软件工程

大连理工大学软件学院

第8章 设计优化

- 高质量的设计，一方面要忠于设计，另一方面要能够尽可能地适应未来的变化，做到以不变应万变。
- 绝对应万变的设计是不存在的，我们可以做的是尽可能提升设计方案对变化的适应能力。
- 高质量的建模要以简单和小为基本观点，从方法级开始，逐渐过渡到组件、架构及更高级别的组织策略，对设计原则进行体现。
- 软件模式是在开发实践中针对经常发生问题的相同或相似解决方案，是设计经验的总结与重用。

小即是美

- “小即是美”体现出设计思想的灵活性、完整性与轻量性，是优化的基础。
- **KISS(Keep It Simple Stupid)：人们应该总是选择那些尽可能简单的实现方案，因为它们既能够全面解决问题，且具有较好的可理解性。**
- **YAGNI(You Ain't Gonna Need It)：这条原则的含义是“你不会需要它”，是指开发者自以为有用的功能，实际上都不会需要。**

设计的“味道(smell)”

- 僵化性：隐藏的设计关联性导致对文档或代码进行小的修正却埋下了不可预期的隐患，这将导致系统的改动困难重重。
- 脆弱性：在进行一个改动时，可能会导致程序许多没有概念关联的地方出现问题。
- 顽固性：分离设计中包含的有价值的部分并进行重用的付出和风险是巨大的。
- 粘滞性：在添加新功能时只是在现有代码的基础上拼凑代码，不愿意也不敢去触碰现有代码，不对代码重构，导致原有设计的破坏和退化。

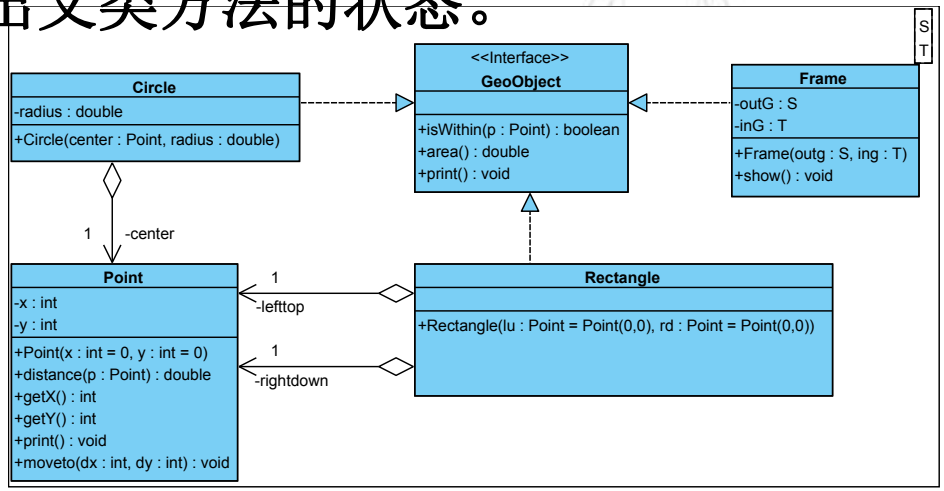
设计的“味道(smell)”

- 不必要的复杂性：设计人员预测需求的变化，为过多的可能性做准备，致使设计含有了绝不会用到的内容，无法带来回报。
- 不必要的重复性：设计中含有重复的内容，而这些重复的内容本可以使用单一的抽象进行统一，导致修改无法保持一致。复制和粘贴的编码习惯是导致大量重复代码的主要根源。
- 晦涩性：模块难以阅读和理解，代码随着时间而演化，变得越来越晦涩，逐渐丧失清晰性和表达力。

设计的优化

- 运行时的多态：多态性在结构上形成类的继承层次。
- 重写（Override）的要求：
  - 重写的方法本质上与父类方法具有相似的行为，但在细节上进行了有针对性的调整。
  - 重写的方法与原方法在相同的条件作用下工作，子类的方法不应具有比其父类更严格的条件限制。
  - 重写的方法最高不能超出父类方法的状态。

运行时多态使方法的调用动态绑定，可减少**switch**的使用，减少分散的重复的但又相同的计算逻辑。

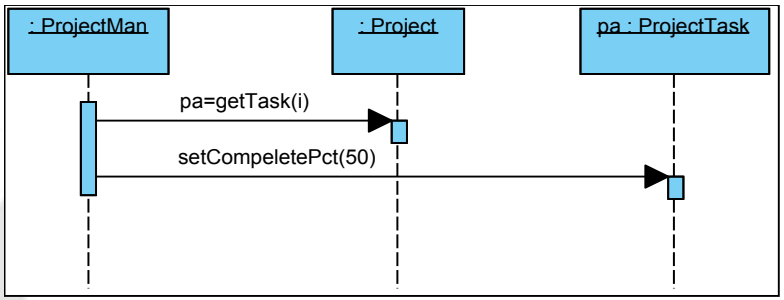


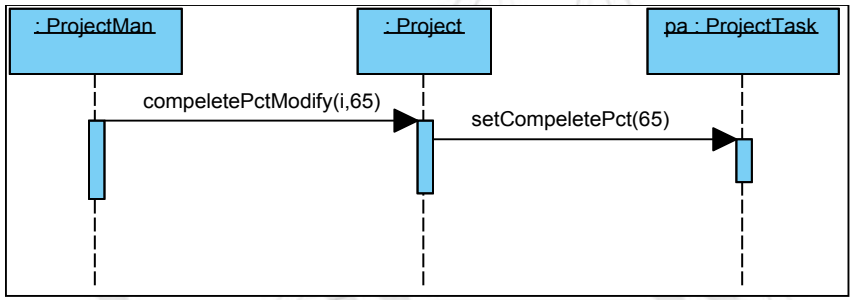
设计的优化

- 耦合的消息链：交互集中的设计与交互分散的设计。

this.getSelectedProject().getTask(i).setCompeletePct(50)

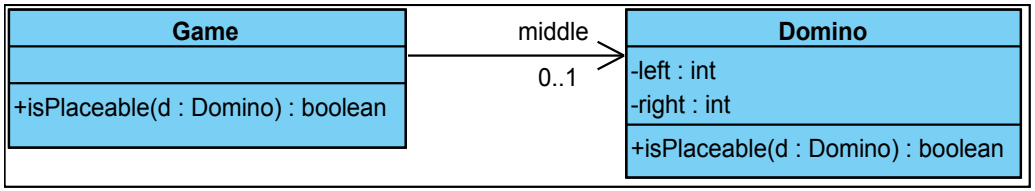
- 分散交互设计的好处是每个类只需了解与其本身相关的子任务，避免了不得不去了解其他所有类的情况，分散的交互方式使得模块的组织简单化。





设计的优化

- 狎昵关系




```
//in Game：两个类间表现出过分亲密的关系，即高耦合。
public boolean isPlaceable(Domino d){
   return d.getLeft() == middle.getRight()
                        || d.getRight() == middle.getLeft();
}
```

```
//in Game Class
public boolean isPlaceable(Domino d){
   return middle.isPlaceable(d);
}
//in Domino Class
public boolean isPlaceable(Domino d){
   return left == d.right || right == d.left;
}
```

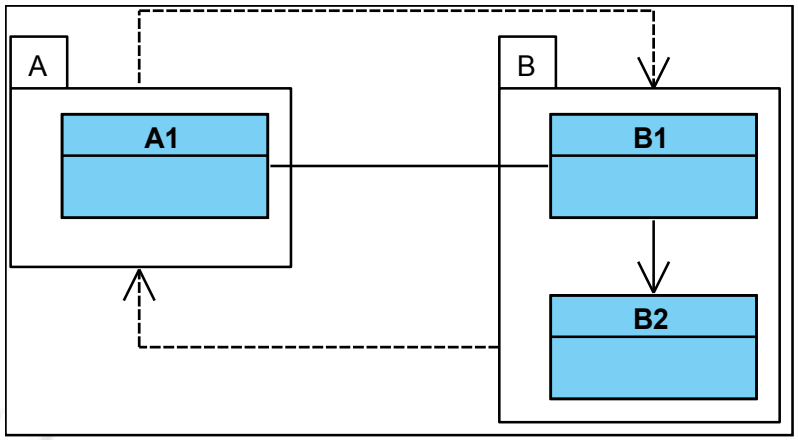
- 面向对象提供的封装机制并不能完全保证模块的独立性
- 必须要在设计中充分考虑业务结构的特点并做出利于稳定的设计决策才行

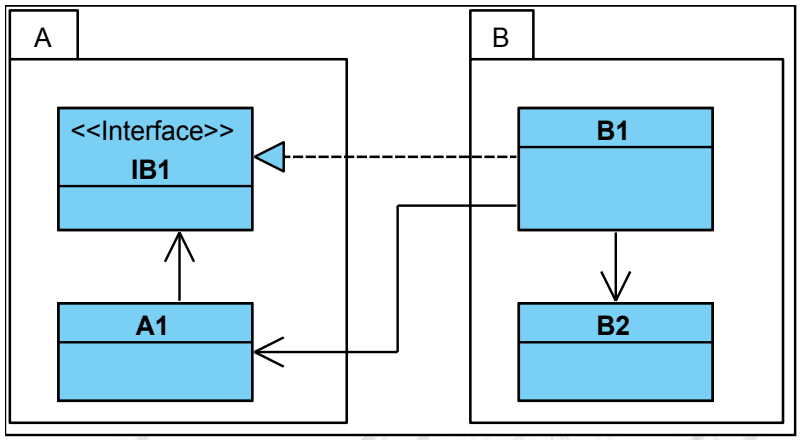
设计的优化

- 被拒绝的遗赠：
- 类功能和结构的扩展：
- 一是借助继承的方式。但当子类只需要重用父类的某些功能，对父类的行为并不想保持一致，这就是拒绝的遗赠现象，预示着继承层次的误用，应考虑第二种方式进行扩展。
- 二是通过所谓的委托(Delegation)进行，其基本形式是新类与原有类构成关联关系，即具有一个它的实例变量。
  - 新的功能会非常容易地在新类中进行重用，而且与上层类之间的依赖也不再强烈。

设计的优化

- 循环依赖：如果两个类分别处于不同的包中，并且具有双向导航的关联关系，这样的循环依赖好像是不可避免的。





- 通过调整打破两个包间的循环依赖状态，整体上只保留包B对包A的依赖。

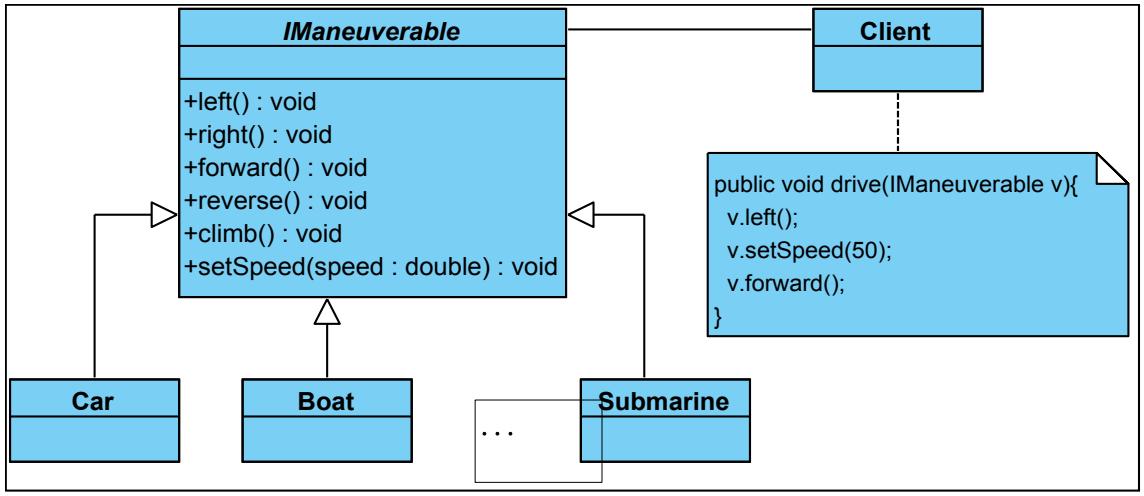
设计的基本原则

- 软件设计是一门艺术，是对“变化”的辩证处理：发现变化，隔离变化，以不变应万变。
- 最基本的设计原则是对面向对象思维的深化和具体。
- 基本设计原则的综合使用是产生高质量软件设计的基石。

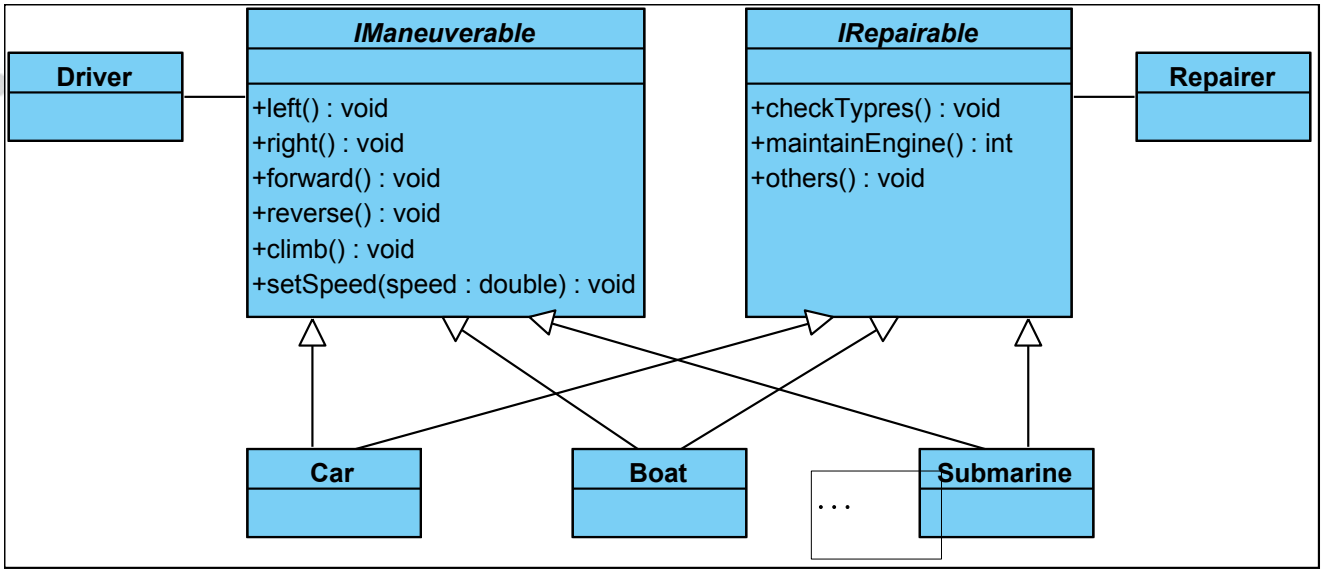
接口隔离原则

- 接口隔离原则（The Interface Segregation Principle, ISP）有两层含义：
- 应尽量使用“接口继承”，而非“实现继承”。接口关注对象的概貌，将对象中“不变”的信息抽象出来，不涉及细节，因此是“稳定”的。
- 通过接口只将需要的操作“暴露”给客户类，而将不需要的操作隐藏起来。接口在这里充当类的视图。

- 接口类IManeuverable是具体交通工具的抽象，提供了驾驶的接口，如加减速、转弯
- 类Client在使用（drive）交通工具时，使用的不是具体类，而是其接口类
- 好处就是当业务需求变化时，更容易发生改变的是具体类，而这些变更可以通过稳定的抽象类进行隔离，使得Client不受变化的影响，从而提高了系统的可维护性。



- 接口指的是广义上的抽象类，可以是编程语言中的**interface**或**abstract class**。如**C++**通过继承纯虚类来实现接口继承；**Java**对接口继承具有单独的语言构造方式。



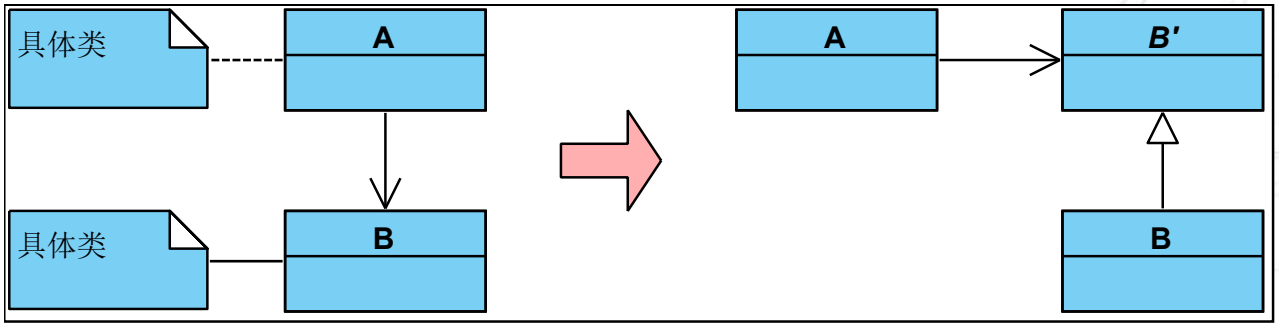
- 如果系统中还存在另一类使用者，比如维修者，则他们专注的应该是这些交通工具的运转，而与驾驶相关的操作是不应该暴露给这些维修者类。

面向接口的作用

- 面向接口的设计能够使Client只需关注如何进行业务活动（如：驾驶），而不必关心其使用对象的具体实现。
- 一个对象可以很容易地被（实现了相同接口的）另一个对象所替换，这样对象间的连接不必硬绑定（hard wire）到一个具体类的对象上，因此增加了灵活性。
- 这是一种松散的耦合，同时增加了重用的可能性。

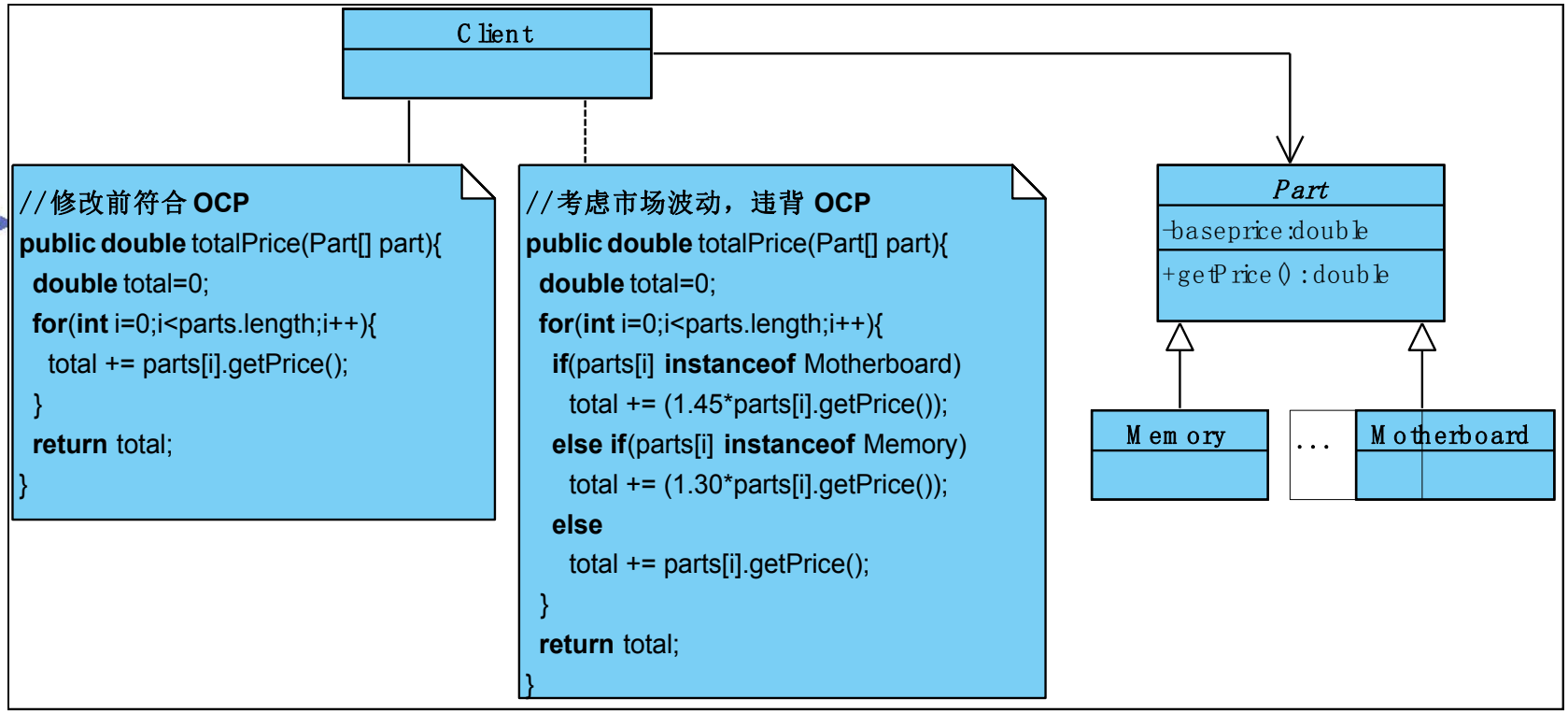
依赖倒置原则

- 依赖倒置原则（Dependency Inversion principle, DIP）宗旨是应依赖于抽象，而不要依赖具体
- 扩展的基础越具体，扩展的难度也越大，具体类的变化无常势必造成扩展类的不稳定
- 依赖倒置原则使细节和具体实现都依赖于抽象，抽象的稳定性决定了系统的稳定性
- 一个基础稳定的系统要比一个基础不稳定的系统在整体上要更“稳定”一些

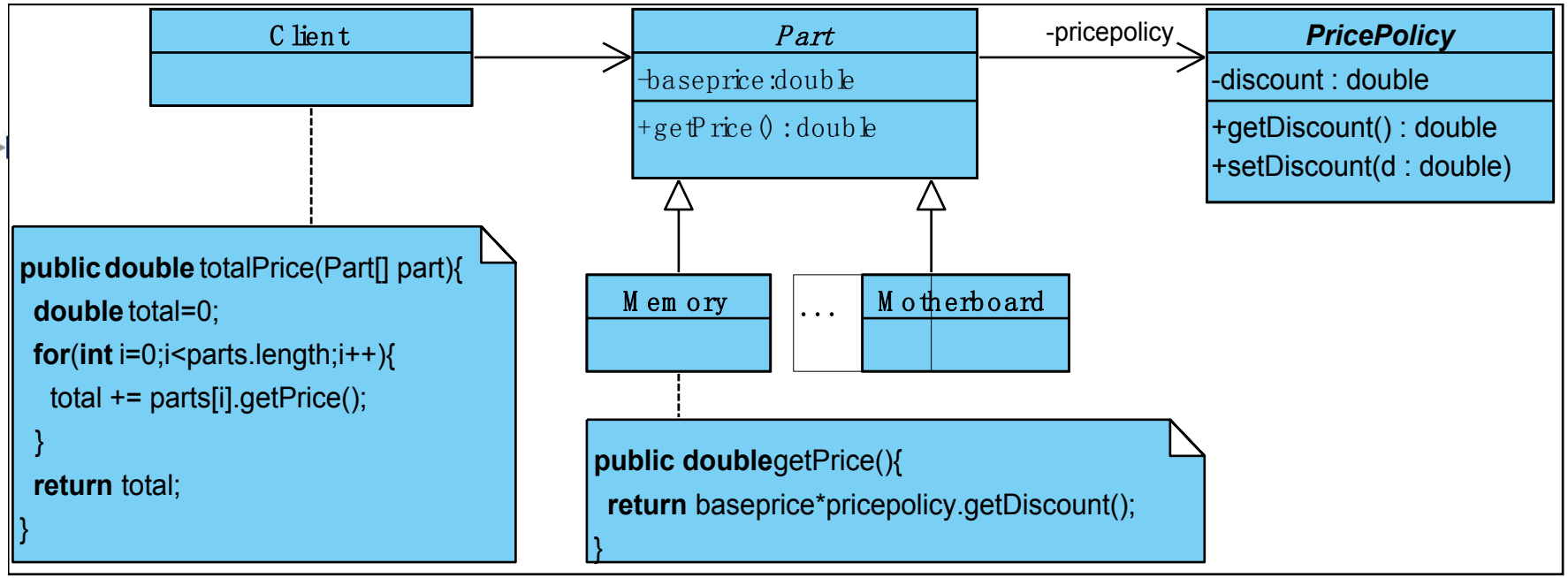


开放封闭原则

- 开放封闭原则（The Open-Closed Principle, OCP）：一个模块对扩展应是开放的，而对修改应是封闭的。
- 这条原则是面向对象思想的最高境界，即设计者应给出对于需求变化进行扩展的模块，而永远不需要改写已经实现的内部代码或逻辑。
- 两个基本的特点：
  - 模块的行为可以被扩展，以需要满足新的需求。
  - 模块的源代码是不允许进行改动的。
- OCP是相对的，没有绝对符合OCP的设计，而且一个软件系统的所有模块不可能都满足OCP，要做的是尽量最小化不满足OCP的模块数量。



- 一个满足**OCP**的设计，**Client**很容易的计算所有零件的价格之和，并且当有新的零件加入时只需继承**Part**抽象类即可，现有的代码不需要做任何修改。
- 可是，但需求发生了变化，如在计算零件价格和的时候还需要考虑市场的价格波动情况，则可以修改代码如图中的注释部分，但这将导致该设计不符合**OCP**。



- 作为对以上设计的一种改进，需要将波动系数的处理融入业务逻辑中，引入折扣（**discount**）属性。
- 但这一属性放在什么地方？显然它不是类**part**的自然属性，因为这一属性是与价格策略相关的，为此引入类**PricePolicy**，与类**Part**呈关联关系，这样做的好处是零件对象与价格策略对象的对应是动态的，可以在程序运行时（**runtime**）动态改变。

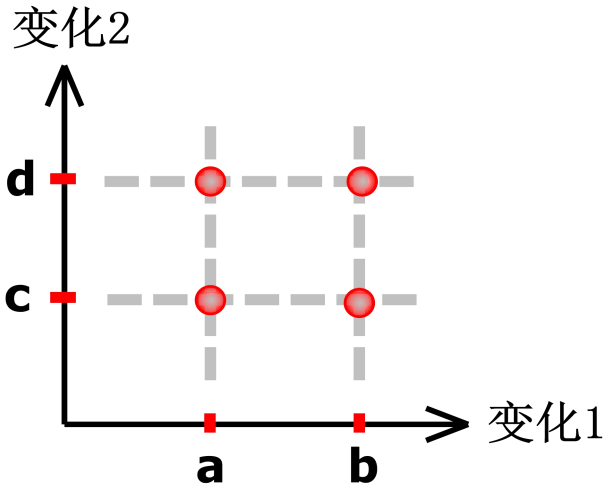
Liskov替换原则

- **Liskov替换原则（Liskov Substitution Principle, LSP）：任何出现父类的地方应能使用子类对其进行无条件的替换，即当使用子类对其父类进行替换时，该组件仍象替换前一样正常工作。**
- **LSP要求对象间的继承关系既与静态属性相关又与动态行为相关。**
- **通常规定父类型在使用前和使用后都要具备必要的条件——前置条件和后置条件。**
- **当子类型替换父类型后不能违反父类型中的前置条件和后置条件，即一个子类型不得具有比父类型更多的限制，这是因为可能对于父类型的某些使用是合法的，但是会因为违背子类型的其中一个额外限制，从而违背了LSP！**

对应代码

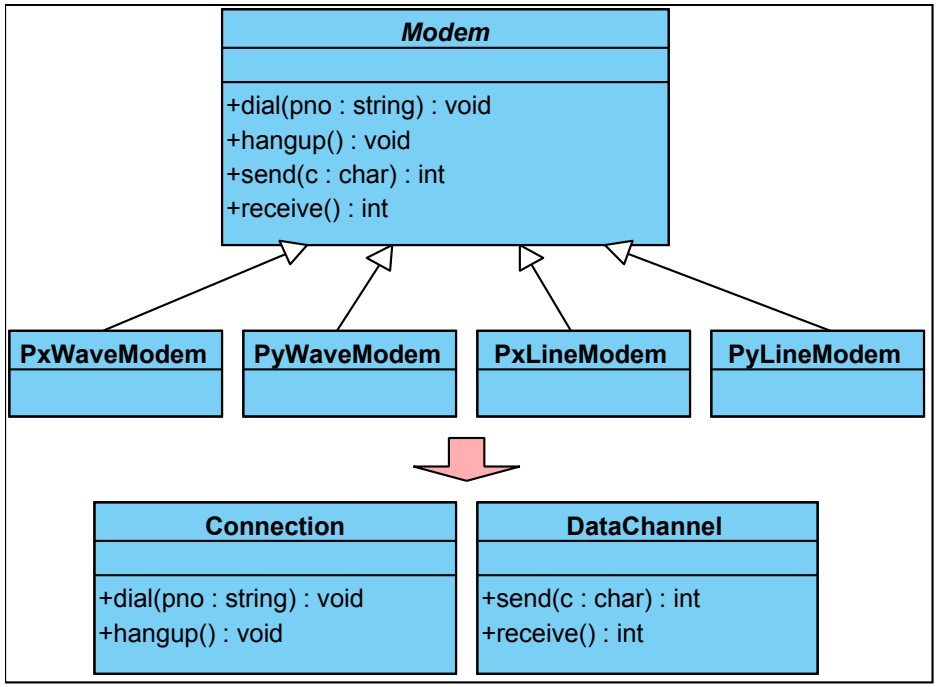
单一职责原则

- 单一职责原则（Single Responsibility Principle, SRP）中所谓职责，可理解为功能，就是设计的类功能应该只有一个，而不应为两个或多个。
- 职责是引起“变化”的原因：当一个类中有两个以上的变化方向，会产生过多的变化点。



- 如果有n个变化方向，每个变化方向上有m个变化点呢？
- 而且很多子类存在浪费，如果用户在此变化点上根本没有需求！

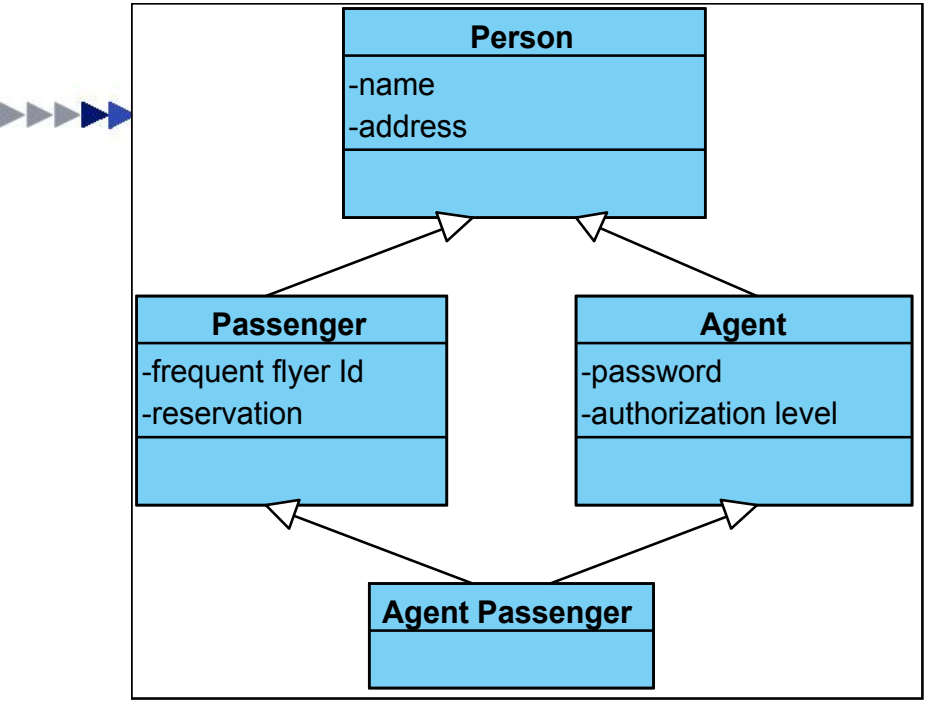
- 如何应对这样的状况？答案就是拆分变化！
- 拆分后设计并未结束，两个拆分的抽象类肯定不是也不应该独立存在，下一步的设计？
- SRP条件是比较苛刻的，一个类真的要做到只能有一个功能而一点儿其它功能也不能具有？答案同样是否定的。



- 多个功能在一个类中是可以同时存在的，但这里有个前提：是否能够成为变化的方向。
- 如果成为单独的变化方向，则应该按照SRP进行类职责的拆分，否则可以保留功能共存（装饰模式）。

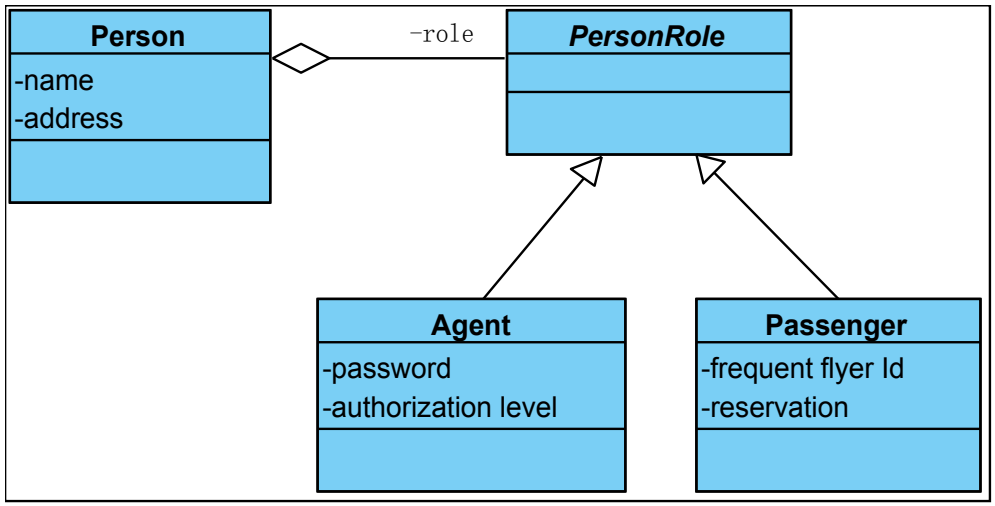
合成/聚合复用原则

- 合成/聚合复用原则（Composite/Aggregate Reuse Principle, CARP）中的合成与聚合是两种特殊的关联关系，是以委托方式实现对象间功能的重用(另外一种面向对象特有的重用方式是继承)。
- 委托重用与继承重用是两种本质上不同的重用方式，委托重用追求的是对象间的独立性即低耦合，而继承重用追求的是对象间应能尽可能的高内聚。
- 合成/聚合复用原则指的是应尽量使用合成/聚合形式的委托重用，尽量不使用继承重用。



一个违反了委托重用优先的例子

- 多重继承中类型的确定是静态的，即人的身份已经确认，无法在运行时动态更改，而“乘客”的身份本身就是动态的。
- 多重继承需要生成大量的底层类，以适应具有不同身份组合的人群，如同时具有学生、代理、乘客、实习者等身份的人。
- 这些问题产生的原因就是类“人”和类“身份”其实是两个耦合性很低的实体，我们把他们生拉硬套放在了一起，违反了事物的本质特性。



- 利用CARP原则降低两者之间的耦合，从而将两者之间的真实特性表现出来。
- 因此，优先使用关联的聚合或组合可获得重用性与简单性更佳的设计。
- 另外，配合使用继承，可以扩充可用的组合类集，加大重用的范围。

继承的使用

- 当然，在条件满足的时候继承也是推荐使用的，如：
- 子类表达了“是一个父类的特殊类型”，而非“是一个由父类所扮演的角色”。
- 子类的一个实例永远不需要转化为其它类的一个对象。
- 子类是对其父类的职责进行扩展，而非重写或废除，因为这会增加违反LSP的可能性。

模式

- 模式是表示周境（Context）、动机（System of Forces）、解决方案（Solution）三个方面关系的一个规则，每个模式描述了一个在某种周境下不断重复发生的问题，以及该问题解决方案的核心所在。
- 模式是一个经验提取的“准则”，并且在一次一次的实践中得到验证，在不同的层面上，模式提供不同层面的指导。
- 根据处理问题的粒度不同，从高到低，模式分为3个层次：架构模式（Architectural Pattern）、设计模式（Design Pattern）和实现模式（Implementation Pattern）。

模式的层次

- 架构模式是模式中的最高层次，描述软件系统里的基本的结构组织或纲要，通常提供一组事先定义好的子系统，指定它们的责任，并给出把它们组织在一起的法则和指南。比如N-层架构、MVC架构模式等。
- 一个架构模式常常可以分解成很多个设计模式的联合使用。设计模式是模式中的第二层次，用来处理程序设计中反复出现的问题，比如GOF总结的23个基本设计模式——Factory Pattern，Observer Pattern等等。
- 实现模式是最低也是最具体的层次，处理具体到编程语言的问题。比如，类名、变量名、函数名的命名规则以及异常处理的规则等。

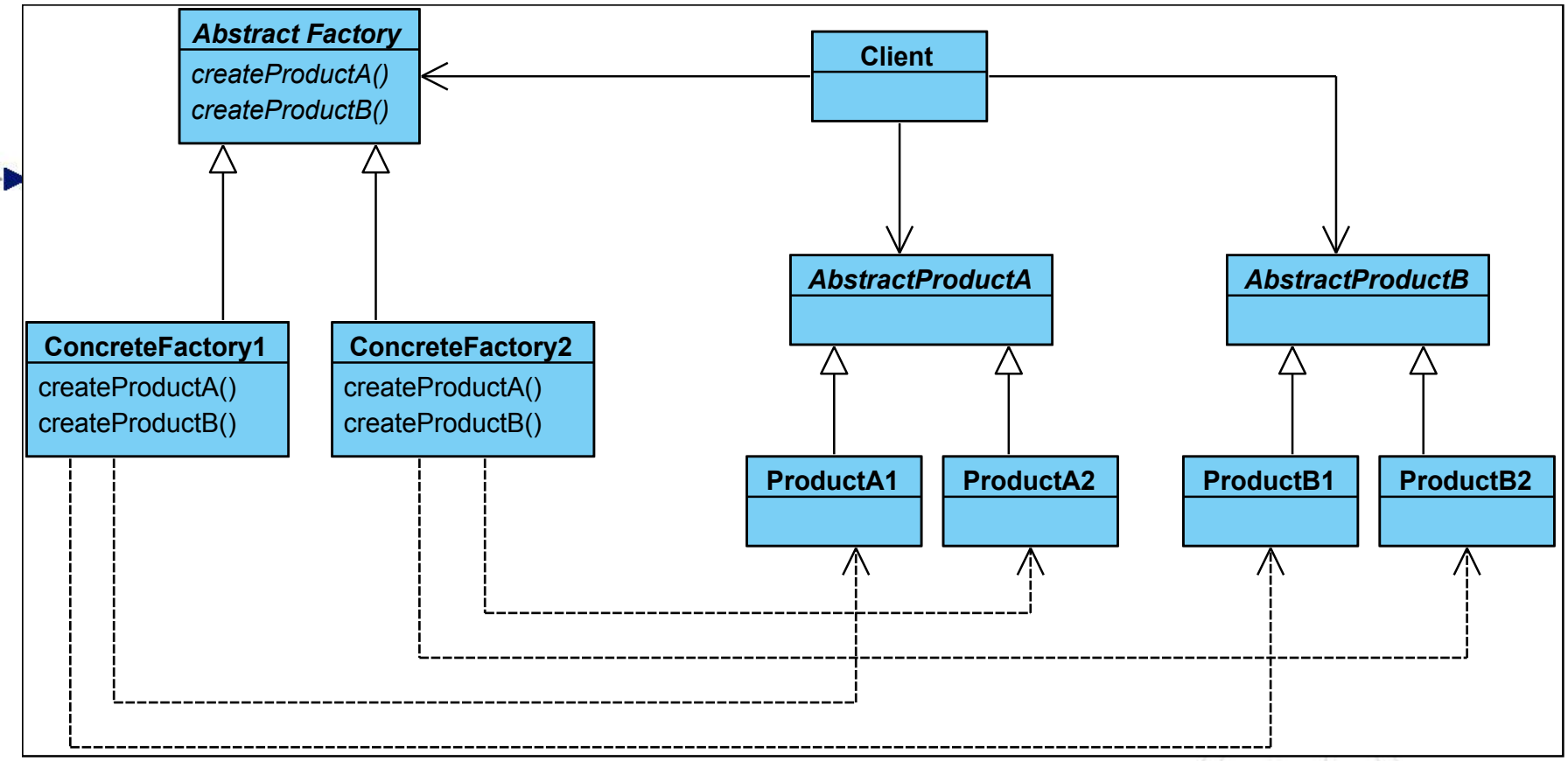
设计模式

- 设计模式提供了对相似的程序设计任务中经常出现的相同问题的解决方案。
- 基于工程化的方法能够保证这些相似任务被识别出来然后通过对设计模式进行较小的调整从而快速产生对问题有效的解决方案。
- 设计模式提供了一种在设计层次上的重用机制，其特点是对解决方案的进一步抽象，一方面模式针对的是抽象的设计思想的重用，而非代码的物理重用；另一方面模式并不对具体问题提供完整的解决方案，而是提供方案的一种结构。

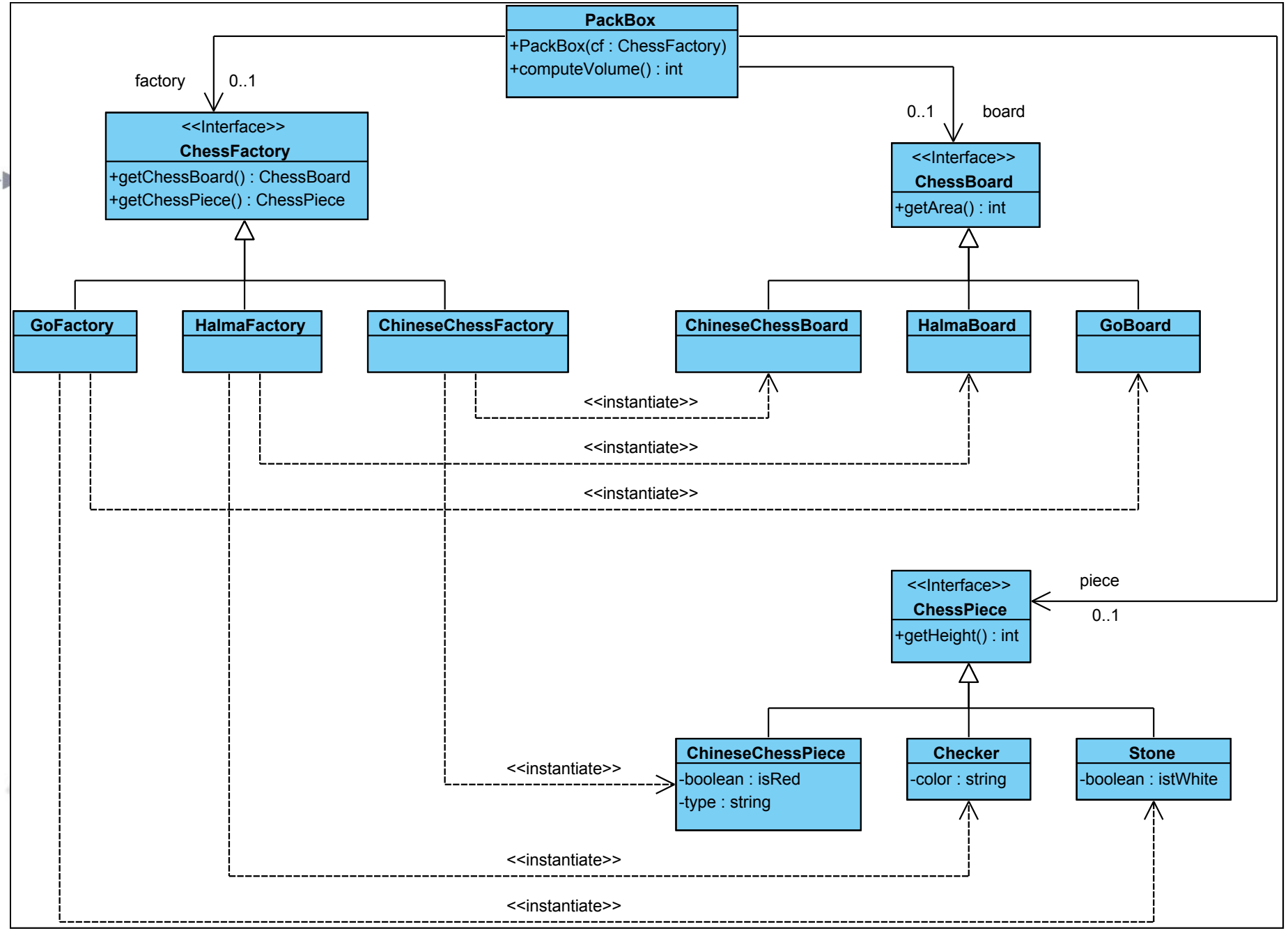
		目的		
		创建模式	结构模式	行为模式
范围	类	Factory Method	Adapter	Interpreter
				Template Method
	对象	Abstract Factory	Adapter	Command
		Builder	Bridge	Observer
		Prototype	Composite	Visitor
		Singleton	Decorator	Memento
			Facade	Strategy
			Proxy	Mediator
			Flyweight	State
				Iterator
				Chain of Responsibility

抽象工厂模式

- 抽象工厂（Abstract Factory）模式的主要作用是实现了客户类在创建产品类时引入的耦合，如在对具体对象创建时使用的new操作，需要指定一个具体的产品类的名字，这样就在客户类和具体产品类之间引入了依赖关系，而这种依赖关系按照面向接口编程等原则是应该进行优化处理的。
- 将产品的创建过程从客户类中分离，通过使用一个类似系统服务的工厂类来解决这个问题，工厂类提供了一个创建一系列相关或相互依赖对象的接口，而客户类无需指定它们需要的具体产品类。



这个类图中客户类对具体产品类没有直接的依赖关系，这是通过抽象工厂类进行了分离。另外，每个抽象工厂类提供了对不同系列产品的创建。


PackBox
+PackBox(cf : ChessFactory)
+computeVolume() : int
factory
0..1
<<Interface>>
ChessFactory
+getChessBoard() : ChessBoard
+getChessPiece() : ChessPiece
0..1
board
<<Interface>>
ChessBoard
+getArea() : int
GoFactory
HalmaFactory
ChineseChessFactory
ChineseChessBoard
HalmaBoard
GoBoard
<<instantiate>>
<<instantiate>>
<<instantiate>>
<<Interface>>
ChessPiece
+getHeight() : int
piece
0..1
ChineseChessPiece
-boolean : isRed
-type : string
Checker
-color : string
Stone
-boolean : istWhite
<<instantiate>>
<<instantiate>>
<<instantiate>>

- 一个抽象工厂模式的具体应用，其中主要关注一个棋类游戏的包装盒对象，该包装盒中又含有棋盘和棋子两种产品对象
- 对于不同的棋类游戏（如中国象棋、跳棋或围棋等），包装盒应具有不同的大小和体积，为了减少包装盒对象与这些棋类对象间的耦合程度，设计中使用了棋盘接口（**ChessBoard**）和棋子接口（**ChessPiece**）
- 同时利用一个抽象的棋类工厂（**ChessFactory**）来消除对这些棋类对象创建的耦合
- 这个例子中通过工厂模式的使用可以使得添加新的棋类游戏更加容易，只需从棋盘类和棋子类继承并进行实现即可；但对于新种类的游戏（比如积木游戏）则需要完全重建一套工厂类和产品体系。

对应代码

单例模式

对应代码

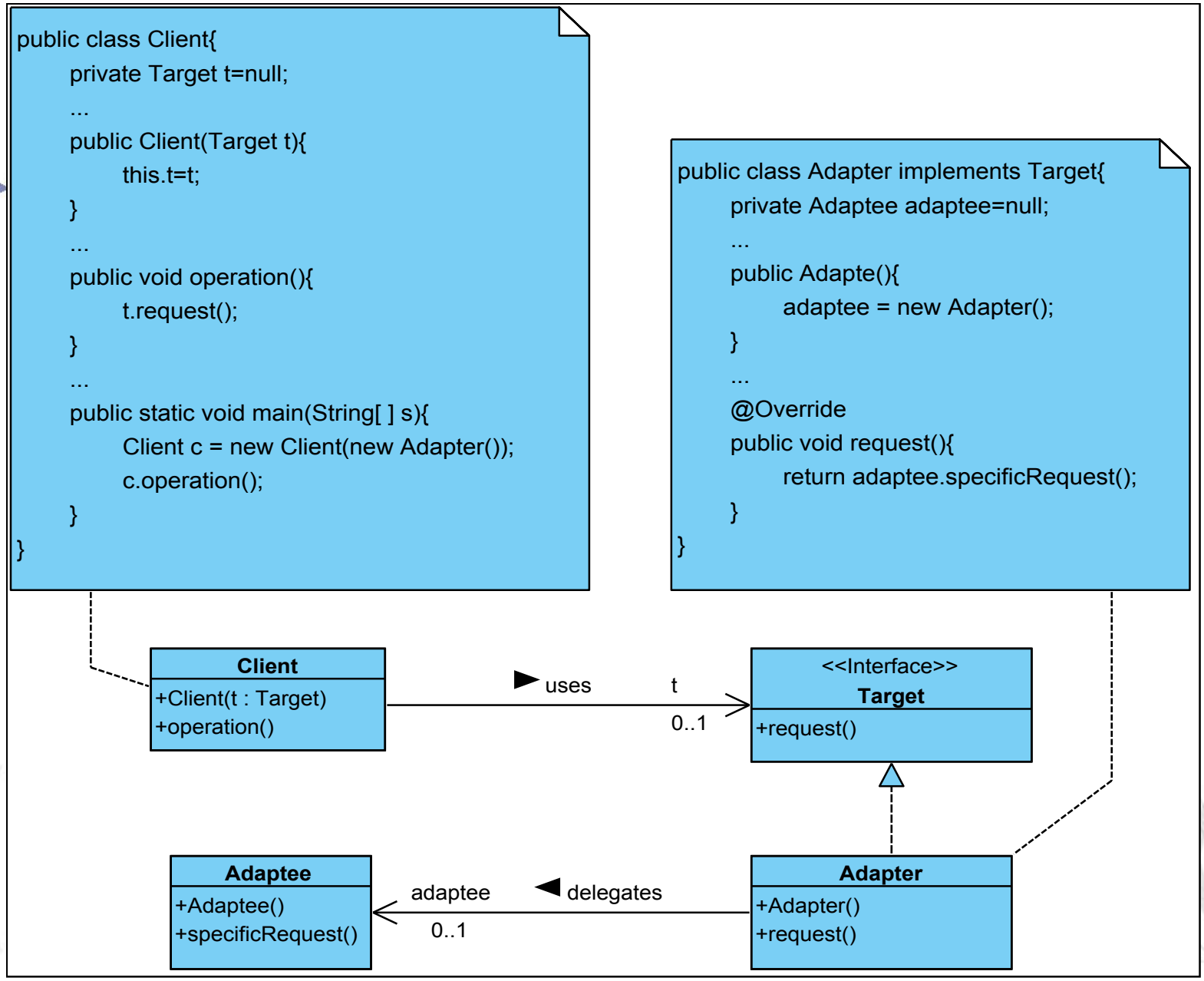
- 前面章节中介绍过了管理类或控制类的概念，它们的使用主要是为了方便对相关的业务类进行管理和维护，因此通常它们在系统中只存在一个实例。类似的，根据业务的需要，在系统中可能还存在一些这样的对象实例，它们只存在单一的实例并且不允许存在其副本，比如银行系统中每个账户类的实例。
- 单例模式（Singleton）保证了一个类仅有一个实例，并提供一个访问它的全局访问点。单例模式要求：(1) 类的所有构造方法都为私有的，防止其被外部创建；(2) 提供一个公有的方法获取该类的实例；(3) 类中的实例变量为私有或受保护的。

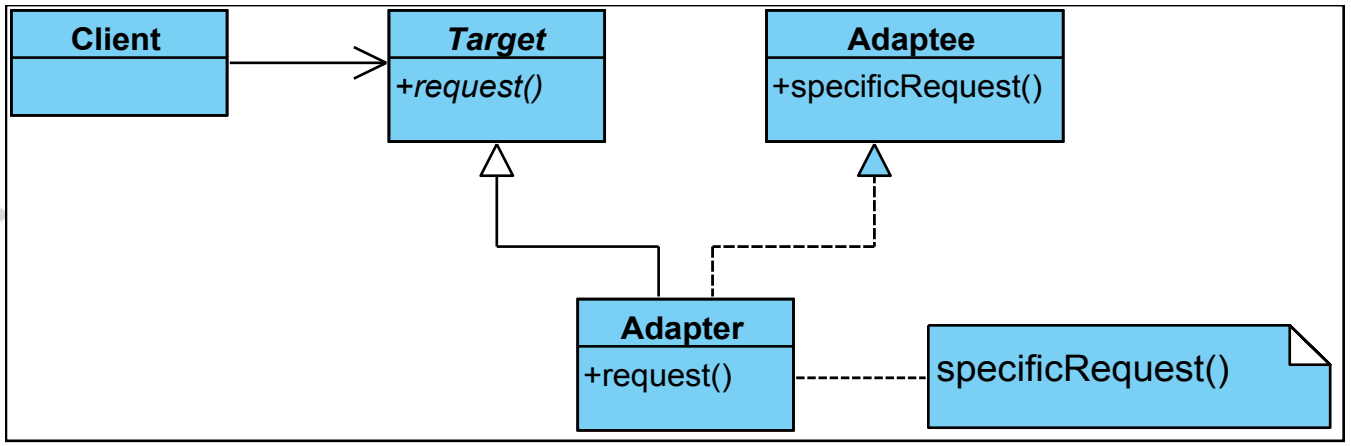
适配器模式

- 适配器模式（**Adapter**）把一个类的接口变换成客户类所期待的另一种接口，从而使原本因接口原因不匹配而无法一起工作的两个类能够一起工作
- 适配器一般有两种工作方式：一种是通过委托的方式，另外一种是通过继承（接口实现）的方式
- 无论哪种方式，适配器都可以充当被适配对象参与与客户类的交互，并可以对基本的适配功能做进一步的扩展，而这个功能扩展的作用又可以通过另外的“装饰模式”进一步描述

```
public class Client{
        private Target t=null;
        ...
        public Client(Target t){
                this.t=t;
        }
        ...
        public void operation(){
                t.request();
        }
        ...
        public static void main(String[ ] s){
                Client c = new Client(new Adapter());
                c.operation();
        }
}
```

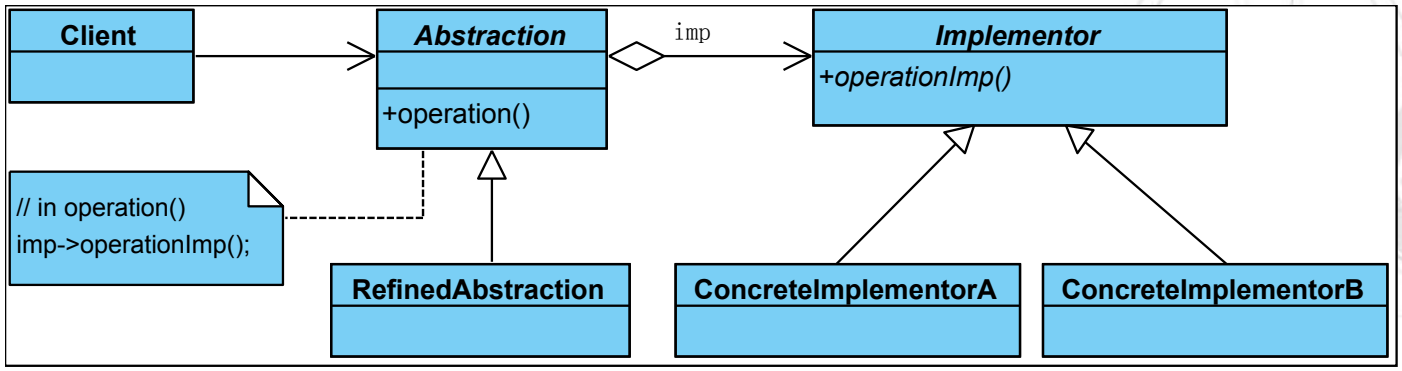
```
public class Adapter implements Target{
        private Adaptee adaptee=null;
        ...
        public Adapte(){
                adaptee = new Adapter();
        }
        ...
        @Override
        public void request(){
                return adaptee.specificRequest();
        }
}
```




Client
Target
+request()
Adaptee
+specificRequest()
Adapter
+request()
specificRequest()

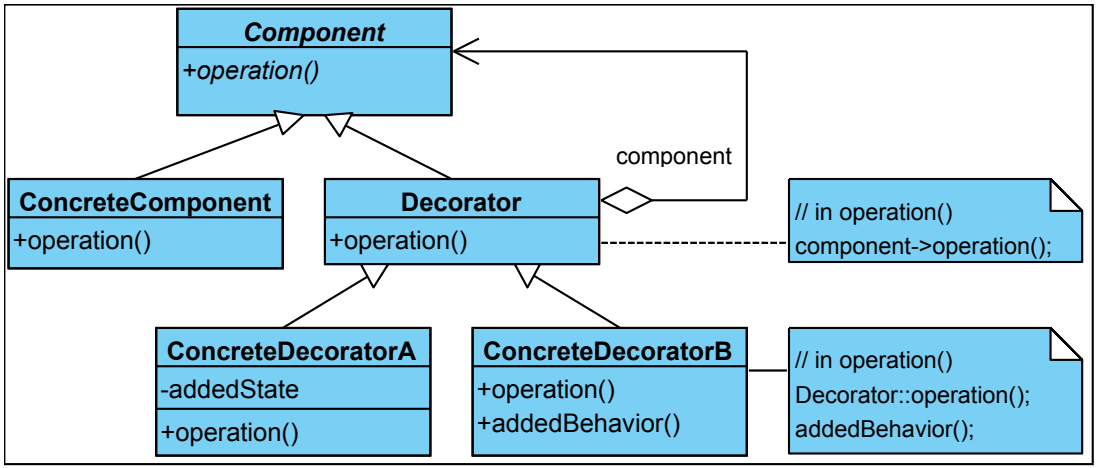
桥模式

- 桥模式的主要思想是将抽象部分与它的实现部分(行为)进行分离，使它们都可以独立地变化。
- 使用桥模式时，首先应该识别出一个类所具有的两个独立变化的维度，将它们设计为两个独立的继承等级结构，为两个维度都提供抽象层，并建立抽象耦合。
- 一般把业务方法和与之关系最密切的维度设计为“抽象类”层次结构(抽象部分)，而将另一个维度设计为“实现类”层次结构(实现部分)。



装饰模式

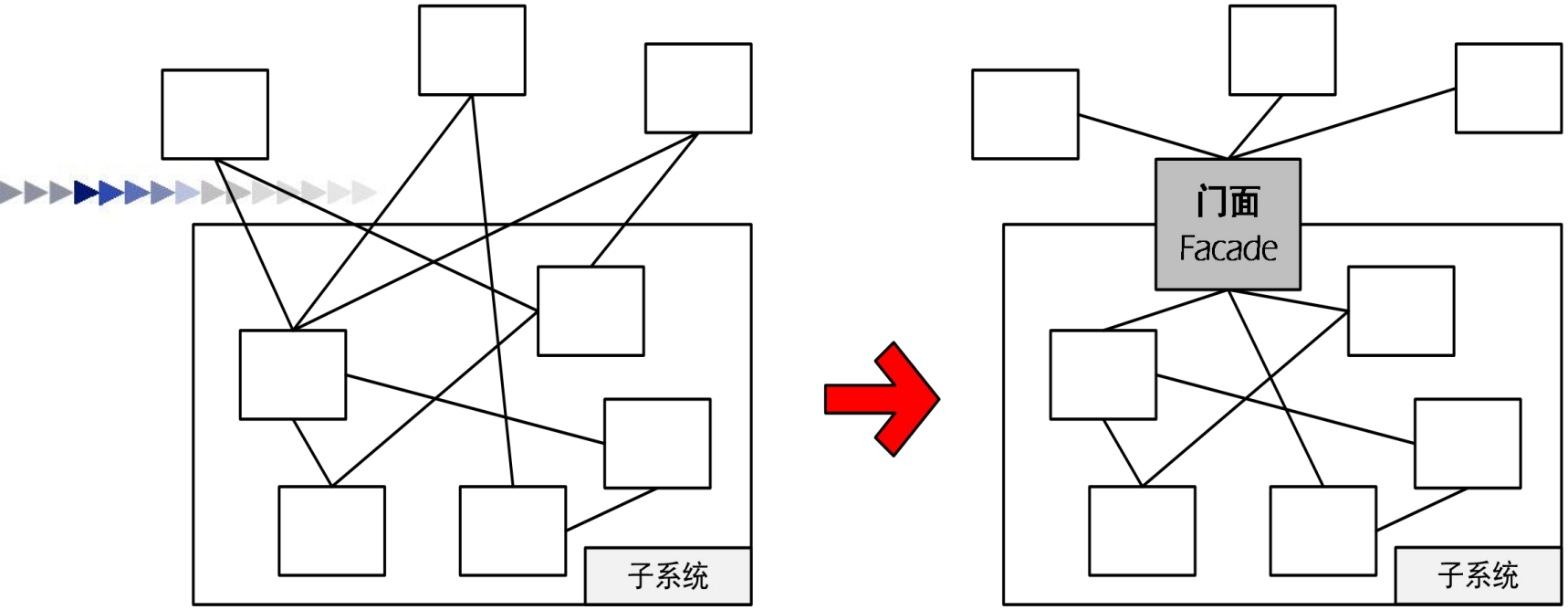
- 一个类可能有些额外的责任(除主体业务)，如加密、缓存、压缩等，这些可能只是辅助主体业务的附着，并不严格按照维度变化。



- 装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案，提供比继承更多的灵活性。
- 动态给一个对象增加功能，并可以再动态的撤消，因此增加由一些基本功能的组合而产生的非常大量的功能。
- 装饰模式中既有继承又存在组合，实际上是将Bridge中的抽象和实现合二为一了，是其特殊形式。

门面模式

- 门面模式（Facade）要求外部与一个子系统的通信必须通过一个统一的门面对象进行。
- 门面模式提供了一个高层次的接口，使得子系统更易于使用。另外，每个子系统一般只要求具有一个门面类，而且此门面类只有一个实例，也就是说它是一个单例模式。
- 这个时候的门面类作用相当于前面介绍的适配器，负责对外部请求的转发，并且可以在此基础上进行功能的扩充，如对传递进来的参数的验证等。整个系统可以有多个门面类。

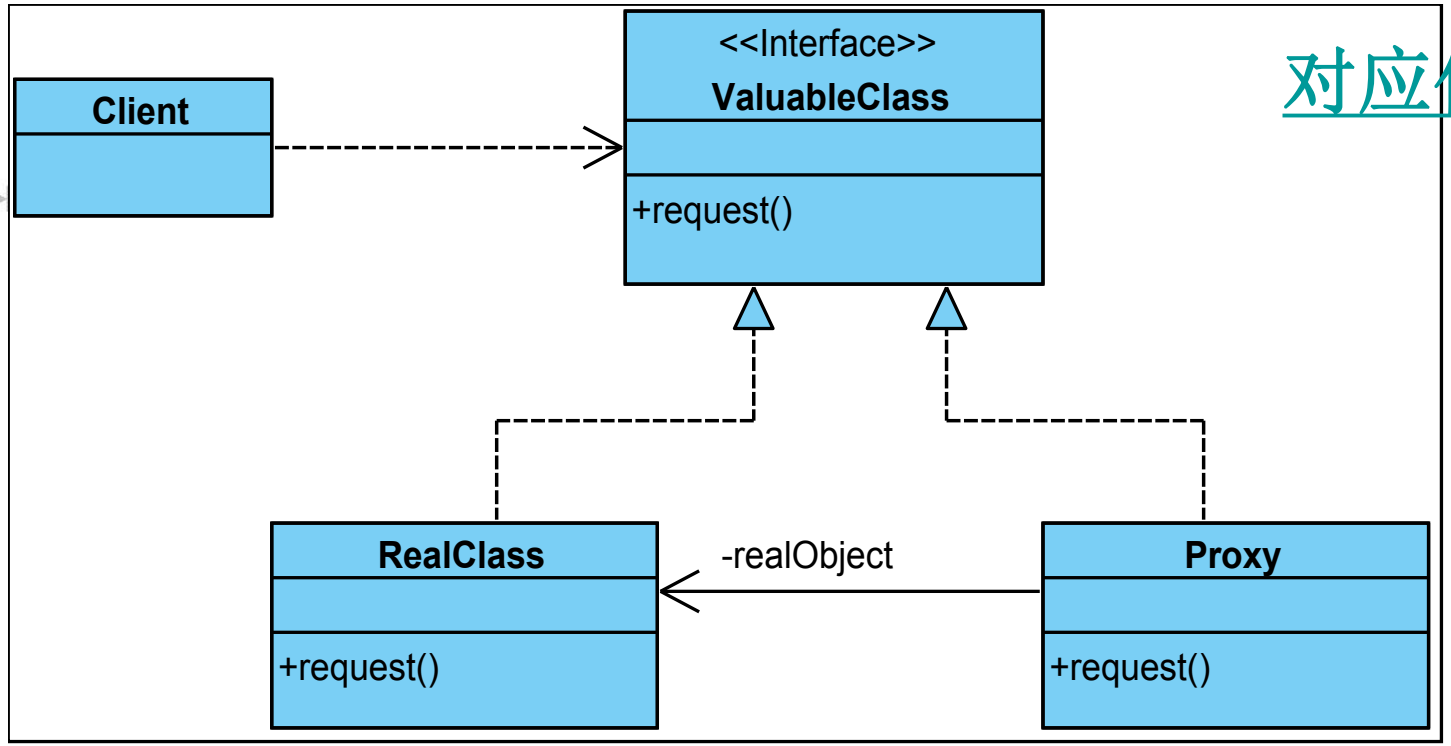


- 更为常见的做法是系统并不提供一个门面类，而是提供一个或多个门面接口。
- 这对于系统内部的开发者非常实用，可以较为自由的实现内部功能，只要保证行为具有门面接口中约定的行为即可，在大型的软件开发中可以事先将各个子系统的外部行为确定下来，即门面接口，然后逐渐完善内部设计和开发，称为“基于契约的设计（Design by Contract）”。

- 同时也注意到，门面类中集成了子系统中不同的内部功能于一身，这是否违反了单一职责的设计原则了呢？
- 门面类虽然具有多种功能，但它每次为外部提供服务的时候一般只涉及其中一类功能，几乎不会做各种功能的联合使用，也就是这些功能多独立变化，不会形成组合在一起形成的多个变化点，因此本质上并不违反单一职责原则的精神。
- 由此也可见，面向对象的设计不能生搬硬套，应视具体情况做具体分析。

代理模式

- 代理模式（Proxy）一般用来对有价值（稀缺）资源的管理，比如数据库的连接等，目的就是为了提高这些资源的利用率或者系统性能。
- 它给这些资源对象提供一个代理对象，并由代理对象控制对资源对象的使用，起到中介的作用。
- 代理对象的存在使得客户类分辨不出代理对象与真实的资源对象。

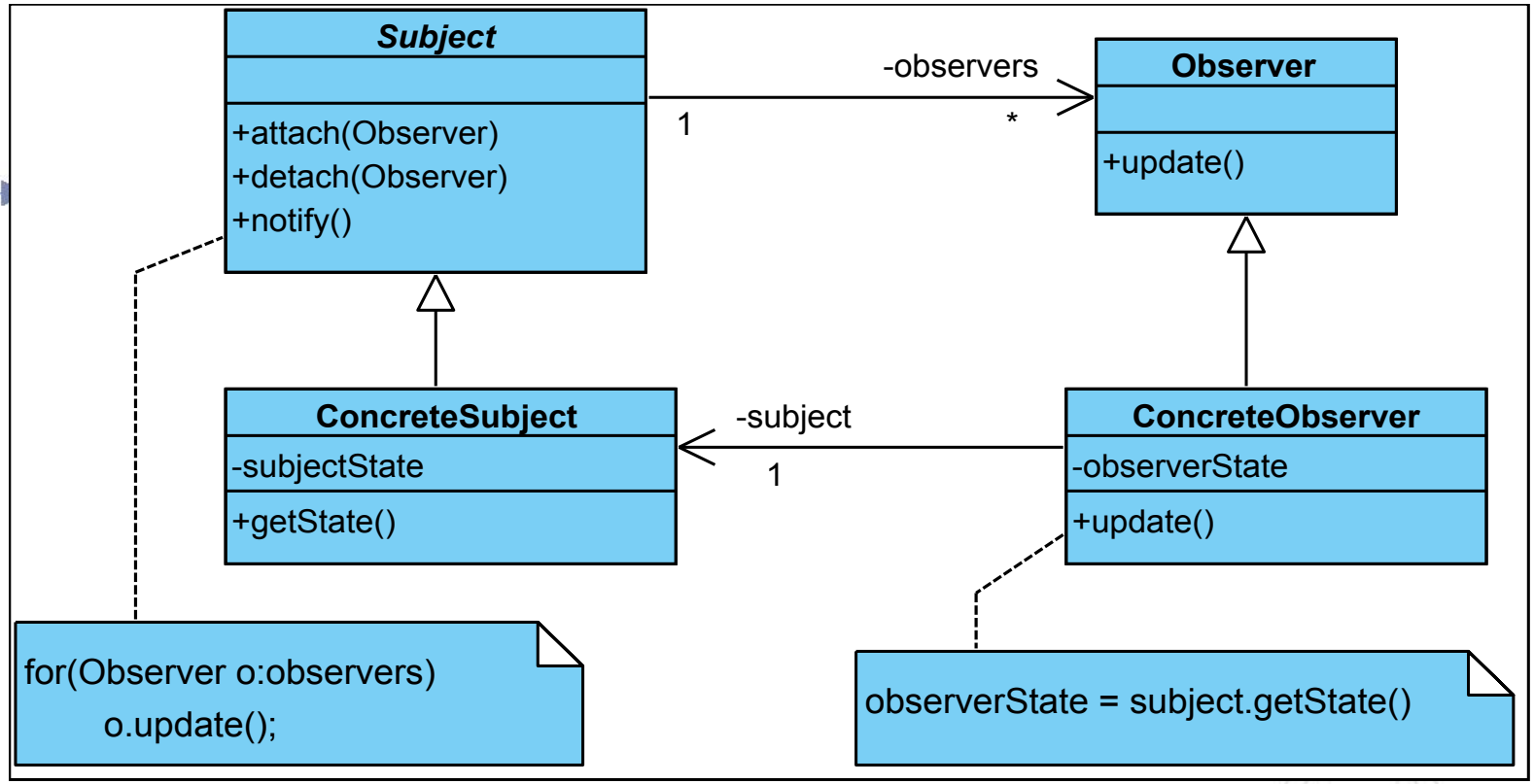


对应代码

- 代理模式也可以并不知道真正的被代理对象，而仅仅持有一个被代理对象的接口，这时候代理对象不能够创建被代理对象，被代理对象必须有系统的其他角色代为创建并传入。

观察者模式

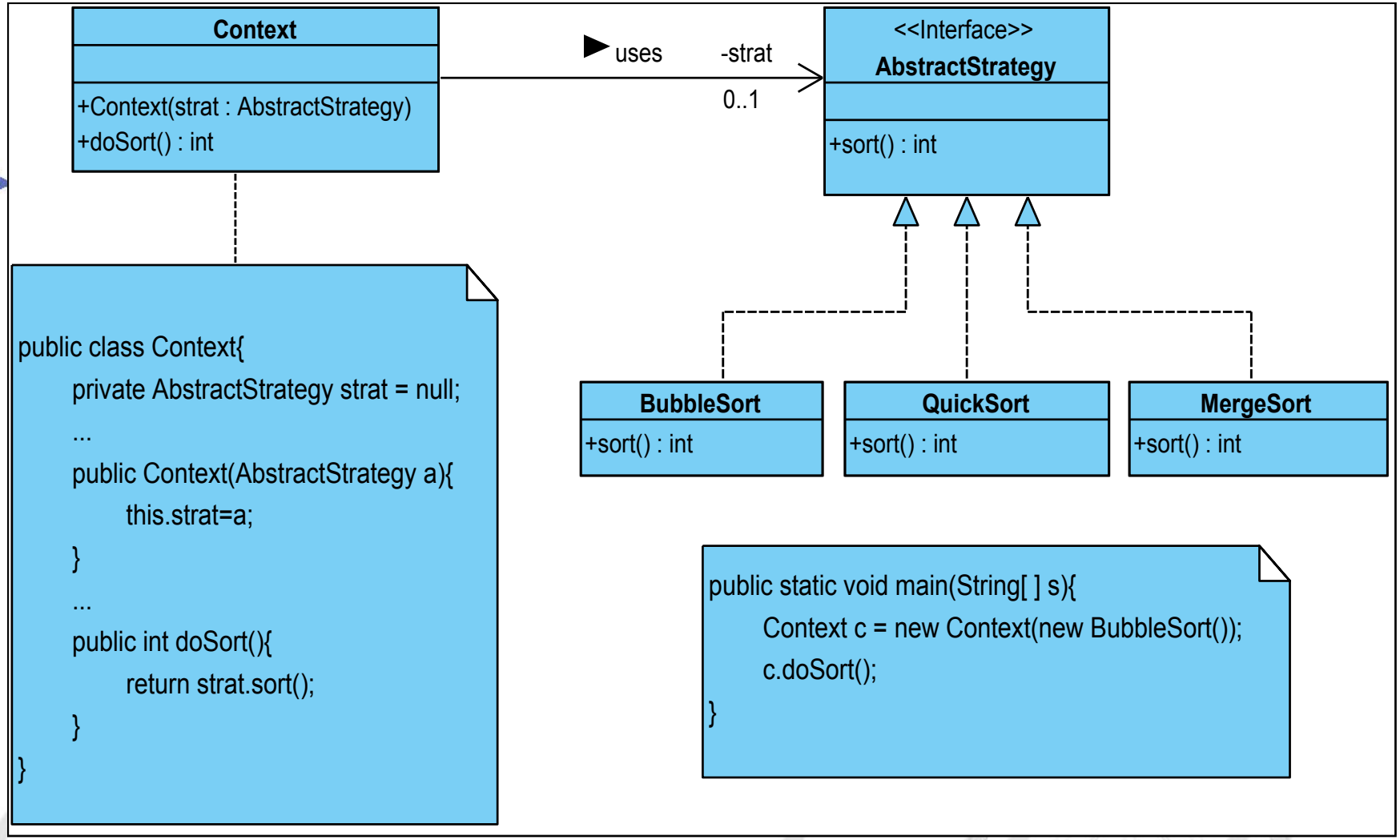
- 观察者模式（Observer）定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象；
- 当这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。
- MVC架构模式在实现上就使用了观察者模式，其中的主题对象就相当于MVC中的模型，观察者对象相当于MVC中的视图。



- 每个观察者对象为了得到主题对象的及时通知，需要事先在主题对象中进行订阅，并且在不需要的时候进行退订
- 每个具体的观察者需要实现自己的更新方法**update()**

策略模式

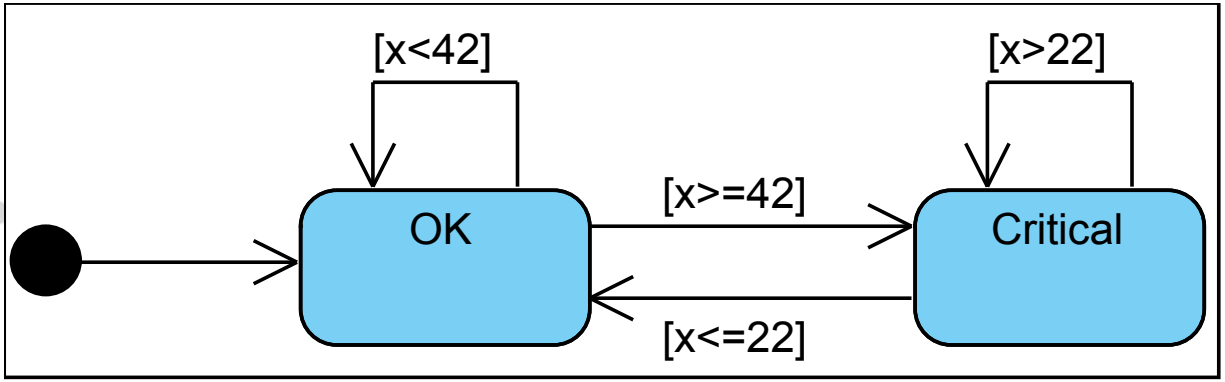
- 工程实践中经常会求解针对复杂问题的解决方案，这些方案多用于某种算法进行描述，而且同一问题在不同的情况下可能会采用不同的算法。
- 策略模式（Strategy）针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。
- 策略模式的好处是能够使得算法可以在不影响到客户端的情况下发生变化，而且将算法的行为和环境分开，环境类负责维持和查询行为类，各种算法在具体的策略类中提供。

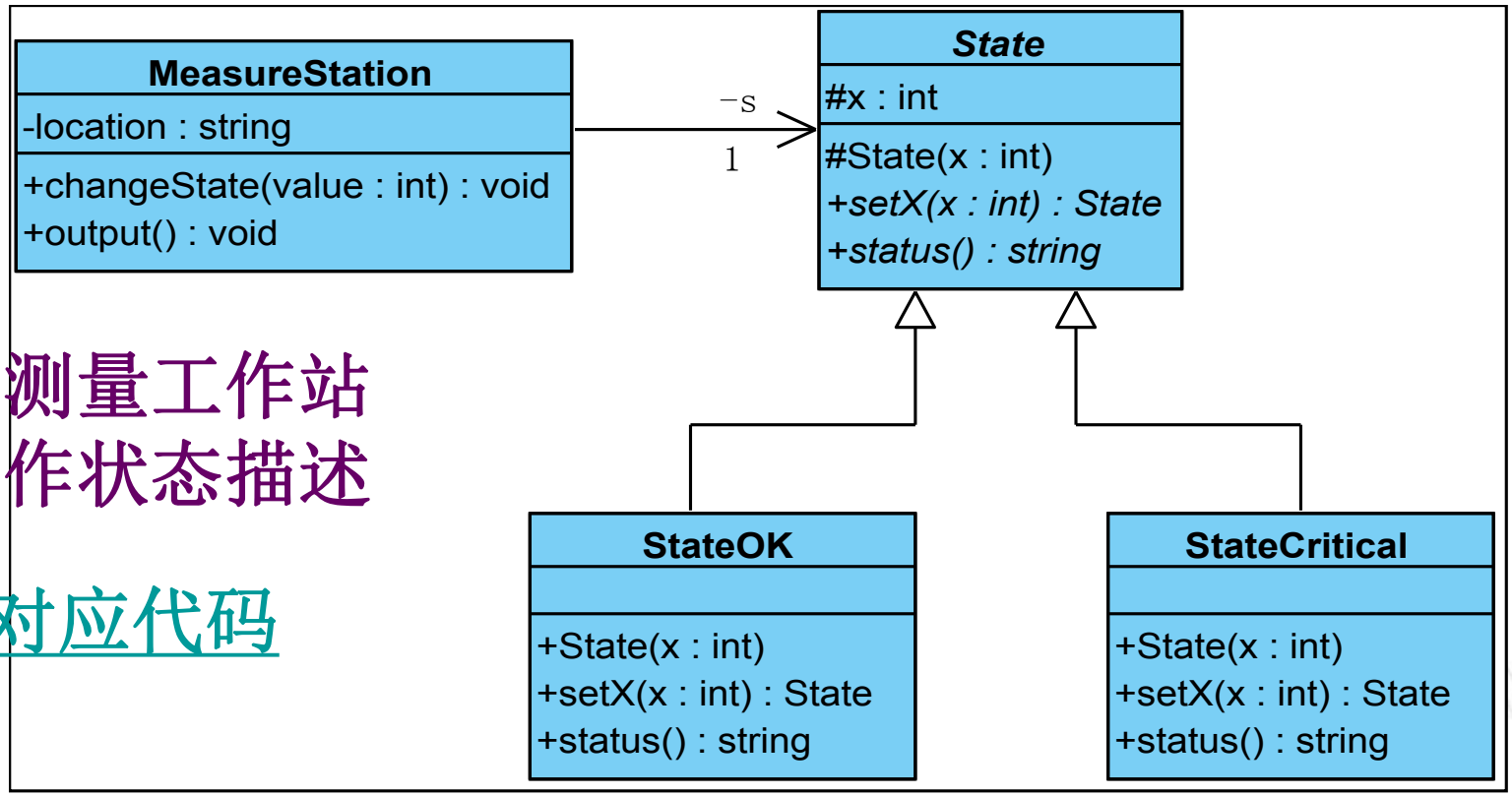


由于算法和环境独立开来，算法的增减、修改都不会影响到环境和客户端。

状态模式

- 状态模式（State）可以看作是策略模式的一种应用，状态模式允许一个对象在其内部状态改变的时候改变行为。
- 状态模式把所研究的对象的行为包装在不同的状态对象里，每一个状态对象都属于一个抽象状态类的一个子类。
- 状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。
- 状态模式需要对每一个系统可能取得的状态创建一个状态类的子类。当系统的状态变化时，系统便改变所选的子类，从而对类在不同状态下的行为进行管理。





一个测量工作站的工作状态描述

对应代码

作业

- 习题1、3（选做）
- 习题2