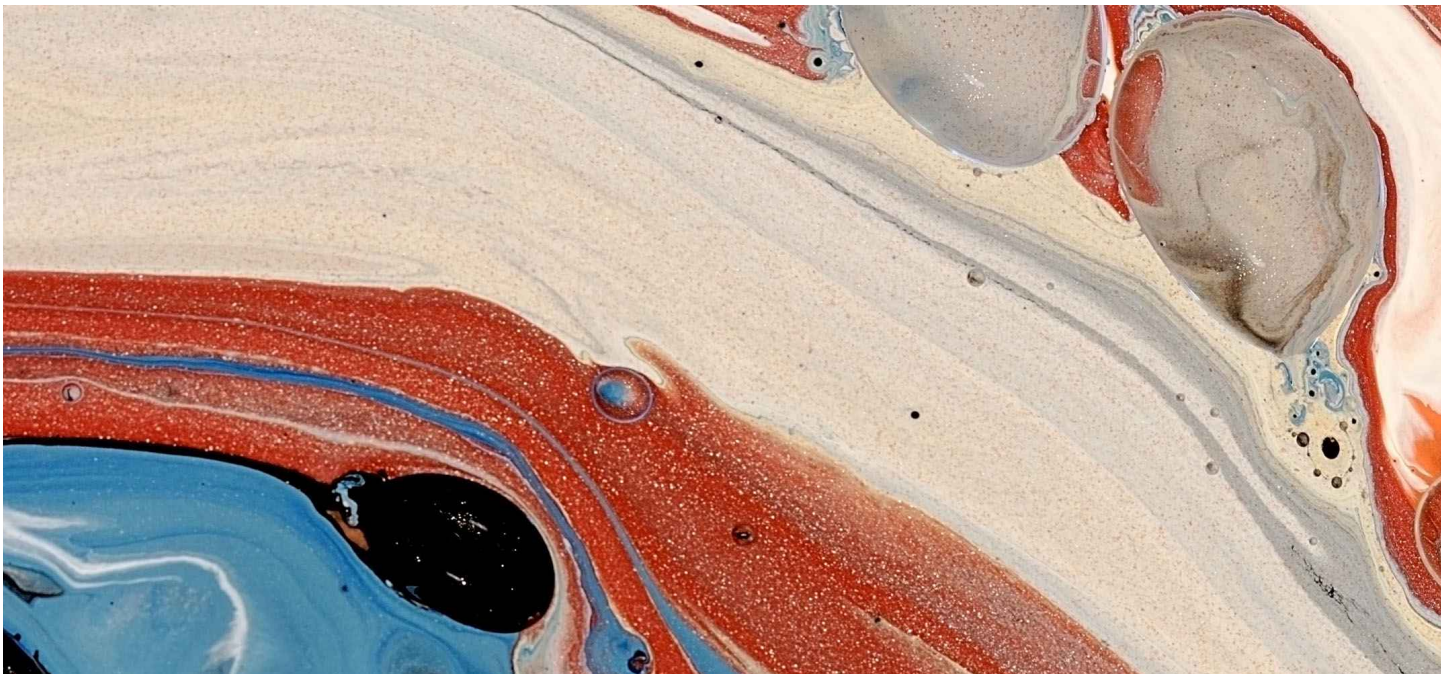


软件设计模式项目设计文档与其他说明

本项目已经发布release版本在Github，如需要，可使用以下的访问链接进行访问：
https://github.com/GyxPillar/TongjiUniversity_2022FallSemester_SoftwareDesignPatterns

项目背景

本系统运用软件设计模式的知识，实现了整个电脑的生产与销售过程的模拟。包括原材料的挑选与采购、供货商的挑选、零件的生产、质量的检验、电脑的装配、物流的运输、网点的销售等。该系统通过命令行实现一个各个流程的运行模拟情况展示，用户可通过命令行输入与其进行简单的交互，例如选择执行的功能以及在各个功能点内选择不同的分支等。

项目目标

1. 设计若干个流程模拟场景，使用所学过的**全部23种GoF设计模式**以及空对象模式模拟各个流程的过程信息。
2. 流程场景设计要合理，模式选择要适当。
3. 对每个系统模式结构图进行展示，并进行详细的描述。
4. 实现一定的交互性，在程序中用户可自由选择想要模拟实现的功能等。

项目结构

TongjiUniversity_2022FallSemester_SoftwareDesignPatterns

```
├── 1-Purchase-gjc
│   ├── 1.cpp
│   └── purchase.h
├── 2-FactoryStructure-gyx
│   ├── 2.cpp
│   └── FactoryStructure.h
├── 3-ProductionProcess-hjh
│   ├── 3.cpp
│   └── production.h
├── 4-Assemble-tarq
│   ├── 4.cpp
│   ├── builder.h
│   ├── composite.h
│   ├── memento.h
│   └── template.h
├── 5-Examine-lsr
│   ├── 5.cpp
│   ├── adapter.h
│   ├── bridge.h
│   ├── state.h
│   └── visitor.h
├── 6-Transport-ysh
│   └── 6.cpp
├── 7-Sales-xjx
│   ├── 7.cpp
│   ├── xjx.cpp
│   └── xjx.h
├── main.cpp
├── main.exe
├── makefile
├── module
```

```
│   ├── 1.exe
│   ├── 2.exe
│   ├── 3.exe
│   ├── 4.exe
│   ├── 5.exe
│   ├── 6.exe
│   └── 7.exe
├── README.md
├── sdp.zip
└── _diagrams
    ├── assemble
    │   ├── builder.drawio
    │   ├── builder.png
    │   ├── composite.drawio
    │   ├── composite.png
    │   ├── memento .drawio
    │   ├── memento .png
    │   ├── template.drawio
    │   └── template.png
    ├── examine.mdj
    ├── factory.mdj
    ├── production.mdj
    ├── purchase.mdj
    ├── sales.mdj
    └── transport.mdj
```

小组成员分工与贡献

学号	姓名	分工	贡献占比
2053385	高逸轩	项目设计与架构；工厂结构部分代码编写；项目设计文档汇总；PPT制作	12.5%

2053300	胡锦辉	项目设计与架构；生产过程部分代码编写	12.5%
2054205	夏佳幸	项目设计与架构；销售和售后部分代码编写	12.5%
2053280	杨思恒	项目设计与架构；物流部分代码编写	12.5%
2050956	田阿润奇	项目设计与架构；装配部分代码编写	12.5%
2052313	周长赫	项目设计与架构；整体代码的合并与打包；作业内容的审核	12.5%
2053386	高骏骋	项目设计与架构；采购部分代码编写	12.5%
2051499	李上如	项目设计与架构；检验部分代码编写；	12.5%

分功能点

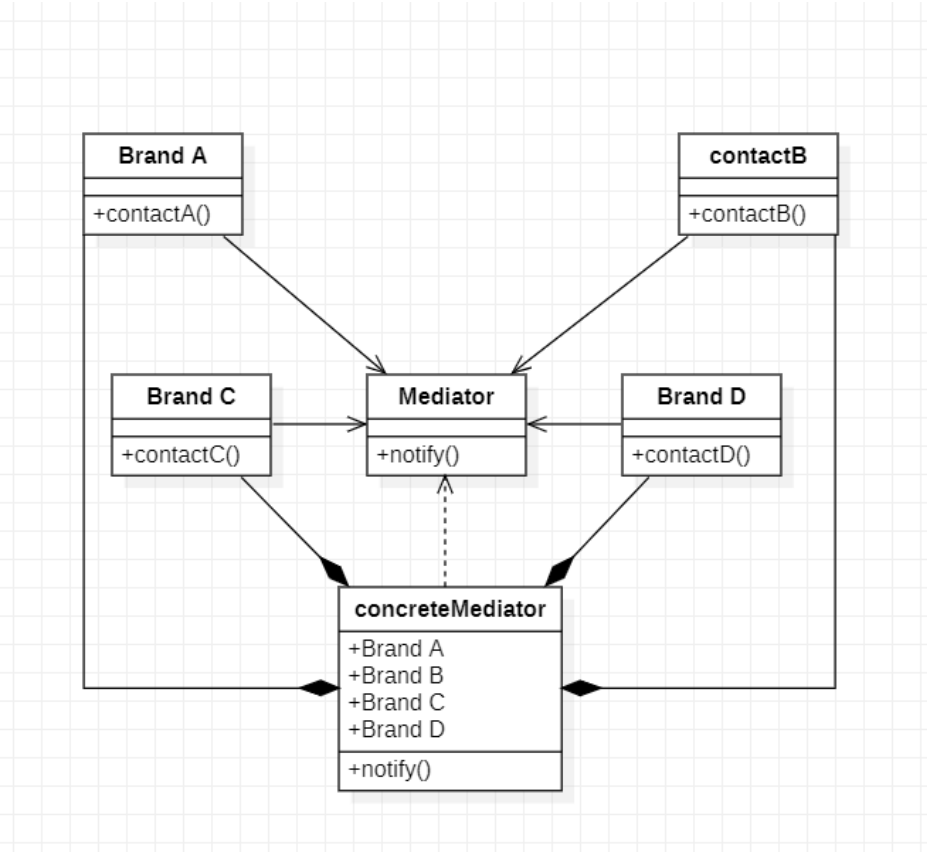
1. 选择供货商与采购

1.1 中介者模式

1.1.1 实现功能

公司联系购买两个购买中介，通过他们联系不同的供货商进行供货。

1.1.2 类图



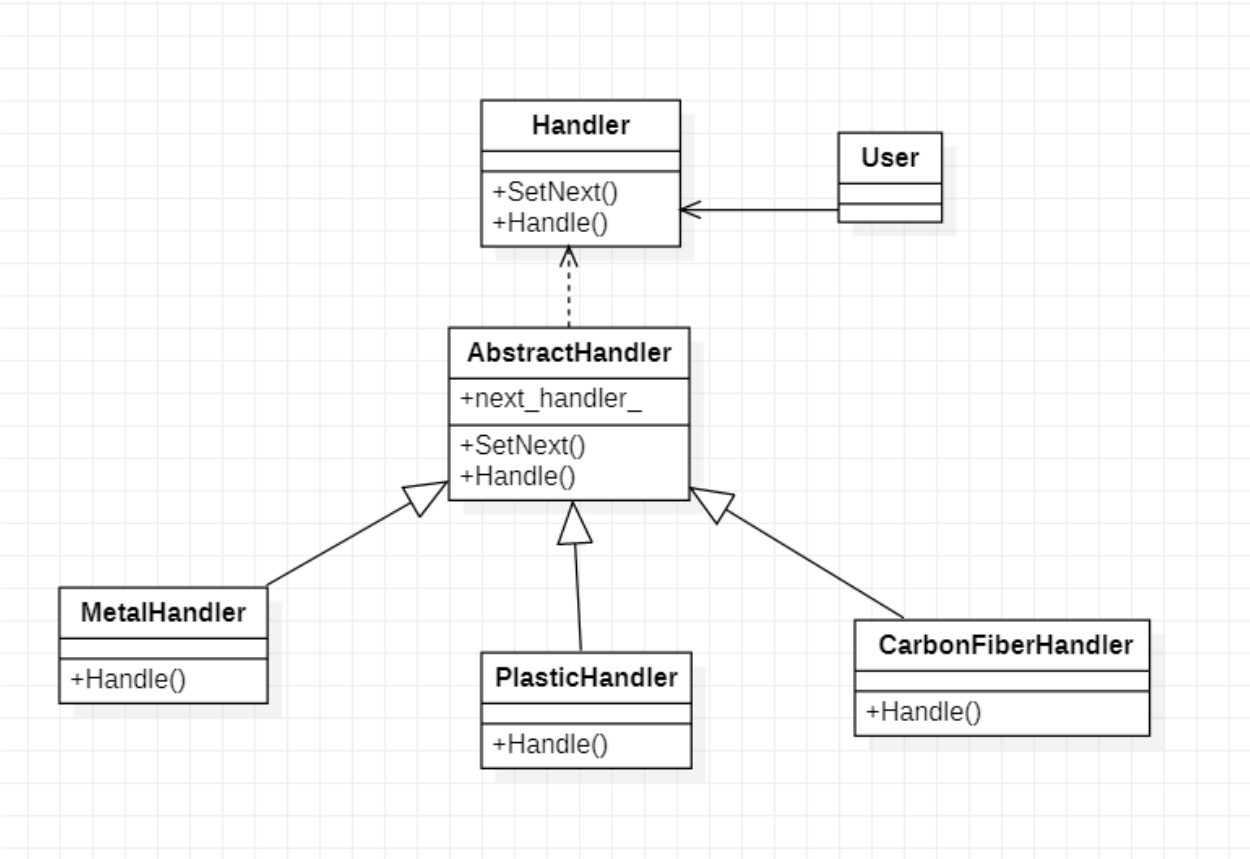
concreteMediator封装了多种供货商之间的关系，进行统一管理。输入选择时，Mediator将会通过notify()进行选择不同的brand执行操作。各个brand组件并不知道其他组件的情况。 如果brand内发生了重要事件， 它只能通知Mediator。

1.2 责任链模式

1.2.1 实现功能

定义三种不同的原材料和负责该种原材料的供应商，通过责任链模式选择不同商家购买其对应的原料。

1.2.2 类图



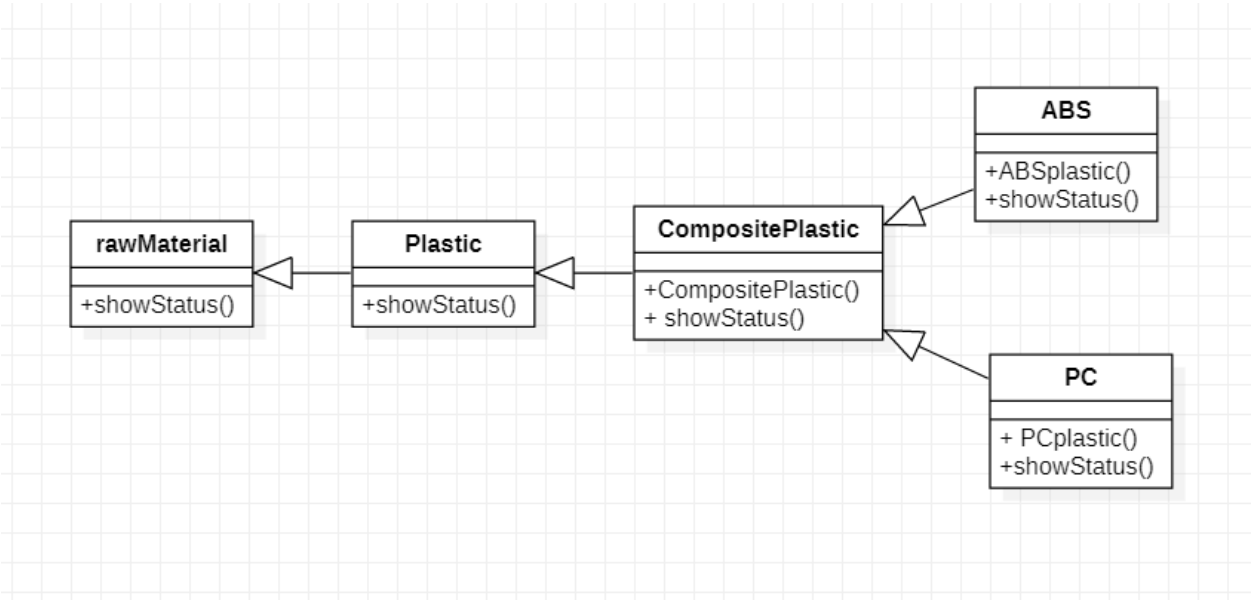
Handler 声明了所有具体处理者的通用接口。 该接口通常仅包含单个方法用于请求处理， 但有时其还会包含一个设置链上下个Handler 的方法。AbstractHandler中定义了一个保存对于下个Handler 引用的成员变量,可通过将处理者传递给上个Handler 的构造函数或设定方法来创建链，随后根据该责任链进行相应供货商的供给。

1.3 装饰模式

1.3.1 实现功能

在不改变现有对象结构的情况下，为对象增加额外功能。应用于购买塑料原材料的过程中，为购买的复合塑料增加ABS和PC两种。

1.3.2 类图



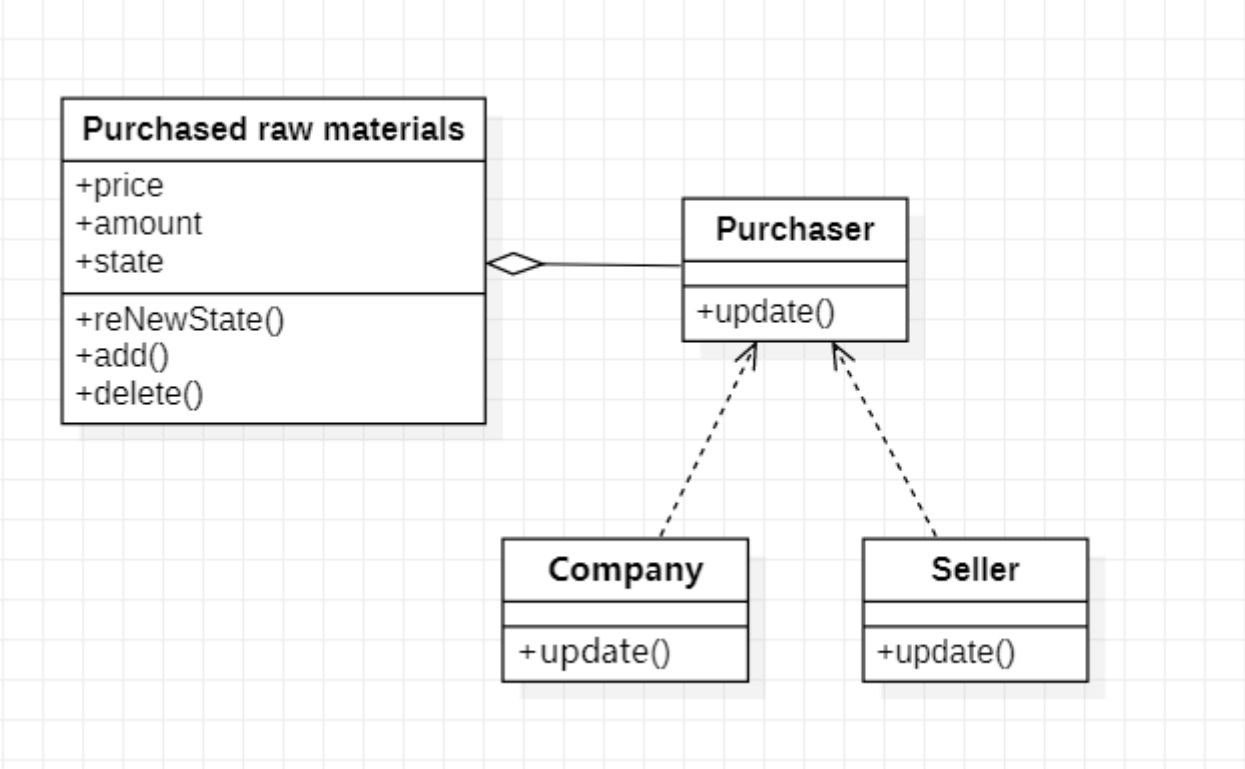
抽象构件rawMaterial给出一个抽象接口，以规范准备接受附加责任的对象。具体构件PlasticCompositePlastic实现抽象构件，通过装饰角色为其添加新的功能，随后ABS和PC是具体装饰，实现抽象装饰的相关方法，并给具体构件对象添加附加的责任（购买相应的塑料材料）。

1.4 观察者模式

1.4.1 实现功能

当采购动作完成时，Purchaser会向采购的电脑公司和供货方更新信息。

1.4.2 类图



主题PurchasedRawMaterials提供了一个用于增加、删除观察者对象的方法，以及通知所有观察者的抽象方法；观察者接口Purchaser包含了一个更新自己的抽象方法，当接到主题的更改通知时被调用，观察者Company和Seller实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。

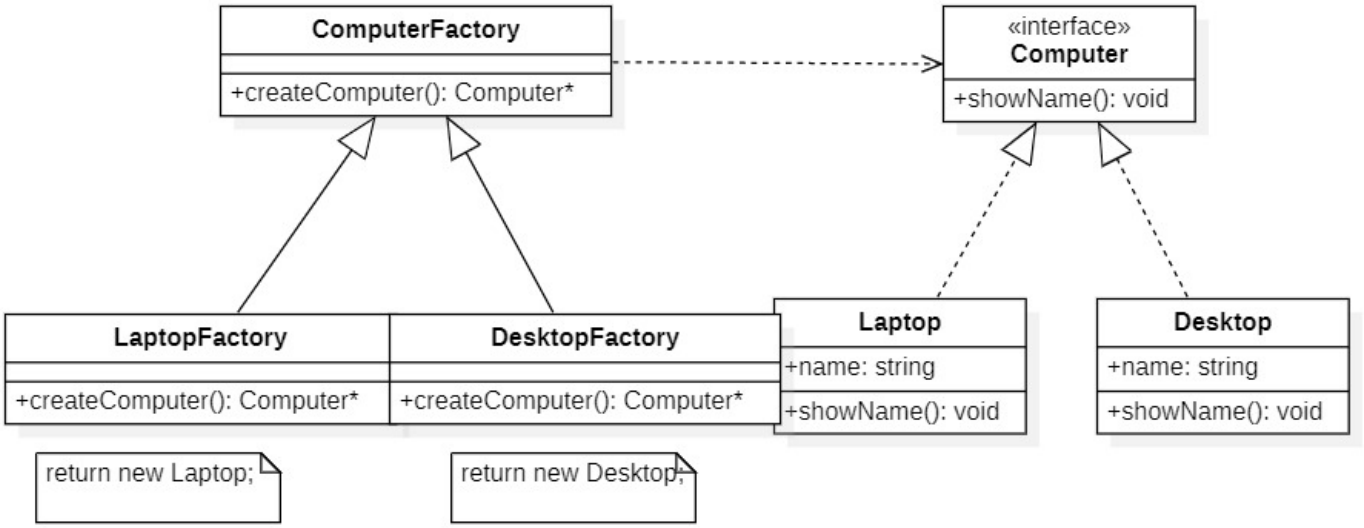
2. 生产与零件质检

2.1 工厂模式

2.1.1 实现功能

通过用户选择生产的电脑类型为笔记本电脑/台式电脑，模拟实现不同的工厂生产不同类型电脑的过程，并将生产过程与结果的信息进行展示。

2.1.2 类图



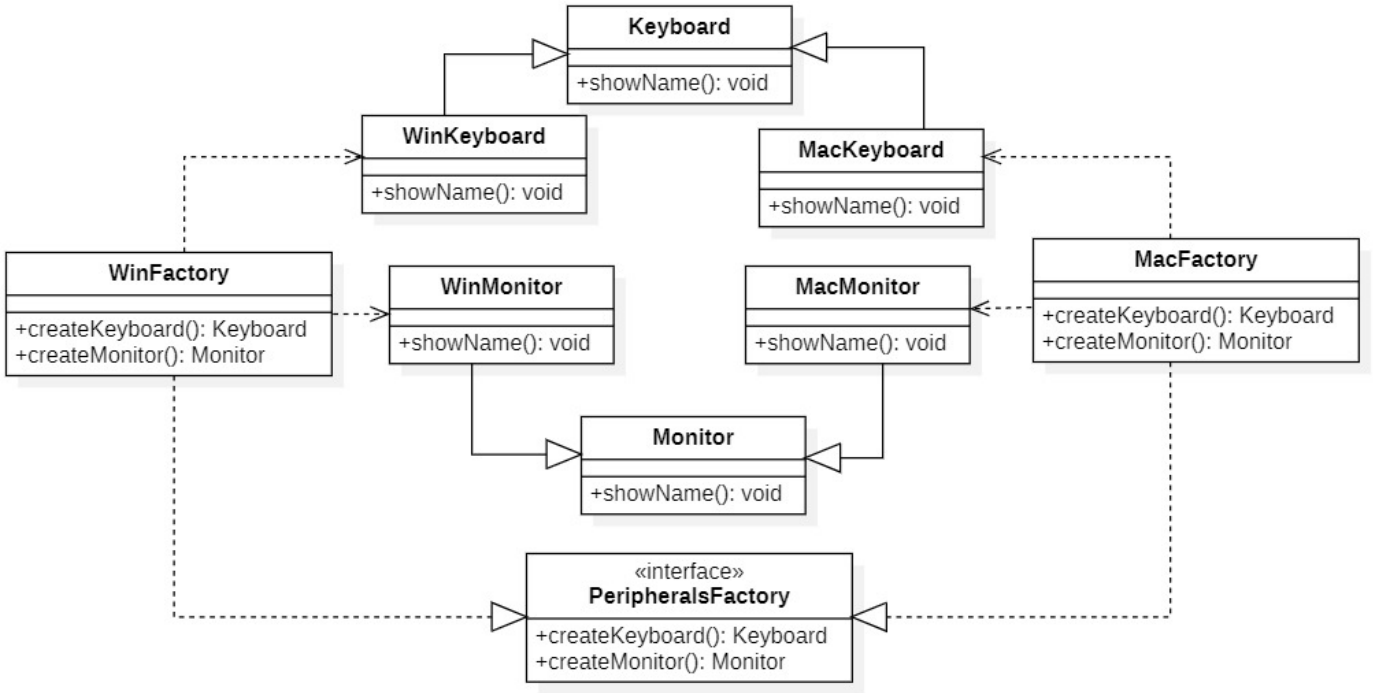
在本图中，Computer接口代表产品（即各种类型电脑），其可以具体实现为Laptop（笔记本电脑）、Desktop（台式电脑）两种类型，在某一台任意类型的电脑被生产出来后，会调用其中的方法showName()来进行生产结果信息的展示，展示此时被生产的电脑类型名称。ComputerFactory为抽象的电脑工厂，依赖于Computer，其可以继承实现为LaptopFactory和DesktopFactory两种类型，调用其中重写的方法createFactory()，不同类型的工厂可以对应生产不同类型的产品，并展示对应的工厂生产信息提示。

2.2 抽象工厂模式

2.2.1 实现功能

用户通过自己选择需要生产的外设零件工艺（Win/Mac）与种类（键盘/显示器），模拟不同工艺的Win和Mac工厂生产不同工艺与不同种类的外设零件的过程，并将生产过程与结果的信息进行展示。

2.2.2 类图



在本图中，Keyboard和Monitor为两类抽象产品，为构成键盘与显示器两个系列产品的一组不同但相关的产品声明接口showName()，用于在不同的外设零件被生产后进行生产结果信息的展示。而WinKeyboard、MacKeyboard、WinMonitor、MacMonitor四种具体产品则是抽象产品的多种不同类型实现。 所有变体 （Win/Mac） 都必须实现相应的抽象产品 （键盘/椅子）。

PeripheralsFactory（外设工厂）为抽象工厂接口，其声明了一组创建抽象产品键盘、显示器的方法，分别为createKeyboard()和createMonitor()，其可以在工厂进行外设零件的生产时展示接下来的生产信息。而WinFactory和WinFactory则为实现抽象工厂的构建方法。每个具体工厂都对应特定产品变体，且仅创建此种类型的产品变体，并且依赖于对应的产品变体。

2.3 原型模式

2.3.1 实现功能

我们首先使用 **原型模式** ，抽象出其原型 `Core` ，该原型拥有以下两个成员变量：

- `coreType`：核心类型
- `coreFrequency` ：核心频率

同时为了方便生产核心，该原型拥有以下两个接口

- `Clone()` ：克隆一个核心
- `Product()` ：描述核心的生产过程，包括定频

之后有 `CentralProcessUnit` 以及 `GraphicProcessUnit` 两个类继承自 `Core` ，分别代表 `CPU` 以及 `GPU` 。这两个类分别重写了原型中的 `Clone()` 方法：

```
1 // 以CPU为例，创建一个CentralProcessUnit对象， 然后将原型Core对象中所有的成员变量值复制
```

```
2 Core *Clone() const override {
3   return new CentralProcessUnit(*this);
4 }
```

使其分别返回 `CentralProcessUnit` 以及 `GraphicProcessUnit` 的实例，这样在生产过程中就能通过克隆的方式正确生产 `CPU` 以及 `GPU` 。

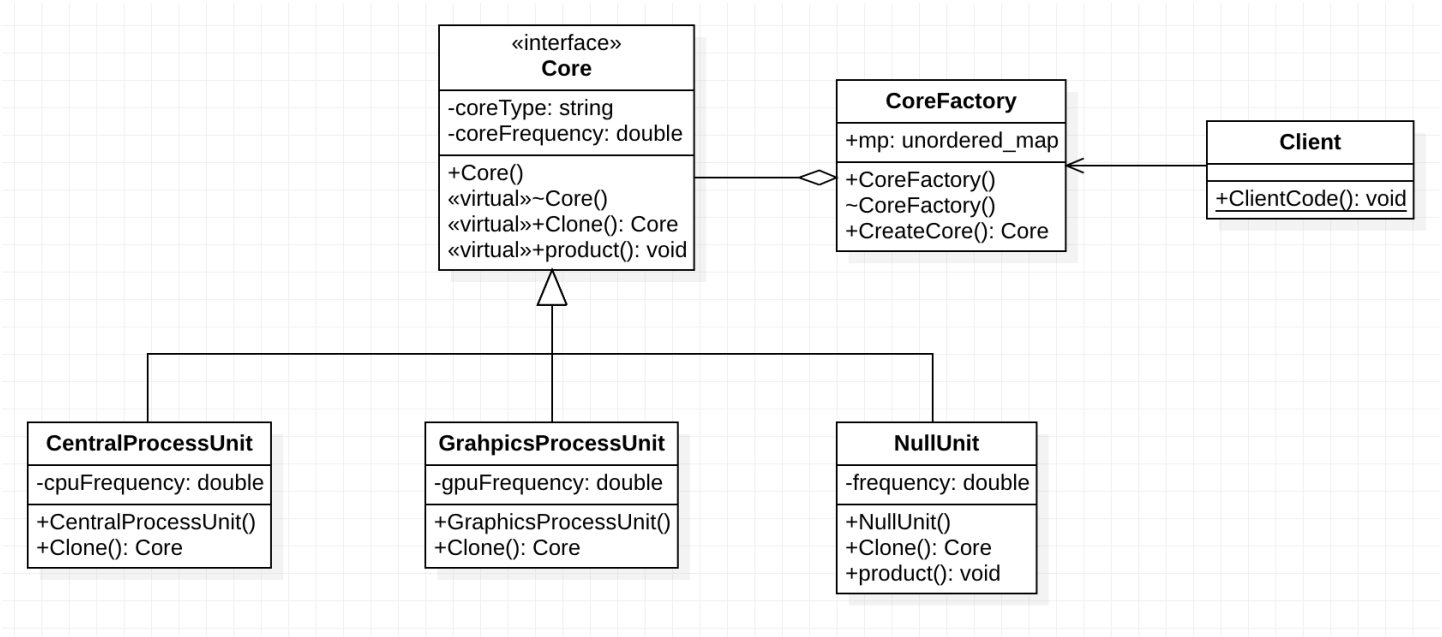
生产过程由核心工厂 `CoreFactory` 控制，该工厂拥有一个成员变量以及一个用于生产核心的方法：

- `mp` ：原型注册表
- `CreateCore()` ：根据核心类型生产核心

原型注册表 是一个 `名称 -> 原型` 的哈希表， 其中存储了一系列可供随时复制的预生成对象，提供了一种访问常用原型的简单方法。

在每次生产时， `CoreFactory` 都会根据生产核心的类型访问原型注册表，并返回一个相应类型的克隆。

2.3.2 类图



2.4 空对象模式

2.4.1 实现功能

由于生产过程中，客户端向核心工厂 `CoreFactory` 发送生产请求中的核心类型可能并不合法，所以可以引入一个同样继承自原型 `Core` 的空对象 `NullObject` 类。由于这种不合法的核心并不能通过一般过程生产，所以 `NullObject` 需要重写 `Core` 中用于描述生产过程的方法 `Product()` ，用于向客户端提示错误信息。

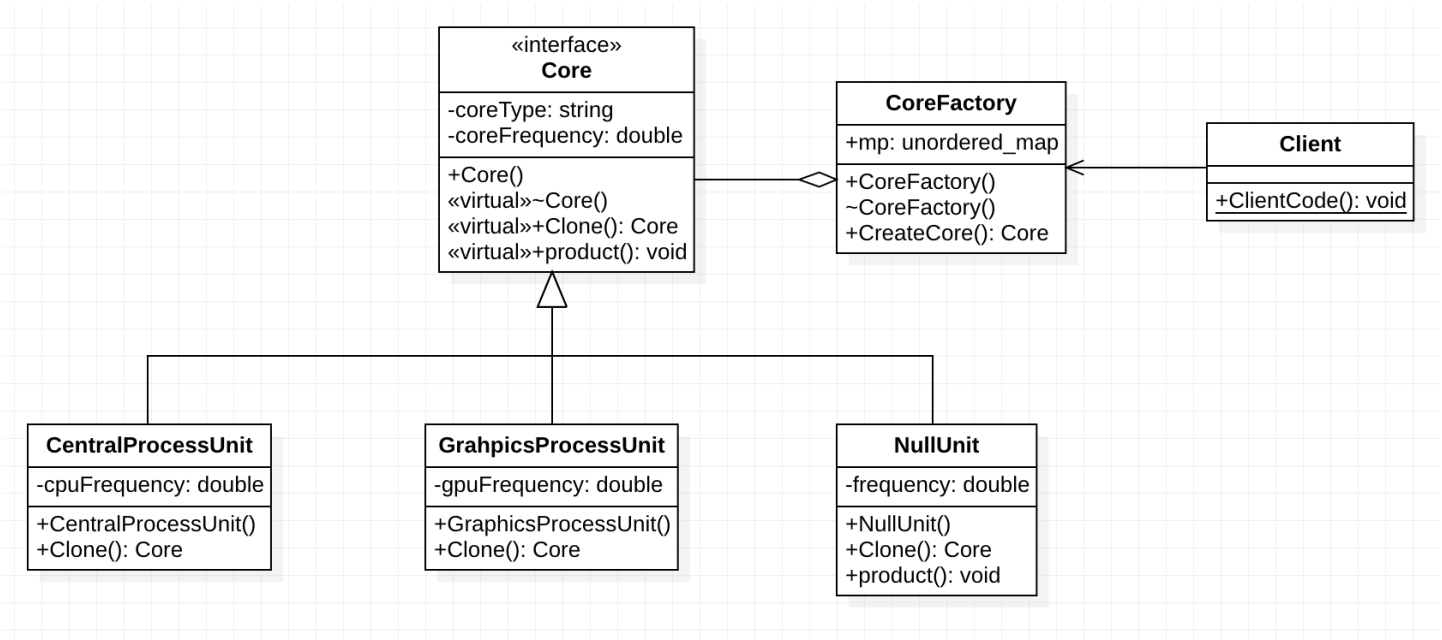
```
1 void Product(std::string coreType) override {
2   std::cout << coreType << " is not a legal core type!" << '\n';
3 }
```

执行逻辑通过客户端 `Client` 类控制， 该类只有一个方法 `ClientCode` ，即客户端程序需要执行的代码段，主要为以下两步：

- `CreateCore` 通过核心工厂生产一个核心
- `Product` 描述该核心的生产过程

方便起见该方法为被置为 `static` 供主程序通过类名直接调用。

2.4.2 类图



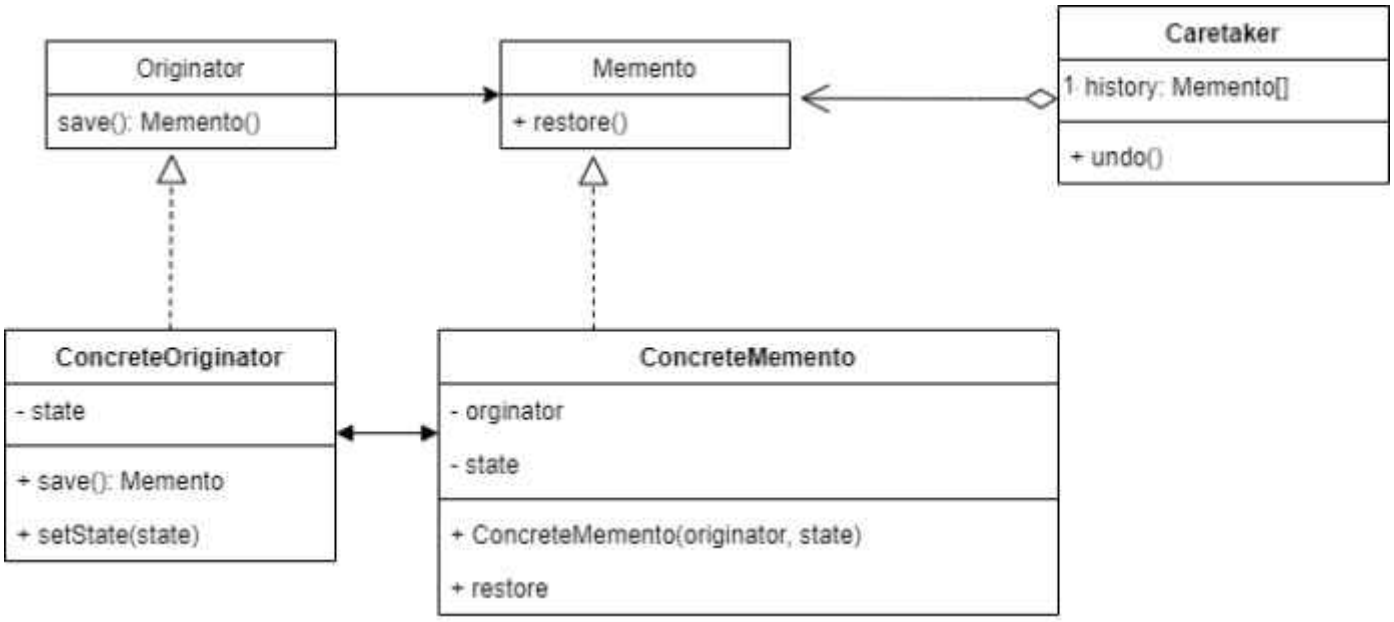
3. 整机装配与质检、包装

3.1 备忘录模式

3.1.1 实现功能

在装配环节，记录装配前的状态和装配后的状态，若装配后有故障，则按照备忘录返回到上一步，直到状态正常。

3.1.2 类图



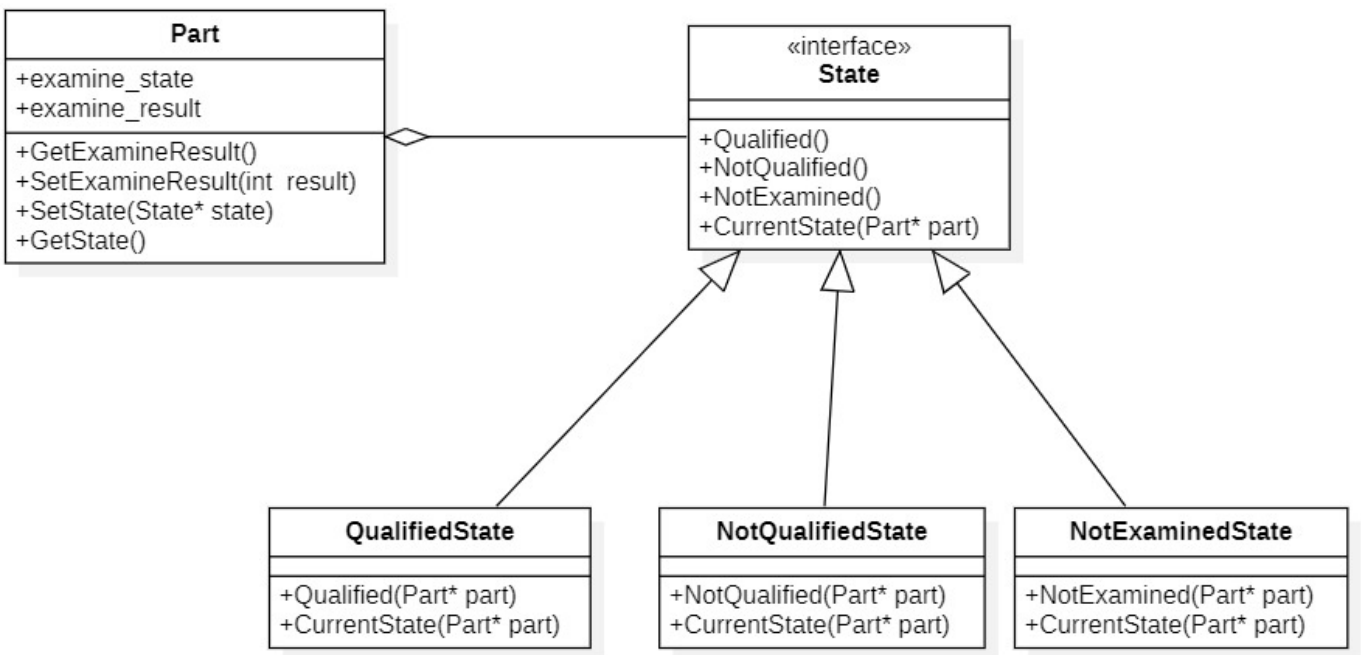
图中，Originator类可以生成自身状态的快照， 也可以在需要时通过快照恢复自身状态。Memento是原发器状态快照的value object。 Caretaker仅知道 “何时” 和 “为何” 捕捉原发器的状态， 以及何时恢复状态。电脑制造者通过保存备忘录栈来记录安装各个配件前的历史状态。 当原发器需要回溯历史状态时， 制造者将从栈中获取最顶部的备忘录， 并将其传递给原发器的restoration 方法。

3.2 状态模式

3.2.1 实现功能

对于零件和整机，需要进行质检。质检前，状态为未检验；质检后，质量合格即为合格状态，质量不合格即为不合格状态。

3.2.2 类图



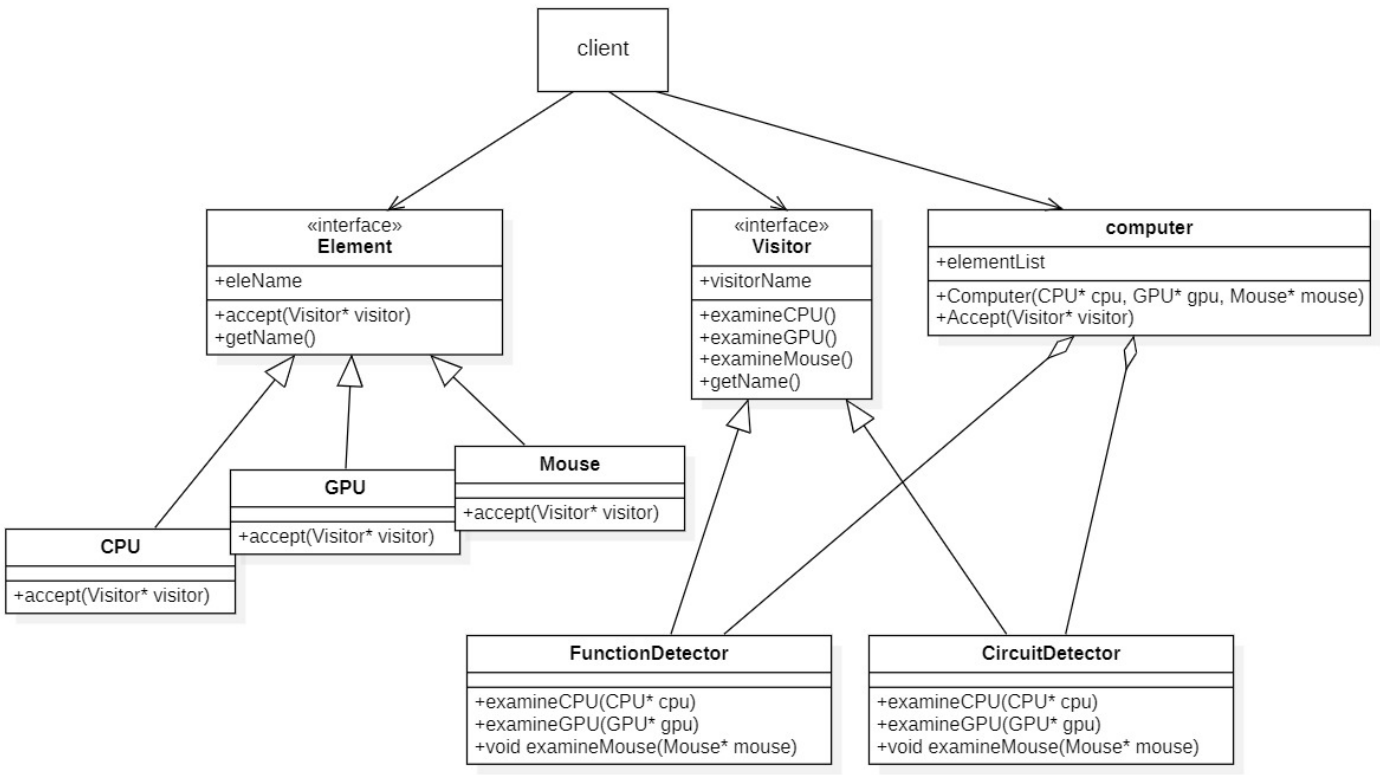
图中，Part类代表需要经历检验的具体对象，State类是Part的状态的基类。QualifiedState、NotQualifiedState、NotExaminedState三个类都是State的基类，分别代表合格、不合格和未检验三个状态。Part类初始化时，examine_state为未检验。根据Part的examine_result的值（它是被用户设定的），可以对Part中的examine_state进行从未检验到合格/不合格的转换。

3.3 访问者模式

3.3.1 实现功能

针对不同配件或整体电脑，需要进行质量评估或功能评估。对于不同的评估，可能会有不同的测试者，以及不同的测试方法。

3.3.2 类图



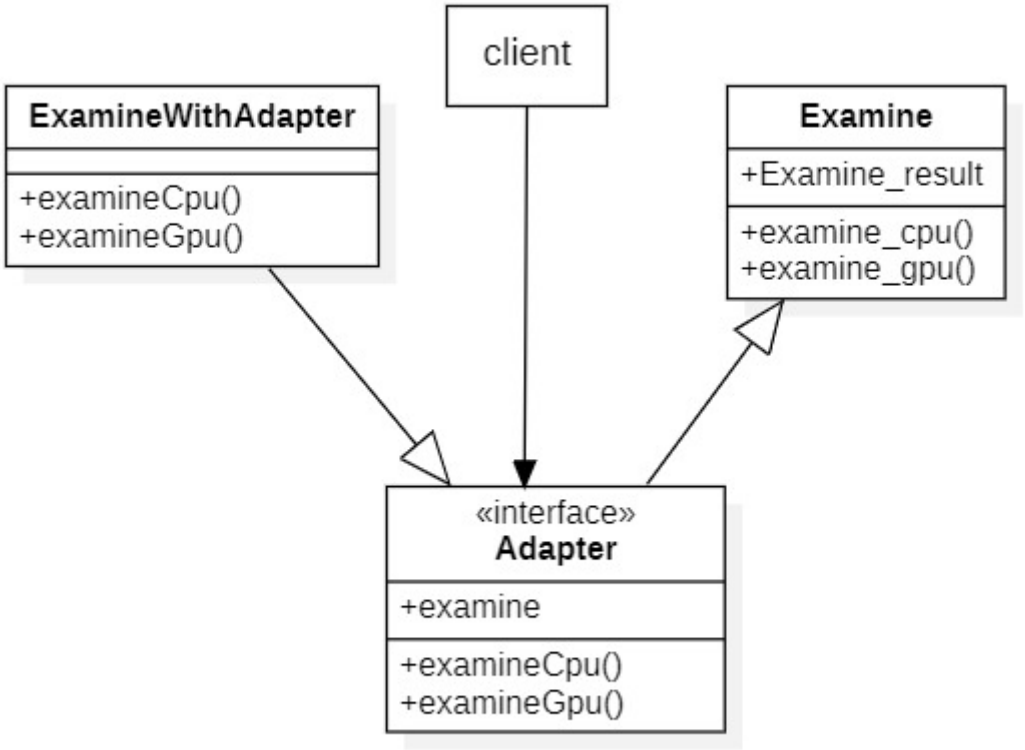
本图中，Element为接口，其具体实现为CPU、GPU和Mouse，即组成电脑的零件。Computer是具体的电脑，由CPU、GPU、Mouse组成。Visitor类是一个接口，其具体实现类是FunctionDetector和CircuitDetector，两种测试者分别测试功能和质量。通过Computer类的Accept函数，实现访问者的检验。

3.4 适配器模式

3.4.1 实现功能

检验过程中，对不同型号电脑的零件提供检验访问接口，使得能够使用已有的方法。

3.4.2 类图



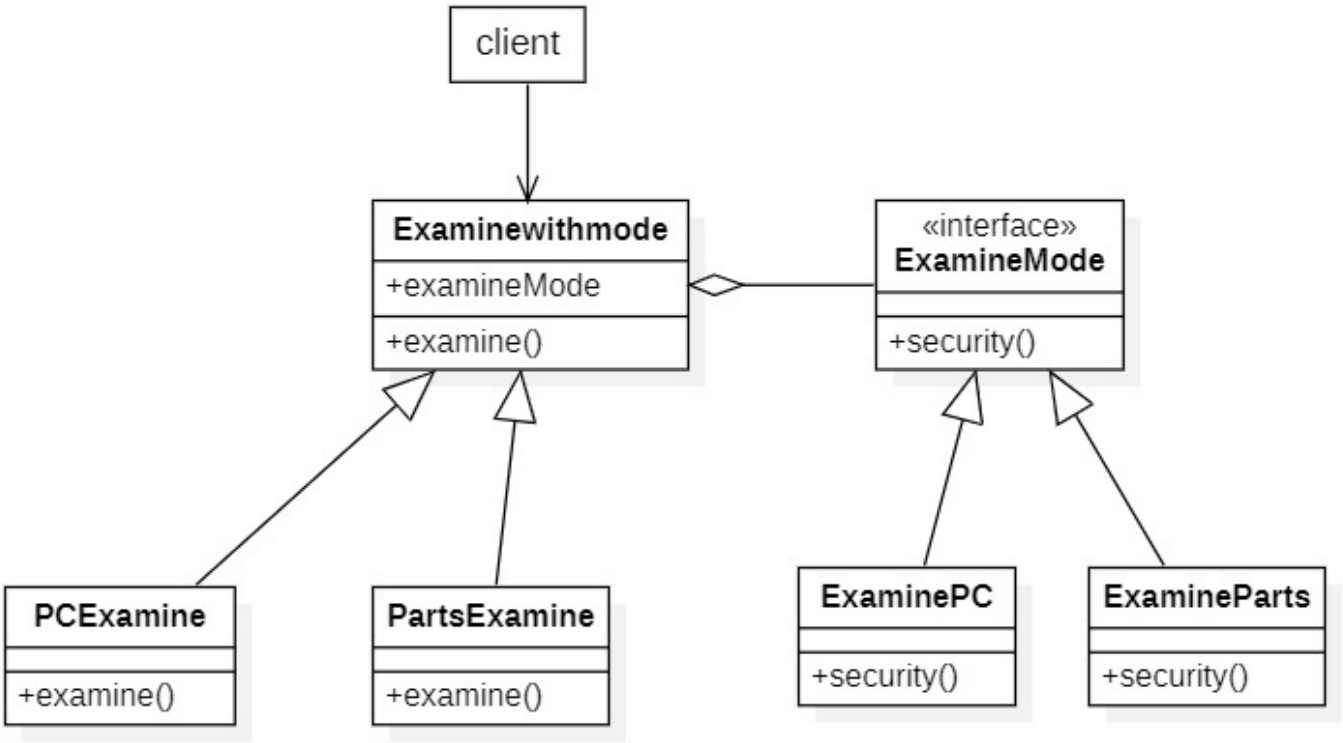
本图中，Examine类即为已有的检验类，其中的方法即为实际有的方法。Adapter类为接口类，它单继承一个目标类，使用组合的方式实现。ExamineWithAdapter类就是实际的类，不再是接口。这样，ExamineWithAdapter就可以使用Examine中已有的方法，对不同型号的电脑进行检验。

3.5 桥接模式

3.5.1 实现功能

电脑检验，定义一个检验过程的接口类，子类提供实现

3.5.2 类图



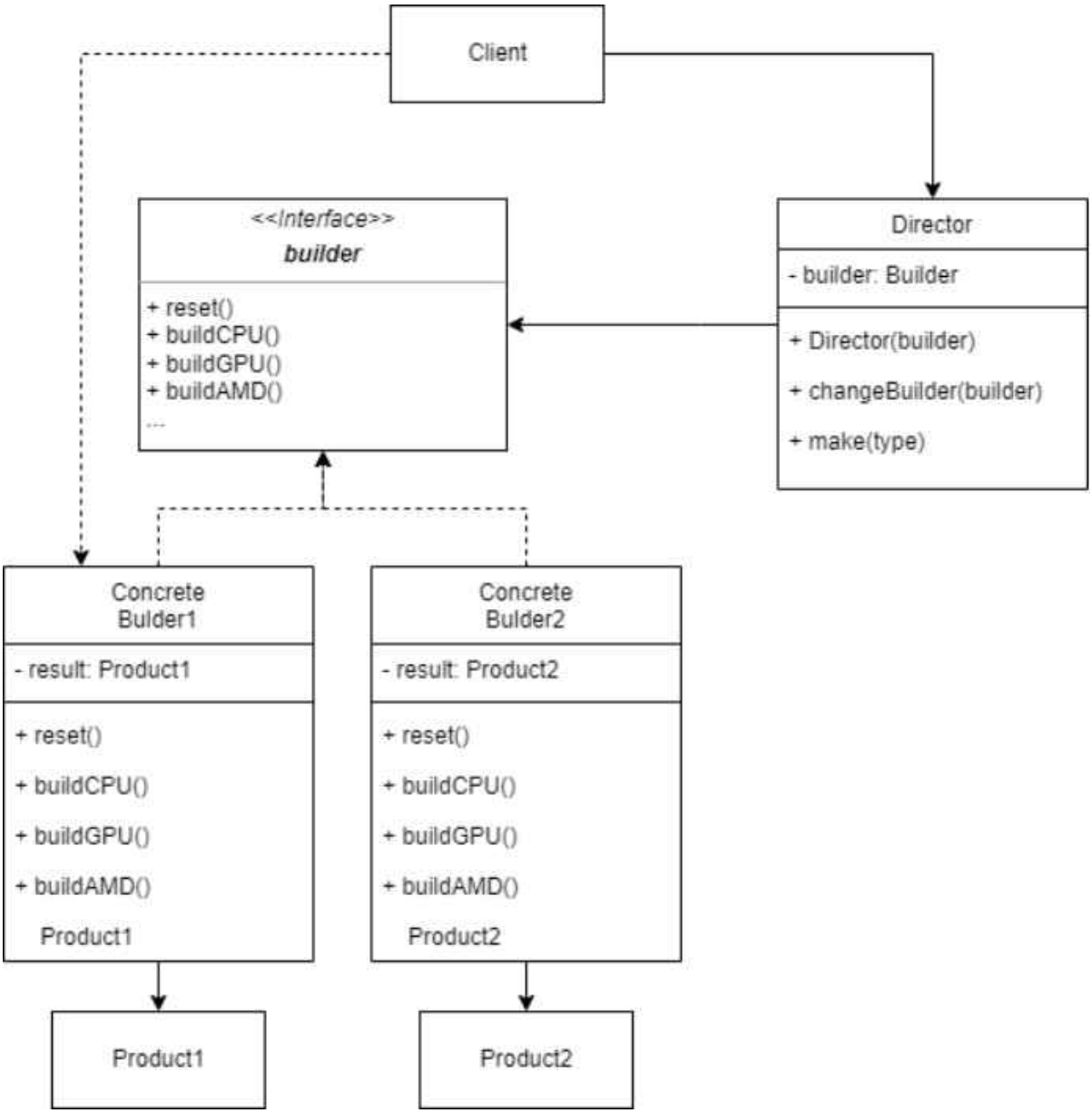
本图中，ExamineMode为实现类接口，其具体实现类有检验零件的ExamineParts和检验电脑的ExaminePC。Examinewithmode为抽象化类，主要完成检验工作，其中，扩展抽象化角色PartsExamine可以调用ExaminePC中的函数以检验零件，扩展抽象化角色PCExamine可以调用ExamineParts中的函数以检验电脑。

3.6 建造者模式

3.6.1 实现功能

将计算机的装配过程划分为一组步骤， 比如 install-CPU和 install-GPU等每次创建对象时，需要通过生成器对象执行一系列步骤。

3.6.2 类图



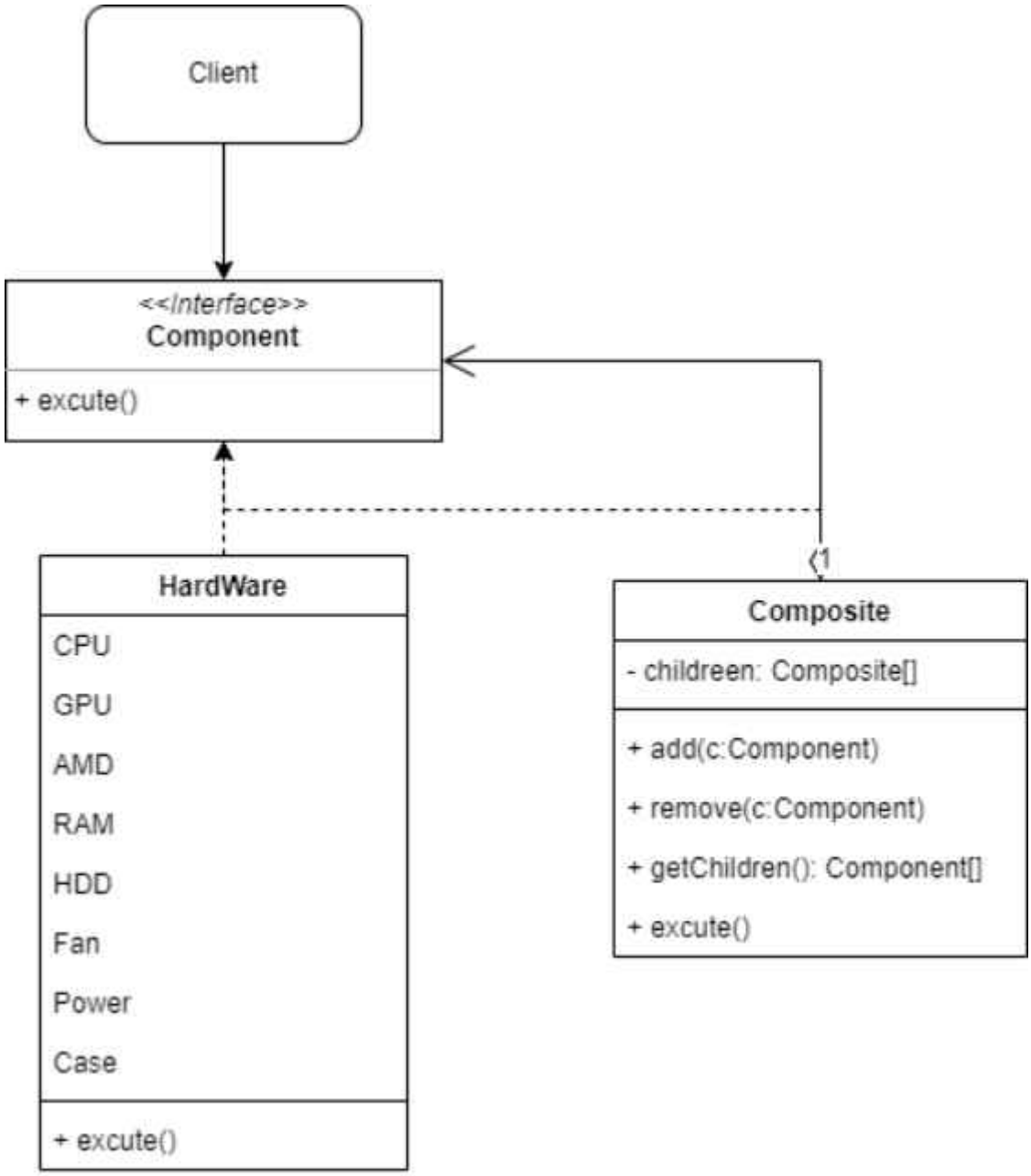
图中，Builder 接口声明电脑中不同配件的构造步骤。Concrete Builders提供各个零件的不同品牌或样式。Products是最终生成的电脑整机对象。 由不同厂家或品牌构造的产品无需属于同一类层次结构或接口。Director类定义调用装配电脑零件的顺序， 这样可以创建和复用特定的产品配置。Client 将某个Builder对象与Director类关联。

3.7 组合模式

3.7.1 实现功能

装配电脑时，每个组件是对象，电脑是一个整体对象。每台电脑包括若干硬件和软件， 硬件由CPU、显卡等构成， 软件由操作系统、应用软件等组成。指令由用户下达， 通过每个层级传递， 直到每个硬件都知道执行应该完成的指令。

3.7.2 类图



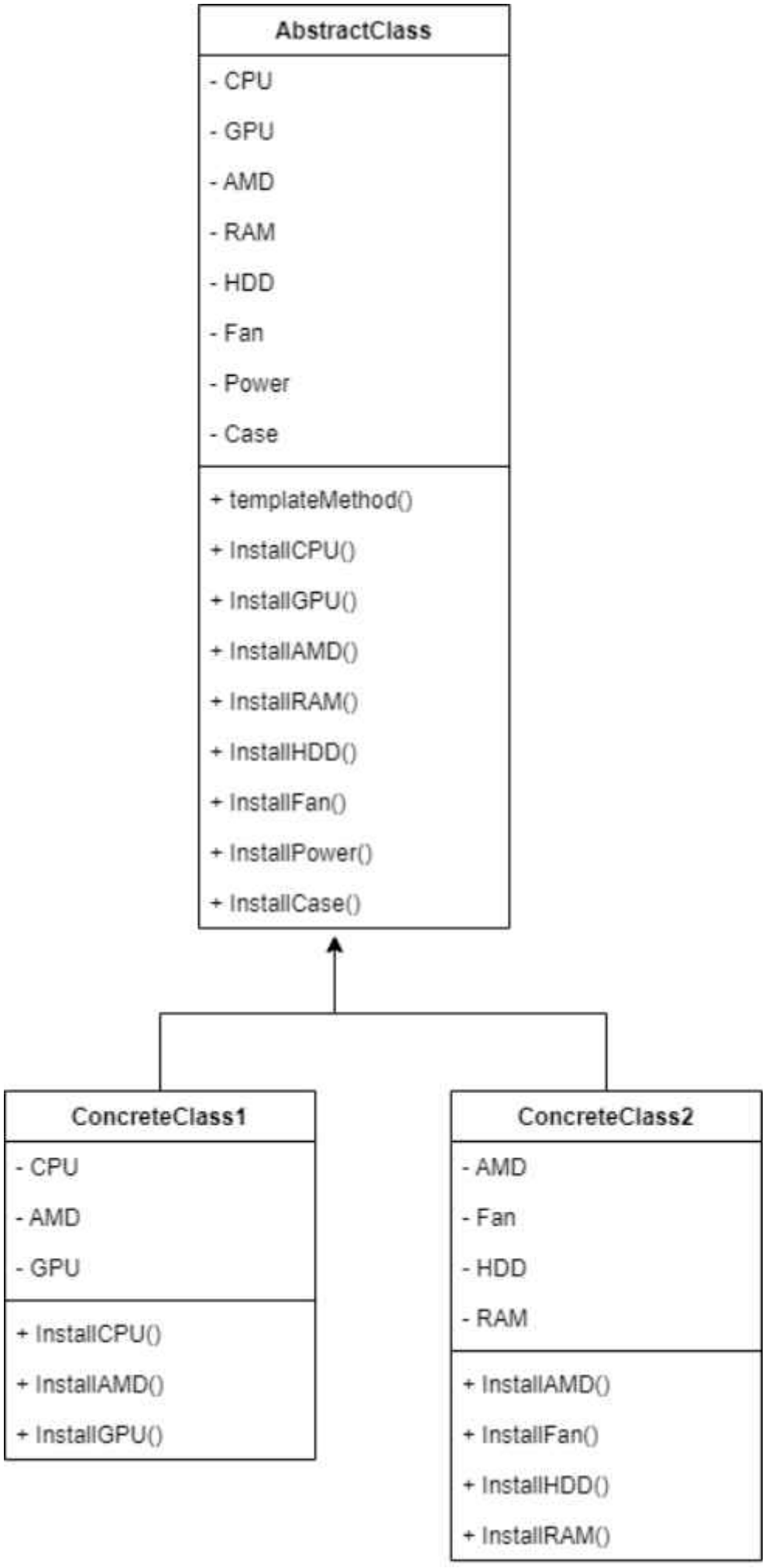
Component接口描述了整体电脑中简单项目和复杂项目所共有的操作。Composite类中规定了各种操作，如添加、移除、获取子节点等。HardWare类中包含了电脑的各个组成部件，包含CPU、GPU、AMD、RAM、HDD、Fan、Power、Case等。

3.8 模板方法模式

3.8.1 实现功能

模板方法用于装配电脑。 标准电脑装配方案中可提供几个扩展点， 允许潜在使用者调整电脑的部分细节。每个装配步骤 （例如安装CPU、GPU、显卡等） 都能进行微调， 这使得组装成的电脑的性能会略有不同。

3.8.2 类图


AbstractClass
- CPU
- GPU
- AMD
- RAM
- HDD
- Fan
- Power
- Case
+ templateMethod()
+ InstallCPU()
+ InstallGPU()
+ InstallAMD()
+ InstallRAM()
+ InstallHDD()
+ InstallFan()
+ InstallPower()
+ InstallCase()
ConcreteClass1
- CPU
- AMD
- GPU
+ InstallCPU()
+ InstallAMD()
+ InstallGPU()
ConcreteClass2
- AMD
- Fan
- HDD
- RAM
+ InstallAMD()
+ InstallFan()
+ InstallHDD()
+ InstallRAM()

AbstractClass 声明装配电脑步骤的方法， 以及依次调用它们的实际模板方法。 ConcreteClass1以及ConcreteClass1分别为电脑装配中的两个子块，它们可以重写所有步骤，但不能重写模板方法自身。

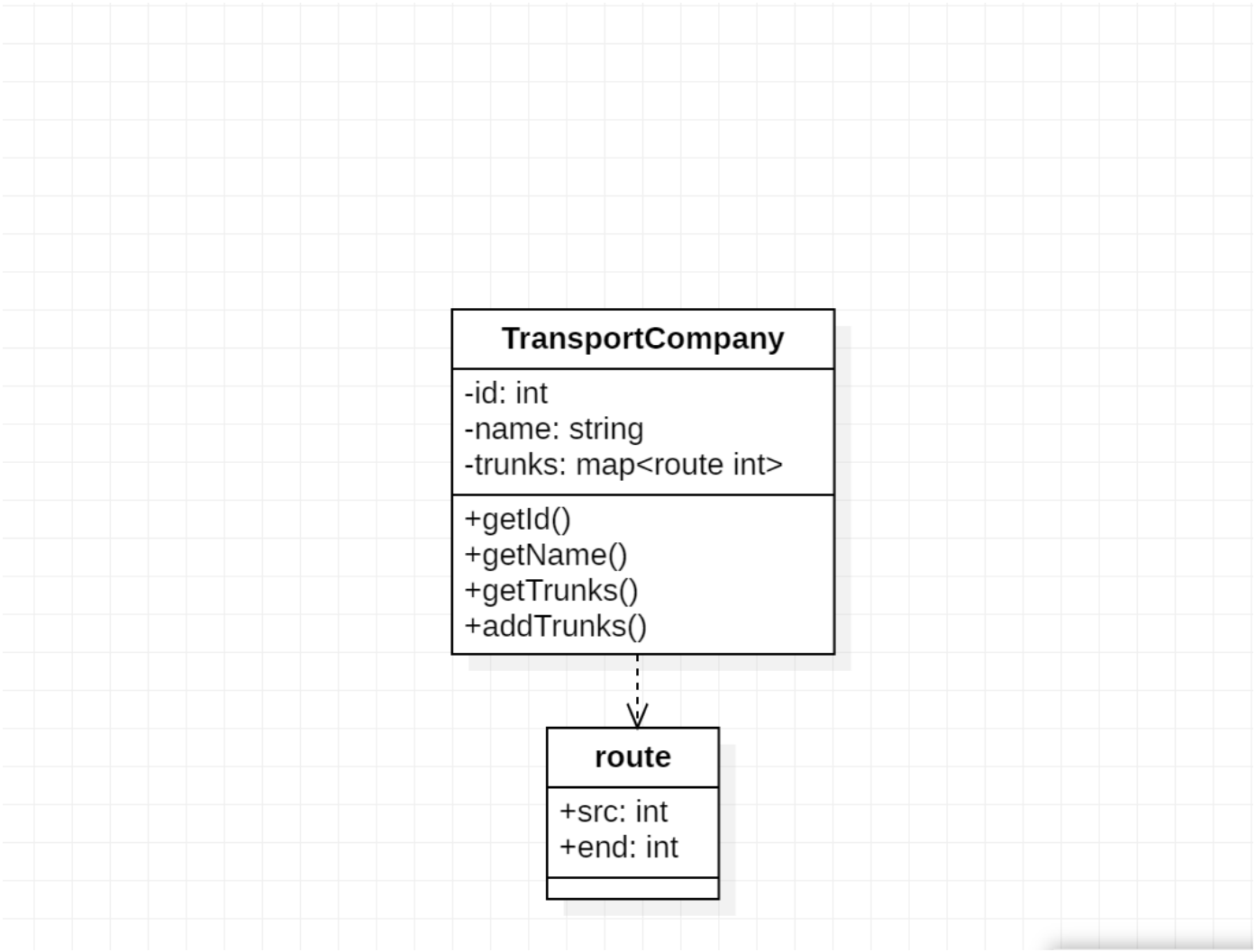
4. 物流过程与网络

4.1 享元模式

4.1.1 实现功能

享元模式用于实现物流车池的共享复用功能，运输货物时，会先查看快递公司的当前路线的运输运力是否充足，如果足够使用则直接复用已有的物流车，否则再添加新的物流车，避免出现不必要的资源浪费。

4.1.2 类图



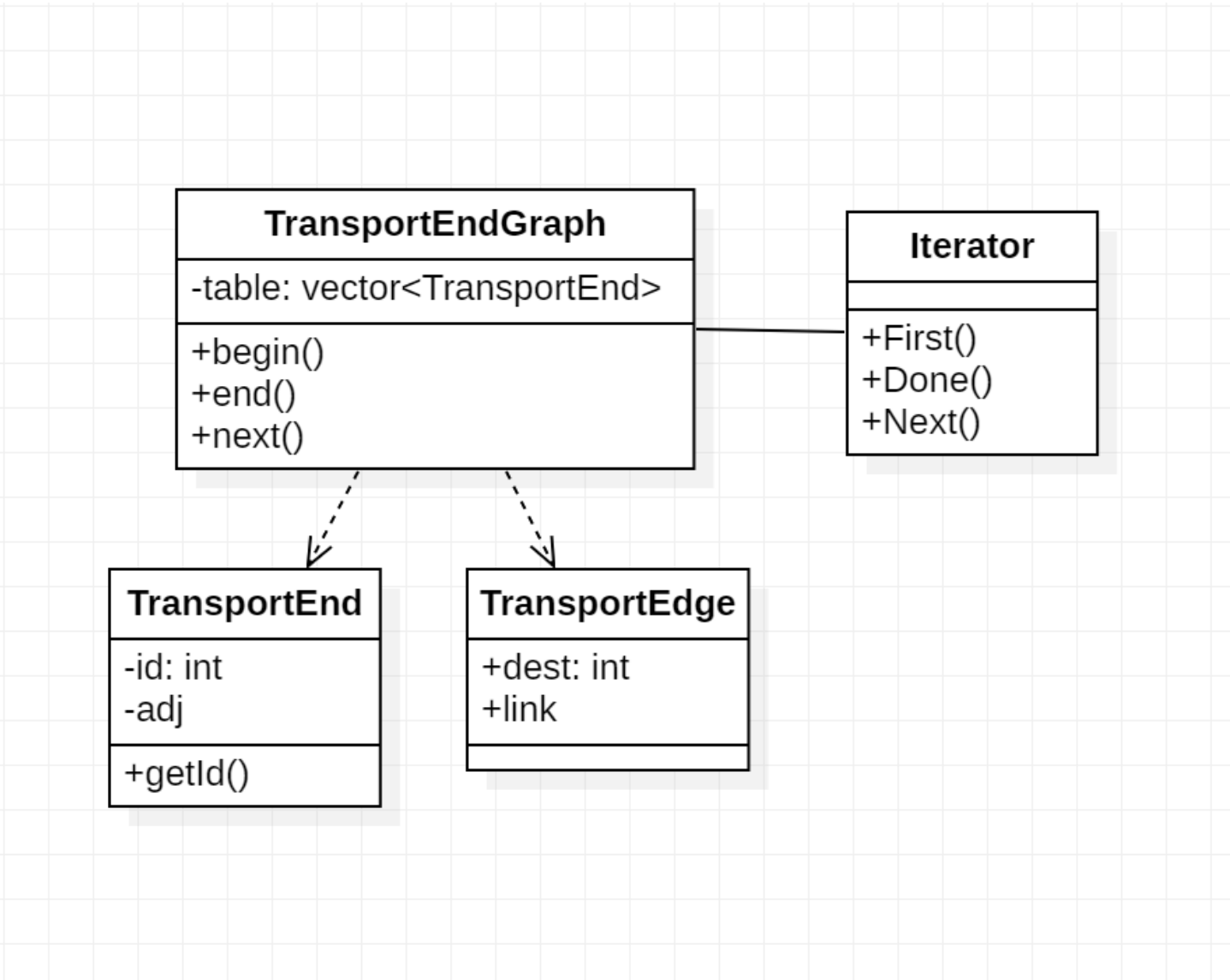
TransportCompany（物流公司）为实体类，其属性trunks（物流车池）依赖于规定的route(路线)类，当需要物流公司发货时，物流公司会根据需求查看是否能够重复利用物流车，如果运力不足则会添加新的物流车，实现共享复用。

4.2 迭代器模式

4.2.1 实现功能

迭代器模式用于遍历物流网络，在不知道物流网络内部结构的情况下，利用迭代器任然可以做到对整个物流网络的遍历。

4.2.2 类图



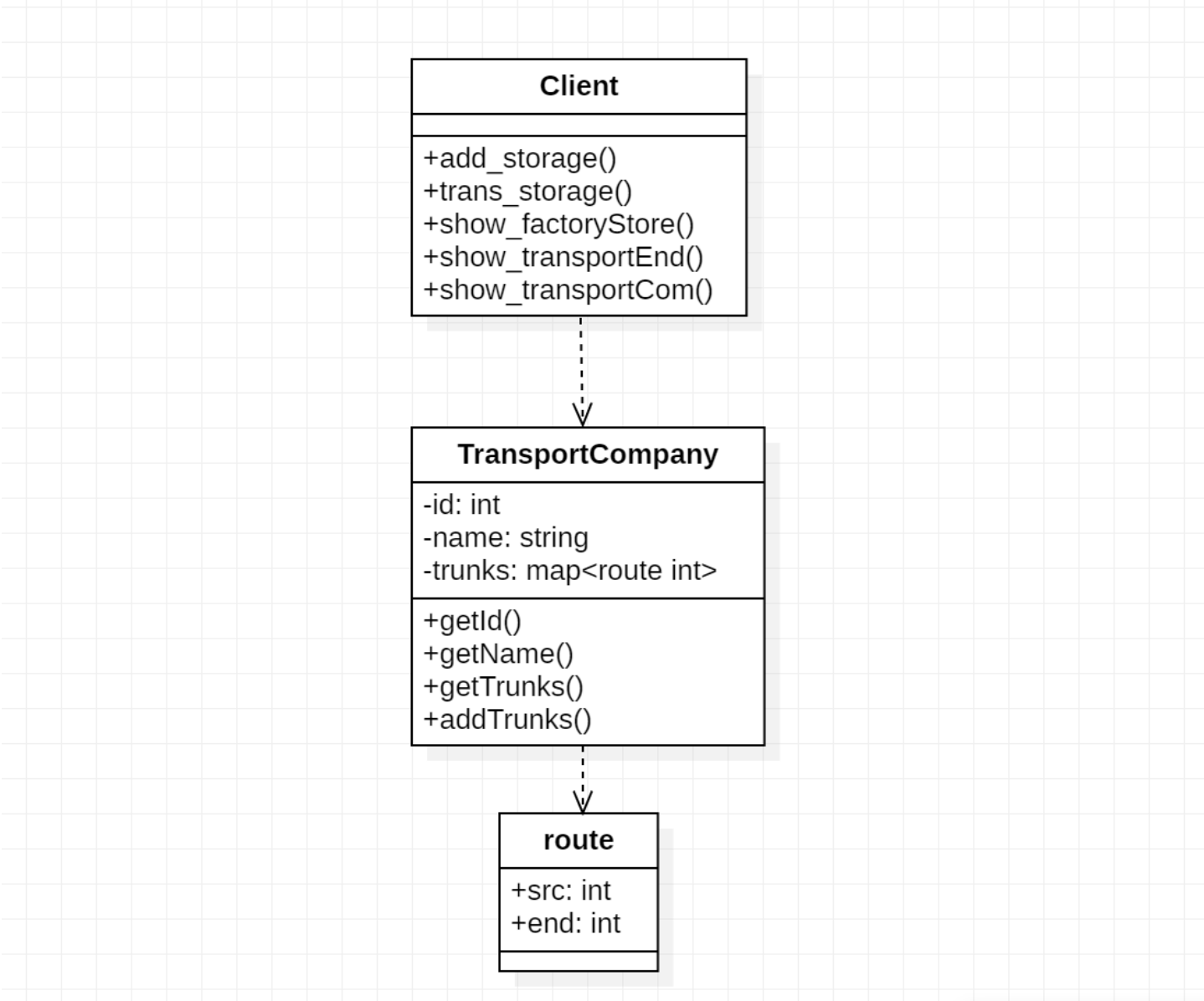
物流网络（TransportEndGraph）为依赖于物流网点（TransportEnd）和物流路线（TransportEdge）的实体类，当需要遍历物流网络时，调用Iterator迭代器中的First方法获取第一个网点，Next获取下一个，直到利用Done确定是否结束。

4.3 策略模式

4.3.1 实现功能：

策略模式会根据执行的实时情况调整不同的执行策略。

4.3.2 类图



每一个TransportCompany（物流公司）都有自己的Trunks （运输车池），当Client选择物流公司后，每个TransportCompany根据自己的运输情况采用不通过的运输策略进行运输。

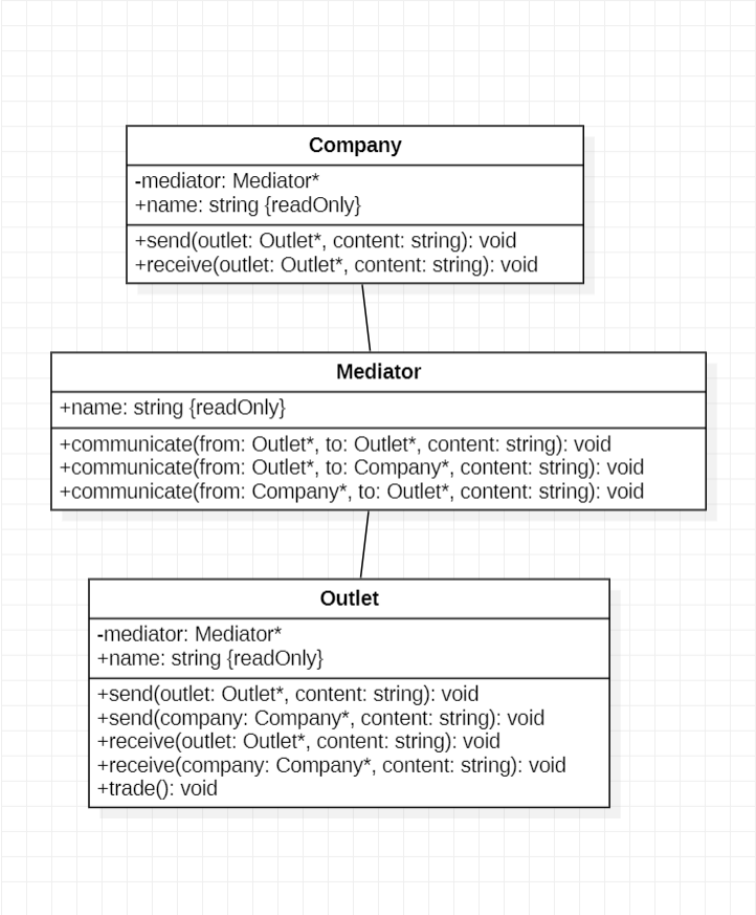
5. 销售与广告、售后服务

5.1 中介者模式

5.1.1 实现功能

此处中介者模式体现在各个销售网点以及总公司之间通过一个中介者实现通信。各个销售网点以及总公司之间直接通信，则会形成一种过度耦合的网状通信结构；引入中介者，各个销售网点互相或与总公司进行通信时，先联系中介者，再由中介者联系目标对象，实现低耦合、可重用、可扩展。

5.1.2 类图



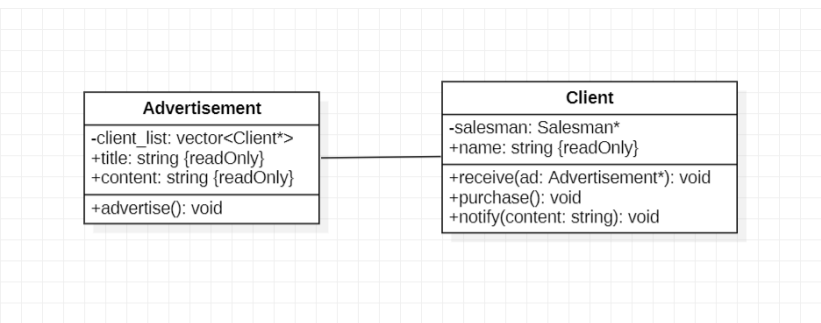
公司类（Company）和销售网点（Outlet）为两个实体类，它们分别与中介者类（Mediator相互关联）。中介者类为接口类，其中包含三个重载的通信方法communicate()以实现公司对象与其下属销售网点对象的自由通信；赋予其字符串常变量属性name的设置是为展示过程中体现中介者对象的存在。公司对象或销售网点对象通过调用与其绑定的中介者对象的方法，经中介者转发至目标对象，并告知目标对象信息来源的对象身份，实现公司与销售网点的解耦。

5.2 原型模式

5.2.1 实现功能

使用原型模式实现将同一广告复制多份发送给不同的顾客。由于广告内容存在较大较多信息，若每次生成一份广告都从数据库调用广告内容信息，将极大程度地影响效率，而原型模式通过对象拷贝的方式解决了这一问题。

5.2.2 类图



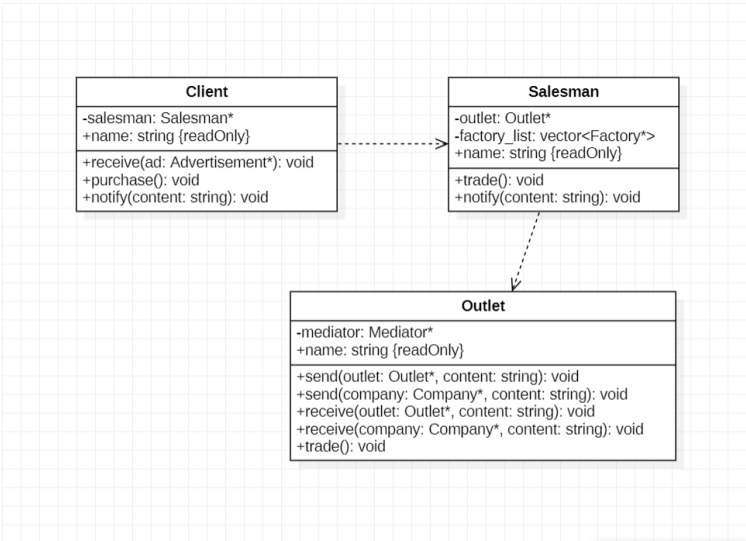
广告类（Advertisement）与顾客类（Client）存在相互关联。广告类的广告投放方法advertise()通过遍历每个广告对象构造时绑定的顾客名单，向名单中的每一个顾客发送一个与当前广告对象完全相同的副本以贯彻原型模式。副本通过广告类的拷贝构造函数构造。顾客对象的接收方法receive()依赖该副本广告对象实现广告的接收和阅读。

5.3 代理模式

5.3.1 实现功能

此处代理模式体现在销售人员代理销售网点，直接与顾客进行电脑产品的买卖功能，当顾客与销售人员交易成功，销售人员将通知其代理的销售网点此次交易成功的信息。

5.3.2 类图



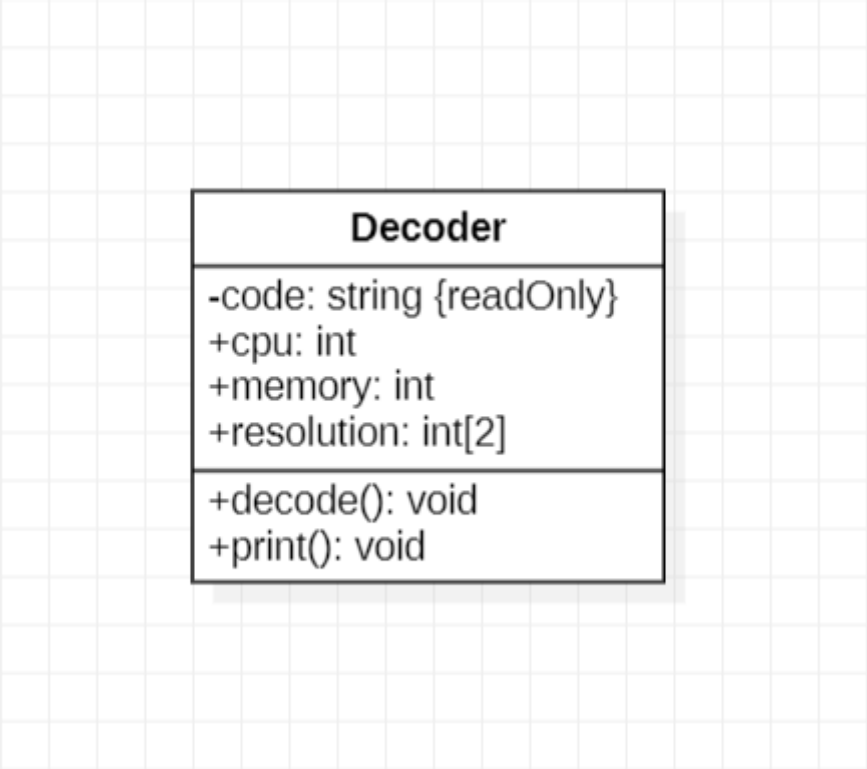
顾客类（Client）依赖于销售人员类（Salesman），每个顾客对象在构造时都绑定了一个专门为其服务的销售人员对象。代理模式体现在销售人员类与销售网点类（Outlet）之间。通常情况下代理模式常用代理类继承抽象父类以继承相同的方法实现代理，此处为了能够配合整个销售与售后子系统的正常调用，使用了另一种实现方式：销售人员类依赖于销售网点类，每个销售人员对象构造时绑定一个销售网点对象以建立二者的联系，且这两个类中有着完全相同的方法trade()，顾客使用购买方法purchase()时与销售人员对象进行交互，而不与销售网点对象发生调用，以体现代理功能。购买结束后，销售人员对象给其绑定的销售网点对象发送交易成功信息。

5.4 解释器模式

5.4.1 实现功能

使用解释器模式，创建一个解码器，对输入的电脑型号编码表达式进行识别和翻译，进而获取电脑产品的基本配置信息。

5.4.2 类图



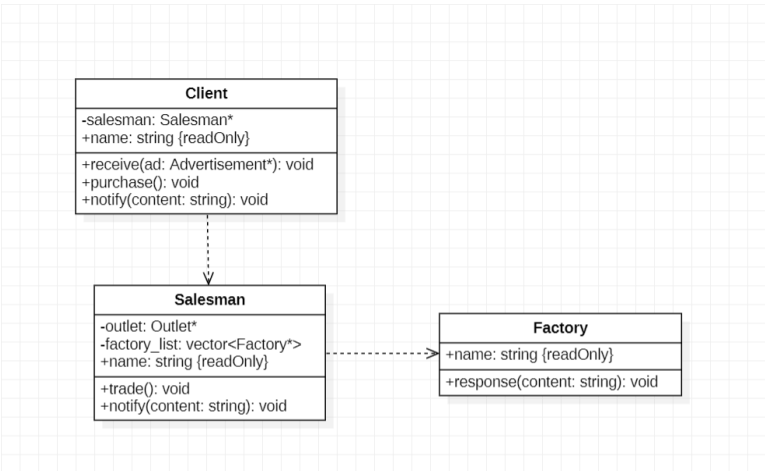
解码器类（Decoder）用于实现对电脑产品型号编码的识别与翻译。其私有属性code为字符串常变量，在构造时传入，每个解码器对象对应一个产品型号编码。使用时通过调用解码方法decode()，将code的内容按照预设算法分解、转换，得到公有属性：CPU核数（cpu）、内存容量（memory）、分辨率（resolution），并可通过打印方法print()输出到终端。

5.5 观察者模式

5.5.1 实现功能

顾客发现购买的产品存在质量问题或有想要提出的改进建议，可将故障或建议信息发送，通知其观察者，即销售人员。销售人员随即响应，并通知其观察者，即销售人员负责的许多工厂。工厂观察到销售人员的通知后，对产品进行整改。

5.5.2 类图



顾客类（Client）依赖于销售人员类（Salesman），每个顾客对象在构造时绑定一个销售人员对象作为其观察者；销售人员类依赖于工厂类（Factory），每个销售人员对象在构造时绑定工厂名单，名单

里的每个工厂对象均为该销售人员对象的观察者。顾客对象和销售人员对象分别通过它们的通知方法notify()以告知其观察者，观察者随即给出响应。销售人员对象的响应即为发送通知给工厂，工厂对象的响应方法response()即整改产品。

参考文献

[1] Deepak Alur, Dan Malks, John Crupi . Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall PTR, 2003. 395-397.
[2] M. Birbeck. Professional XML. Beijing: Machine Press, 2002. 14-16

[3]RobertNystrom.GameProgrammingPatterns[M]ISBN:978-0-9905829-0-8,2004,291

[4] William Crawford, Jonathan Kaplan. J2EE Design Patterns - Patterns in the Real World. [M]DBLP:William Crawford,2003.37:50

[5] Grand M . Patterns in Java: A of Reusable Design Patterns Illustrated with UML, 2nd Edition, Volume 1[J]. computer bookshops, 2002, 21(9):153-162.

[6] Dmitry Zhart https://refactoringguru.cn/refactoring/course