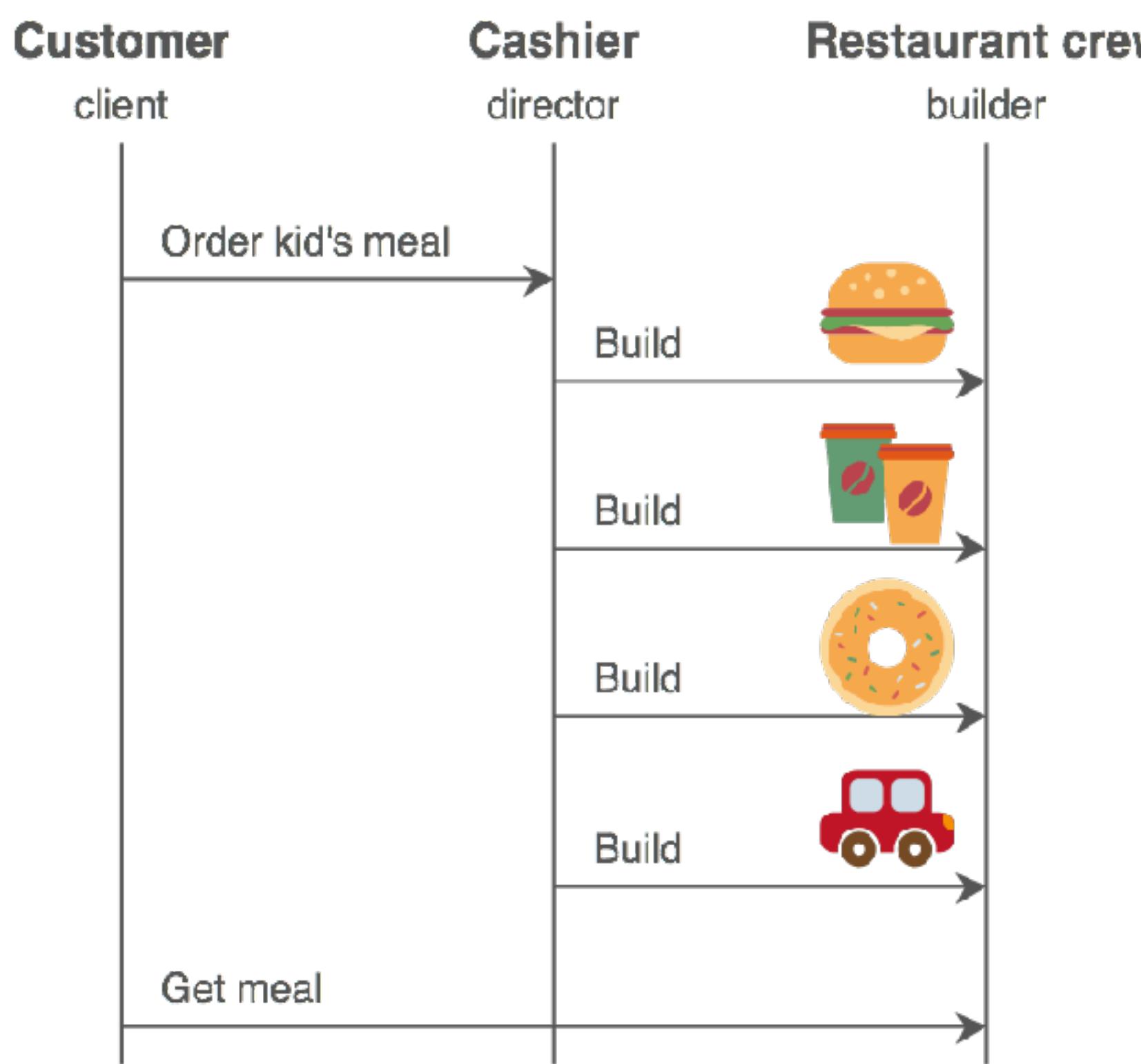
构建模式

JDK实例：

`java.lang.StringBuilder` (unsynchronized),

`java.lang.StringBuffer`(synchronized) 的append方法

案例：儿童快餐店



原型模式

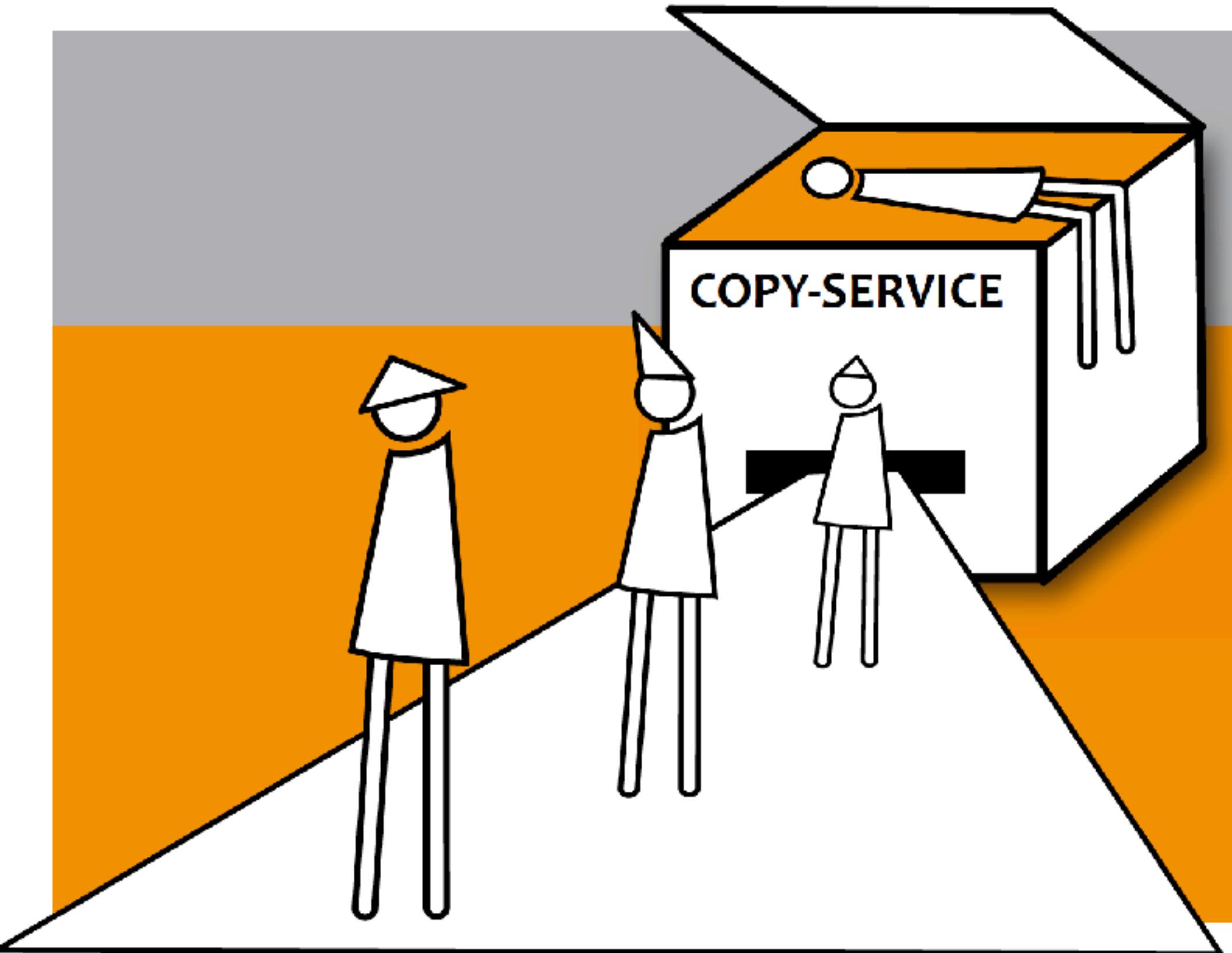
在基本对象的复制过程中产生变化

原型对象 → 新对象1, 2....n

通过克隆和复制一个实例化的对象来产生的对象

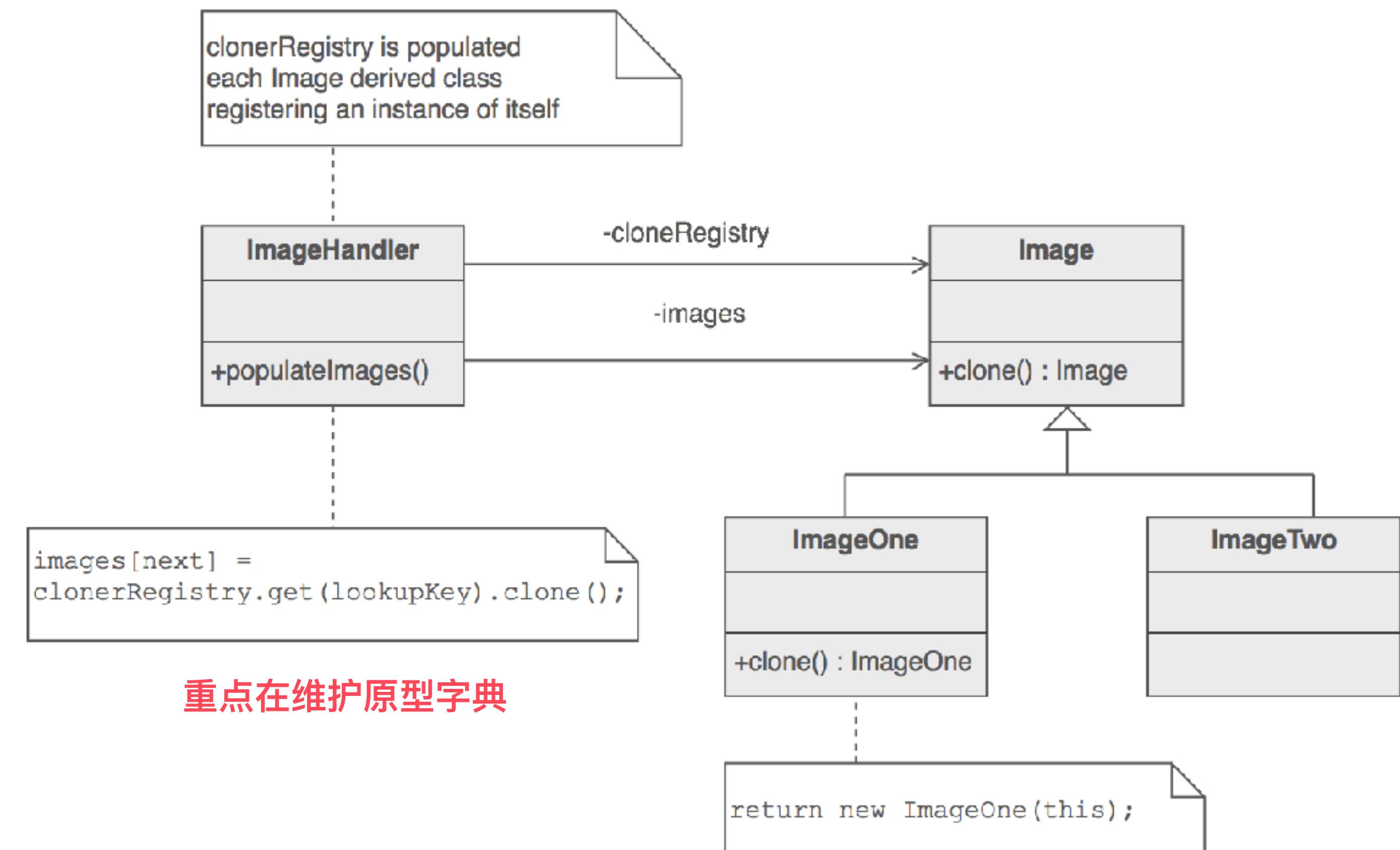
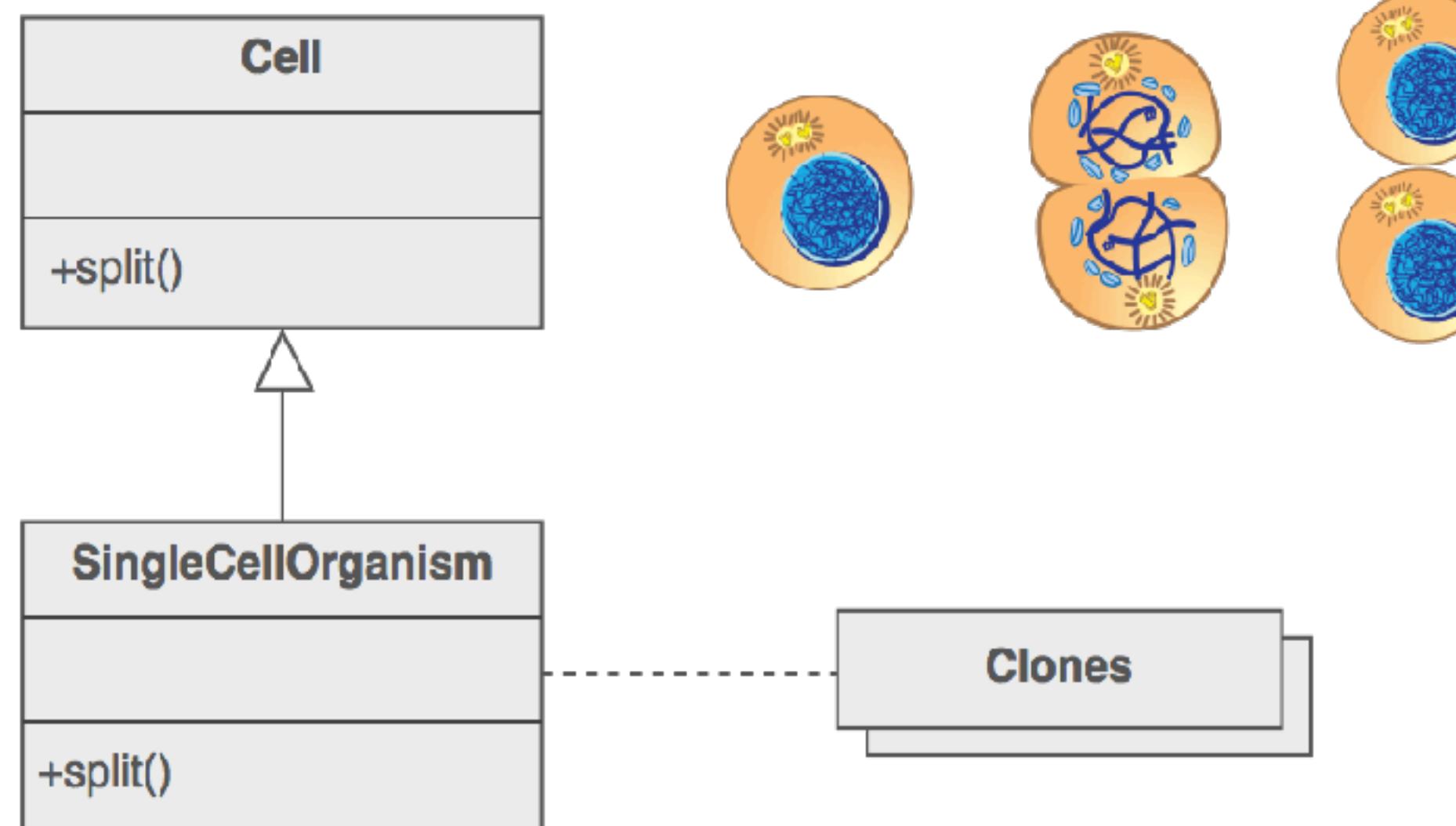
实现clone方法

场景：从数据库读出数据，并创建一个对象。如果需要创建多个类似对象，不能new，而是clone



原型模式

案例：细胞有丝分离



结构型模式



适配器模式

适配接口到另一个已存在的接口

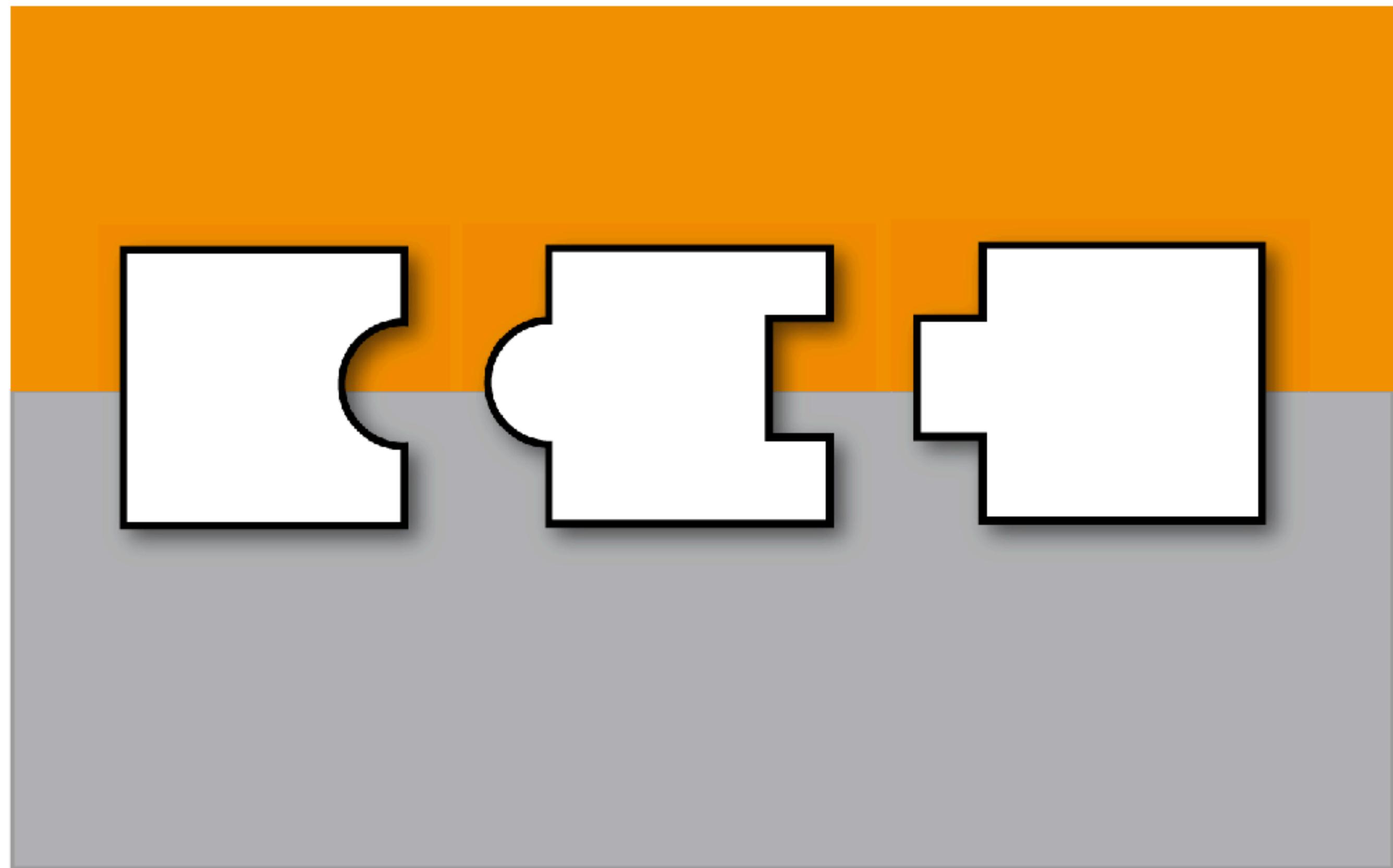
适配器（包装）：连接不相关接口的对象

目的：两个不相关的接口一起工作

解决新旧接口的兼容性问题

1. 类适配器：继承扩展原接口

2. 对象适配器：组合包含原对象



适配器模式

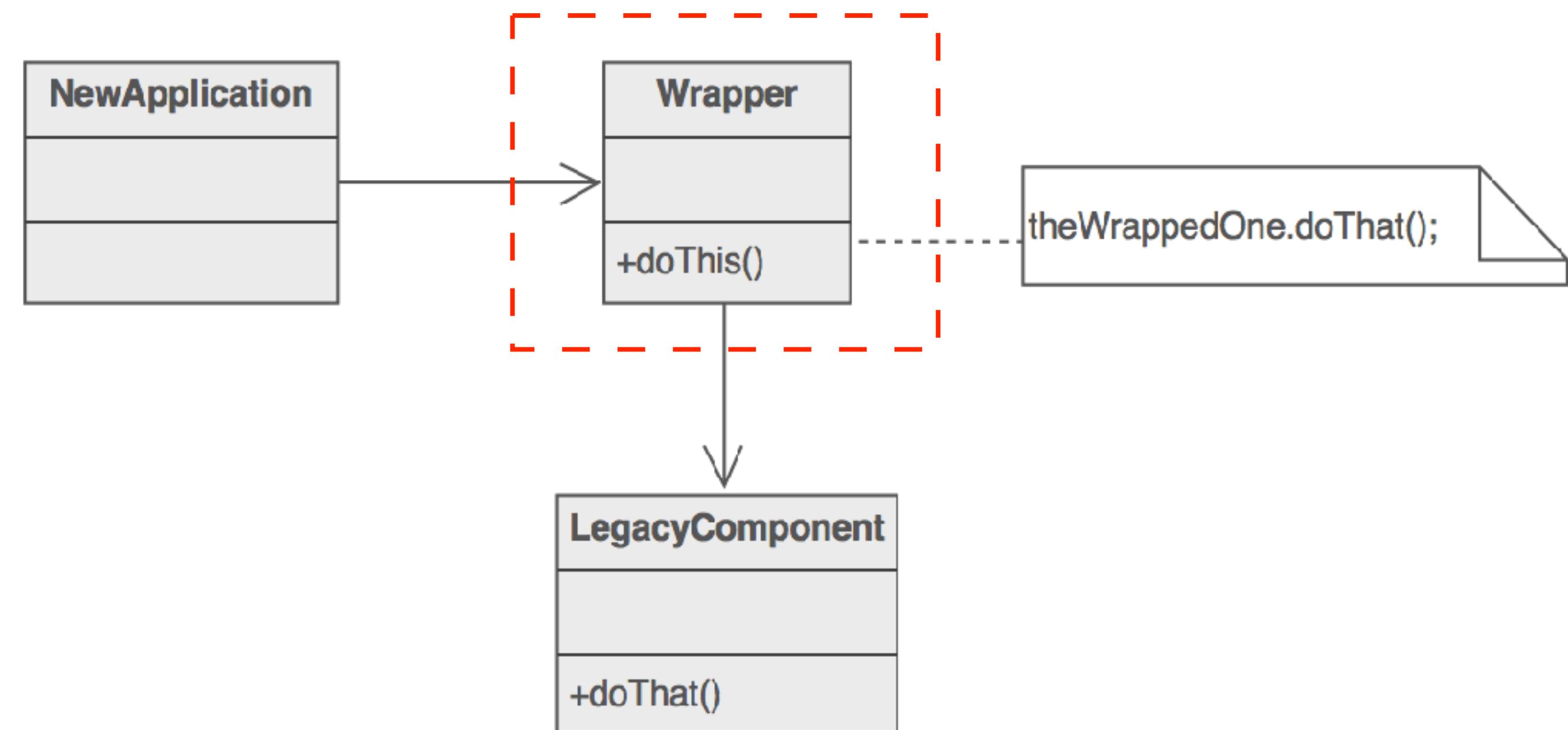
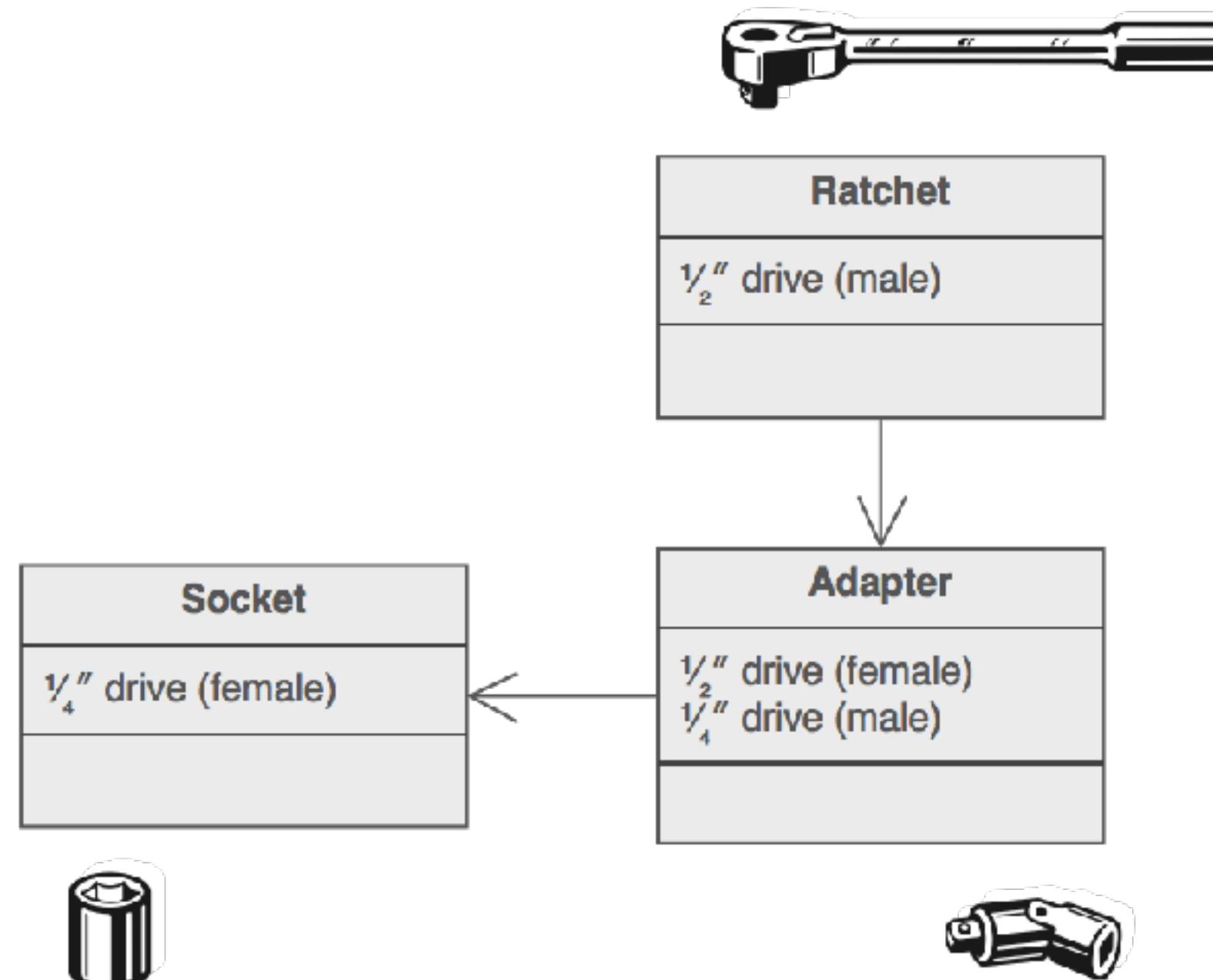
JDK实例：

java.util.Arrays的asList方法

java.io.InputStreamReader从InputStream返回Reader

java.io.OutputStreamWriter从OutputStream返回Writer

案例：套筒扳手



桥接模式

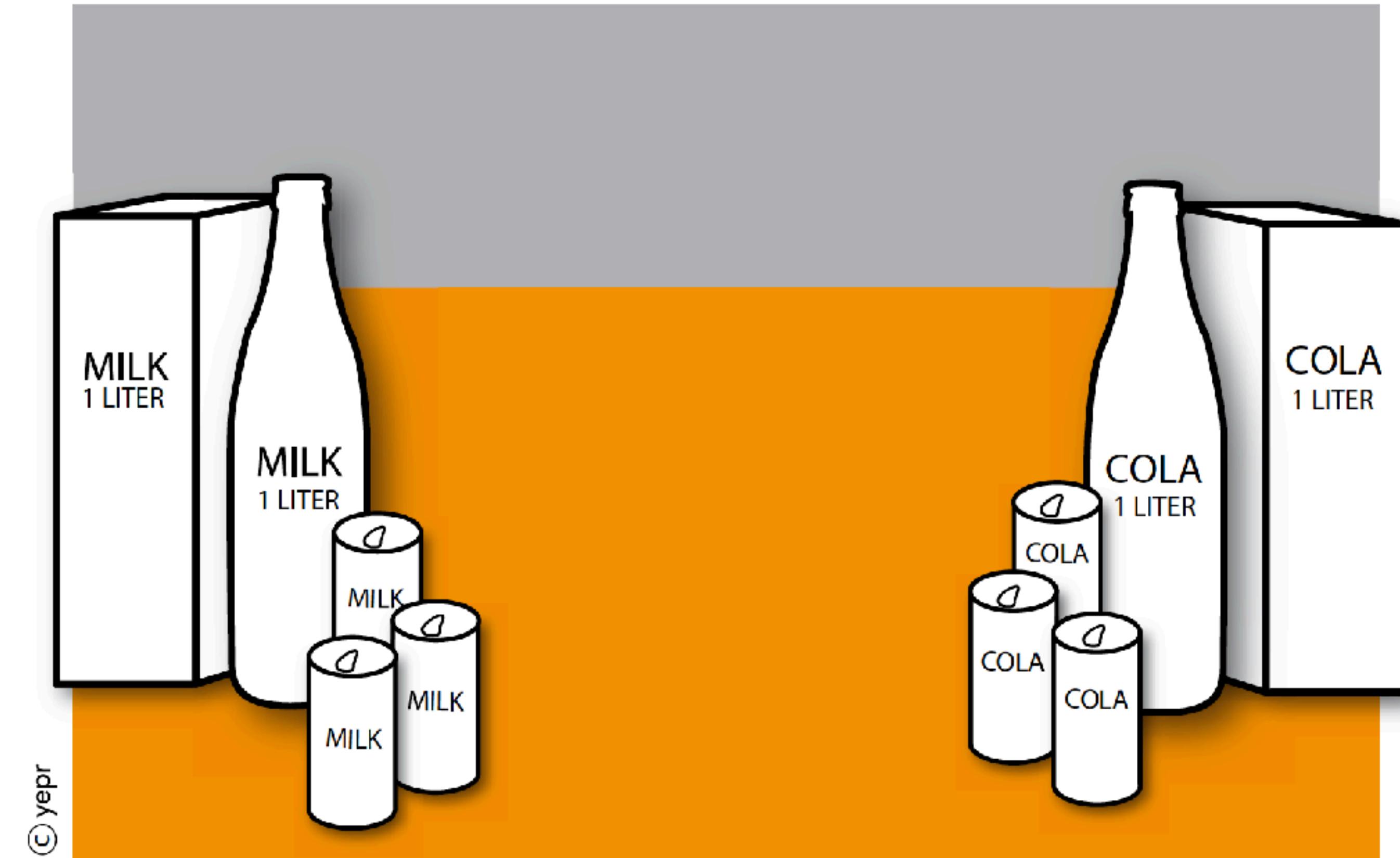
解耦抽象和它的实现，他们是独立不同的

通过组合而不是继承实现

桥接 = handle/body

重构指数级爆炸式的继承层级到正交层级，一个是平台独立的抽象，另一个是平台独立的实现

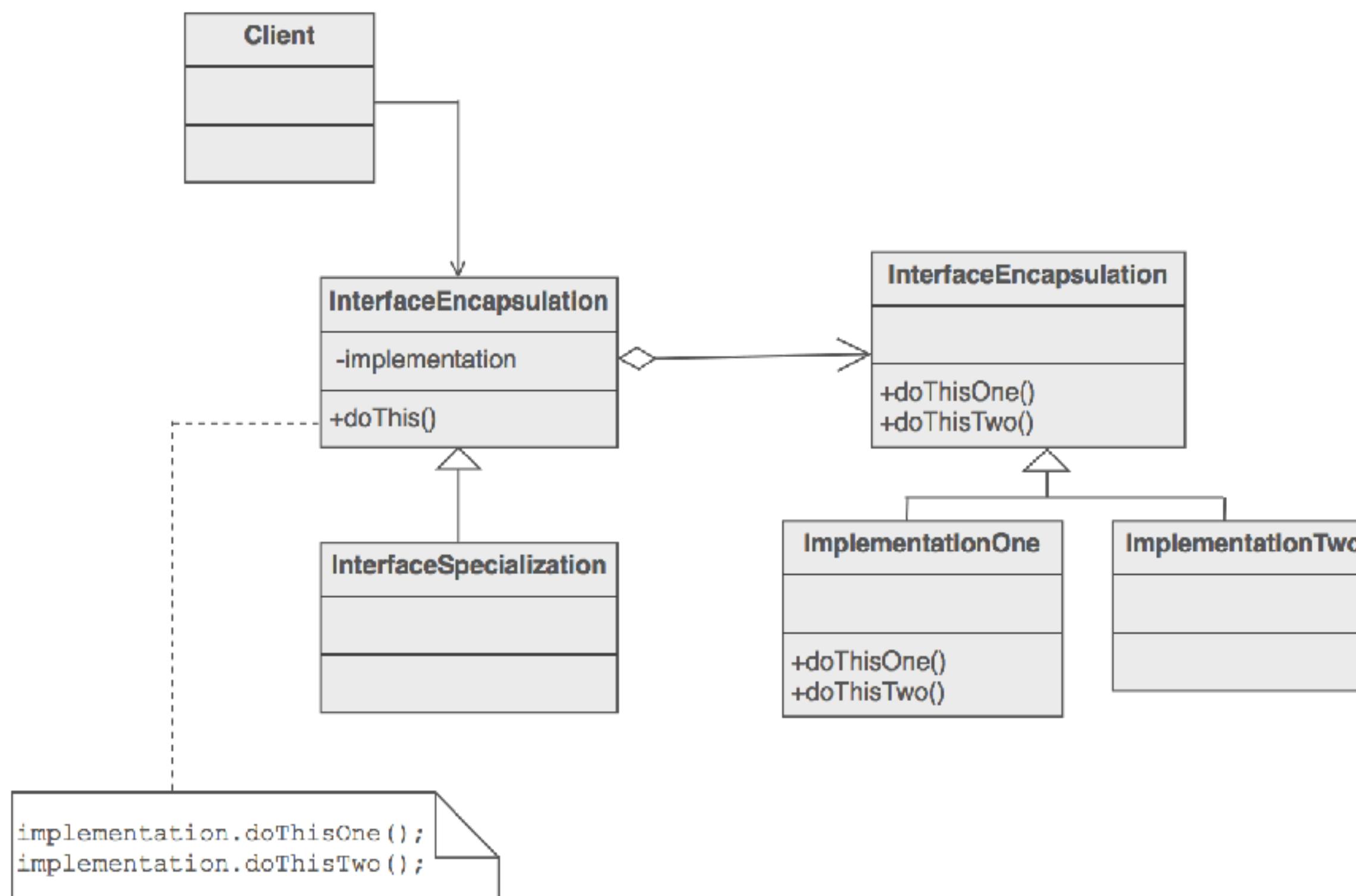
- 运行时绑定实现
- 由于接口耦合而导致的扩散式的类增长
- 共享一个类实现在多个对象中
- 改进扩展性



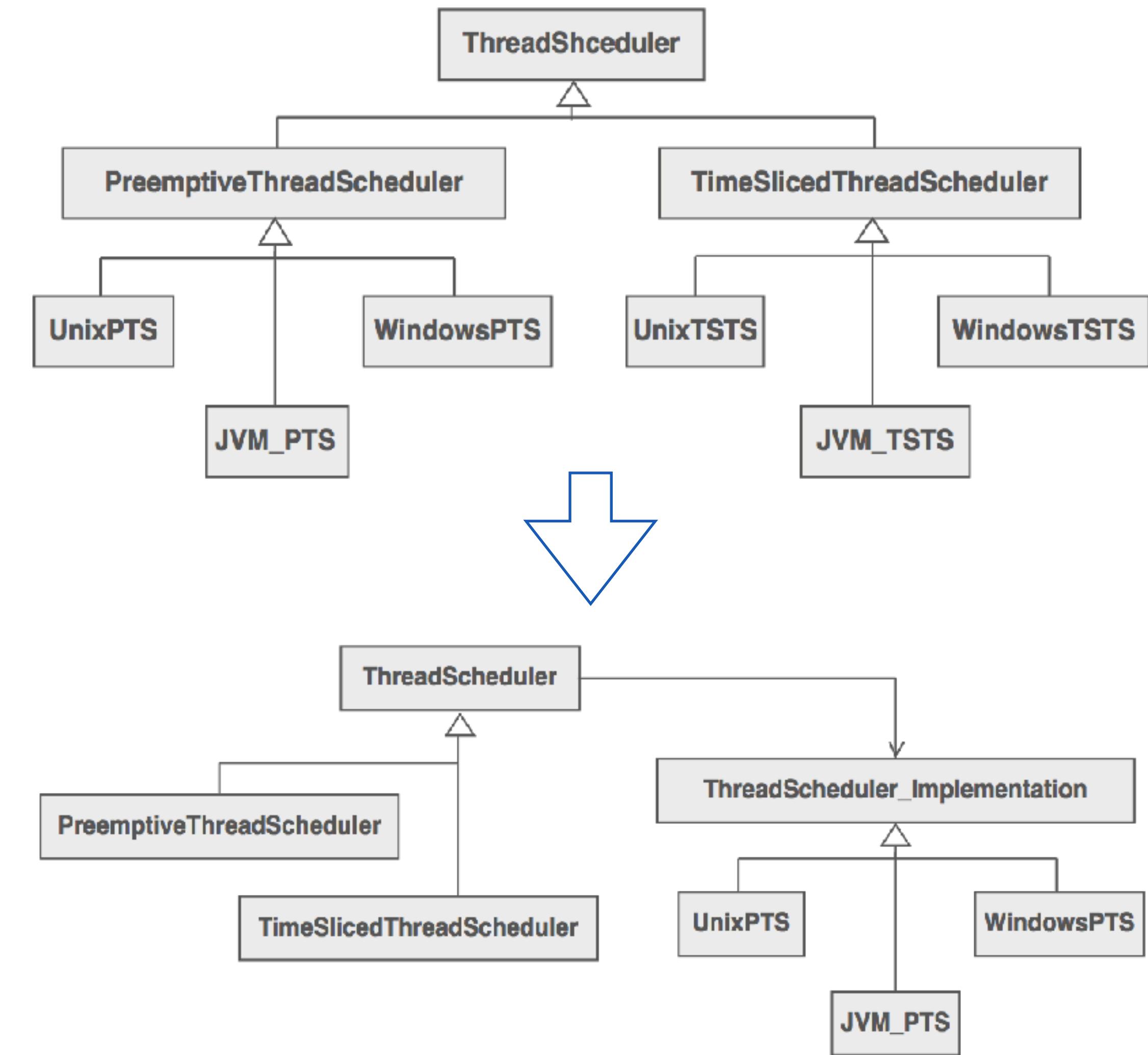
桥接模式

步骤：

1. 创建实现 (body) 抽象
2. 继承抽象实现独立的类
3. 创建接口 (handle) 代理到实现
4. 通过实现接口类来装饰接口类



```
implementation.doThisOne();
implementation.doThisTwo();
```



组合模式

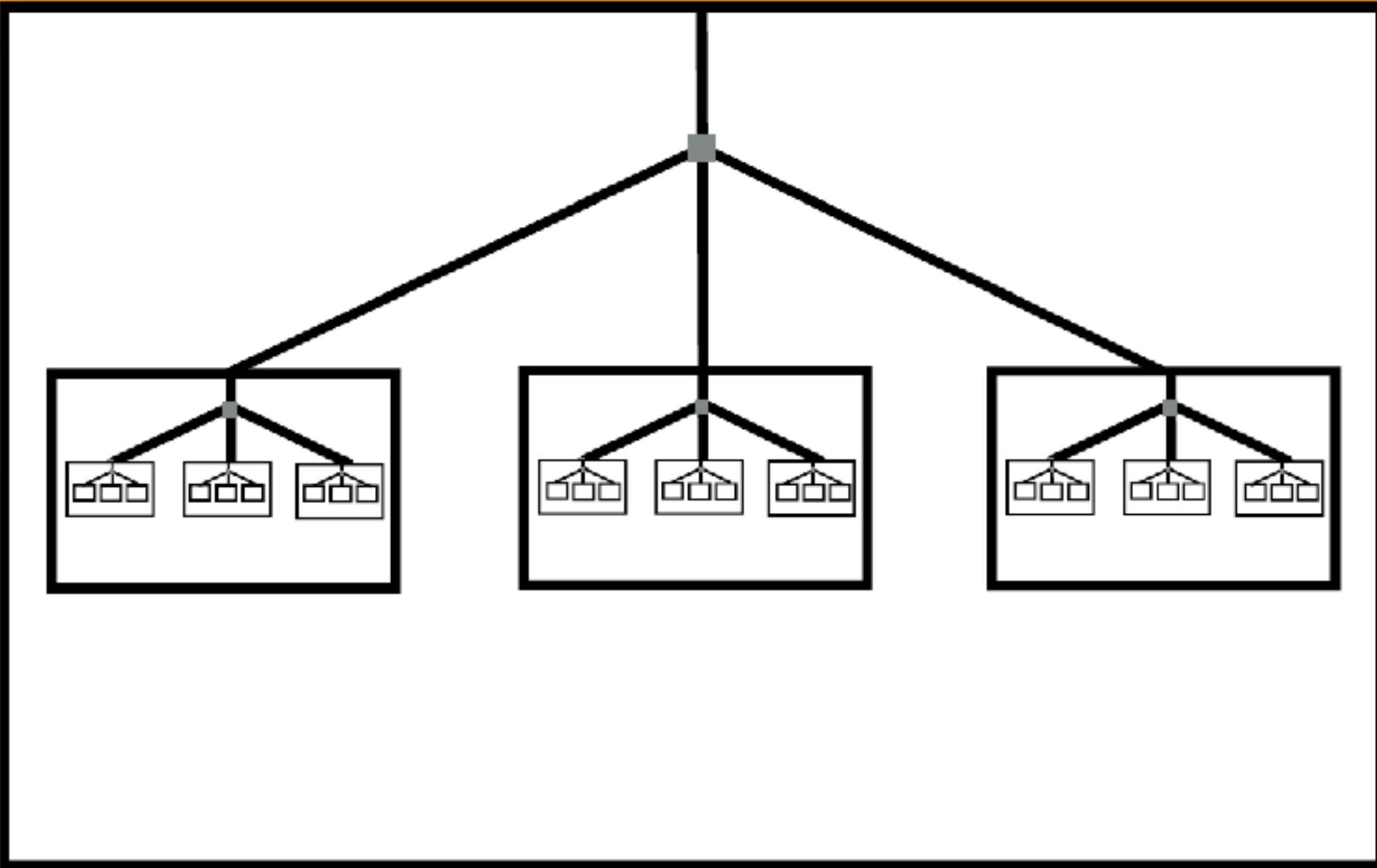
创建一个“整体-部分”层次的树型结构
每个节点都是树的一个部分，迭代下去

一对多，“has-a”关系

1. 基础组件：接口或抽象类，2/3继承于基础组件
2. 叶子结点：最后一级
3. 组合结点：包含组合结点或者叶子结点

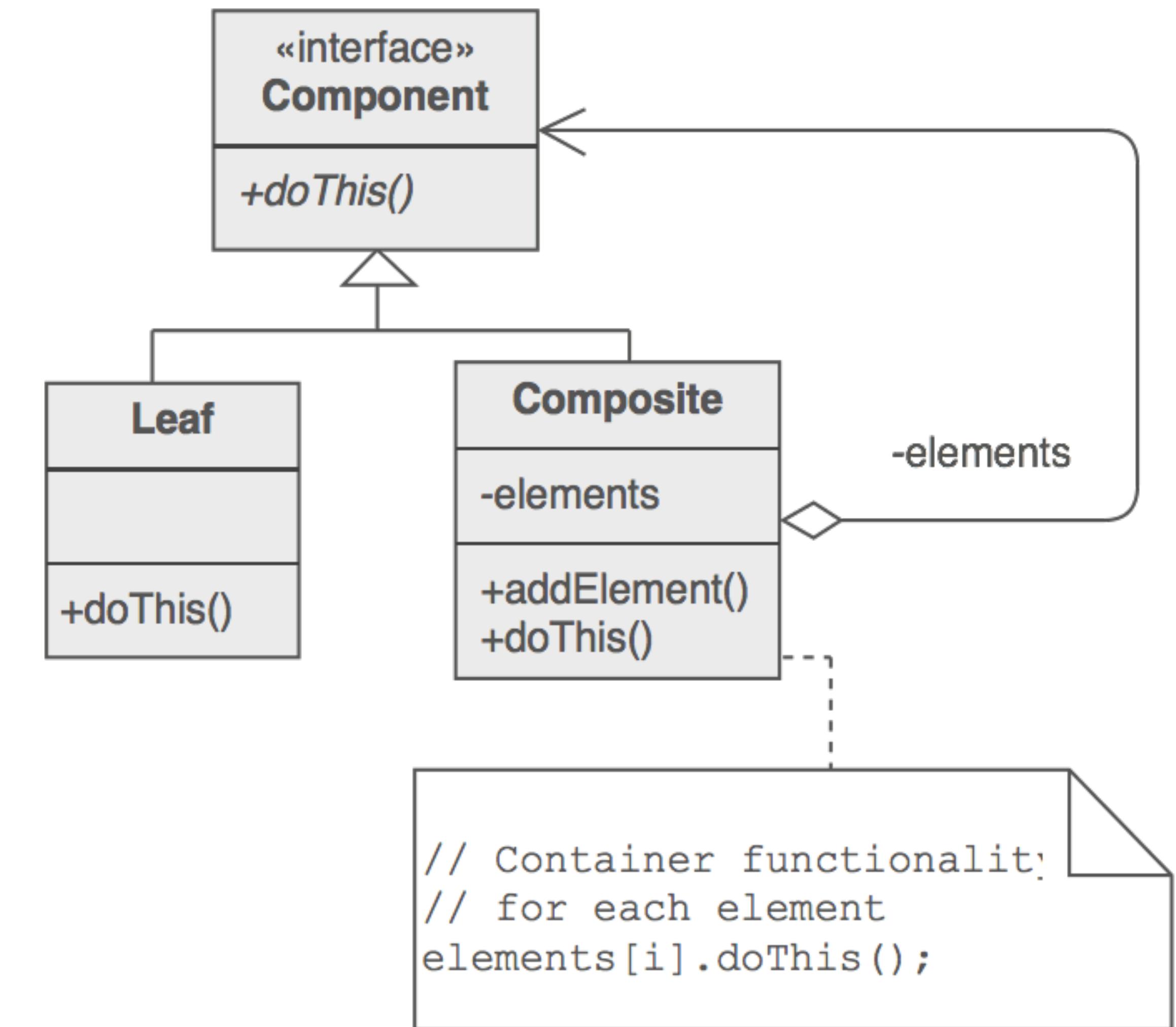
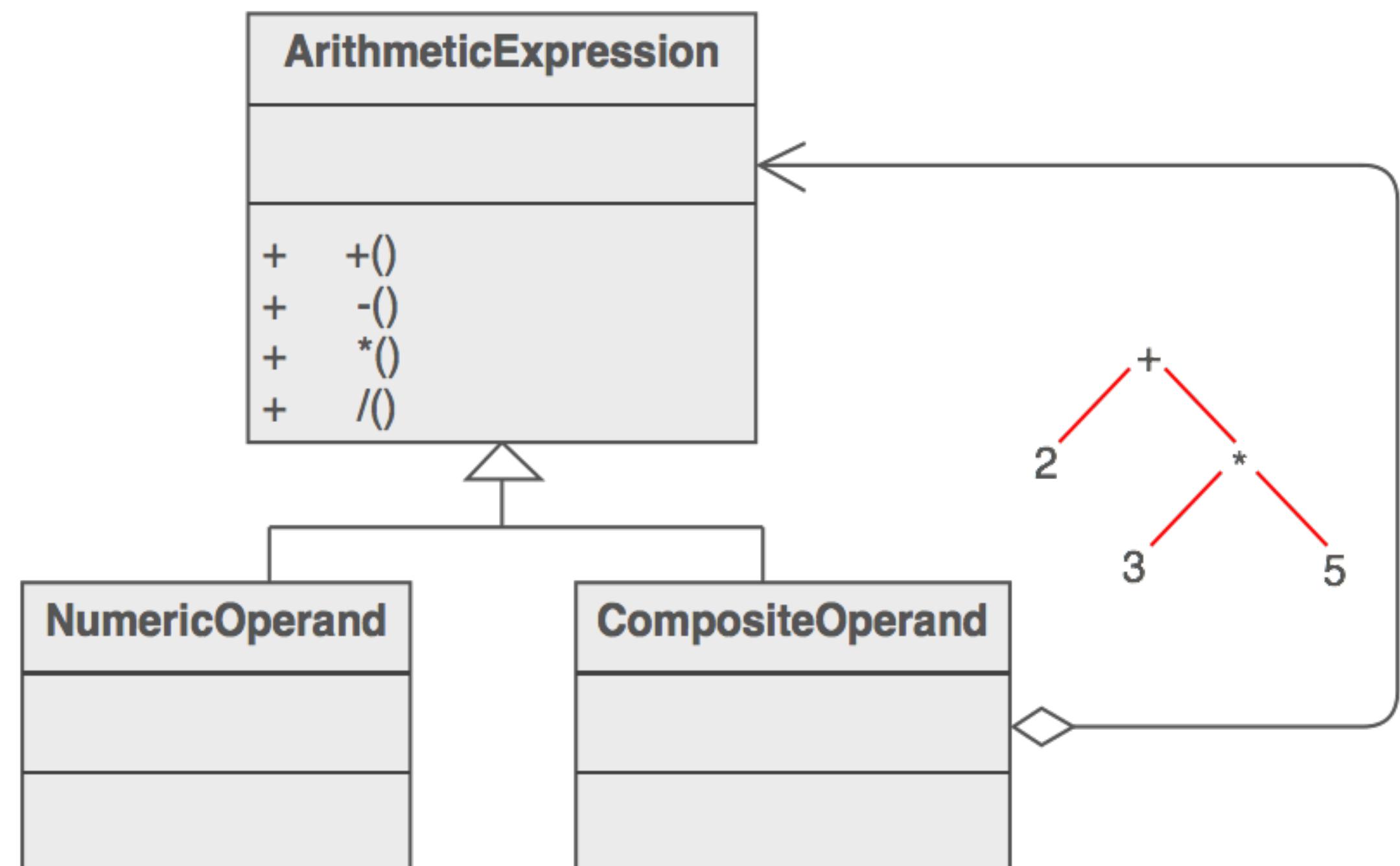
JDK实例：

java.awt.Container的add(Component)方法



组合模式

实例：代数运算



装饰器模式

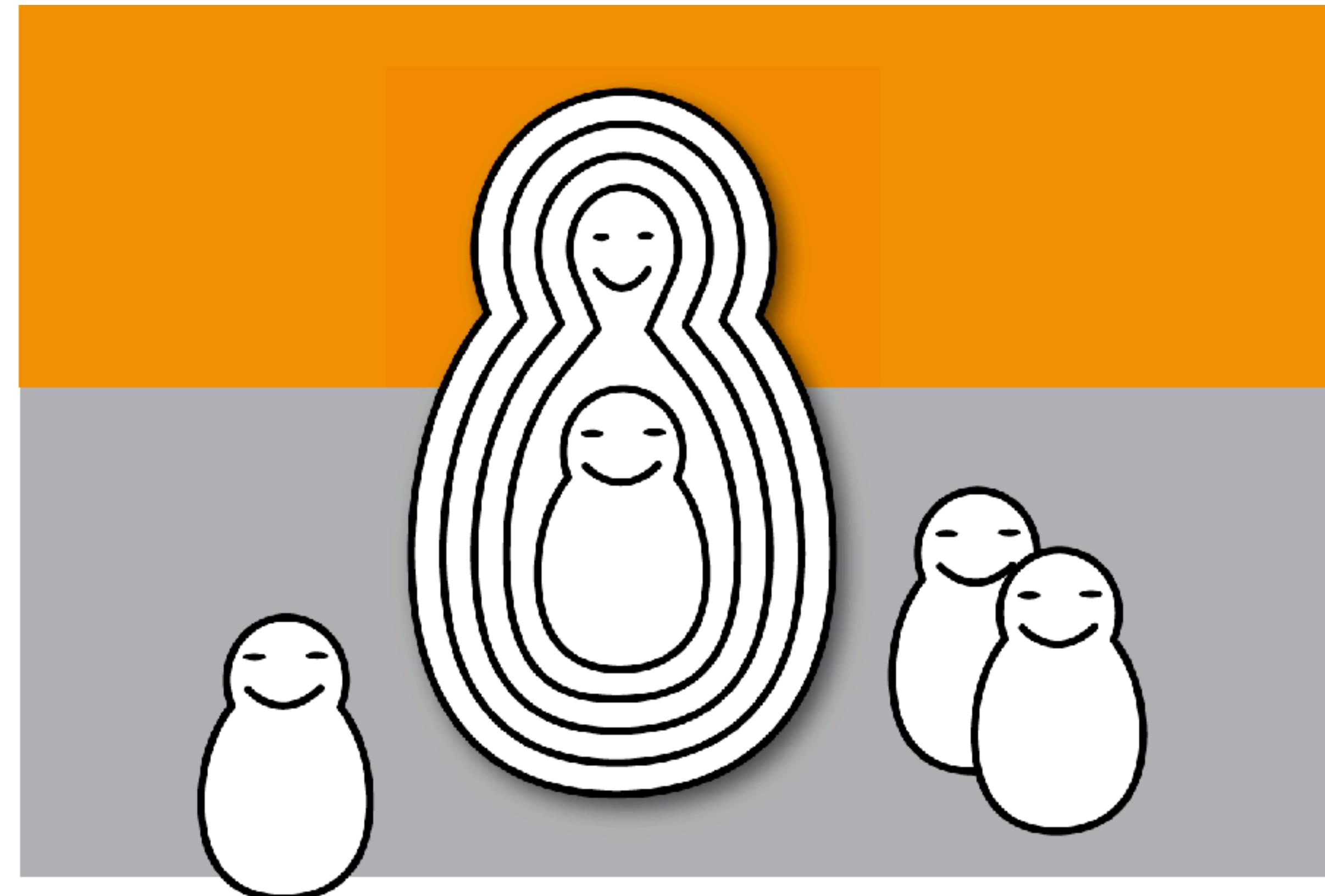
在运行时添加额外功能，同时保证原始接口不变

动态地为对象附加功能，装饰器为子类提供灵活扩展方法

运行时添加行为或者状态

“封装礼物，放入盒子，用彩色带子包装盒子”

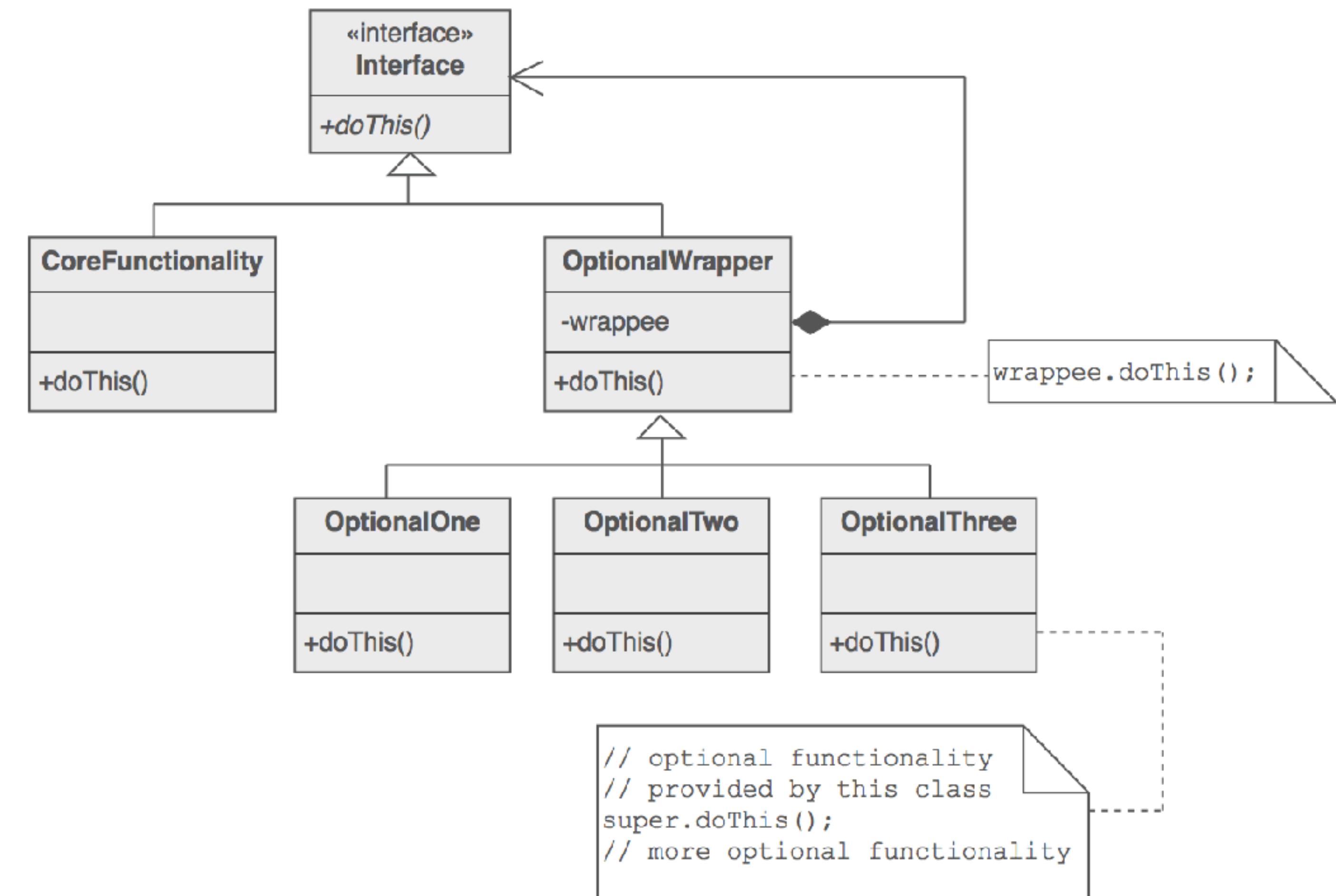
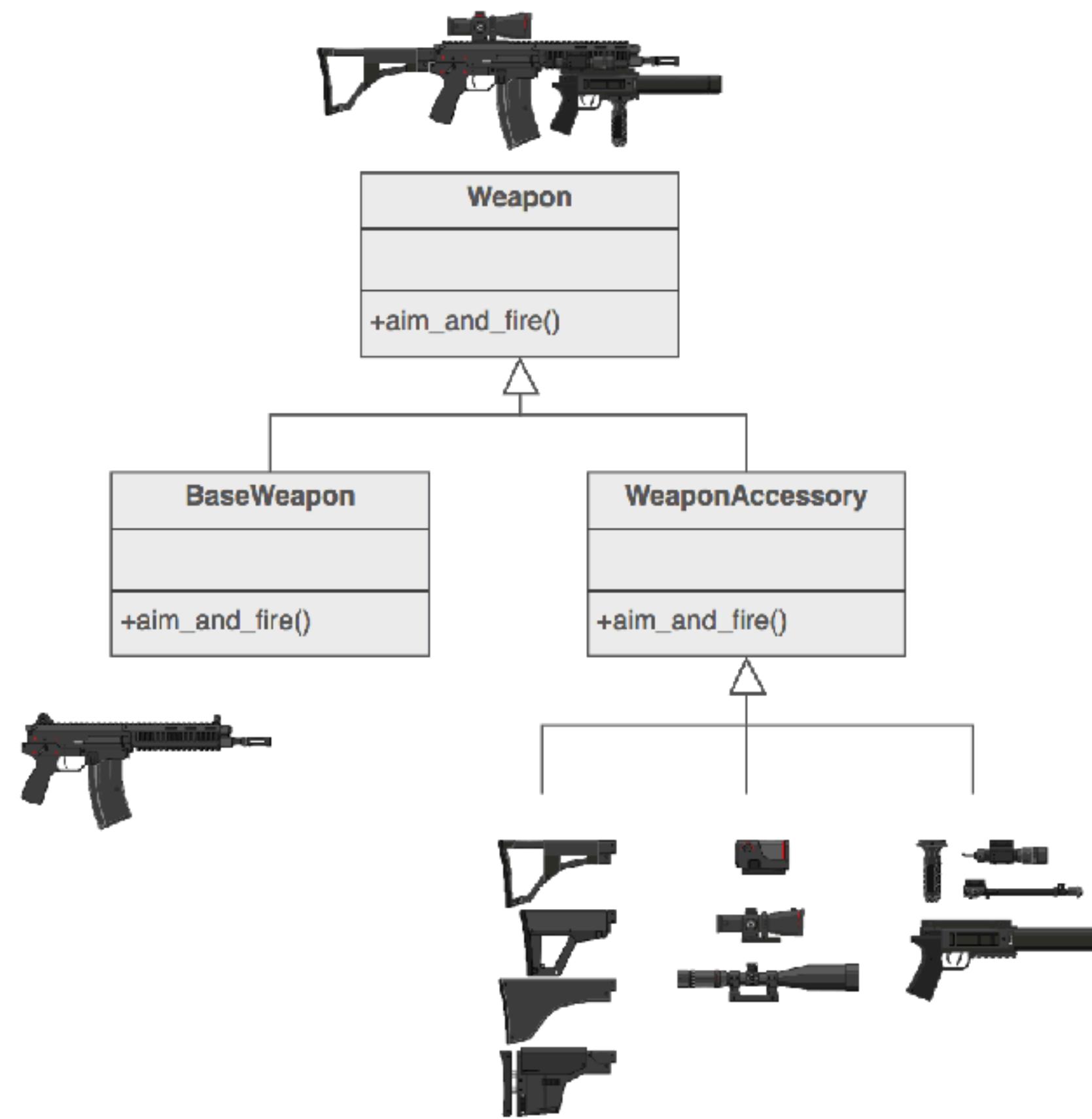
核心功能（必选） + 装饰器（可选）



装饰器模式

应用：java.io.FileReader, java.io.BufferedReader

实例：突击机枪



享元模式

使用类的一个实例提供很多“虚拟”实例

使用共享去支持大量细粒度的对象

“智能”拆分和加载

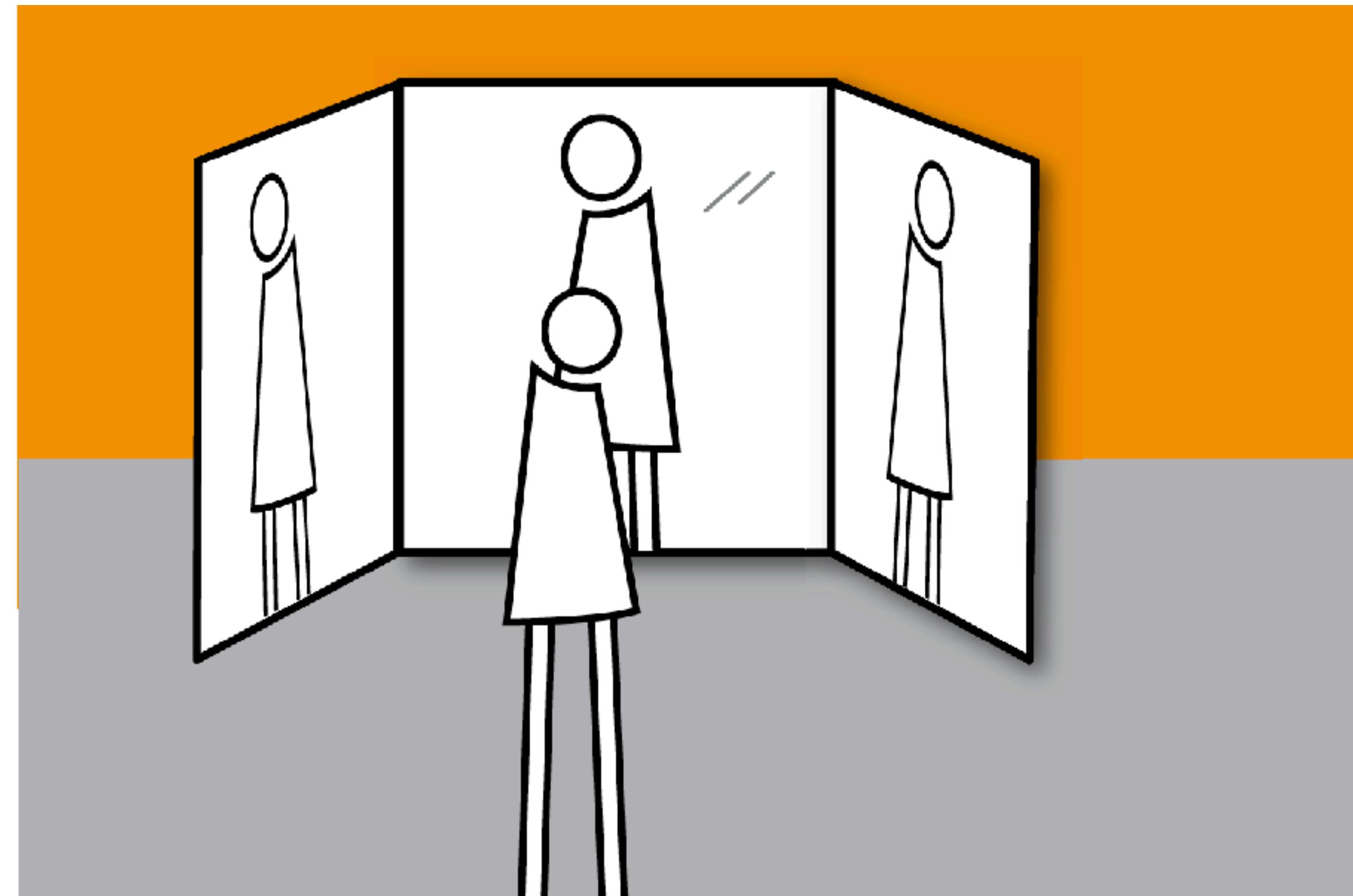
目的是将heavy-weight替换为light-weight对象

当需要创建大量对象，同时内存有限，那么通过共享对象
节约内存

对象可以分为本质属性（独立）和非本质属性（依赖）

本质属性存在享元对象（共享），而非本质属性存在客户
端对象，调用时传给享元对象

并不适合于本质属性过多的情况



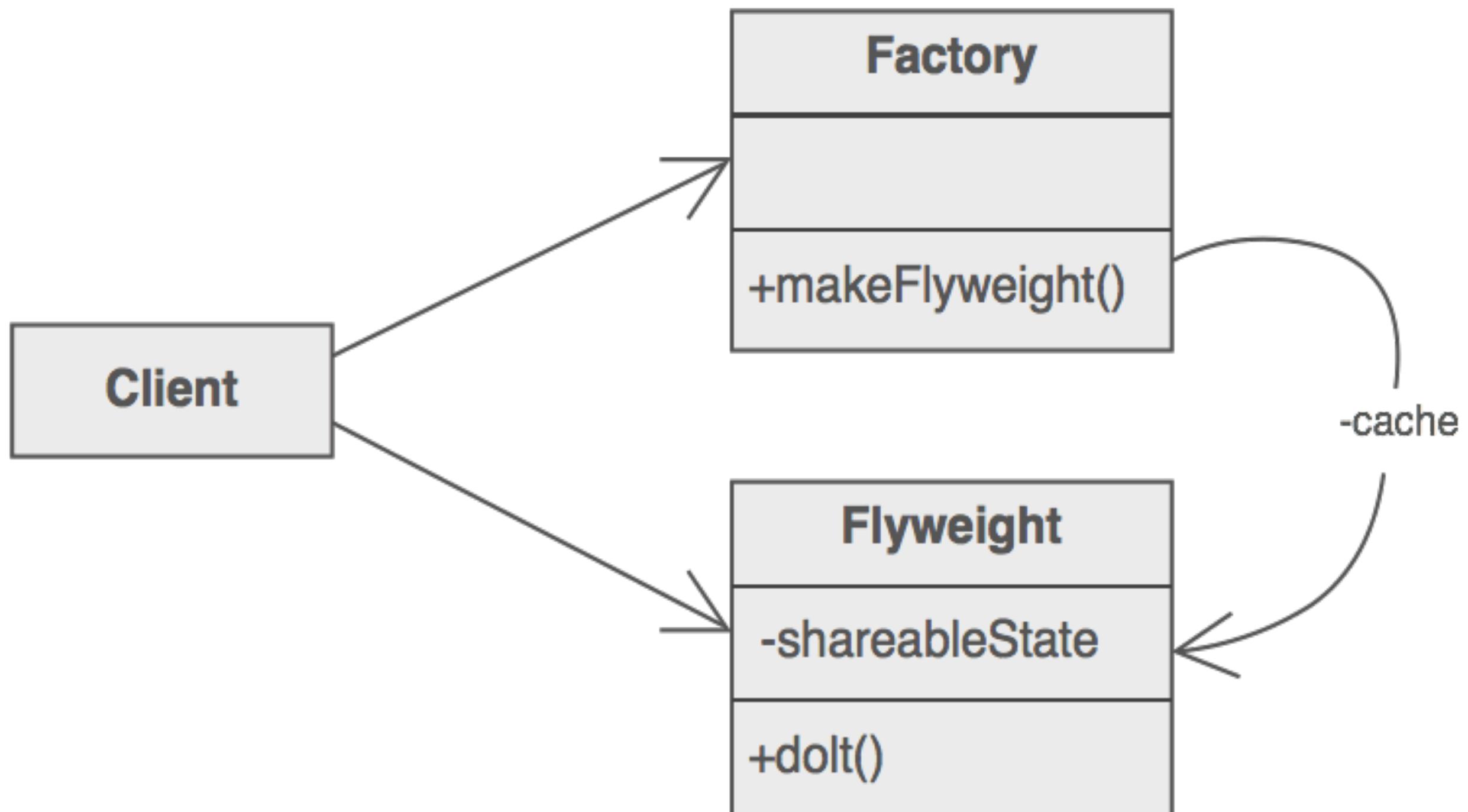
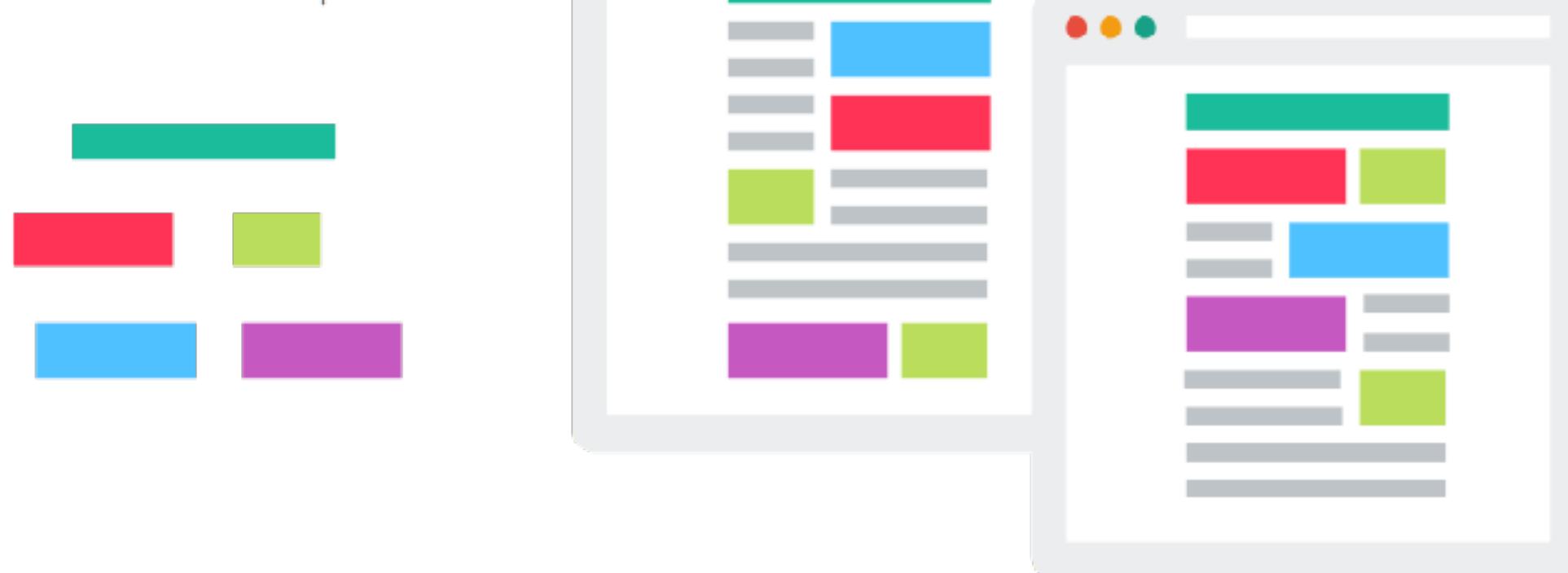
享元模式

- 享元工厂用于实例化对象。它使用数组缓存了可以共享的对象
- 若对象存在，直接返回；若对象不存在，创建一个包含本质属性的对象放进数组，返回对象

应用：`valueOf`方法使用缓存对象，
Java String Pool实现

实例：浏览器缓存

Browser loads images
just once and then
reuses them from pool:



门面模式

为一组接口提供一个通用的，更简单的接口

为很多子系统提供一个统一的接口

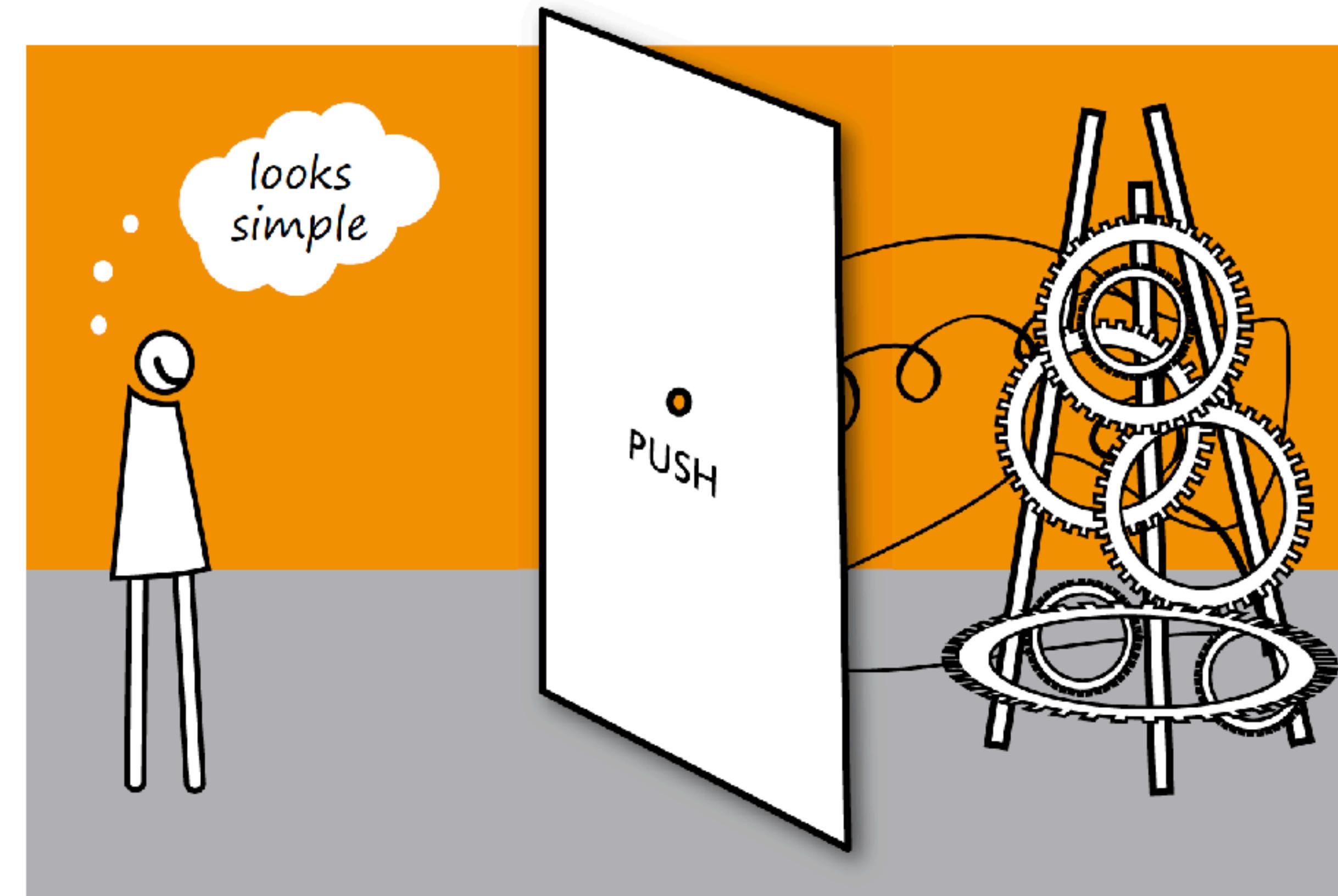
提供高层接口，使子系统更易用，减少子系统的学习曲线

将复杂系统封装成一个更简单的接口

客户仅仅是门面接口，不接触内部接口

限制“超级用户”的能力，无法接触内部接口

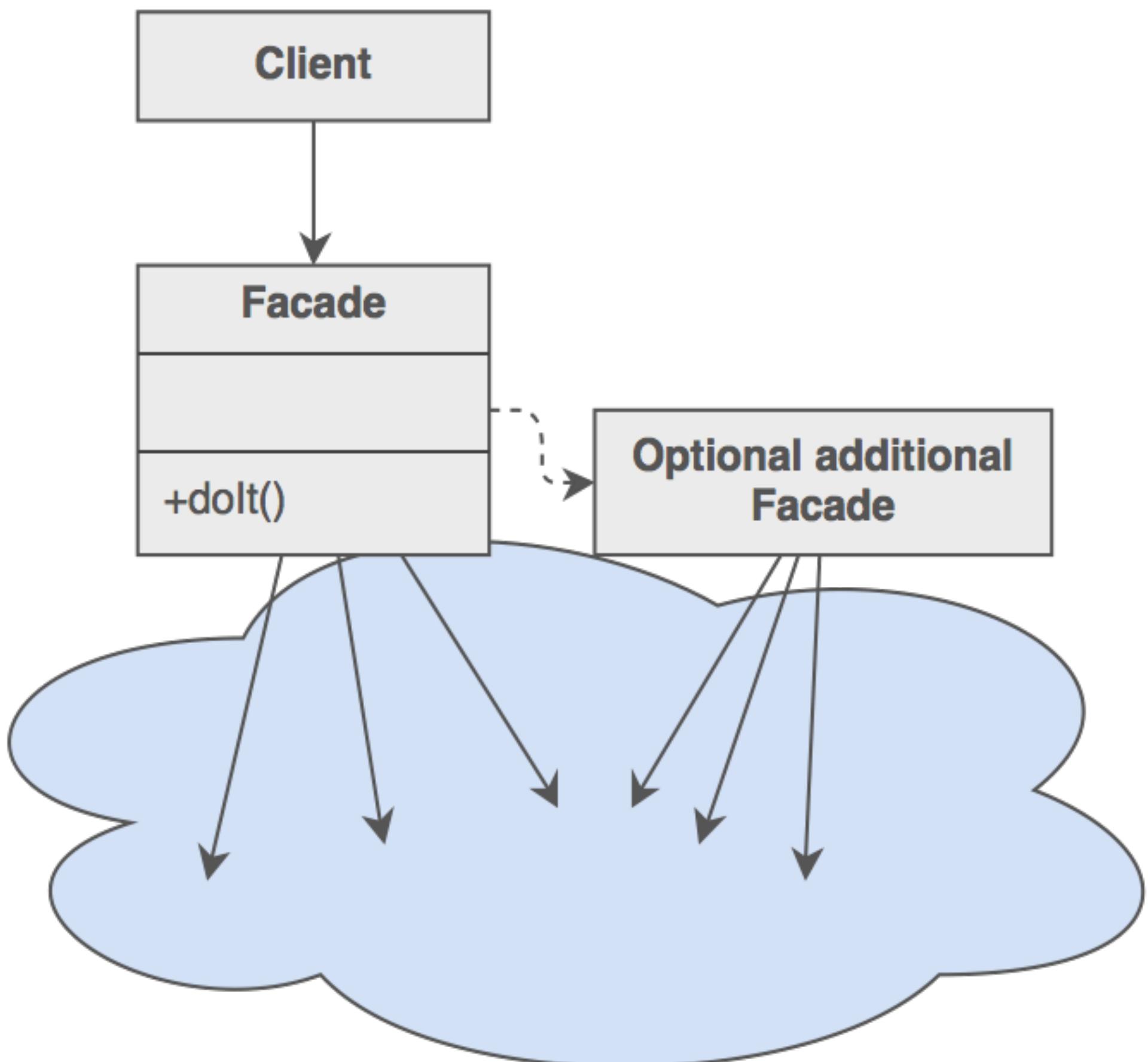
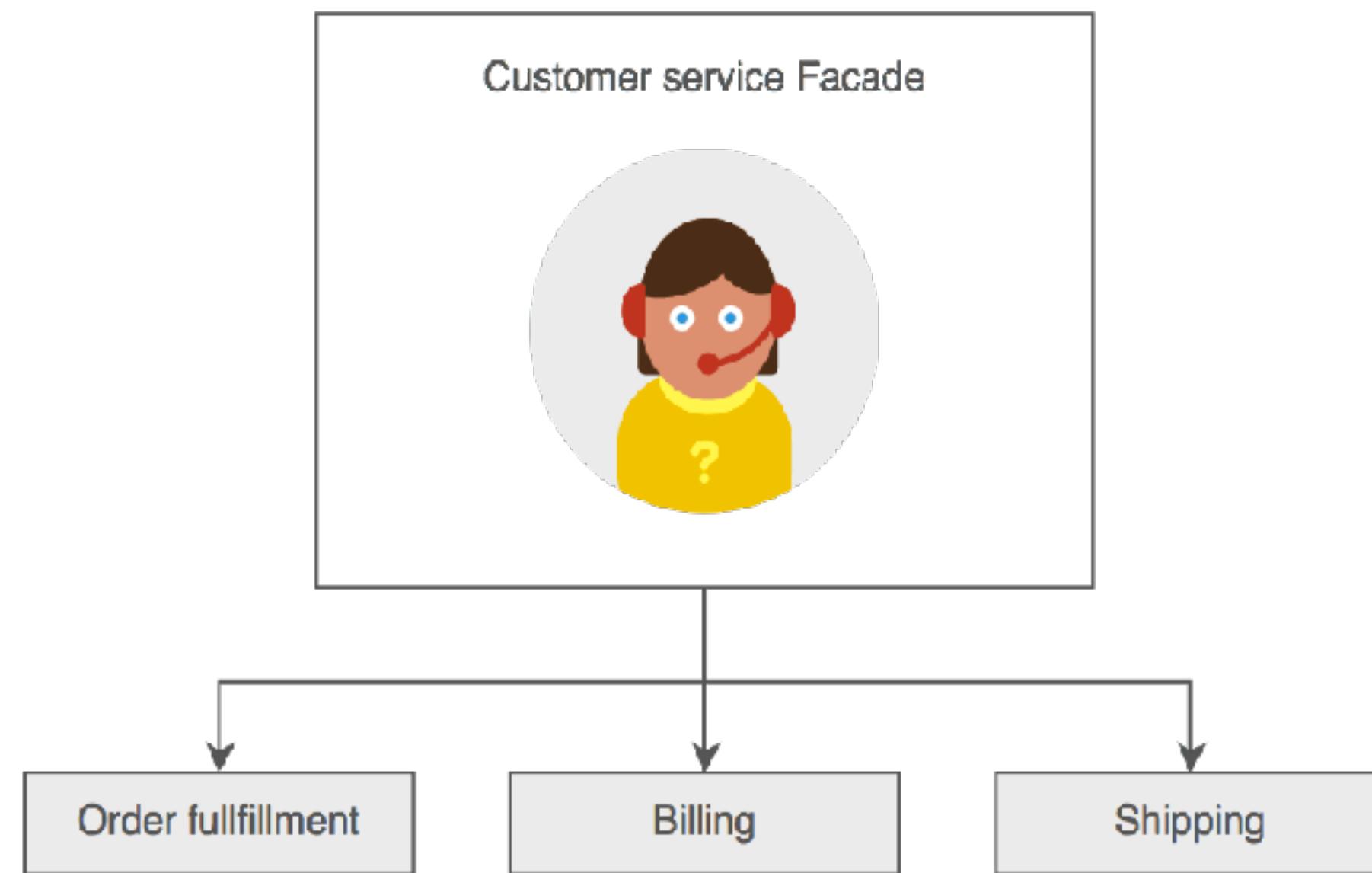
“上帝”对象



门面模式

- 适配器基于老接口，目标是使他们可以相互调用
门面模式则是创建新的接口
- 享元模式用于缓存很多共享对象，而门面通过一个接口代表整个系统

实例：客服系统



代理模式

为对象提供代理，用来替代直接访问该对象

间接实现分布式，访问控制，安全保护和智能访问需求

场景：当客户访问代理时，代理实例化真实对象，然后将客户请求传递给真实对象。

1. 虚拟代理：按需创建“昂贵”对象
2. 远程代理：存在于多个地方的对象的本地缓存 e.g. RPC
3. 保护代理：访问控制 e.g. 权限，安全性
4. 智能代理：自动载入资源（有访问）和释放资源（无访问），安全锁

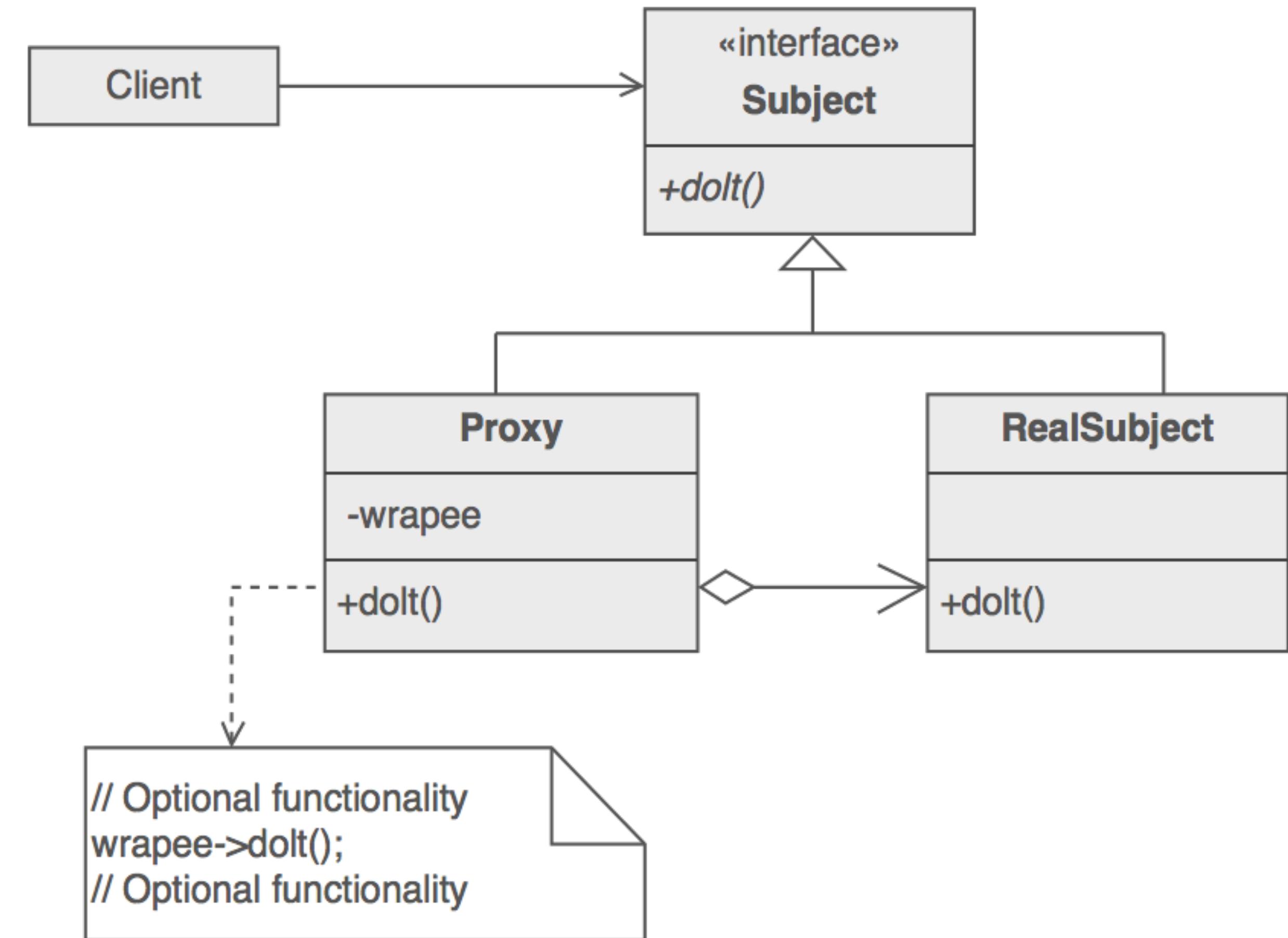
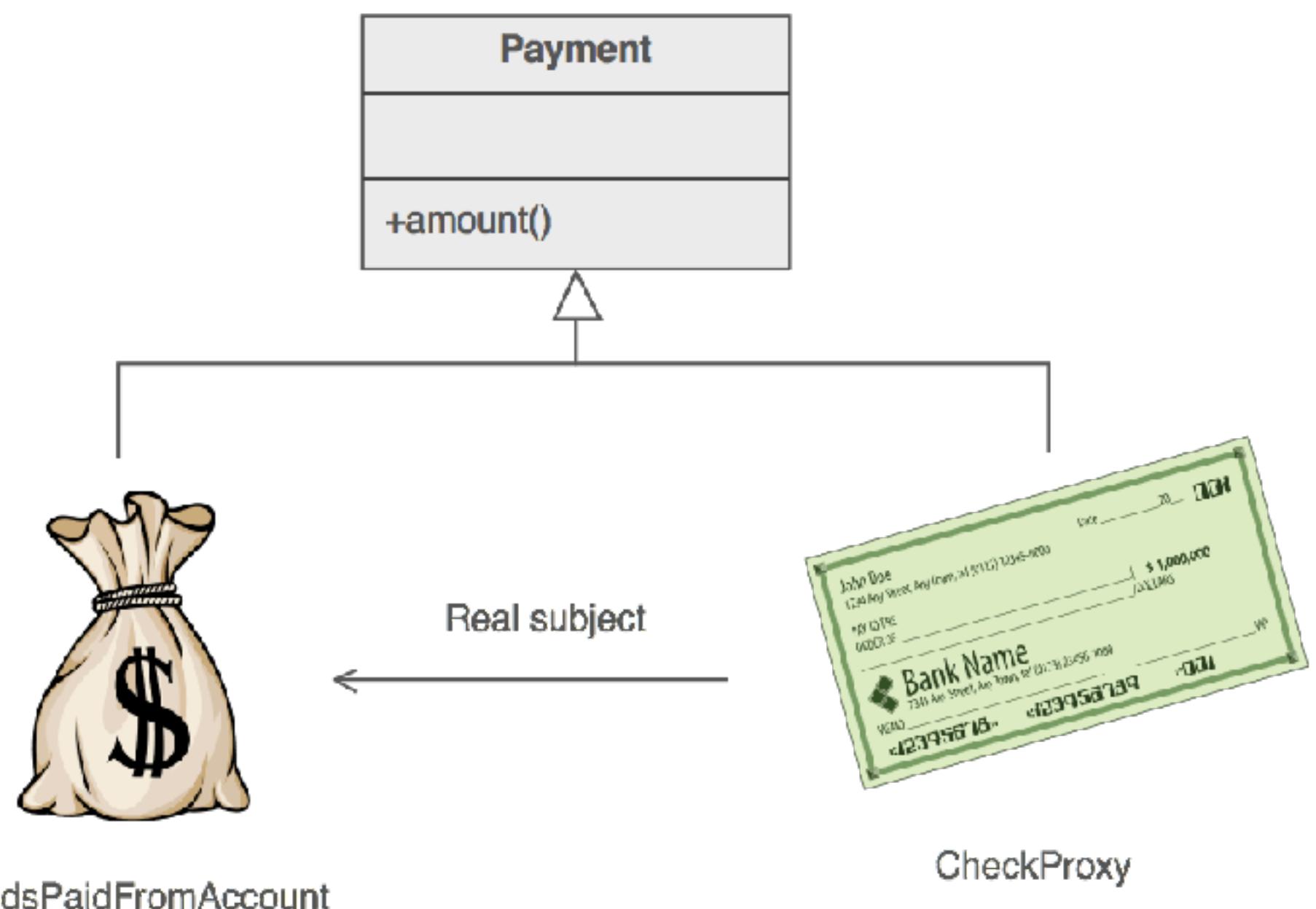


代理模式

- 代理提供相同的接口，适配器提供不同的接口，装饰器提供加强的接口
- 代理和装饰器结构相似：间接访问对象，转发请求

应用：java.rmi包

实例：银行支票 / 虚拟支付



行为型模式



命令模式

在一个对象中封装命令请求

用于实现“请求-响应模式”松耦合，通过命令解耦调用操作的对象和执行操作的对象，即调用者和接收者的分离

一系列的命令可以组成复合命令

如果接收者行为过多，会导致困惑

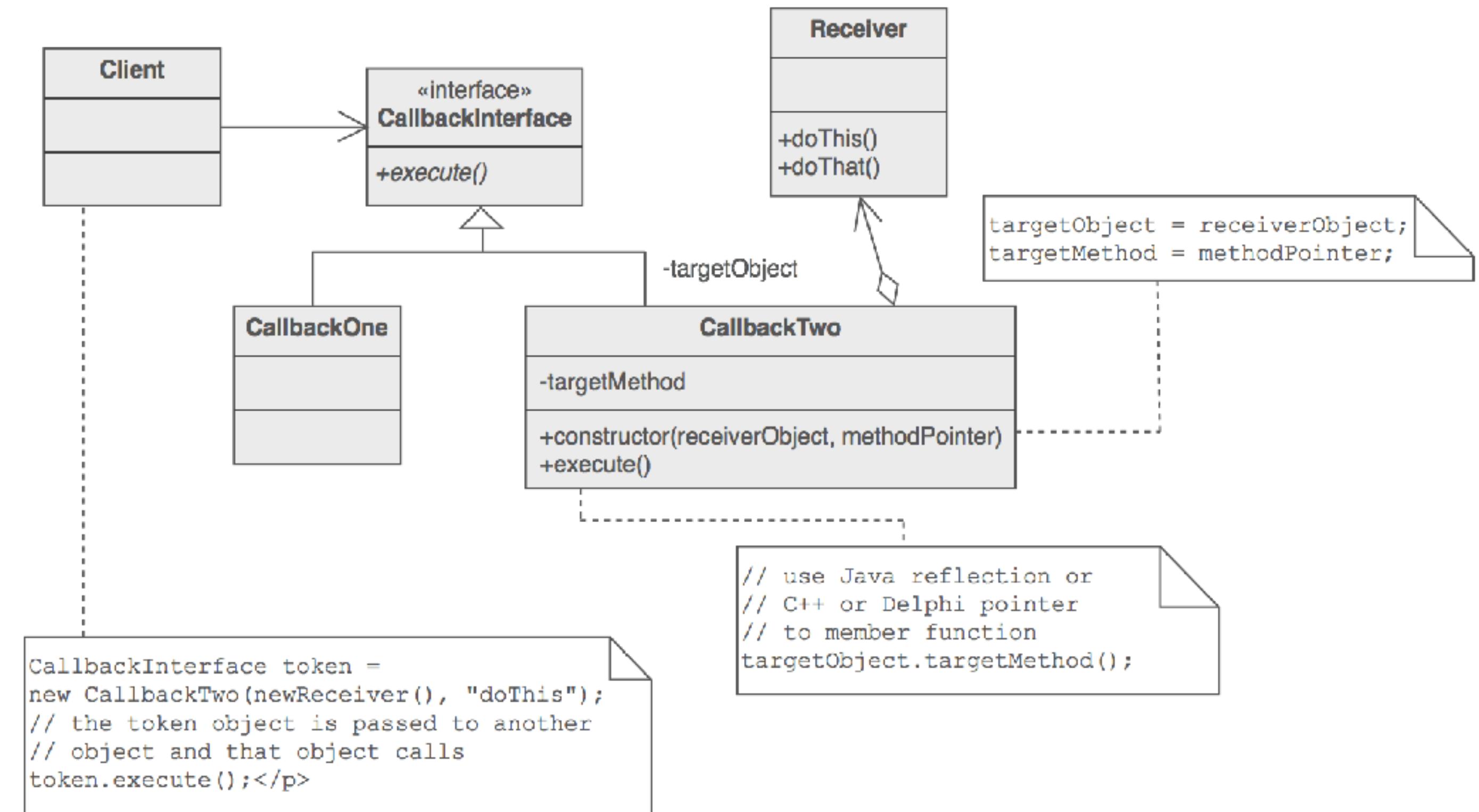
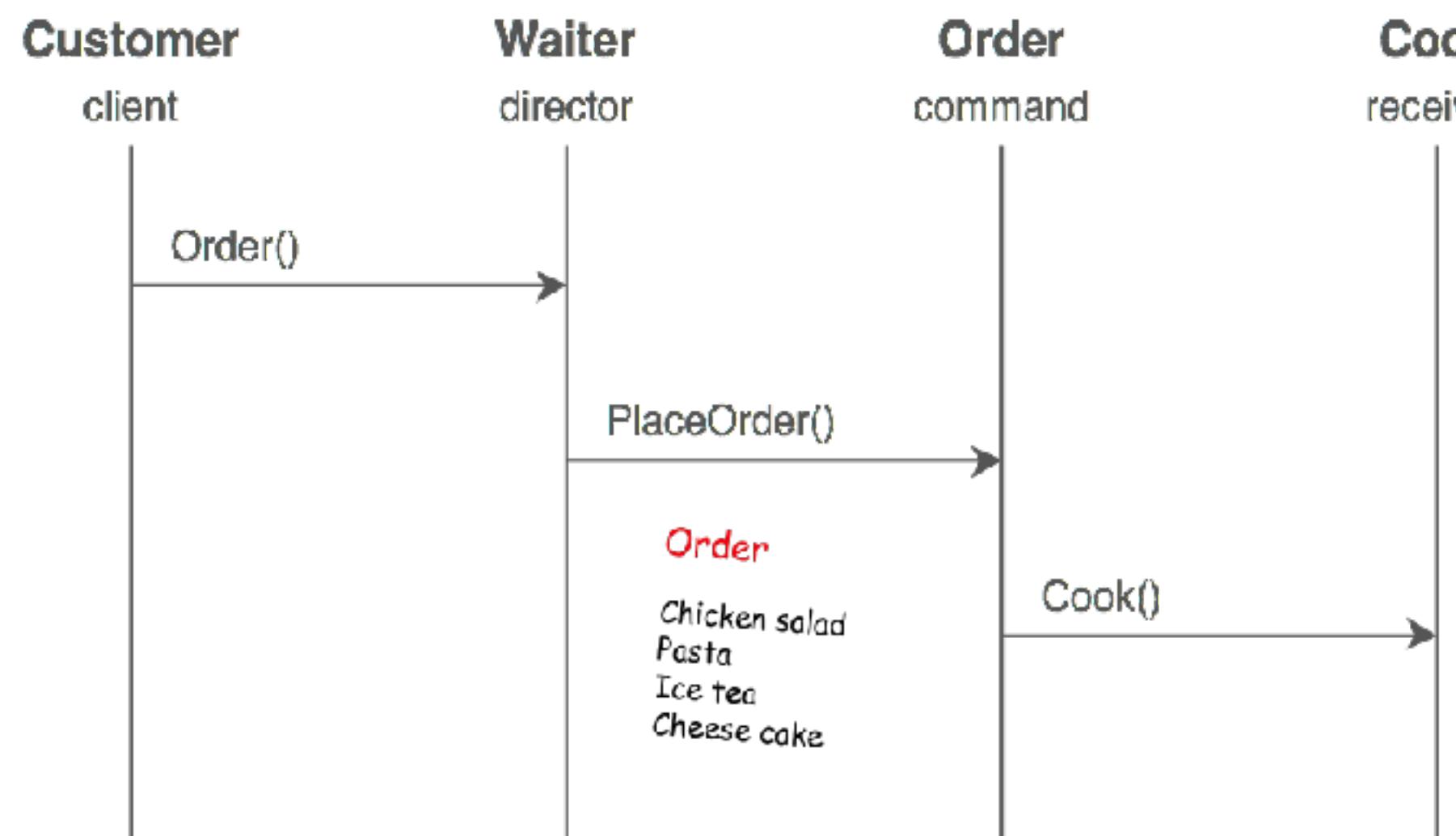
- 分为调用者，接收者和命令
- 请求发送给调用者，调用者将其封装为命令对象
- 接收者传递给命令对象，命令对象调用接受者特定方法执行命令对象
- 客户端用于实例化调用者对象，将接收者和命令组合在一起



命令模式

应用: javax.swing.Action, java.lang.Runnable

实例: 餐厅点餐



责任链模式

通过定义一个方法，将请求在对象链中传递

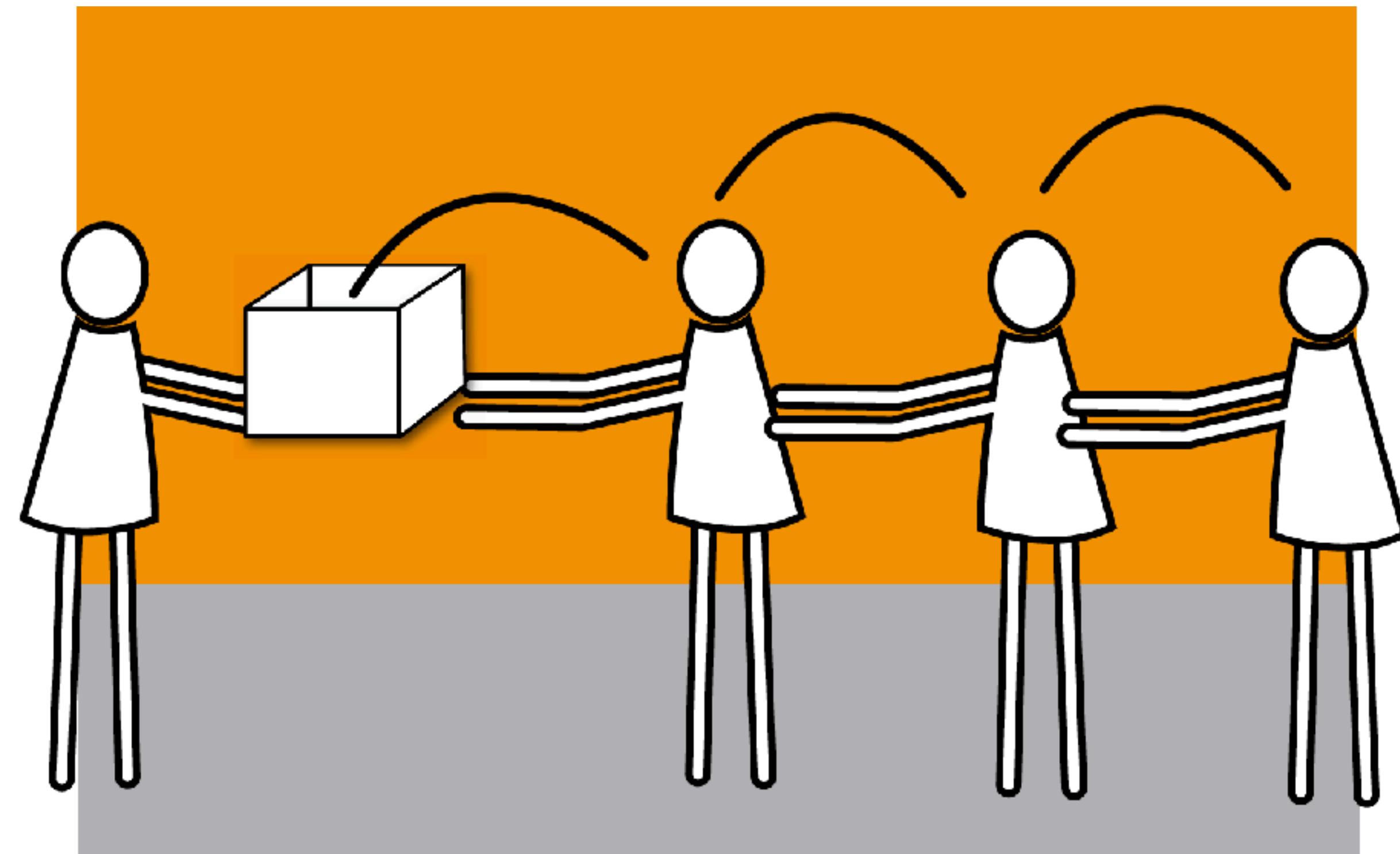
解耦请求发送者和接受者

客户端将请求放入流水线

请求会经过多个“链接”起来的对象，迭代处理，知道最终的处理者

每个对象或者完全/部分处理请求，或者传递下去

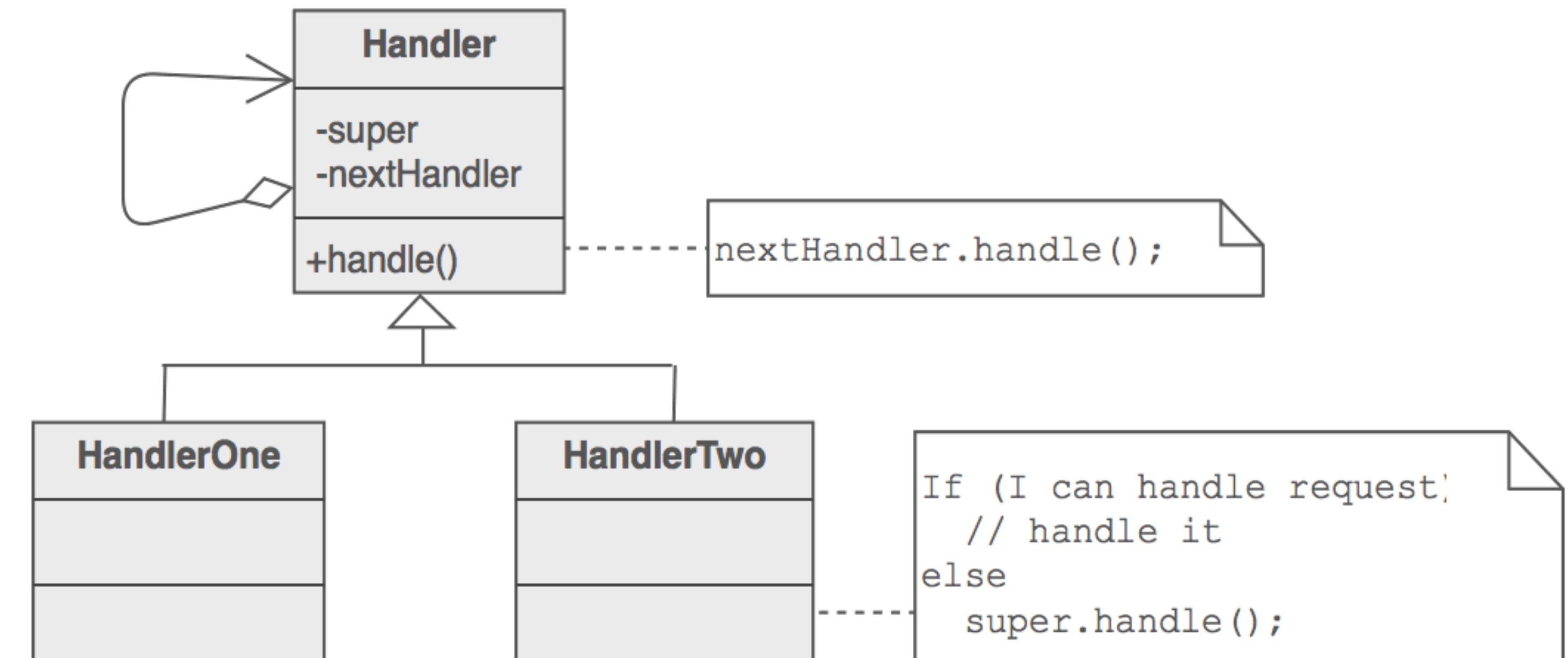
不适用于请求只被一个调用者处理，或者发送者知道接受者是谁



责任链模式

应用：try-catch, java.util.logging.Logger的log方法,
javax.servlet.Filter的doFilter方法

实例：ATM取钱（组合金额）



解释器模式

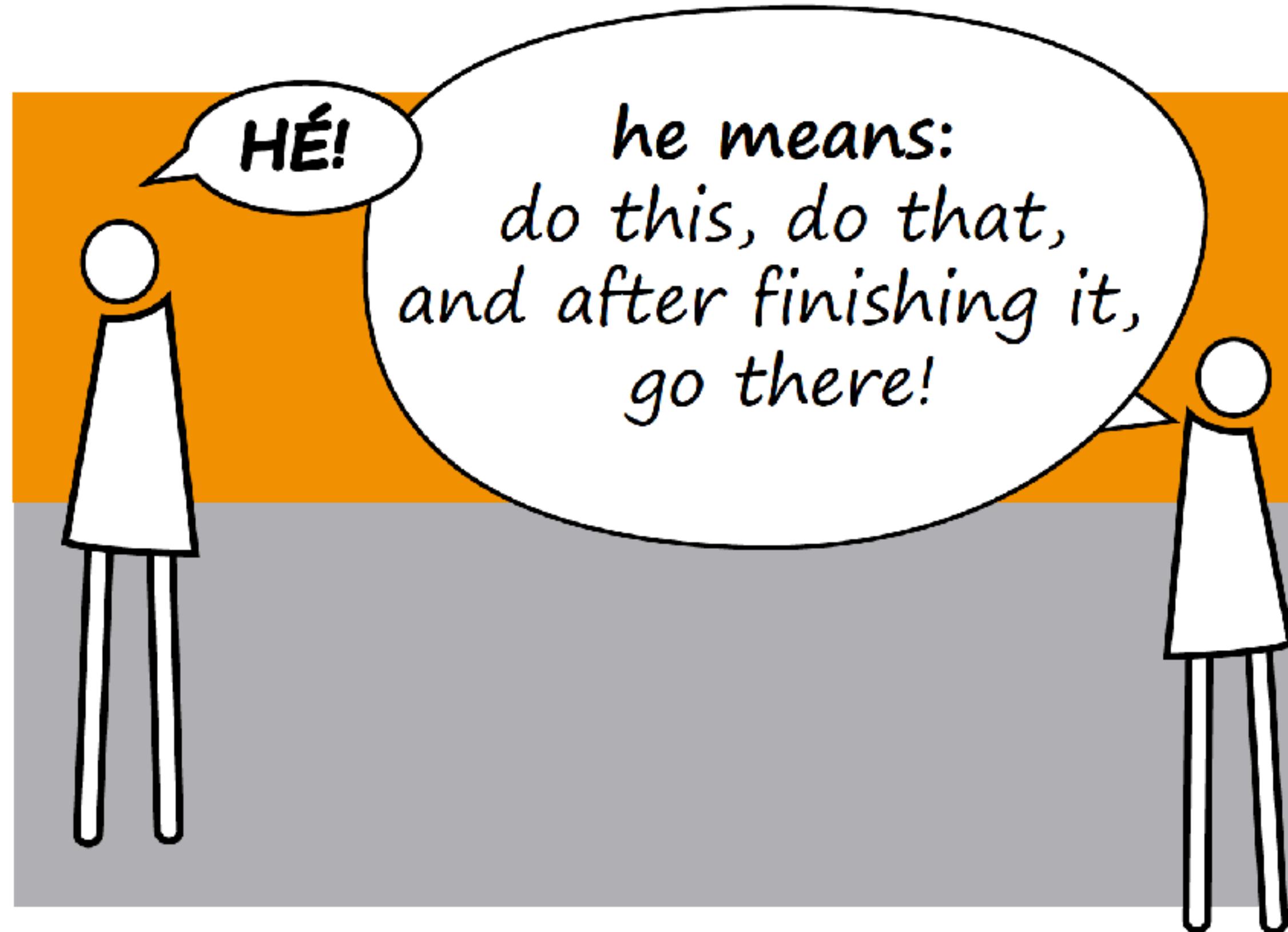
领域 -> 语言 -> 语法 -> 对象 (DSL)

对于一个语言，定义个语法代表，解释器使用该语法解释句子（语法树）

是一个从领域到语言，再到语法，最后到有层级的面向对象设计

适用于拥有层级的，迭代式的语法（或者包含子语法，或者具体语法）的领域，解释器基于该语法“理解”领域“句子”

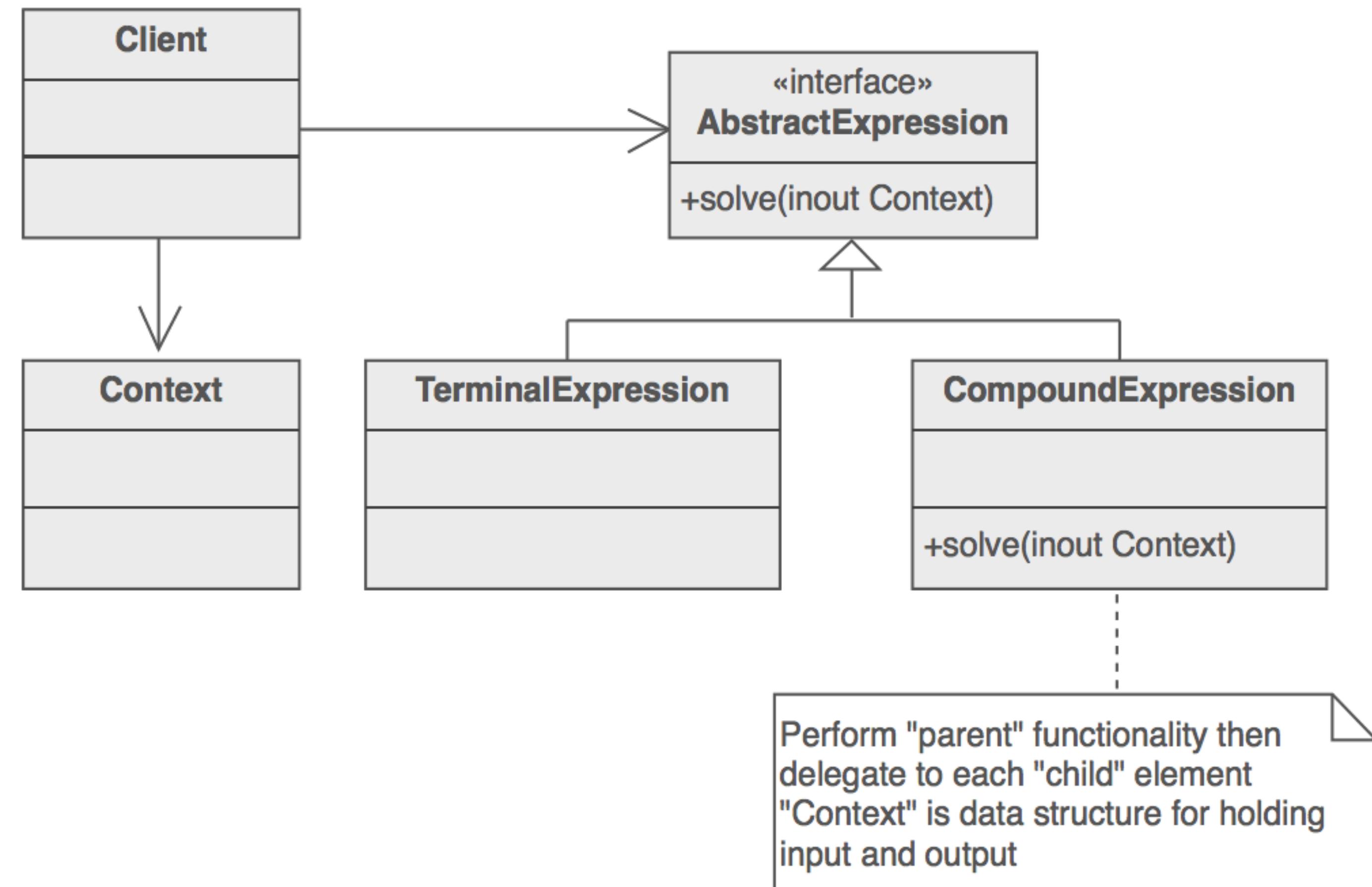
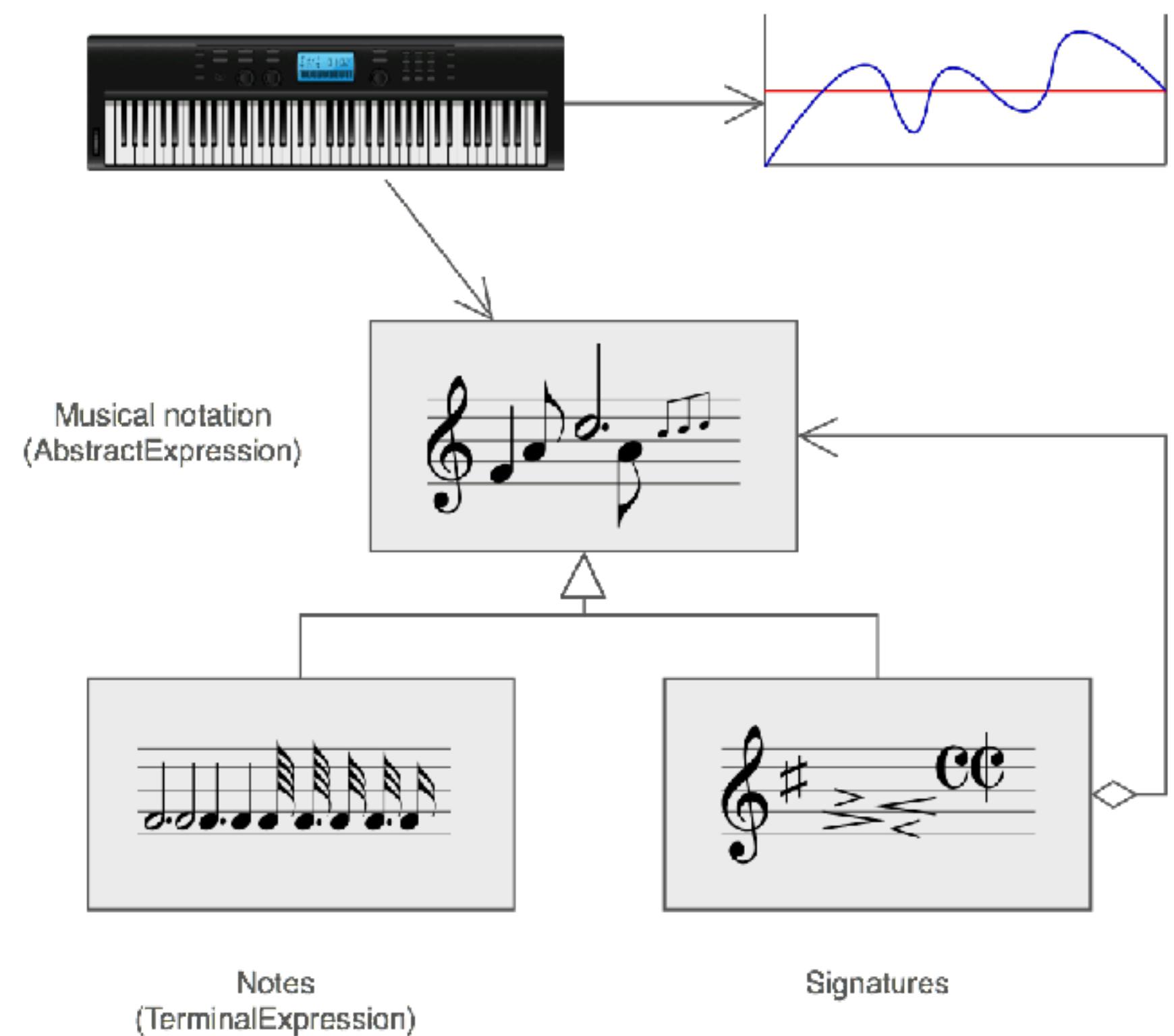
通常组合模式（结构）的使用都伴随着解释器模式（行为）



解释器模式

应用: `java.util.Pattern`, `java.text.Format`, JVM

实例: 音乐家



迭代器模式

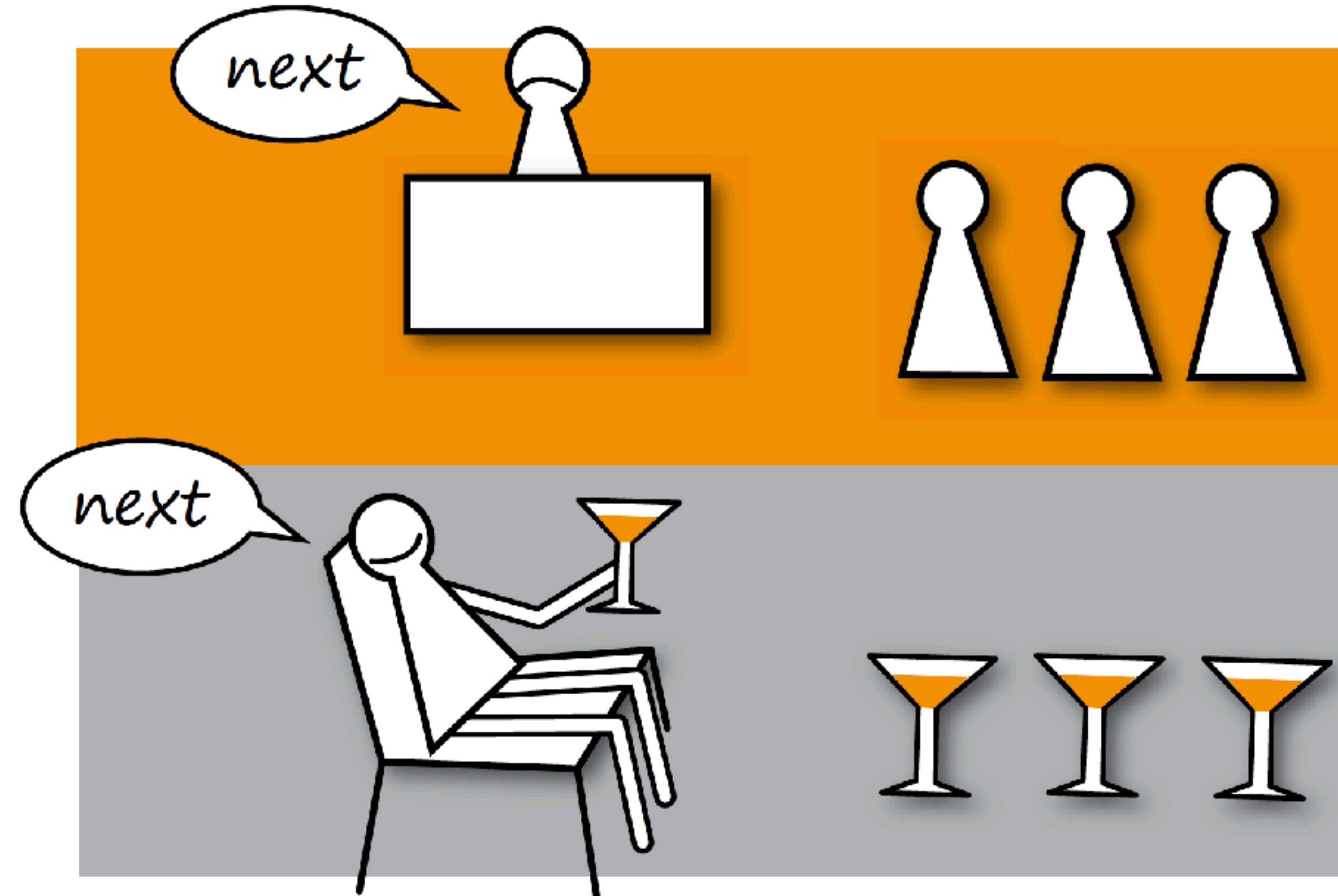
在不显示内在数据结构的基础上，实现集合上的顺序访问

一种顺序访问（遍历）聚合对象的方法，同时不暴露底层表现

迭代器定义标准的遍历协议

迭代器逻辑嵌在集合内部，对外隐藏

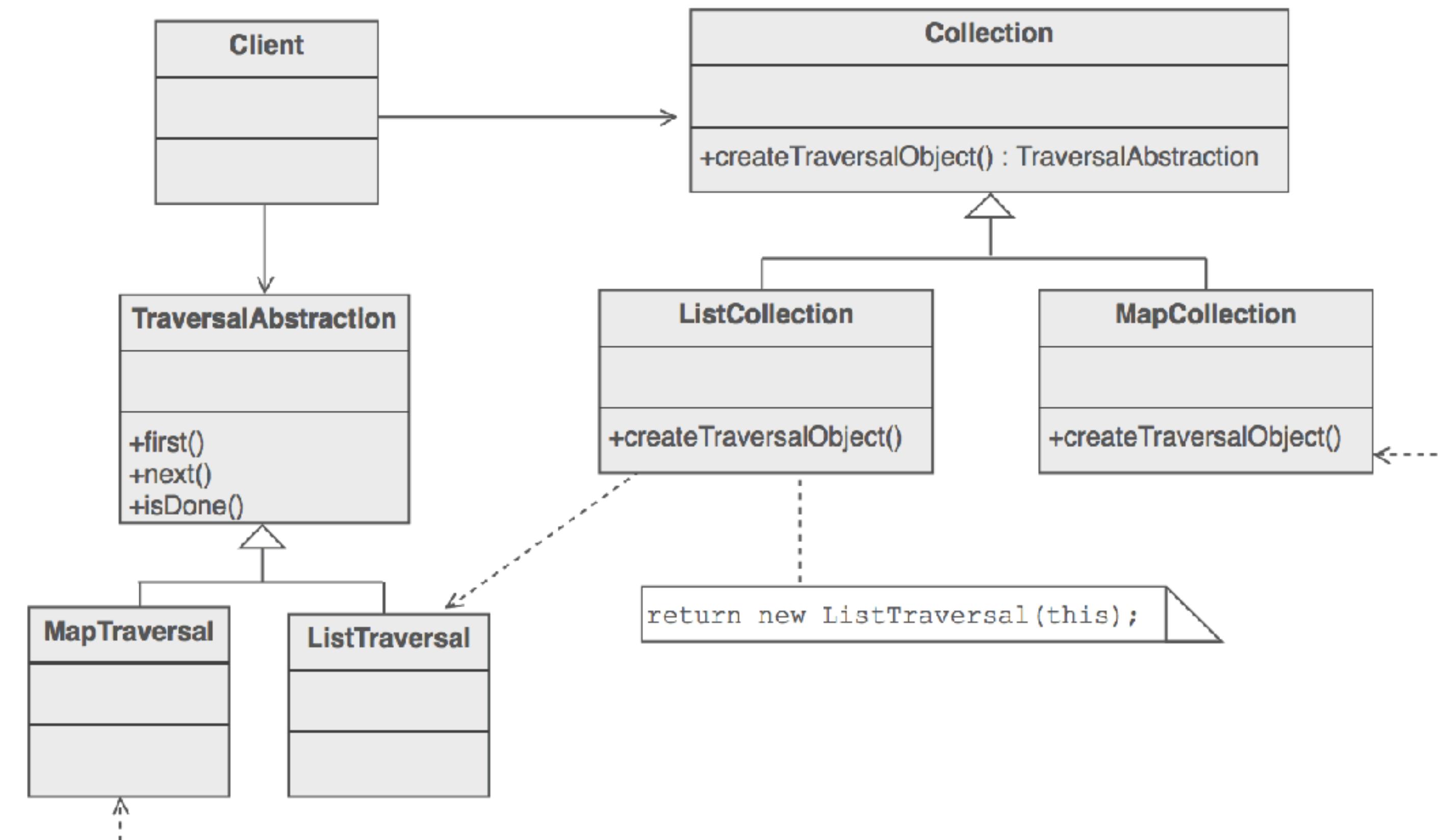
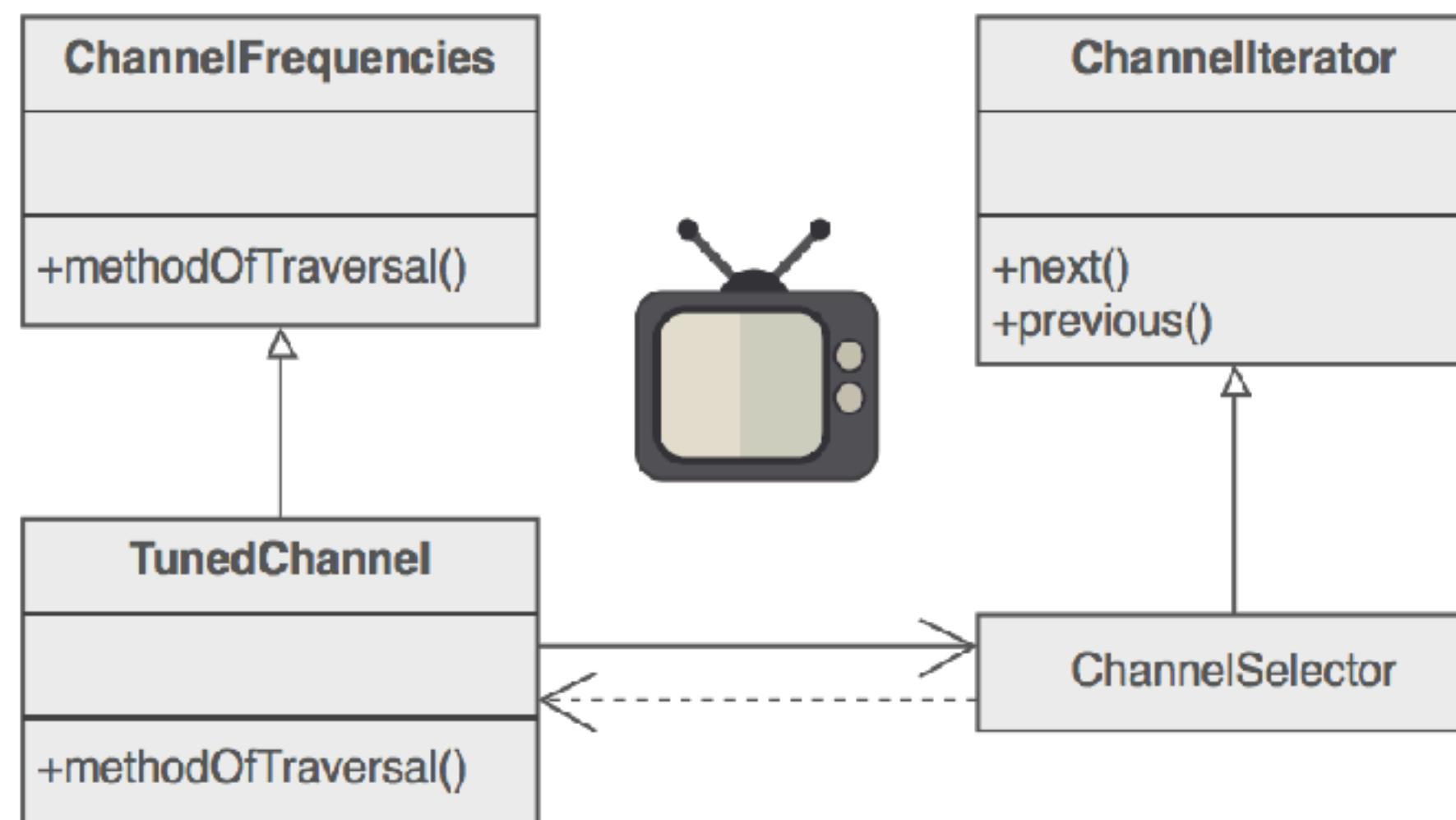
迭代器抽象叫做泛型编程，用于分离算法和数据结构



迭代器模式

应用：Java所有collection接口有iterator，
java.util.Scanner

实例：电视机切换频道



中介者模式

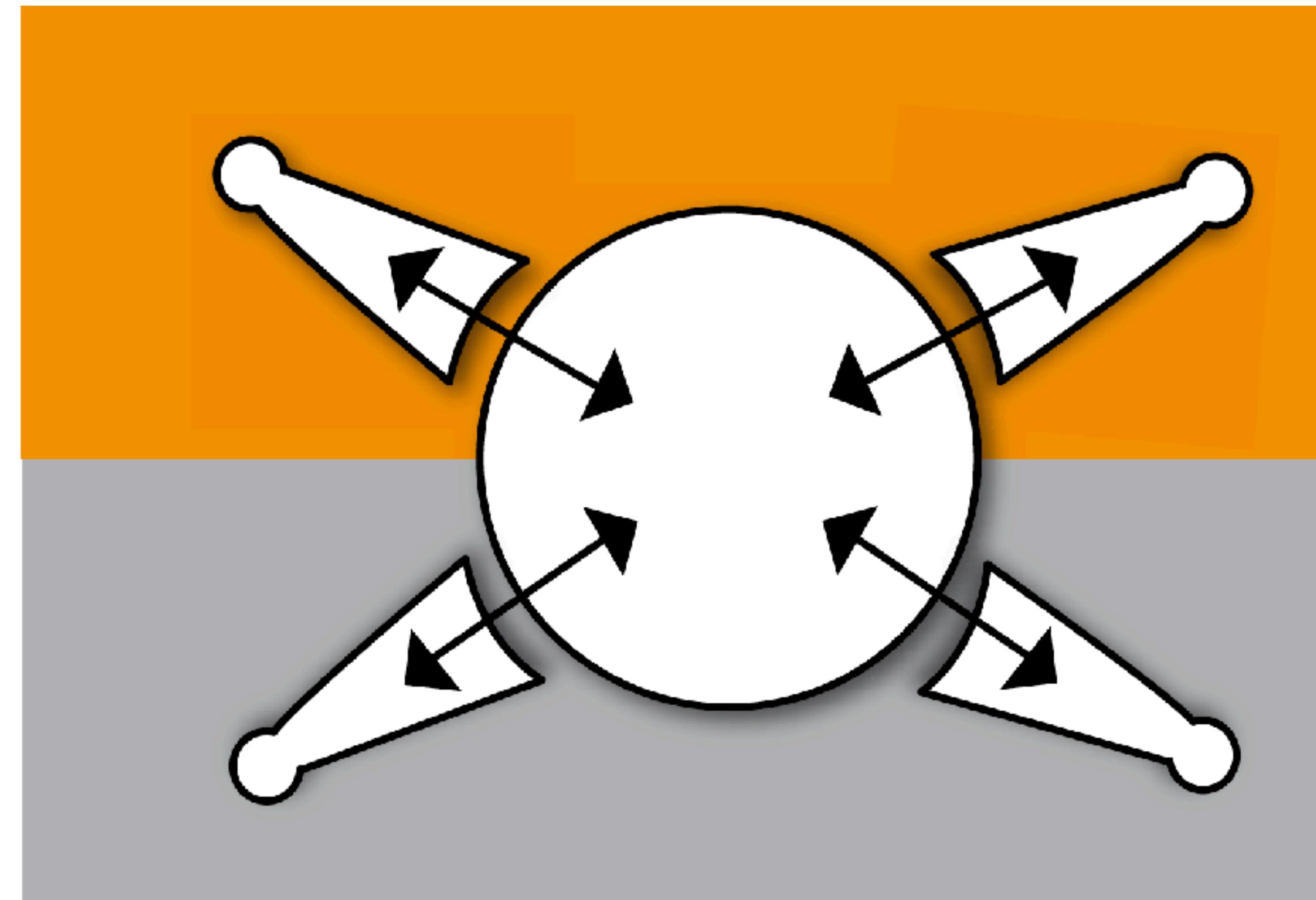
对象中的中间层，通过对对象相互通信

用于封装对象之间的特别复杂通信逻辑

中介者模式解耦（不同类型）对象组之间的直接访问，
这个可以独立的改变他们之间的交互

通过一个中介实现多对多的访问关系

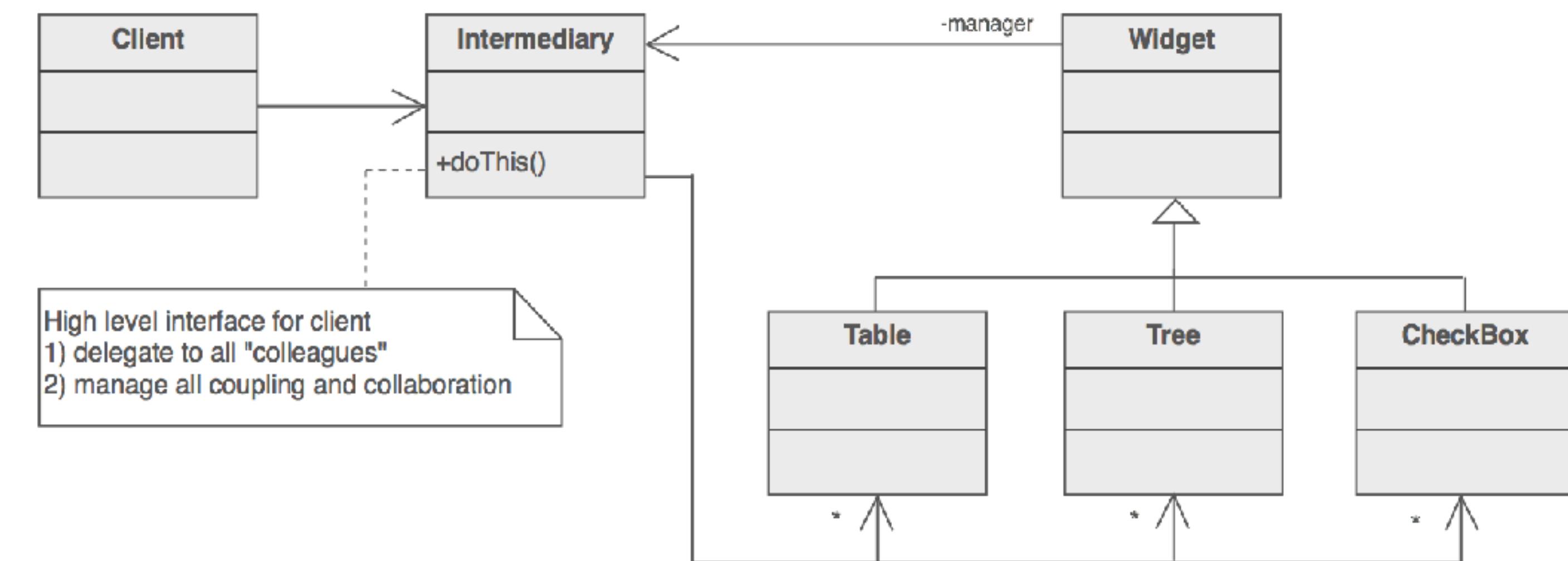
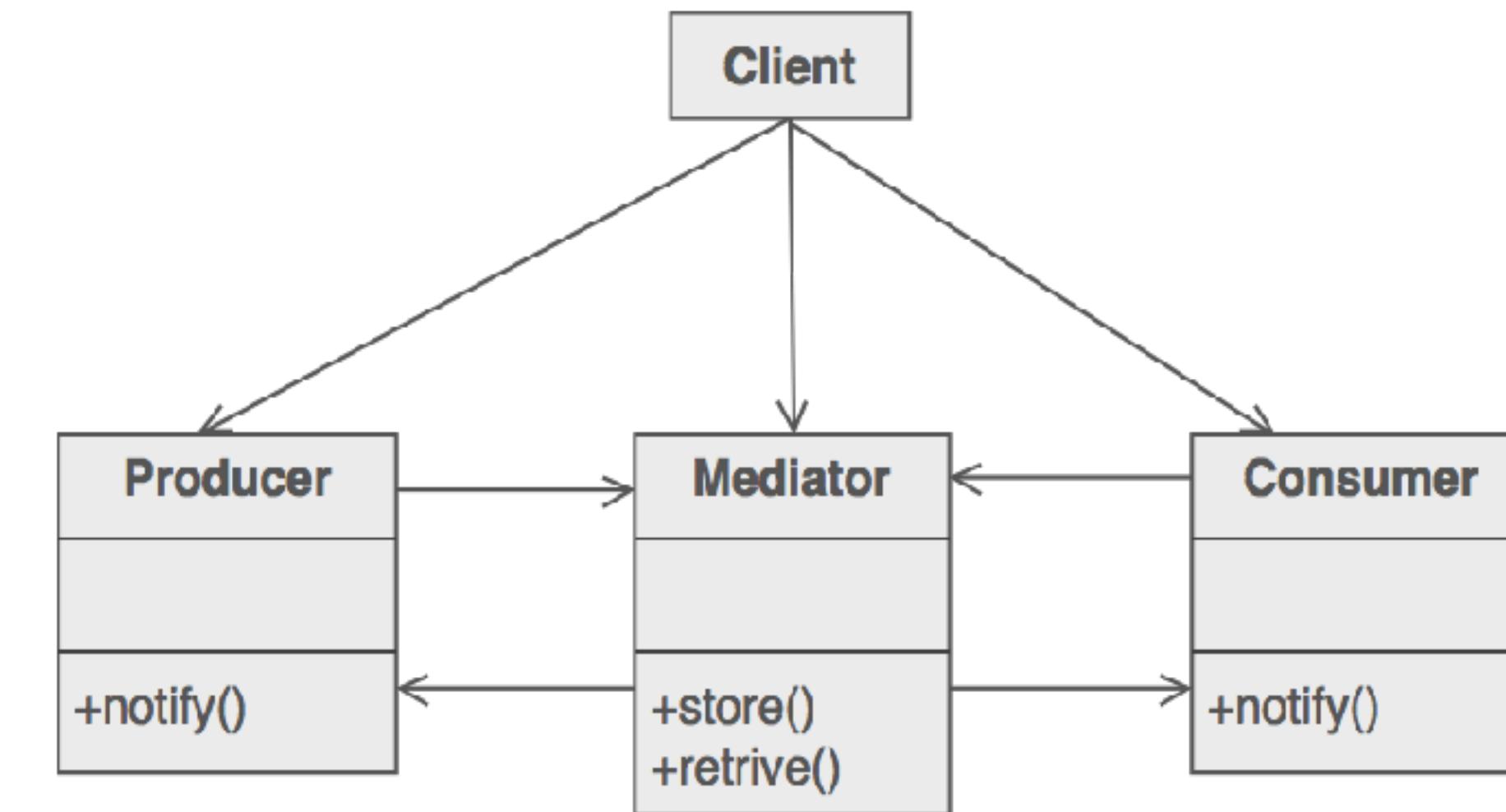
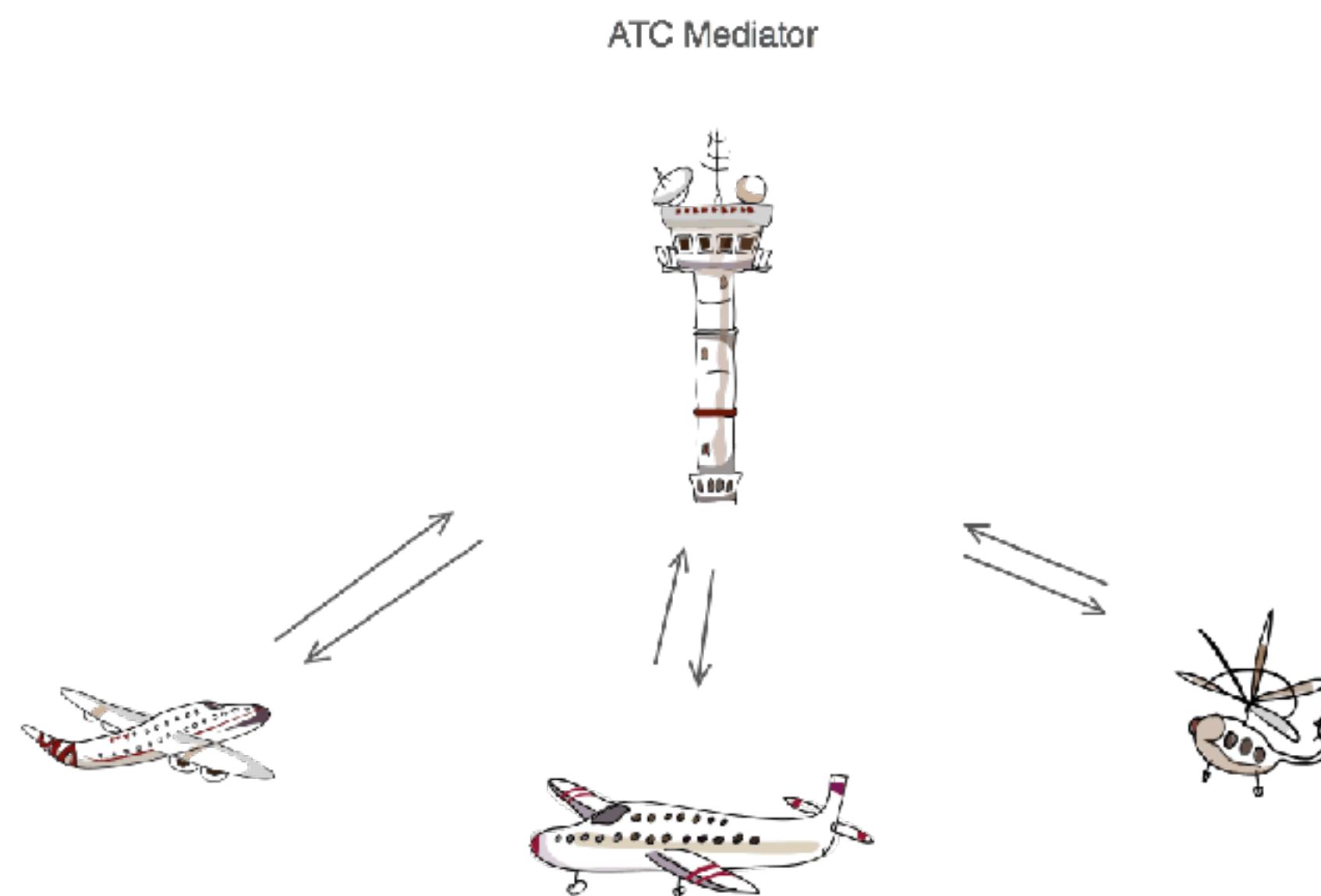
JMS = 中介者 + 观察者



中介者模式

应用：java.util.Timer的scheduleXXX()方法，Java Concurrency Executor的execute()方法，java.lang.reflect.Method的invoke()

实例：机场控制塔台



备忘录模式

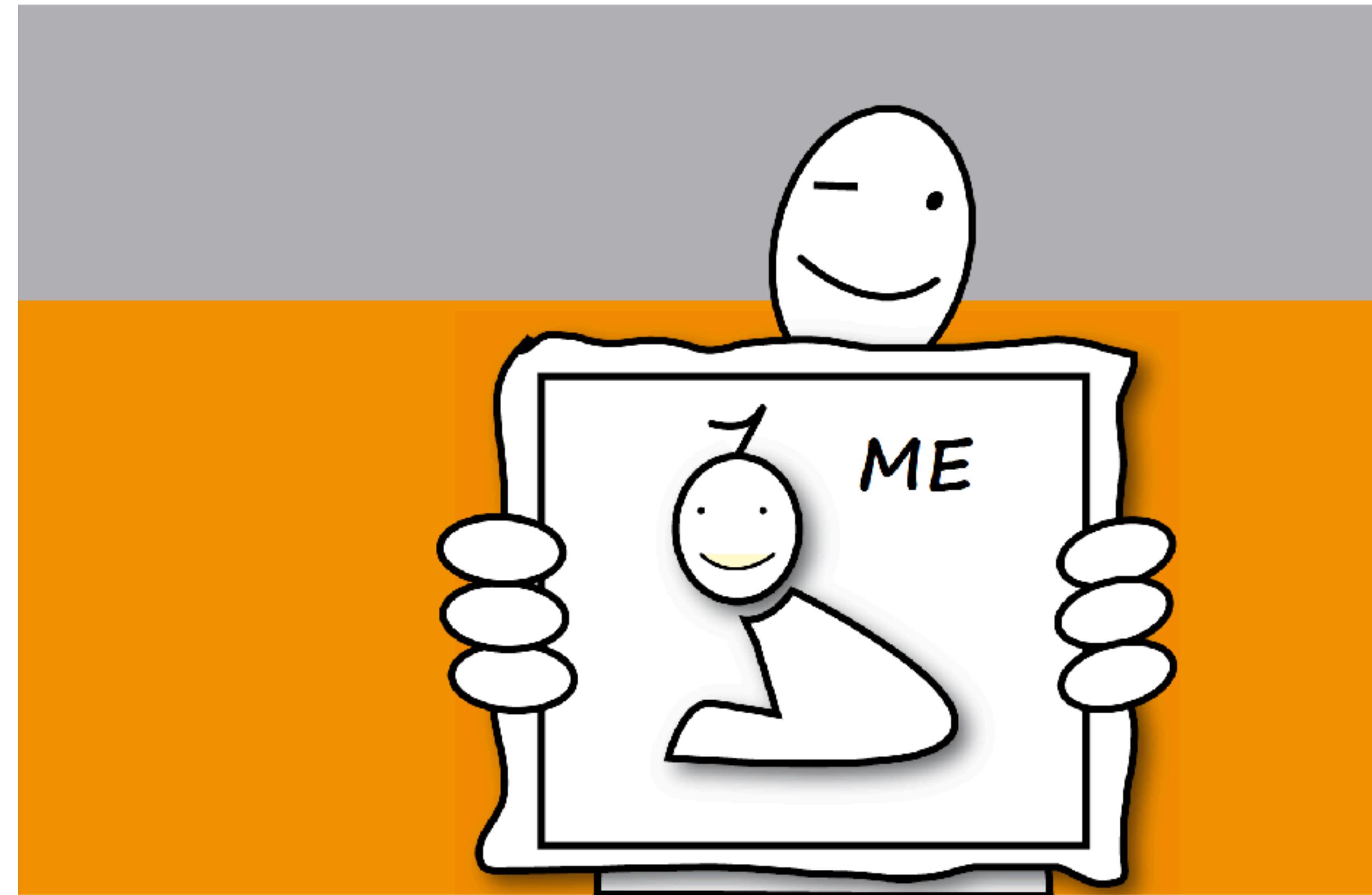
在不影响封装性的前提下，捕获并存储对象的内部状态，用以回滚到之前的状态

恢复对象之前状态，撤销或回滚操作

通过从源对象中请求备忘录来获取历史状态

备忘录通常和命令模式一起使用，记录状态

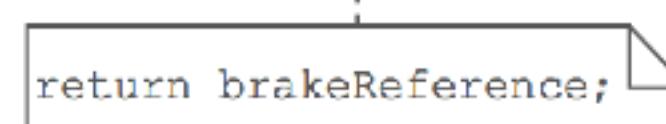
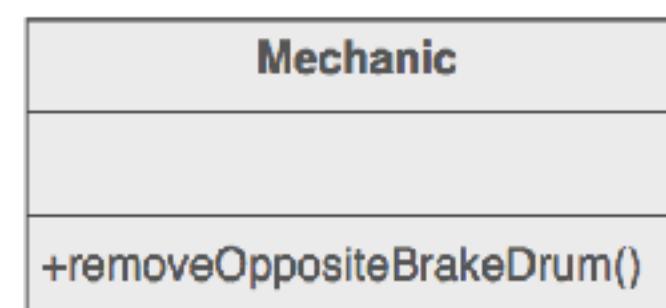
- 发起人 (Originator) : 负责存储对象自身 (how)
- 守护人 (Caretaker) : 负责管理何时存储和恢复对象自身 (why when)
- 备忘录 (Memento) : 由发起人进行读写，而由守护人管理



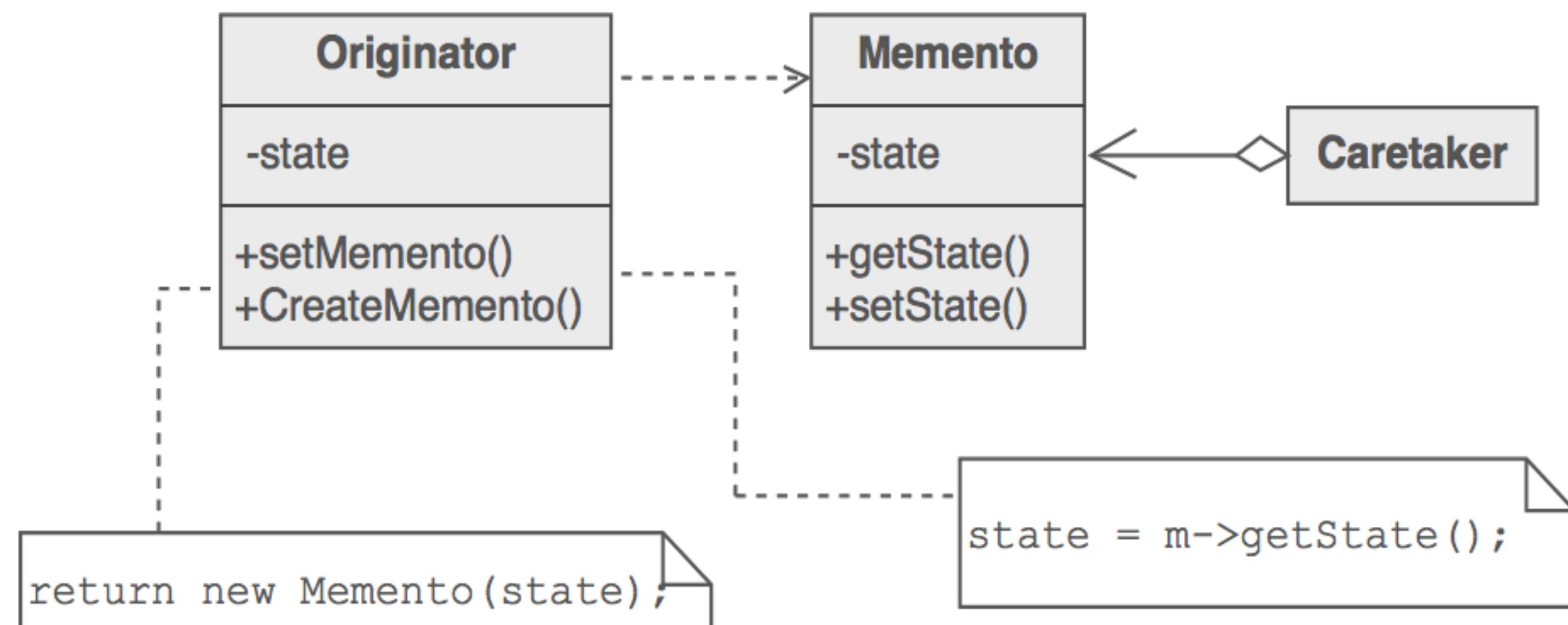
备忘录模式

当备忘录过于庞大，会占用很多内存

实例：鼓式制动器（分开拆卸）



Leave intact until brakes
on Side1 are completed



观察者模式

动态通知观察者/订阅者发生的改变

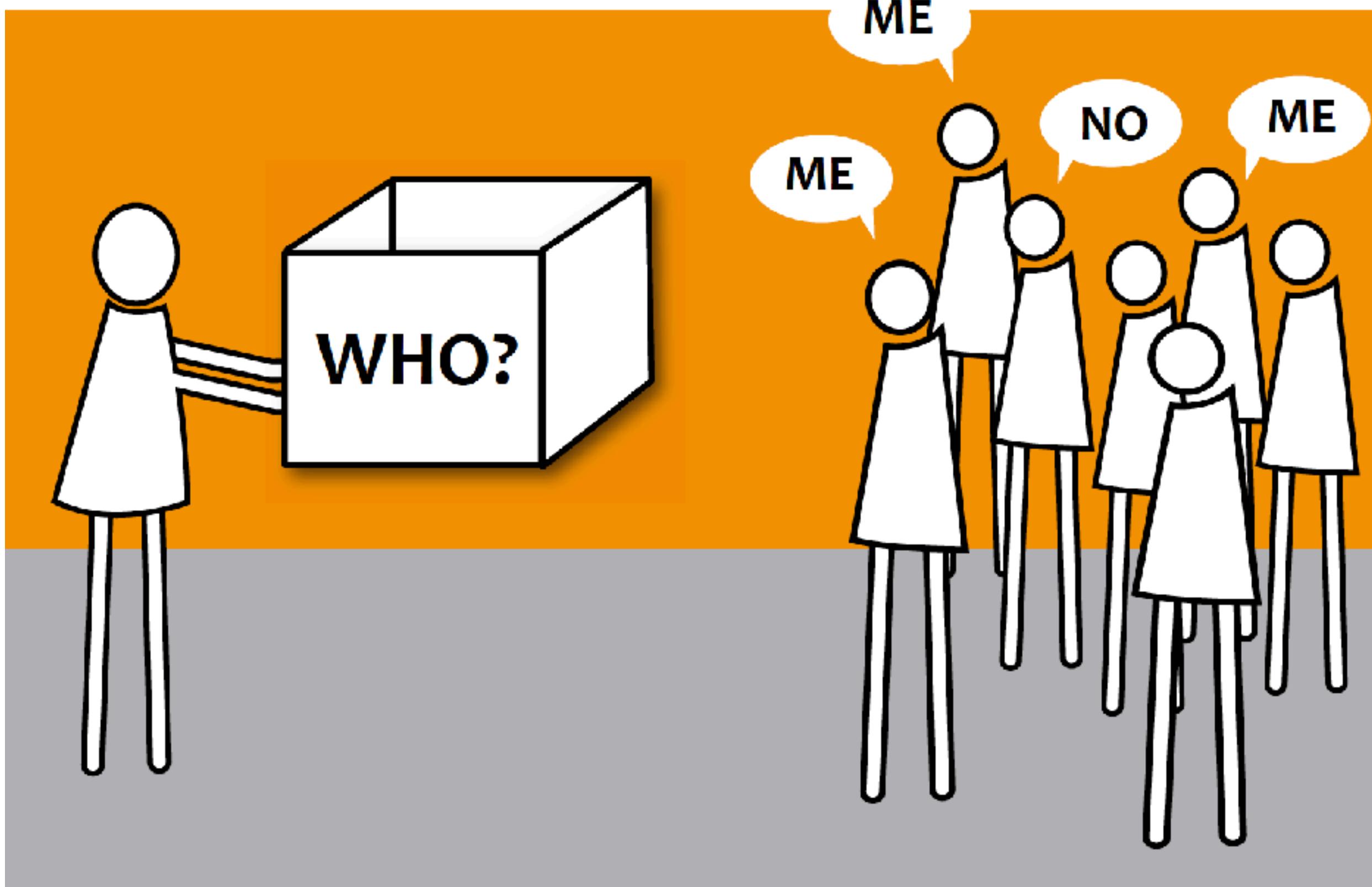
MVC中M是发布者，V层是观察者

订阅者-发布者模式

定义一对多关系，当一个对象状态改变，它的依赖对象就被通知和自动更新

所有订阅对象“拉”的方式获取状态更新，重用性更好，但效率不高

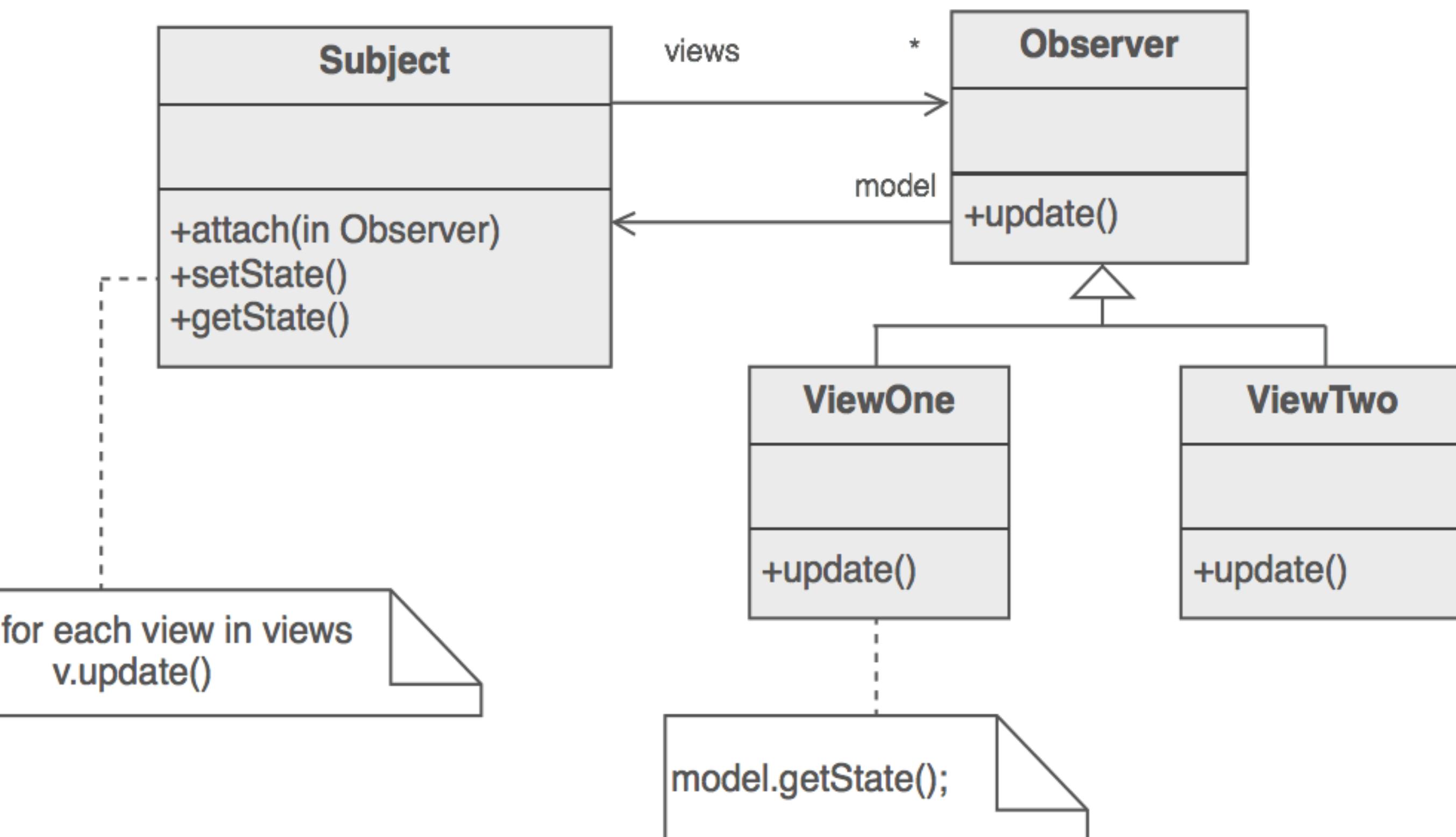
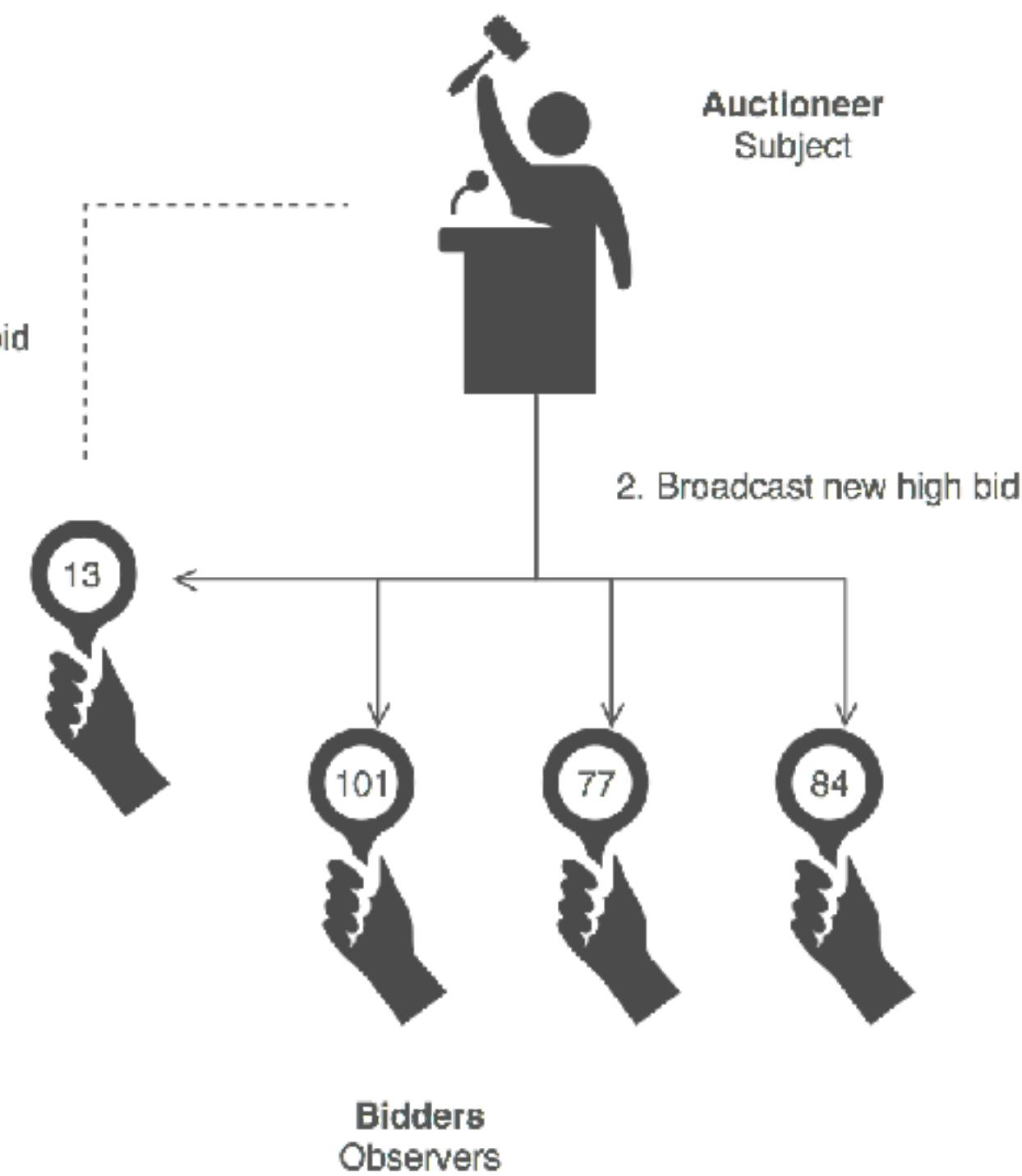
JMS = 观察者 + 中介者



观察者模式

应用: try-catch, java.util.logging.Logger的log方法,
javax.servlet.Filter的doFilter方法

实例: 拍卖



策略模式

当一个功能存在多种方式实现，那么让这些方法成为可交换的

策略 = 算法选择

满足开闭原则，抽象耦合（编程接口，而不是实现）

maximize cohesion and minimize coupling

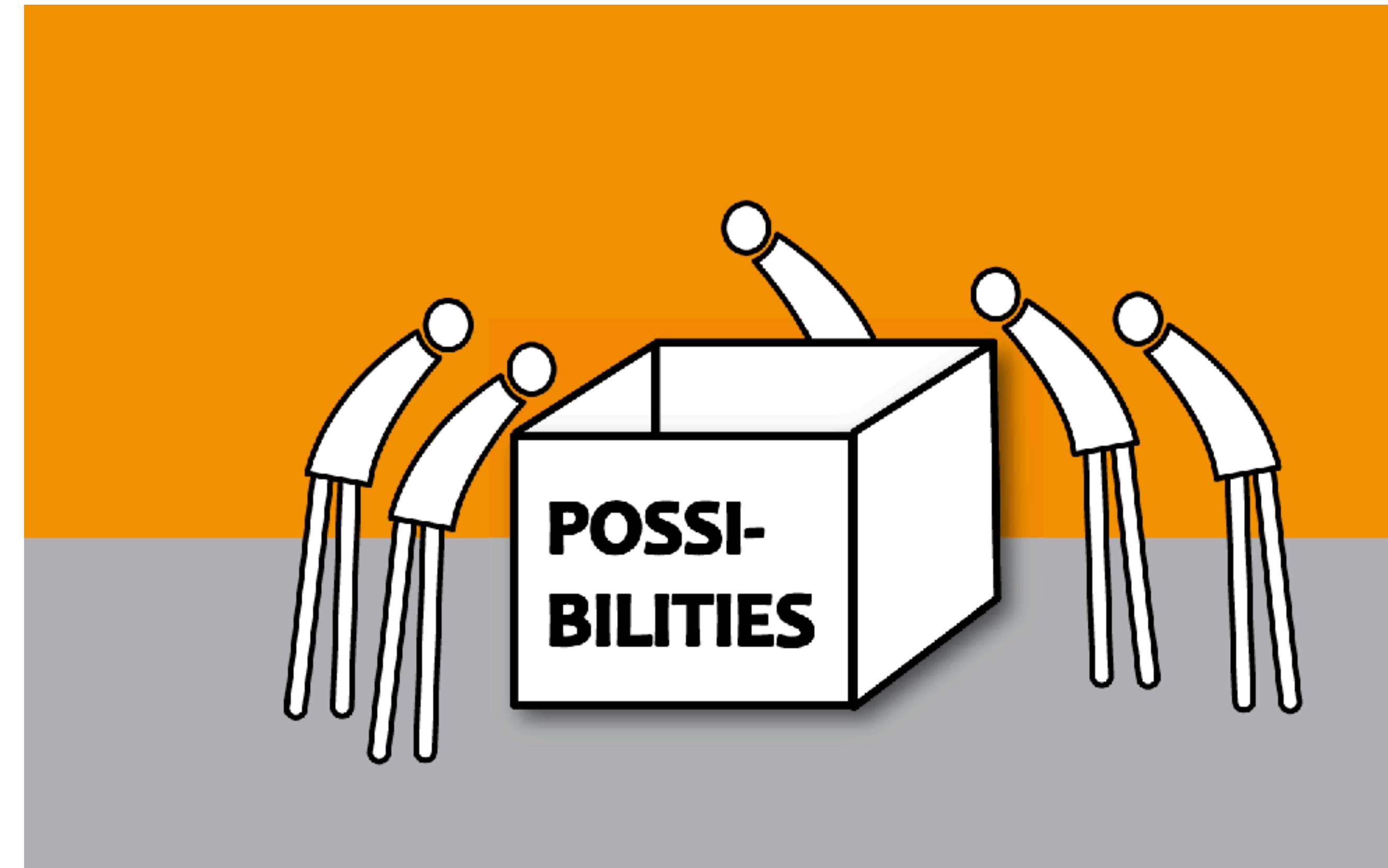
定义一组可以互换的算法，依据用户而变化

用户只和接口交互，算法实现的变化不影响用户体验

策略和模版的区别是在粒度

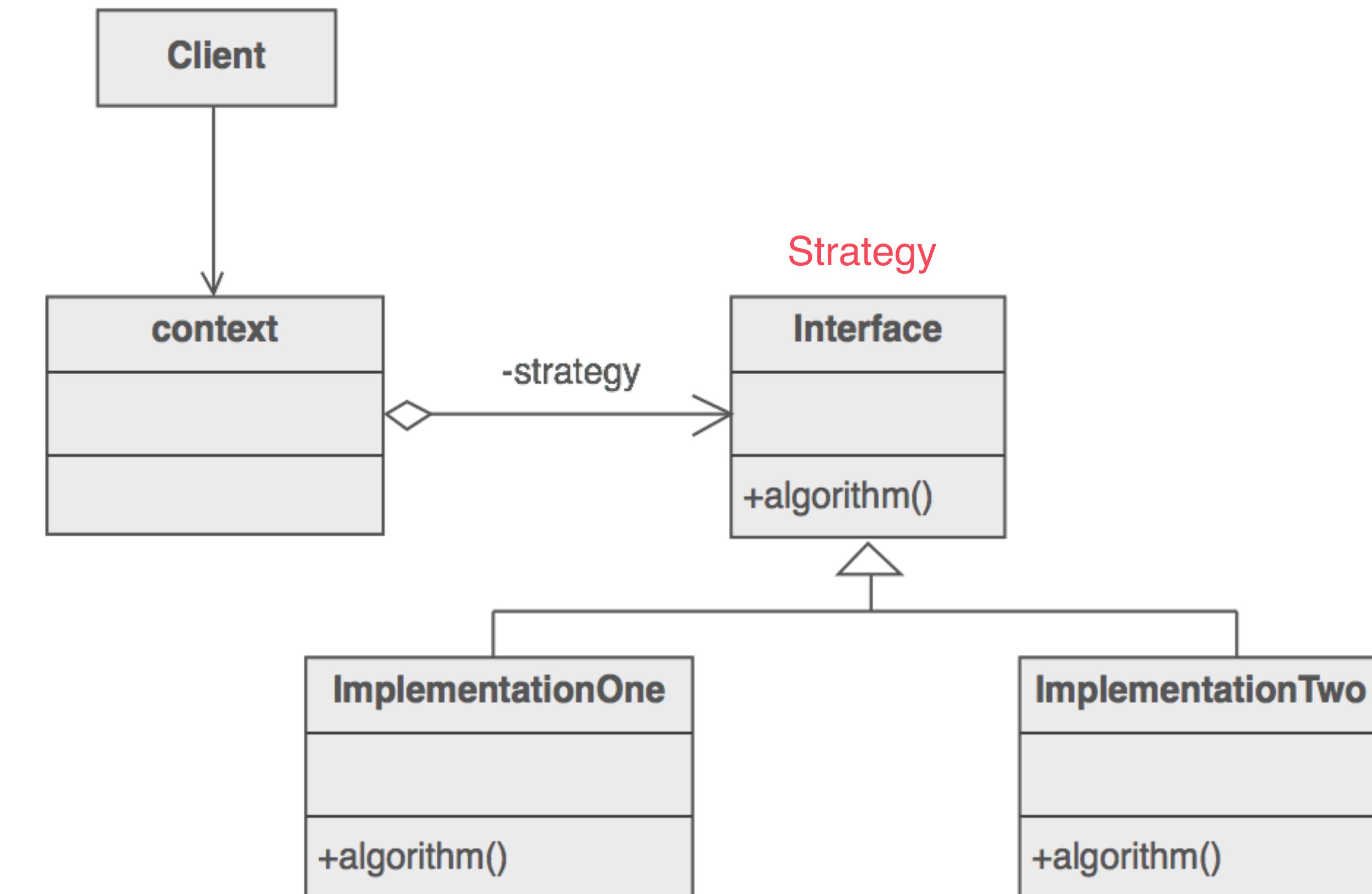
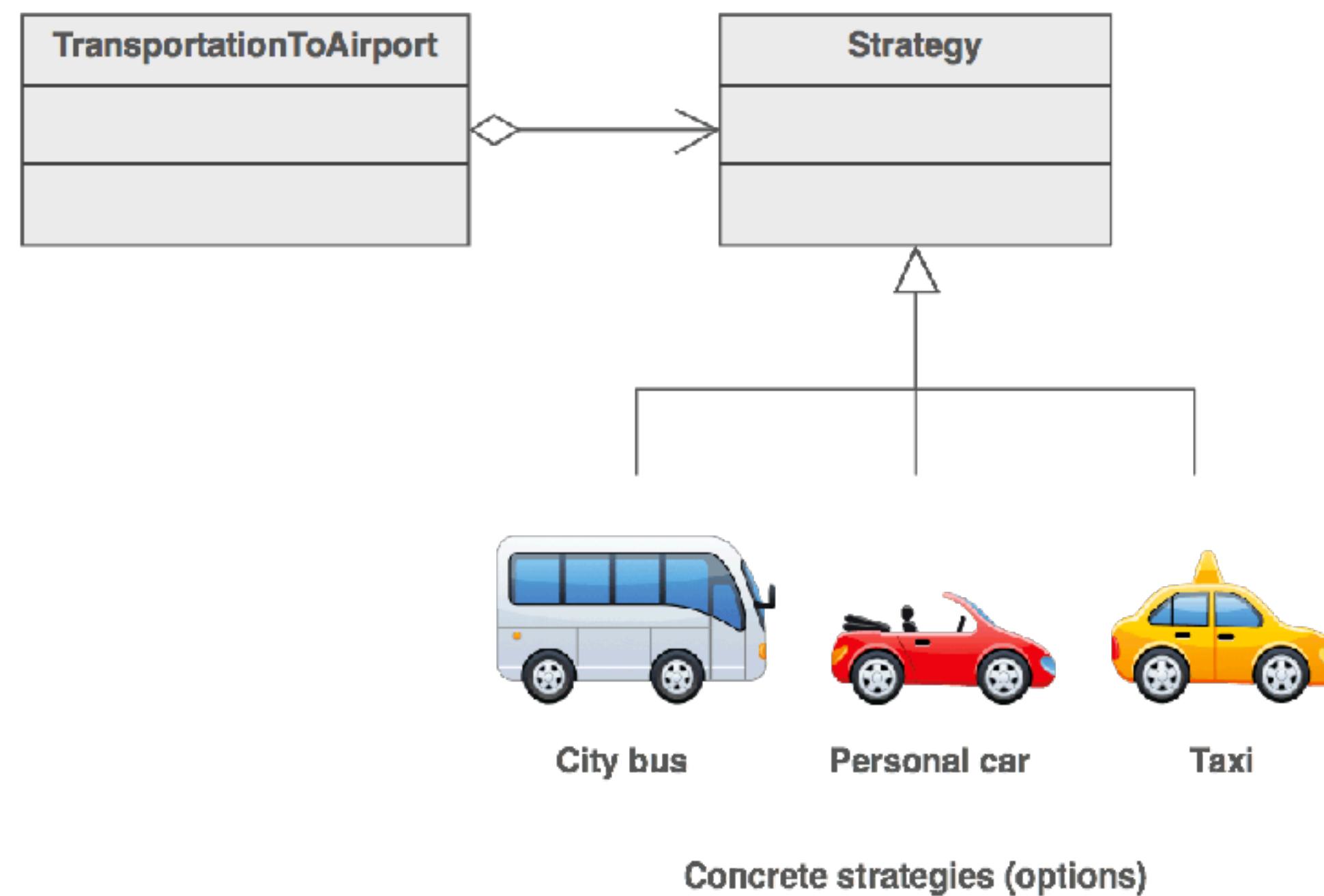
策略重在实现内核，装饰器重在皮肤

策略，状态和桥接具有类似结构，但目的（侧重点）不同



策略模式

实例：机场交通



状态模式

当一个对象内部状态改变后，该对象会显示不同的方法（就像它改变了类）

解决行为依赖于状态的问题

当内部状态发生变化时，允许对象更改它的行为

状态模式 = 上下文 + (状态 + 行为) 集合 + 当前状态

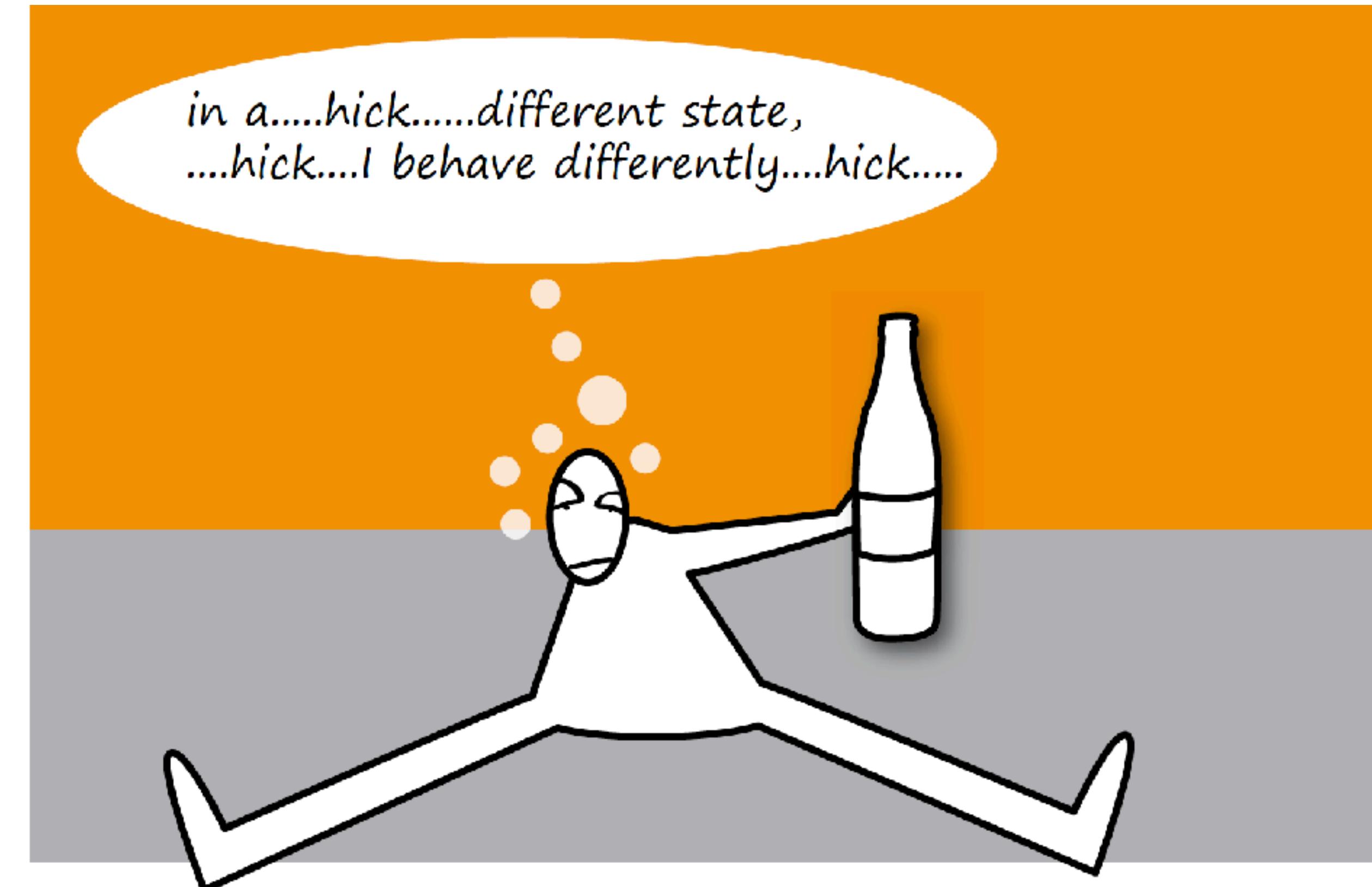
状态转换规则，或者在上下文，或者在每个状态类内部

不是简单的if-else

状态对象通常是单例

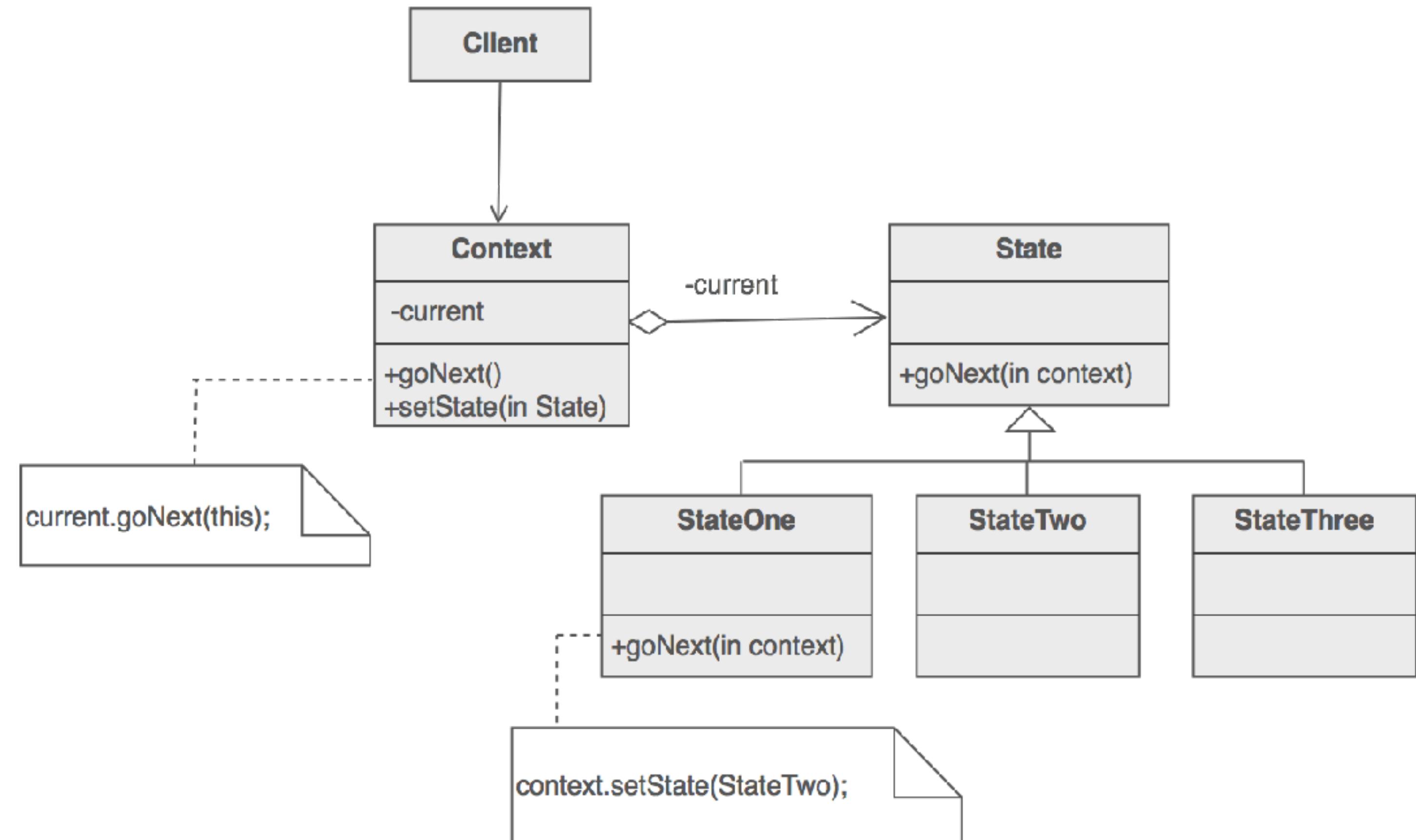
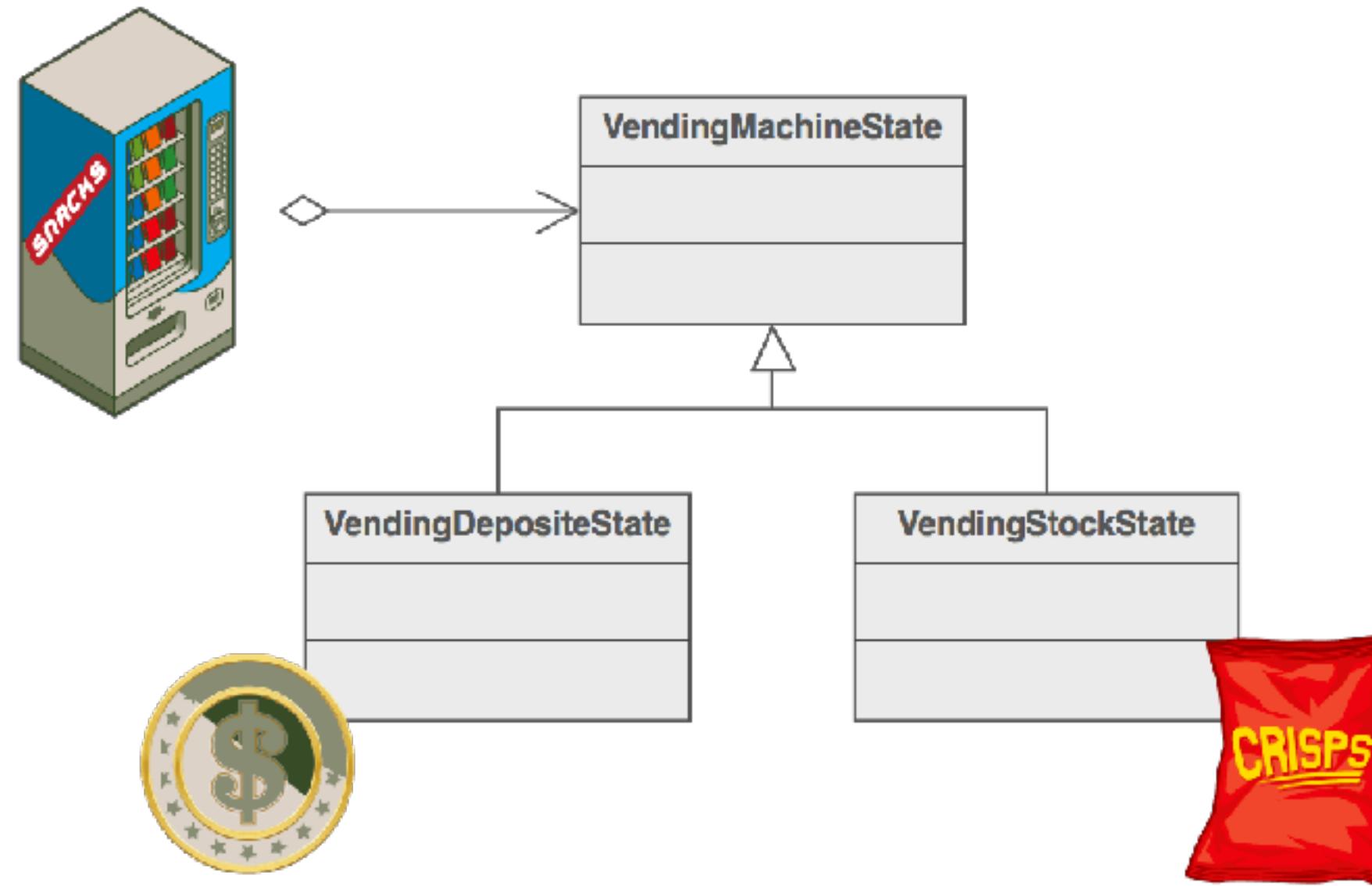
状态模式是基于策略模式实现的。策略重在算法的选择，是稳定的；

状态重在上下文的变化，它决定选择哪个策略



状态模式

实例：自动贩卖机



模版模式

算法的框架是固定的，但部分可以填充为不同的值

子类重构或者修改计算细节不影响框架（步骤）

父类一个方法中定义算法的框架，在子类中实现次框架

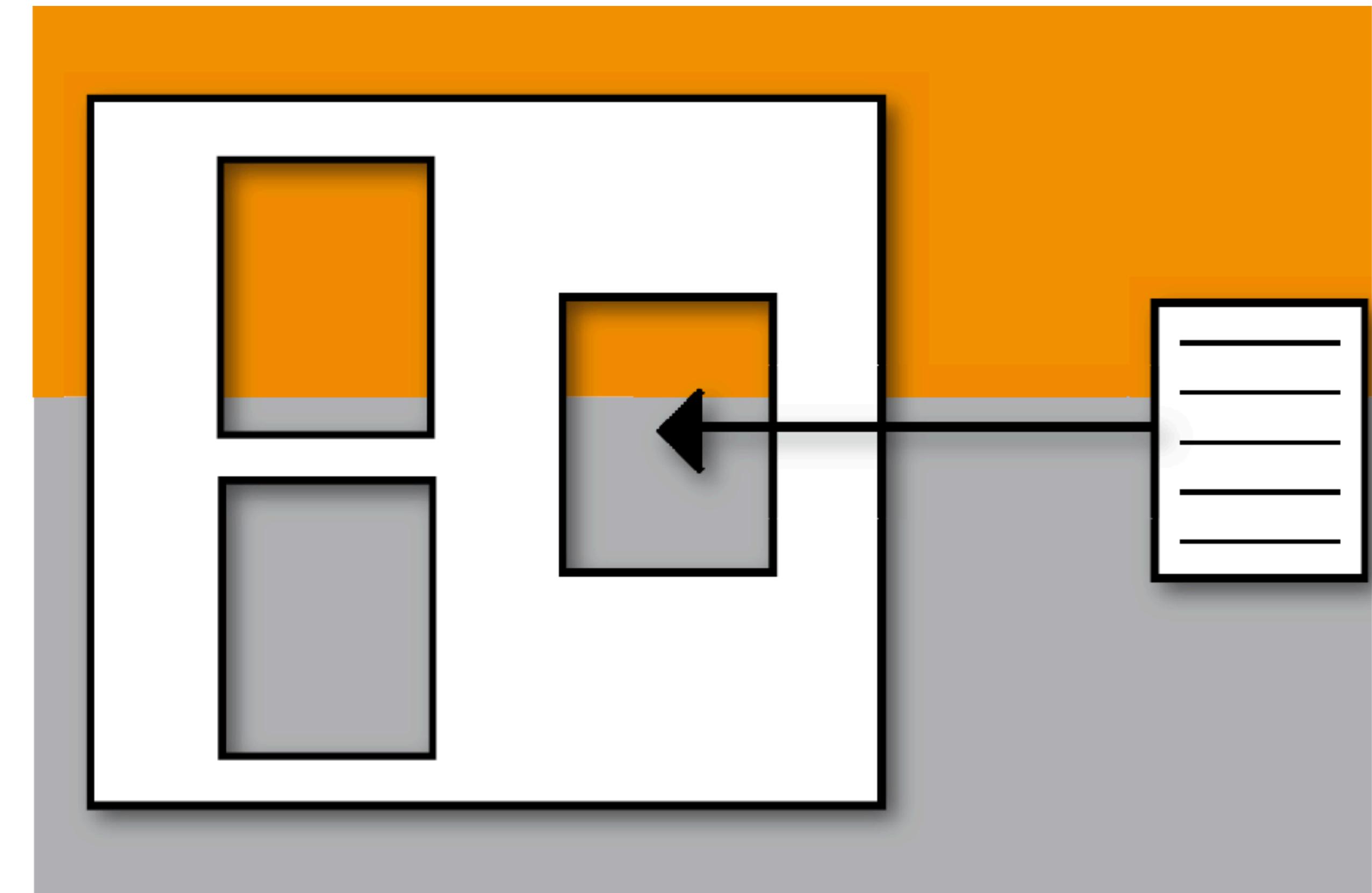
架构设计师要抽取业务共性和业务方法，业务共性通过基类
(可以包含默认方法-hook) 表现，业务方法通过子类表现

框架设计必须课

HollyWood原则：don't call us, we'll call you

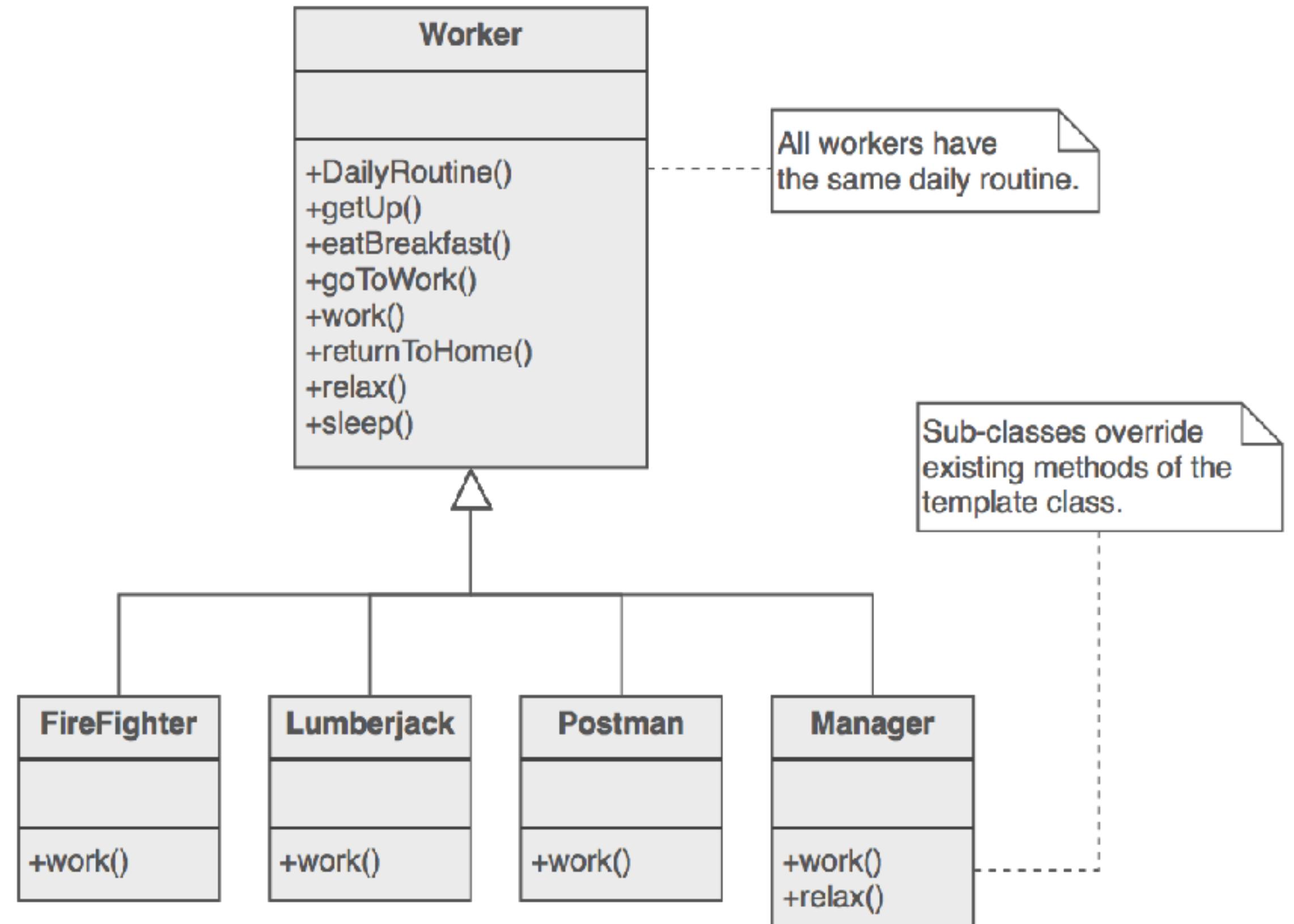
策略实现整个算法，模版实现部分算法（步骤）

工厂方法是策略的一个特例

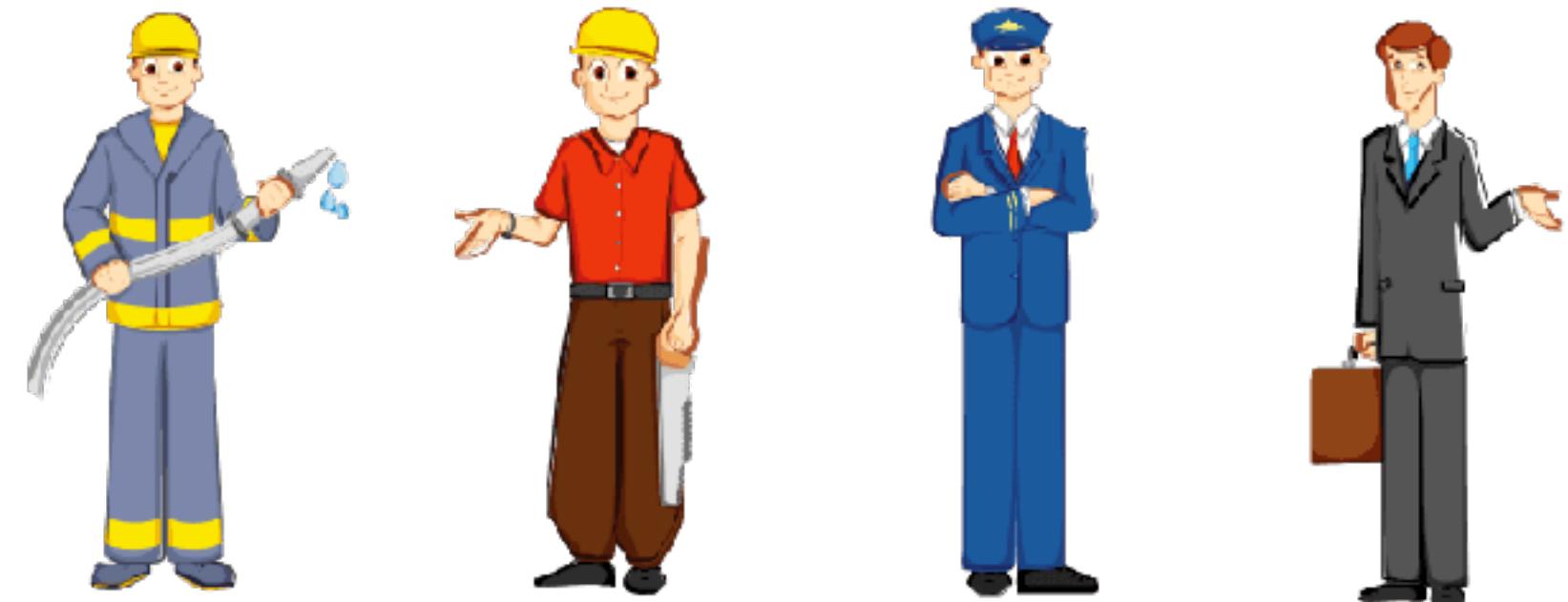
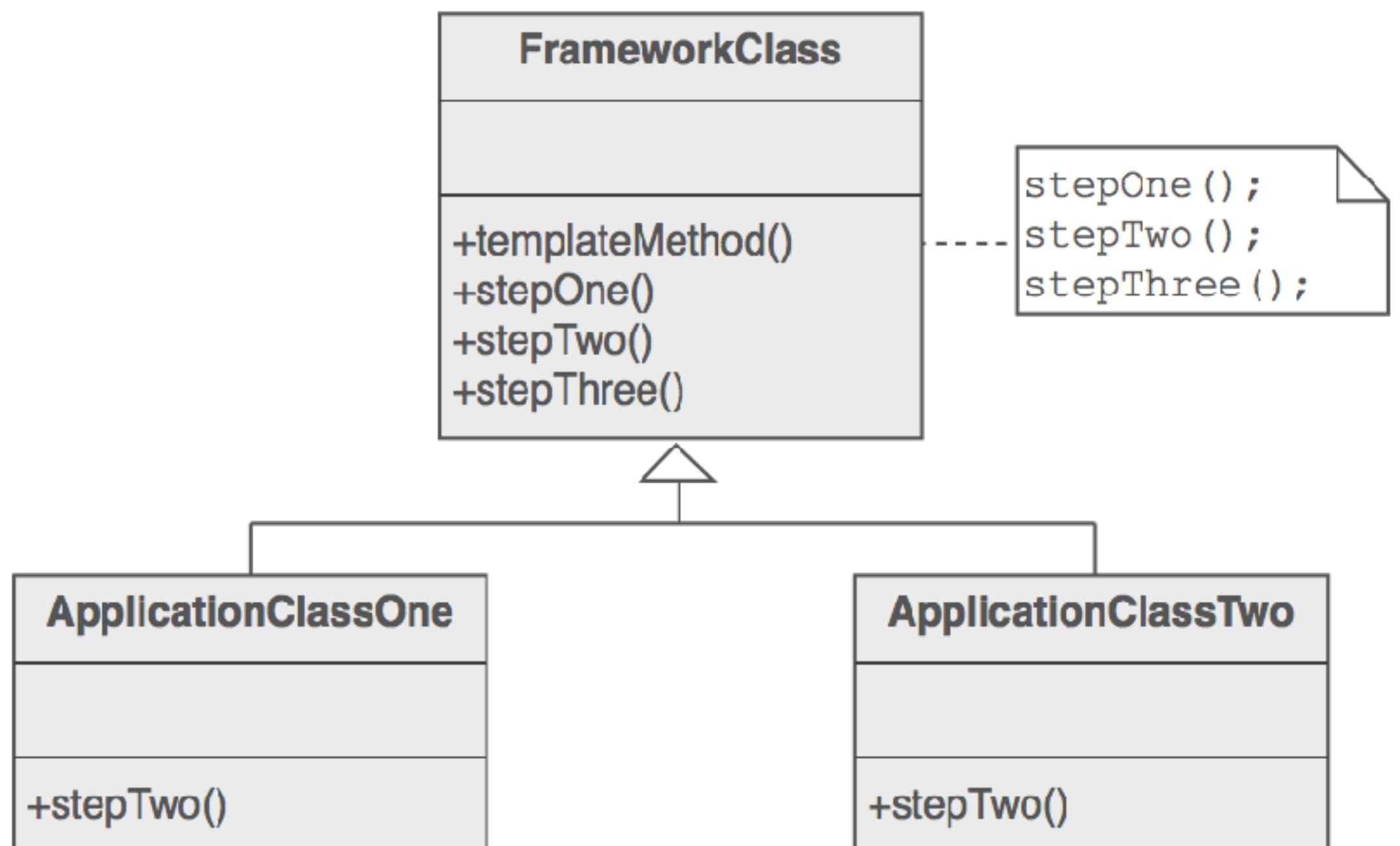


行为型模式

实例：房屋建筑工



模版模式



访问者模式

实现方法插件化，实现运行时动态添加方法

visitor -> element

“全局方法适配器”

用于定义新的操作而不需要改变运行它的类

运行不同的，不相关的操作，而不污染执行操作的类

目的是函数分解-分离算法和数据结构，设计轻量级的元素类

添加新操作变得容易，通过实现新的访问者；而元素类变得复杂

单向分发：操作依赖于方法 + 接受者（element）

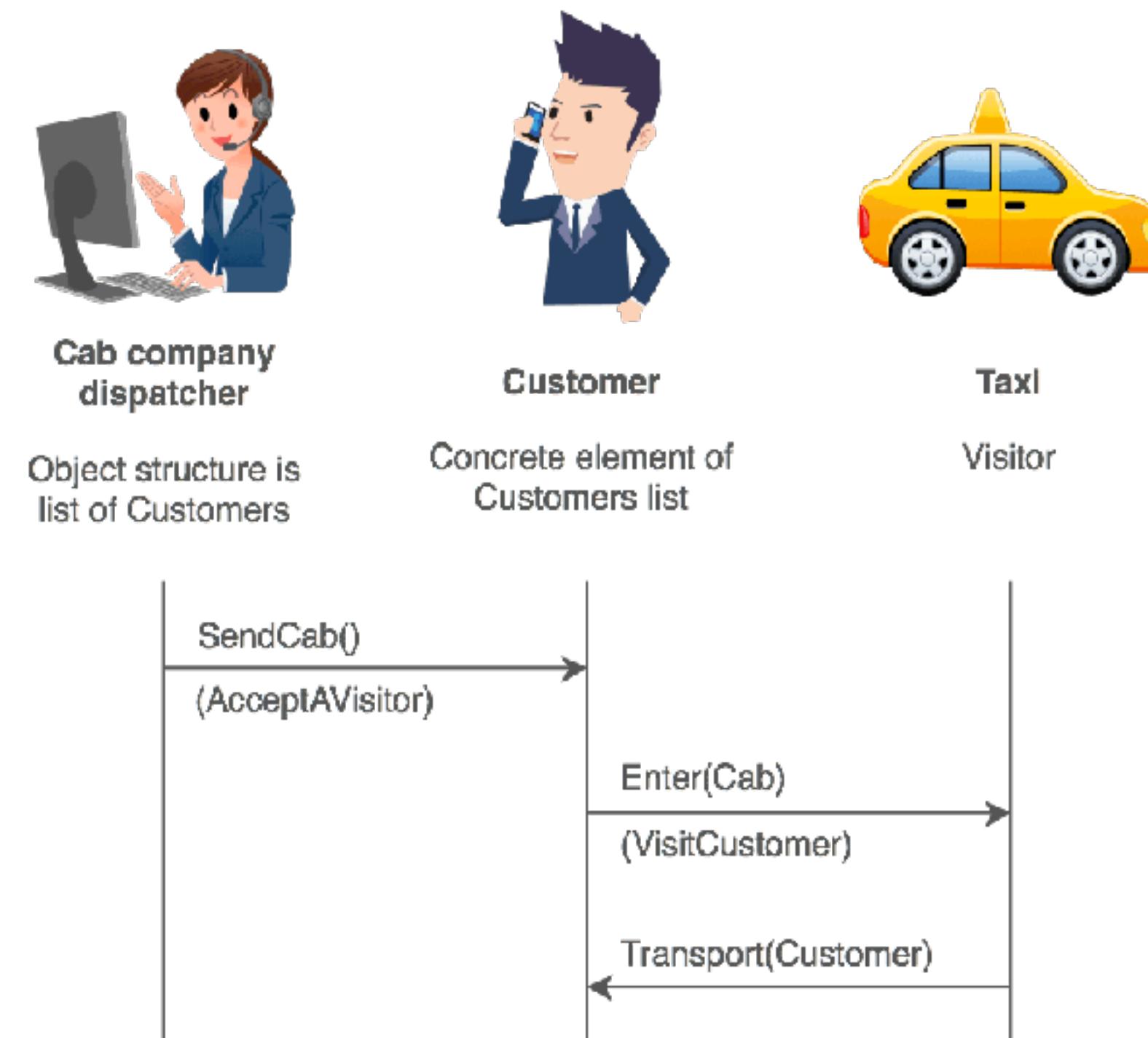
双向分发：操作依赖于方法 + 双方接受者类型（visitor+element）



访问者模式

- 强化的命令模式，访问者可以初始化任何它觉得合理的
- 典型的恢复丢失类型信息的技术，而不是动态计算

实例：出租车公司



作业

1. 选用适当的设计模式，重新实现杂货店程序
2. 通过应用场景理解和记忆设计模式



Thanks!

Any questions?