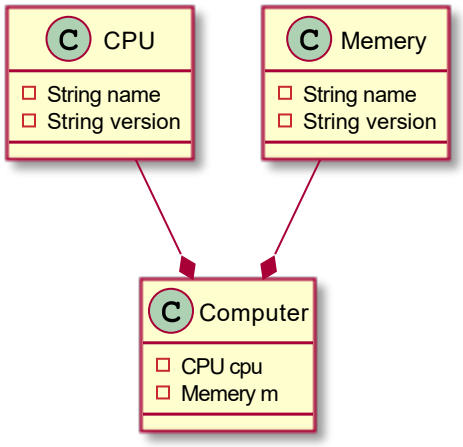
第2章 工厂方法模式

提出问题

> 初始问题：某公司需要采购（buy）一批电脑，决定购买配件后自己组装。请根据需求设计程序模拟业务流程。

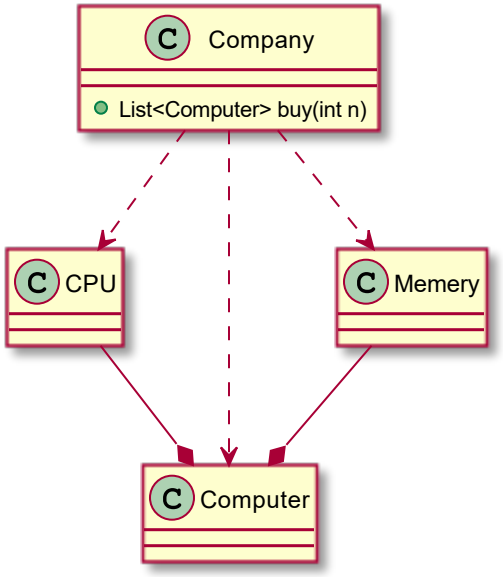
假设电脑由处理器（CPU）和内存（Memery）两个配件组成，其类图结构如下：



根据问题描述，公司购买N台电脑的命令流程设计如下：

```
public class Company {
        public List<Computer> buy(int n) {
                List<Computer> compList = new ArrayList<Computer>();
                for(int i = 0; i < n; i++) {
                        Computer comp = new Computer();
                        CPU cpu = new CPU("Intel", "i5");
                        Memery m = new Memery("Kingston", "8G");
                        comp.setCpu(cpu);
                        comp.setM(m);
                        compList.add(comp);
                }
                return compList;
        }
}
```

在执行购买流程时，`Company`类与`Computer`、`CPU`、`Memery`等多个类存在依赖关系（类图如下），一定程度违背了迪米特法则。

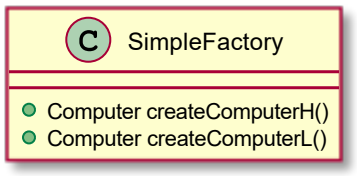


> 问题升级：这里所购买电脑的型号是固定的，假设新需求为购买高低两种配置电脑，以上程序又该如何修改？

首先`buy()`函数的参数设计无法满足新需求，需要增加电脑配置的类型信息，例如将电脑数量参数修改为一个采购单，采购单记录了需要采购的电脑配置类型；另外，在组装电脑时需要建立选择分支，区分高低配电脑的不同组装流程。代码修改如下：

```
public class Company {
        public List<Computer> buy(List<String> orderList) throws Exception {
                List<Computer> compList = new ArrayList<Computer>();
                for(String type:orderList) {
                        Computer comp = new Computer();
                        if(type.equals("H")) {
                                comp.setCpu(new CPU("Intel", "i5"));
                                comp.setM(new Memery("Kingston", "8G"));
                        }else if(type.equals("L")){
                                comp.setCpu(new CPU("Intel", "i7"));
                                comp.setM(new Memery("Kingston", "16G"));
                        }else {
                                throw new Exception("Undefined Computer Type");
                        }
                        compList.add(comp);
                }
                return compList;
        }
}
```

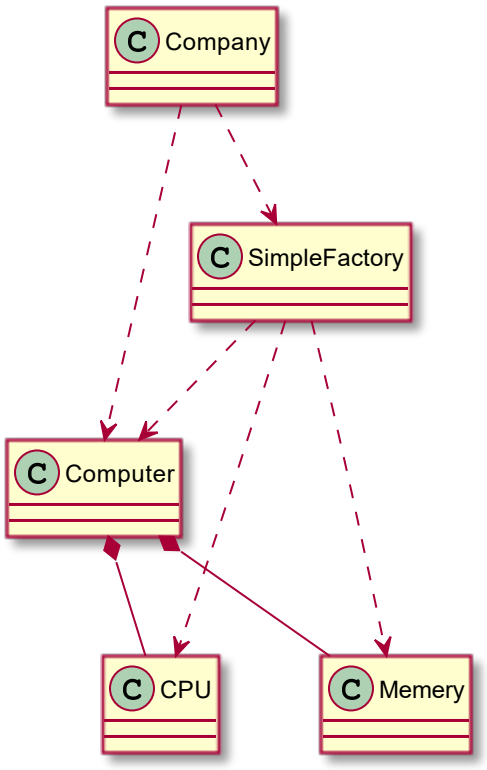
这里看到由于`Company`的设计违背了迪米特法则，当需求变化时，需要修改较多代码，这也违背了开闭原则。思考如何减少修改`buy()`方法的代码？面对复杂对象的创建时，其创建过程的代码是比较多的，因此可以将这部分代码切割出去，由专门的类负责管理，这个类称为简单工厂。如下图所示，`SimpleFactory`类负责生产高配置和低配置两种型号的电脑。



公司的购买流程代码修改如下：

```
public class Company {
        public List<Computer> buy(List<String> orderList) throws Exception {
        SimpleFactory sf = new SimpleFactory();
                List<Computer> compList = new ArrayList<Computer>();
                for(String type:orderList) {
                        Computer comp;
                        if(type.equals("H")) {
                                comp = sf.createComputerH();
                        }else if(type.equals("L")){
                                comp = sf.createComputerL();
                        }else {
                                throw new Exception("Undefined Computer Type");
                        }
                        compList.add(comp);
                }
                return compList;
        }
}
```

可以看出，修改后的设计中，`Company`类降低了与系统其他类的依赖程度，其类图如下：



在目前的设计中，当需要扩展一种电脑型号时，只需要在`SimpleFactory`类中增加一个方法，并在`buy()`中增加一个条件分支。虽然修改原模块的代码量减少了，但终究还是违背了开闭原则，

需要思考如何完全避免修改原模块。这里的关键还在于`buy()`代码的设计太过具体，没有充分做到面向抽象编程，比如设计了`createComputerH()`和`createComputerL()`两种具体的操作。如果这里将操作统一为`createComputer()`，再利用多态实现具体创建产品的多样化，是否就能将`buy()`方法"固定"下来。这种思路就是接下来要介绍的工厂方法模式。

模式名称

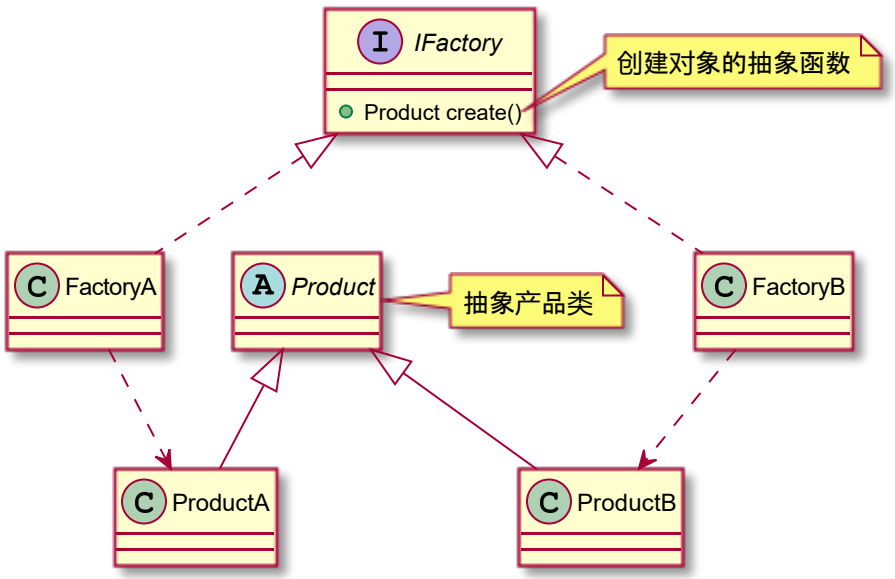
工厂方法：`Factory Method`或则`Virtual Constructor`

设计意图

定义一个接口创建对象，让子类决定哪个类被实例化。这里强调定义一个创建对象的统一操作，也就是抽象方法，可以定义在接口或抽象类中。抽象函数被不同子类重写，由子类确定最终创建的对象类型，这也是典型的多态应用。

> Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

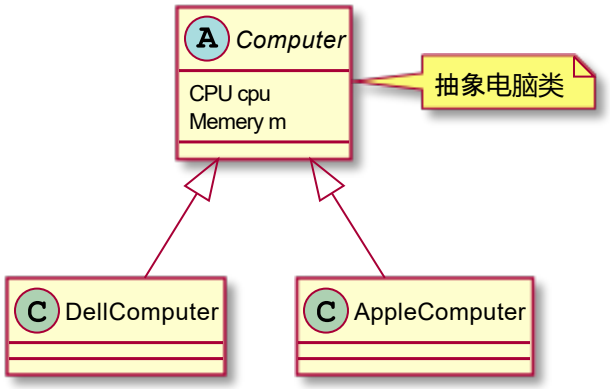
设计结构



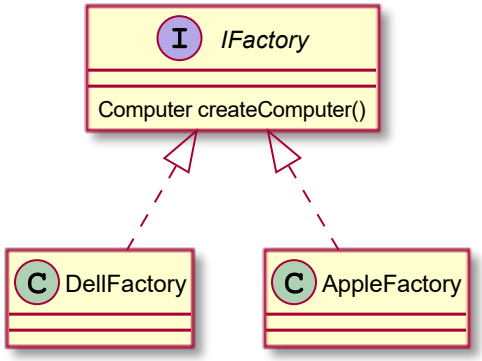
解决问题

> 要解决的问题是如何修改之前的设计，使其符合开闭原则，当产品类型扩展时，不修改原模块代码。

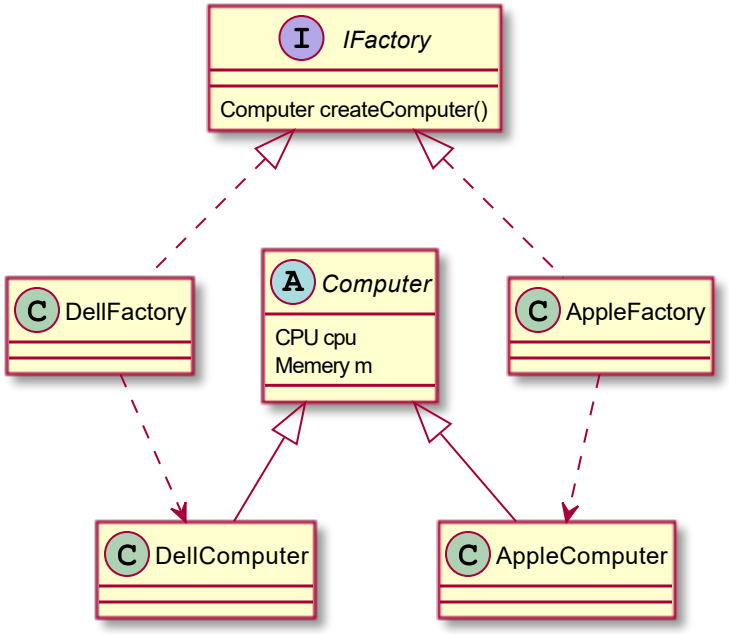
这里稍微规范一下产品类型，假设现在有戴尔、苹果两个品牌的电脑，那么产品类图如下：



要统一操作，就要建立一个公共接口（`IFactory`），针对不同品牌的电脑建立专门工厂，类图如下：



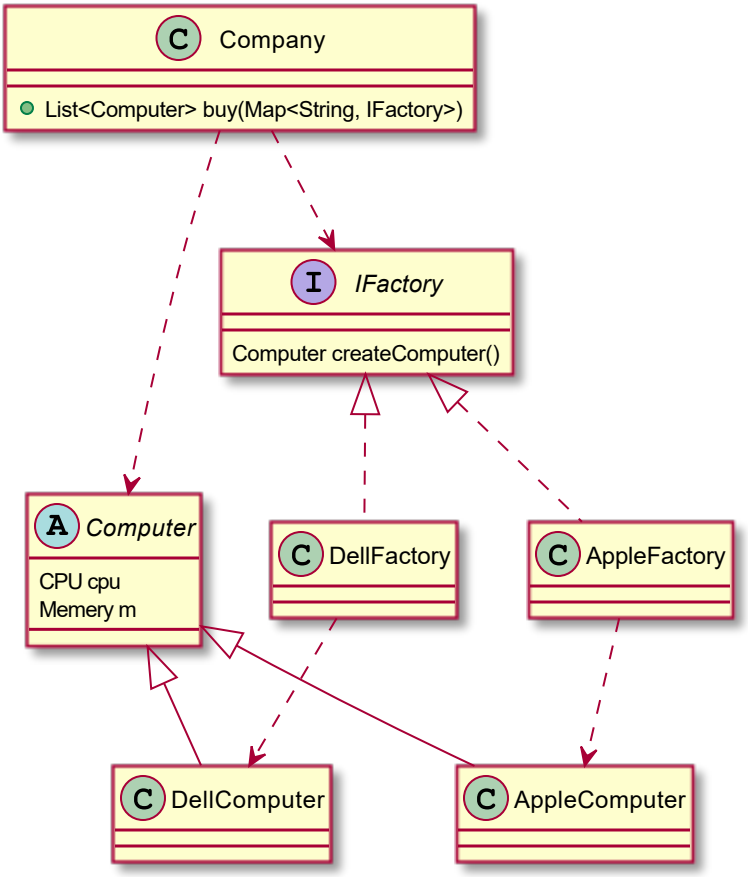
具体工厂生产具体电脑，两者存在依赖关系，将产品类和工厂类结合起来如下：



公司类的`buy()`调整如下，订单增加了工厂信息，具体流程为：遍历订单列表，获取每项对应的工厂，调用工厂的生产函数返回产品。产品类扩展时，`buy()`方法不需修改，符合开闭原则。

```
public class Company {
        public static void main(String[] args) {
                Company company = new Company();
                Map<String, IFactory> orderMap = new HashMap<String, IFactory>();
                IFactory dellF = new DellFactory();
                IFactory appleF = new AppleFactory();
                orderMap.put("001", dellF);
                orderMap.put("002", appleF);
                company.buy(orderMap);
        }
        public List<Computer> buy(Map<String, IFactory> orderMap) {
                List<Computer> compList = new ArrayList<Computer>();
                for(String orderID:orderMap.keySet()) {
                        compList.add(orderMap.get(orderID).createComputer());
                }
                return compList;
        }
}
```

完整的类图如下：



效果与适用性

主要效果体现在降低耦合、提升扩展性、减少维护代价等方面：

- 封装对象创建过程，减少上层模块对具体细节的依赖；
- 设计统一操作接口，通过依赖倒置实现面向抽象编程；
- 将细节延迟到子类，使系统能根据需求灵活扩展。

适用性：

- 当需要创建对象的类信息未知时。例如在上层代码逻辑中要创建很多类型产品对象，但各类产品细节还不知道。
- 当需要由子类来指定创建对象时。
- 将创建对象的逻辑局部化（localize）到具体子类。例如在之前的购买流程中，客户端需要设计针对所有类型电脑的创建流程，进行了全局化设计；而采用工厂方法模式后，创建逻辑延迟到子类，每个子类只需要了解单一类型的创建，不需要知道全局信息。