

Lecture 13. GRASP Patterns in OO Design II

-General Principles in Assigning Responsibilities

责任分配的一般原则

Object Oriented Modeling Technology
面向对象建模技术

Professor: Yushan Sun
Fall 2022

Contents

1. Concept of pattern 模式的概念
2. Information Expert (信息专家模式)
3. Creator (创建者模式)
4. Low Coupling (低耦合模式)
5. High Cohesion (高内聚)
6. Controller (控制器模式)
7. Polymorphism (多态模式)
8. Pure Fabrication (纯虚构模式)
9. Indirection (间接模式)
10. Protected Variations (受保护变化模式)
11. Law of Demeter

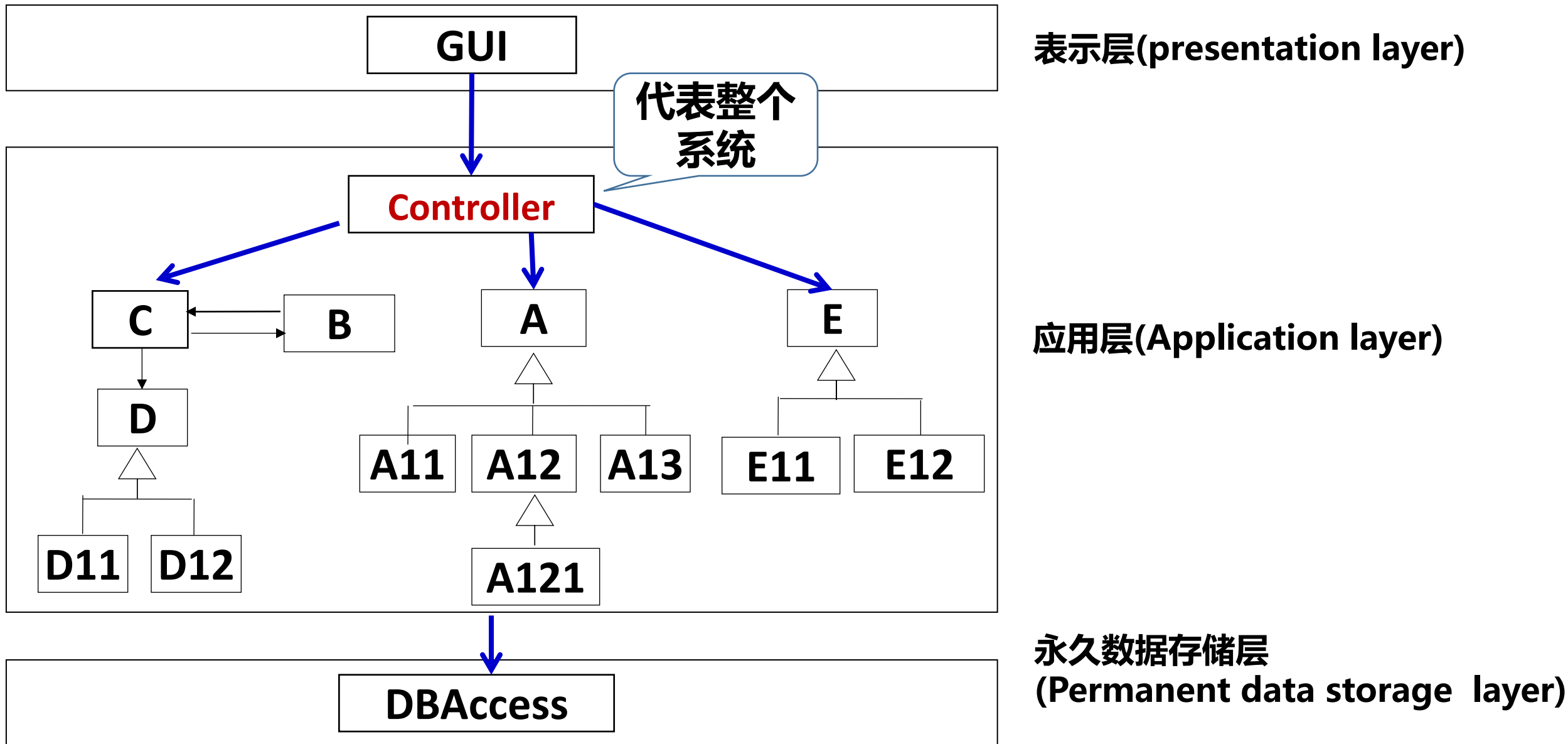
6. Controller Pattern

控制器模式

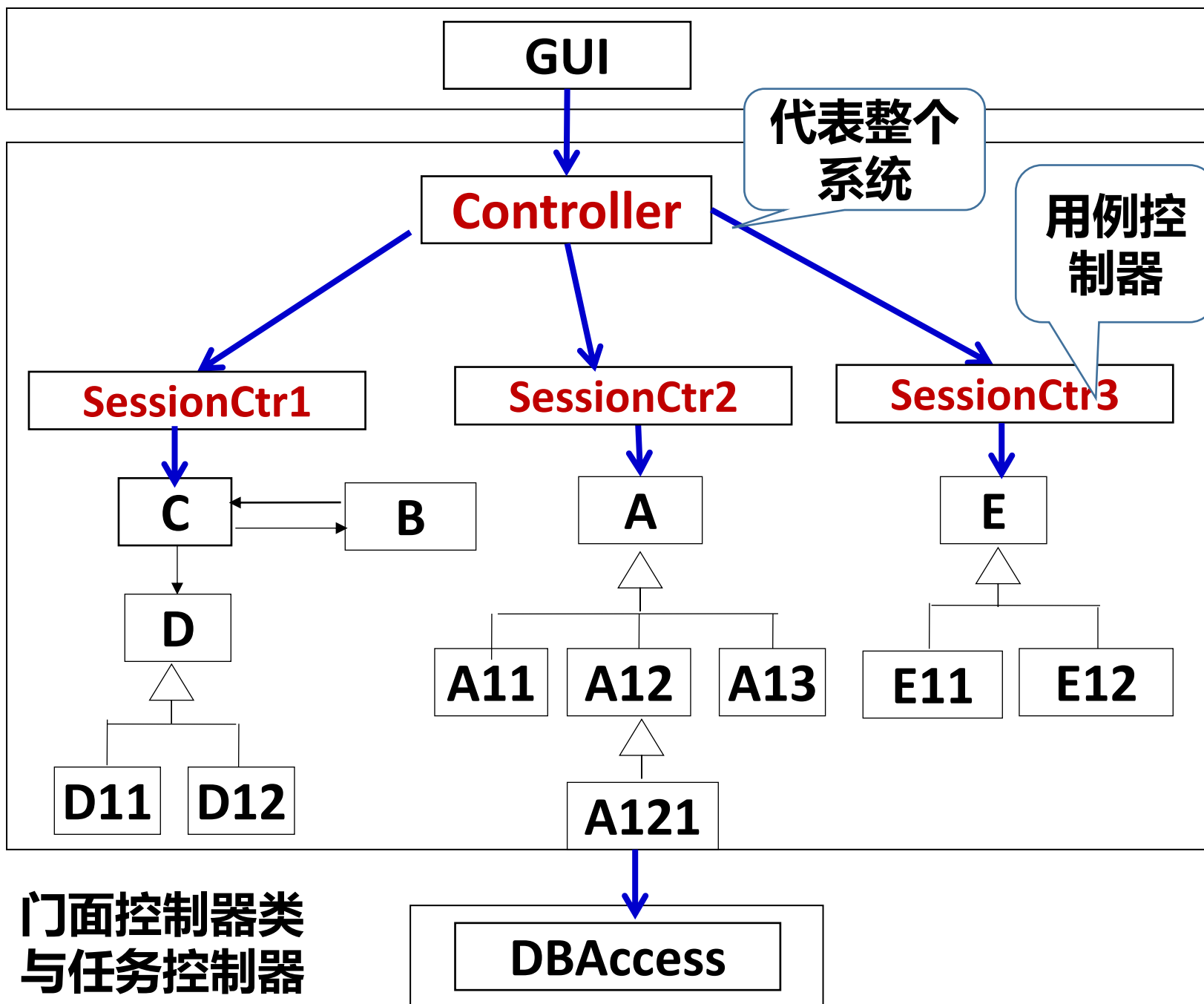
6. Controller Pattern

- **Background:**
- 系统消息由外部用户产生
- 这些消息关系到一些操作，因为系统为了响应这些消息需要一些操作。
- **问题：**谁负责处理输入系统事件。

- **控制器模式：** 将接受（处理）系统事件消息的责任分配给一个
 - 代表整体系统（子系统、设备）的类，叫做门面控制器或者
 - 代表一个用例场景的类，称为用例控制器的类。
- **Controller pattern:** Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:
 - Represents the overall system, device, or subsystem (*facade controller*). 门面控制器
 - Represents a use case scenario within which the system event occurs, (*use-case or session controller*). 任务控制器



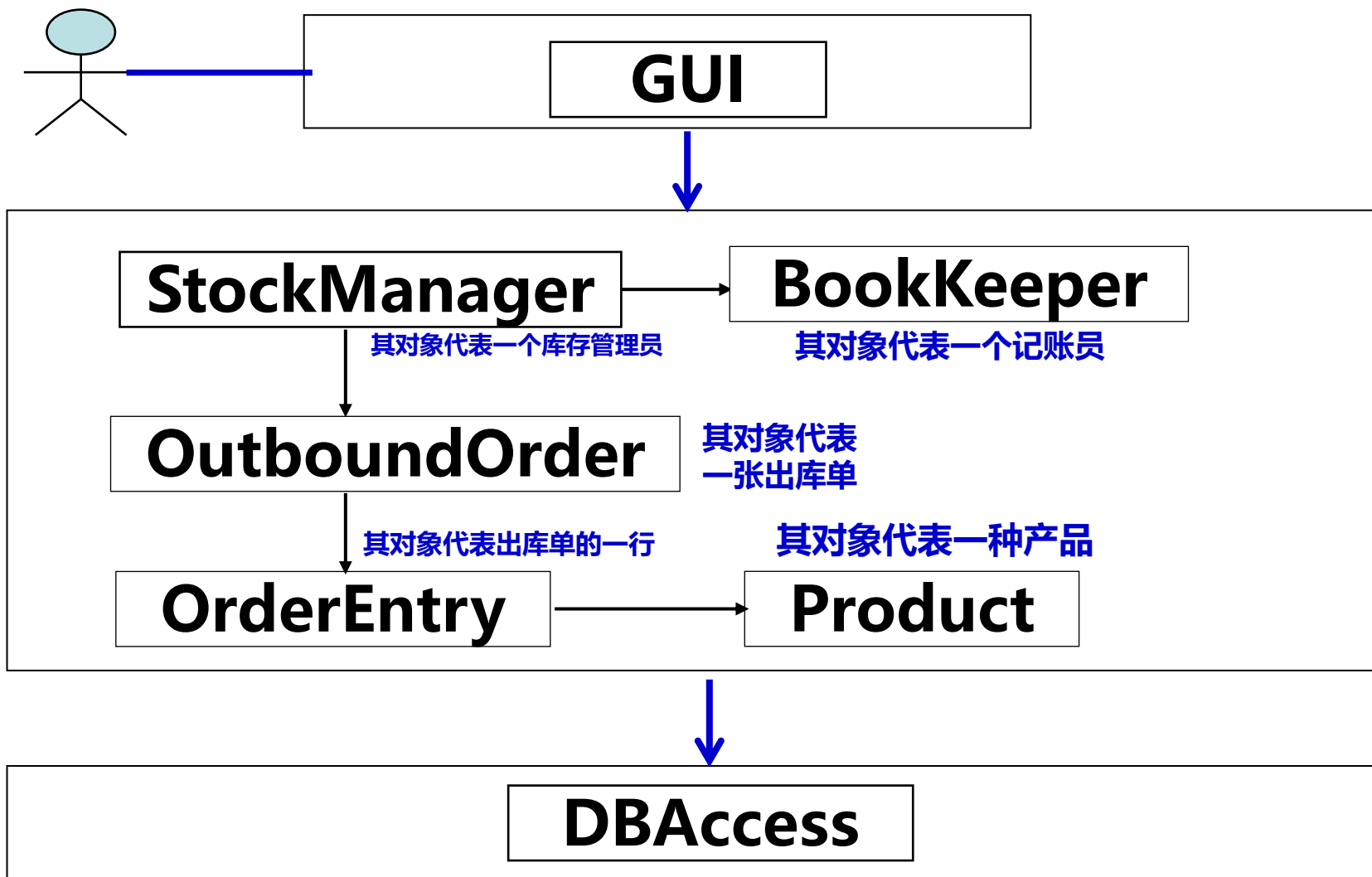
在应用层需要有一个专门的控制器类（Controller）负责接受用户请求



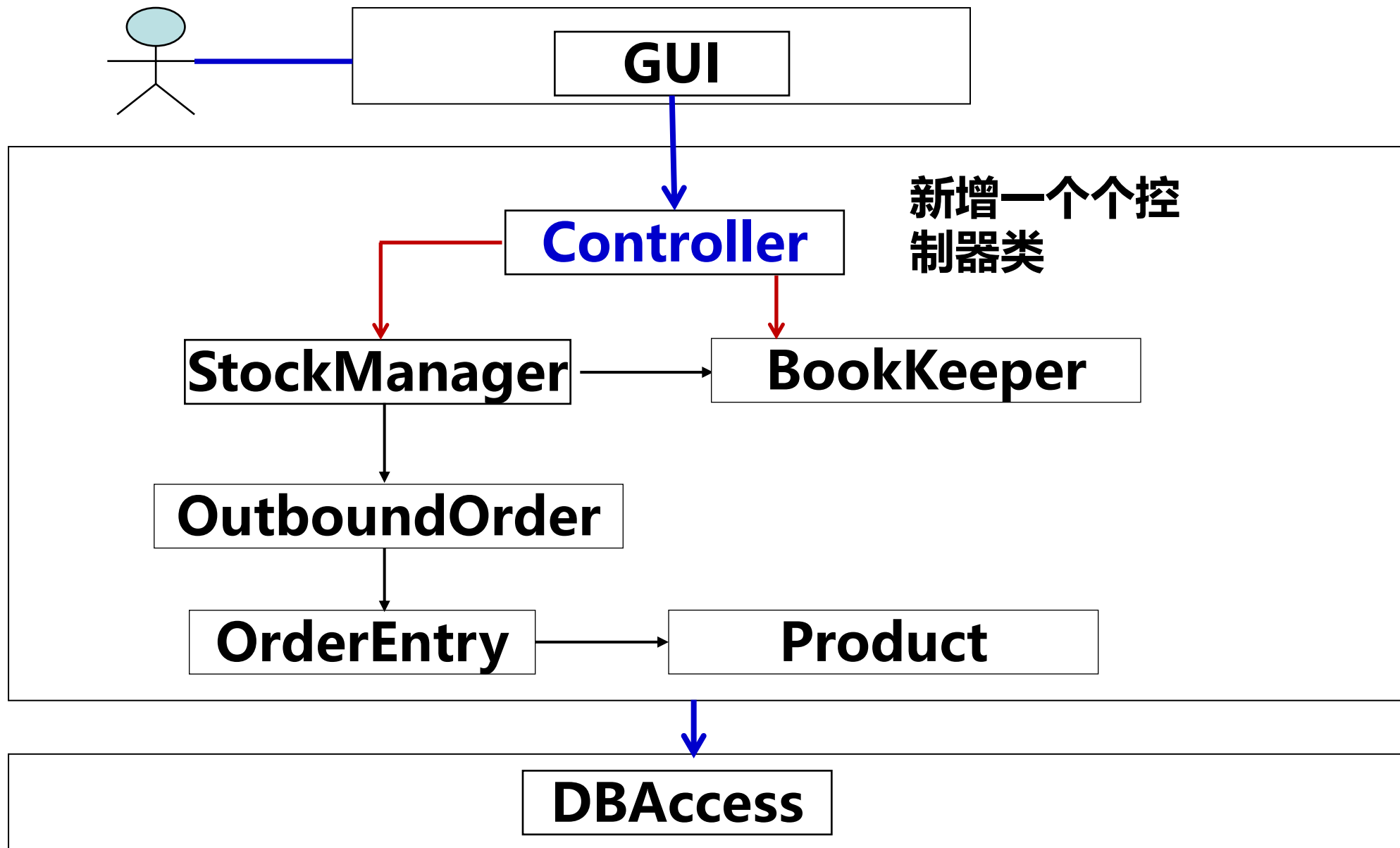
- **控制器是一个非用户图形界面对象。控制器负责接收或处理系统消息。控制器决定调用哪些系统操作。**
- **Note:用户图形界面类不适合做控制器类。**

例9. 出库单的例子(继续)

- 现在在出库单的例子程序中，要看看哪个类负责做Controller 类？



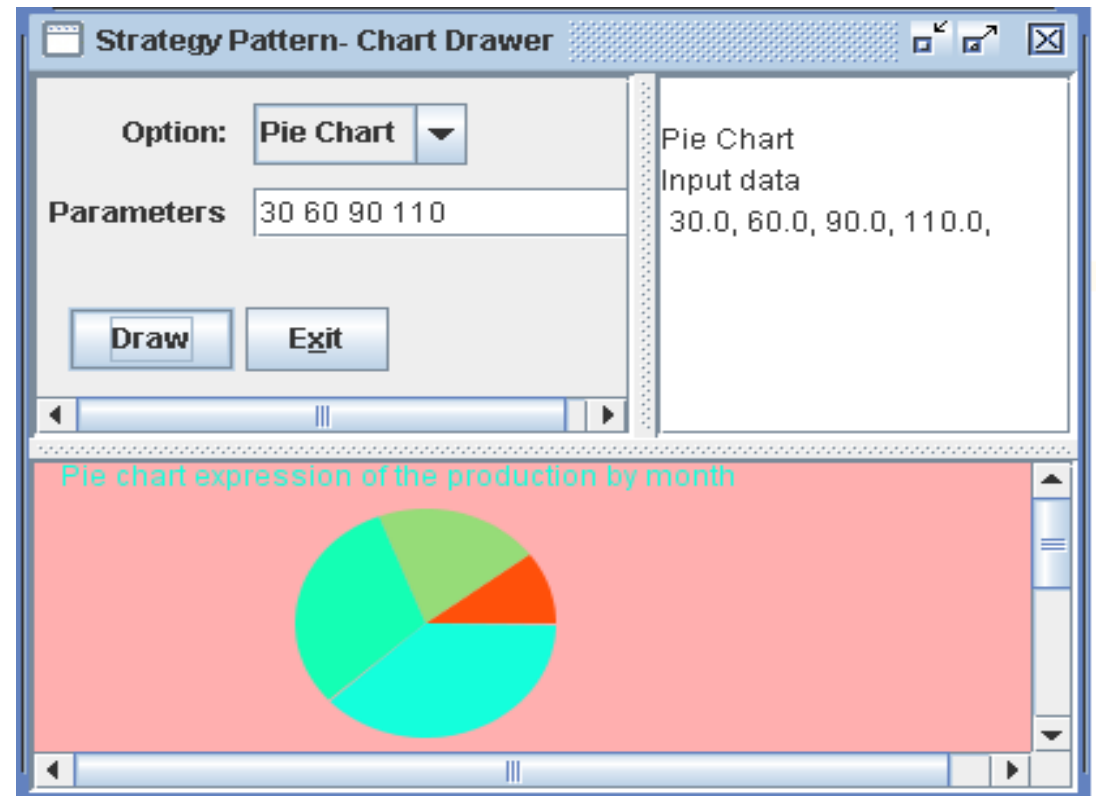
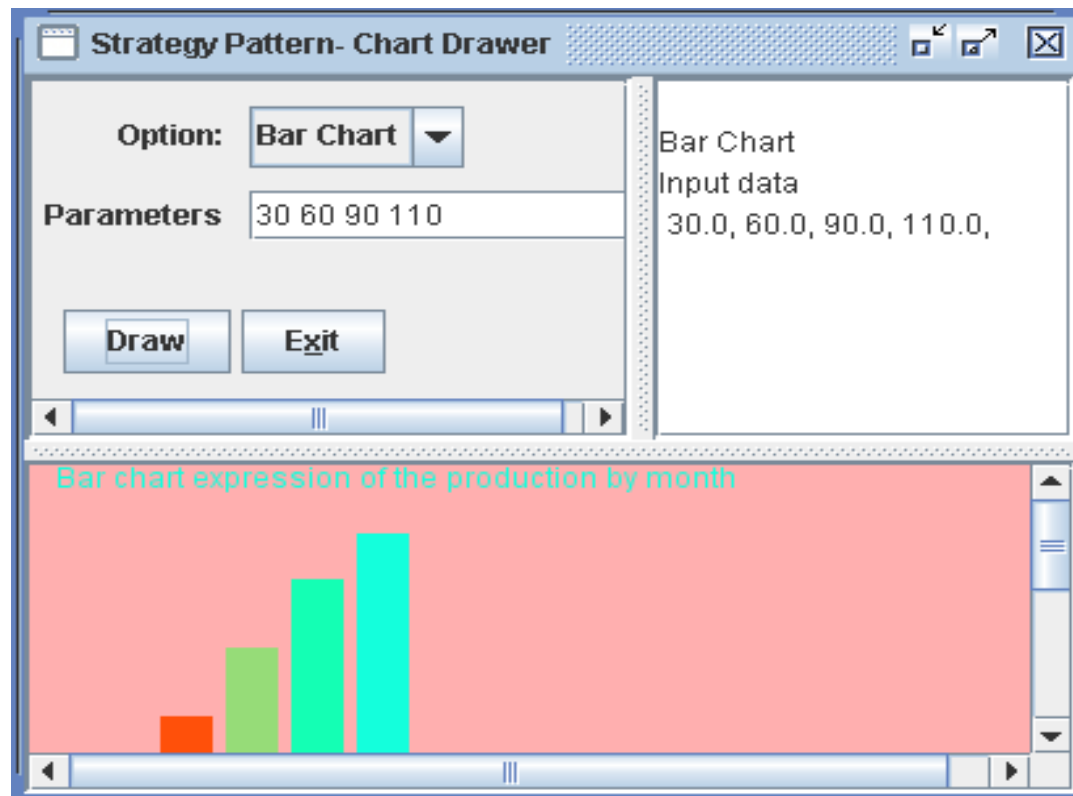
应用层的这几个类都不适合增加控制责任。

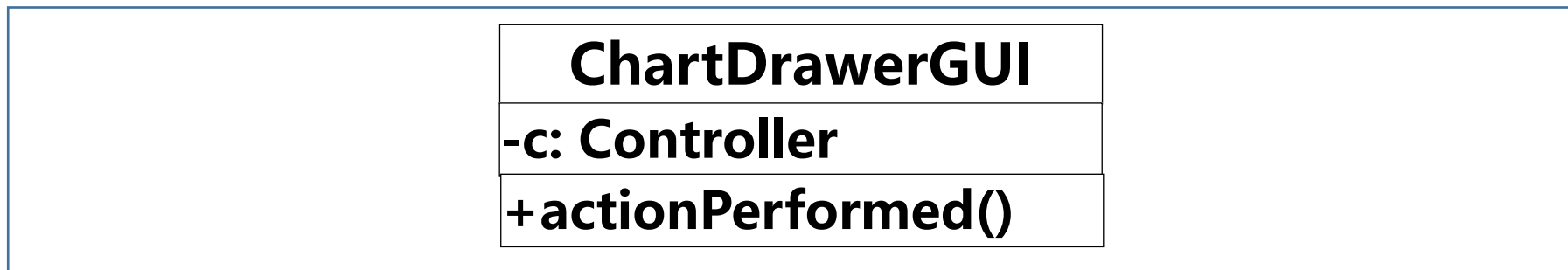


原来的领域类（实体类）都不适合作为控制器类，因此创建一个专门的Controller类较好。

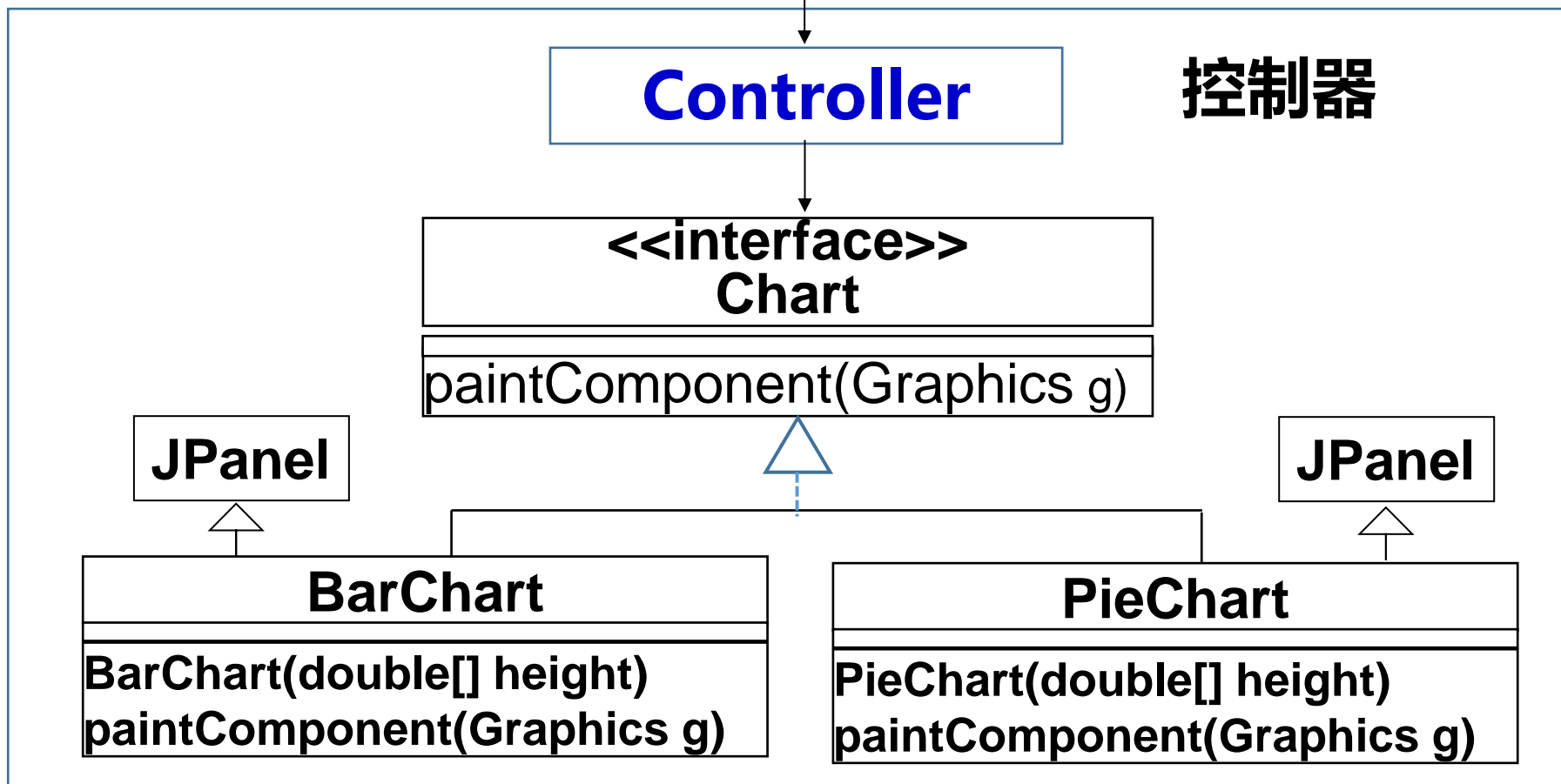
例10. 网站多种图表程序的设计-控制器模式的例子

- 在政府或商业网站上，许多数据需要用图表表示。同一组数据可以通过不同的图表显示，例如柱形图(条形图)，饼图(饼图)。
- 该软件采用3层层次架构进行设计，原型图如下。





表示层



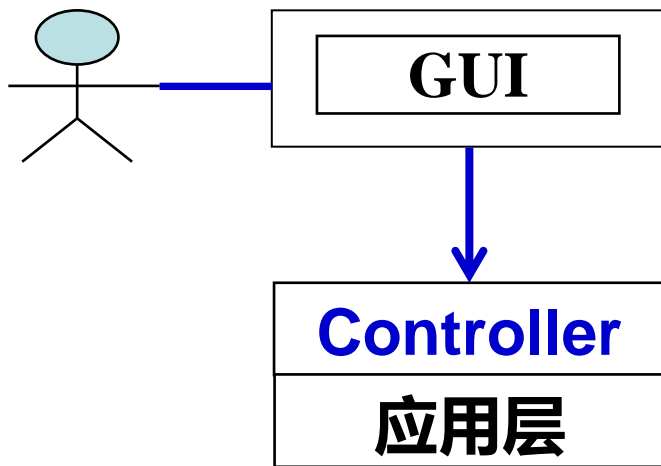
控制器

应用层

优点:

- **Controller类对象代表了应用层的控制器，它提供了用户图形界面所需要的方法。这样，**
- **ChartDrawerGUI类就可以不和Chart层次类产生关联，而只要调用Controller类即可。**
- **用户图形界面ChartDrawerGUI是可插拔的，可以轻松地替换的。**

- 使用控制器模式的好处(Benefits).
- 增加用户图形界面的复用，产生可插拔的用户图形界面可能性。
 - 保证应用逻辑不包含在GUI层。
 - 将系统操作代理给一个Controller类的好处是支持**应用逻辑**在将来的应用程序中的复用。



GUI仅仅负责输入、输出；捕捉了用户输入之后，仅仅将输入传给Controller；而不是调用应用层其它的类；GUI不包含业务逻辑。GUI的复用性好。

对于GUI传递进来的请求，根据请求的不同，而调用应用层的不同的类；充当指挥员的角色；包含了高层的业务逻辑。

Back

7. Polymorphism Pattern

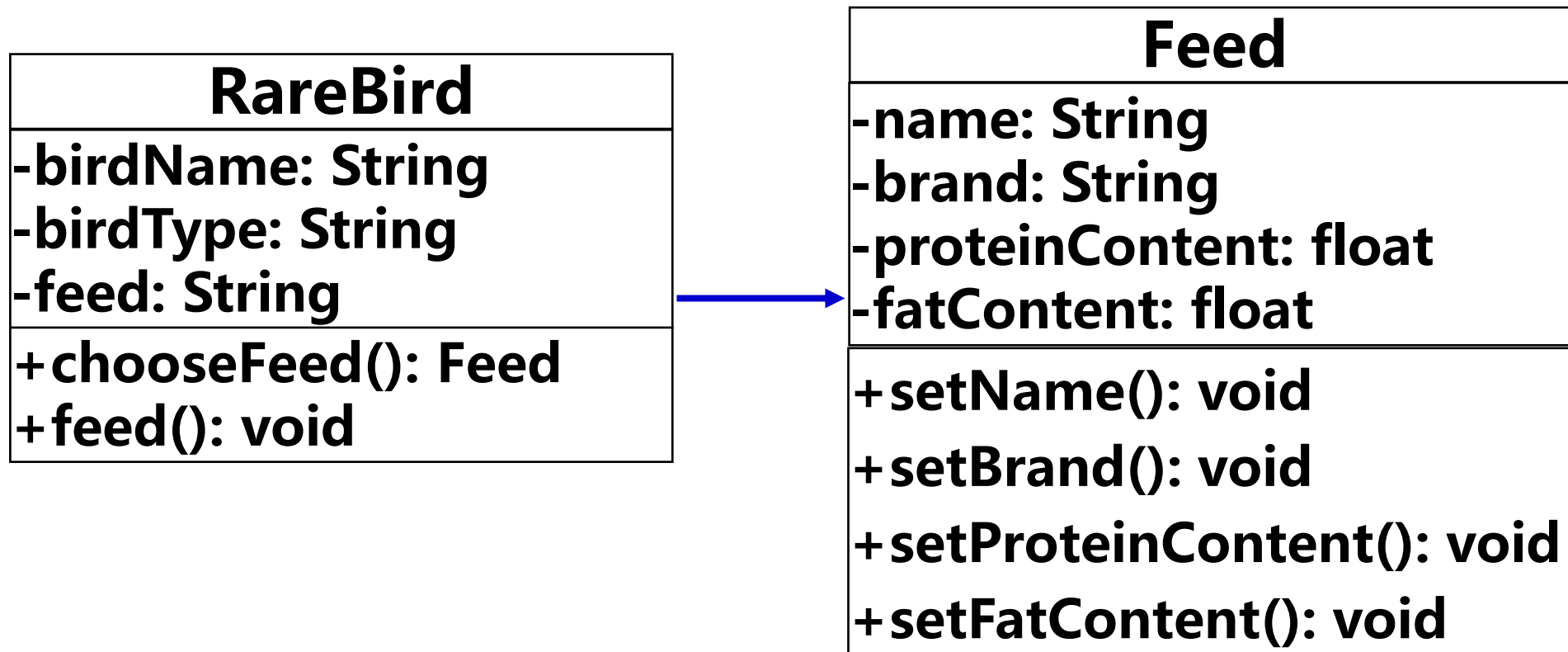
多态模式

6. Polymorphism Pattern

- **背景 Background:**
- **如果一个程序使用了条件语句，则当有新的条件被加入的话，则需要修改原来的条件语句组的逻辑**
- **缺点：程序不容易扩展。原因是如果程序有了新变化的时候，则可能会涉及到多处的修改。**

- **问题: 如何处理基于类型变化的解决方案?**
- **解决方案: 使用多态模式。** 当相关的行为基于类型而变化, 使用多态操作, 将行为责任分配给相关的类型。
- **Polymorphism Pattern:** When related alternatives or behaviors vary by type (class), assign responsibility for the behavior-using polymorphic operations-to the types for which the behavior varies.

例11. 珍稀驯化养殖场管理子系统设计片段



珍稀驯化养殖场管理子系统设计片段

- RareBird类里面的chooseFeed 方法与 birdType相关, 所以代码包含了许多类型相关的条件语句

if (birdType is Ostrich)

choose ostrich feed

else if (birdType is turkey)

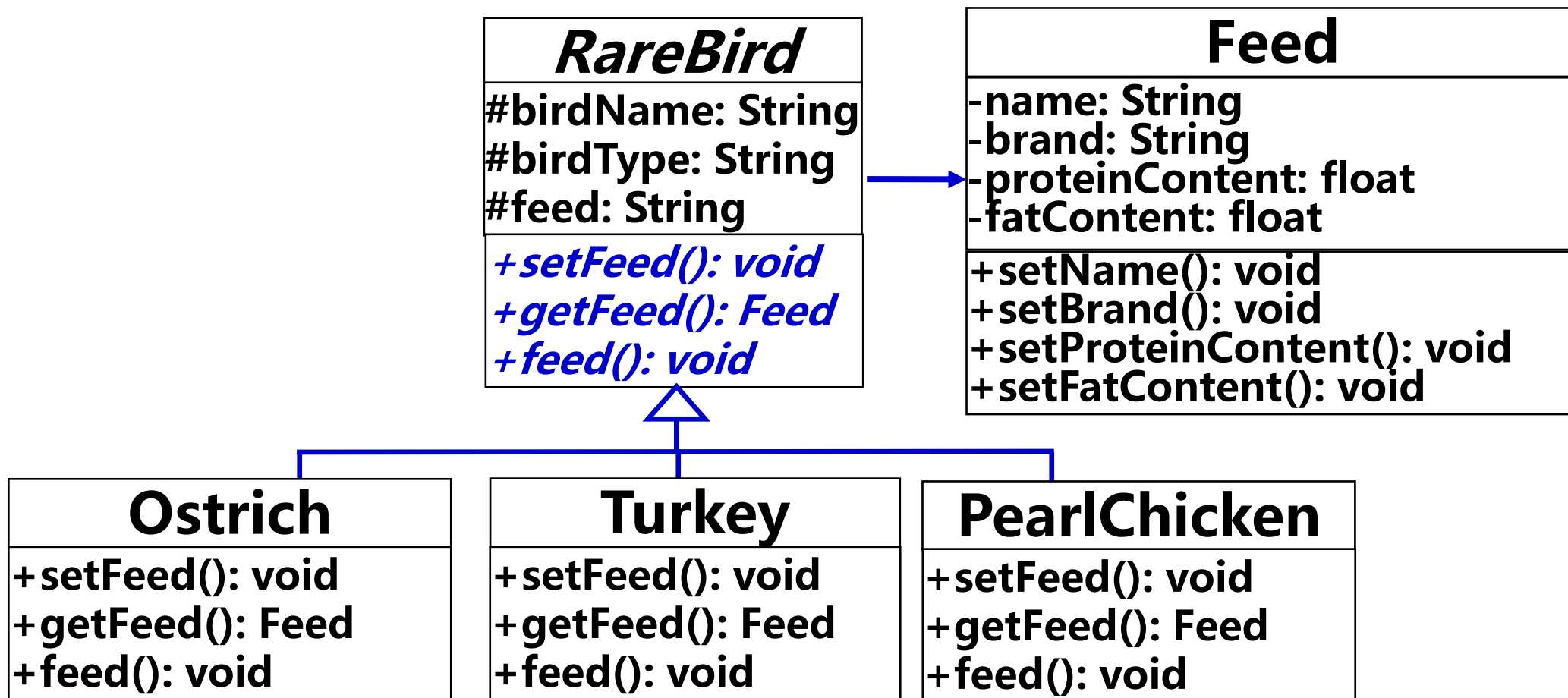
choose turkey feed

else if (birdType is Pearl Chicken)

choose Pearl Chicken feed

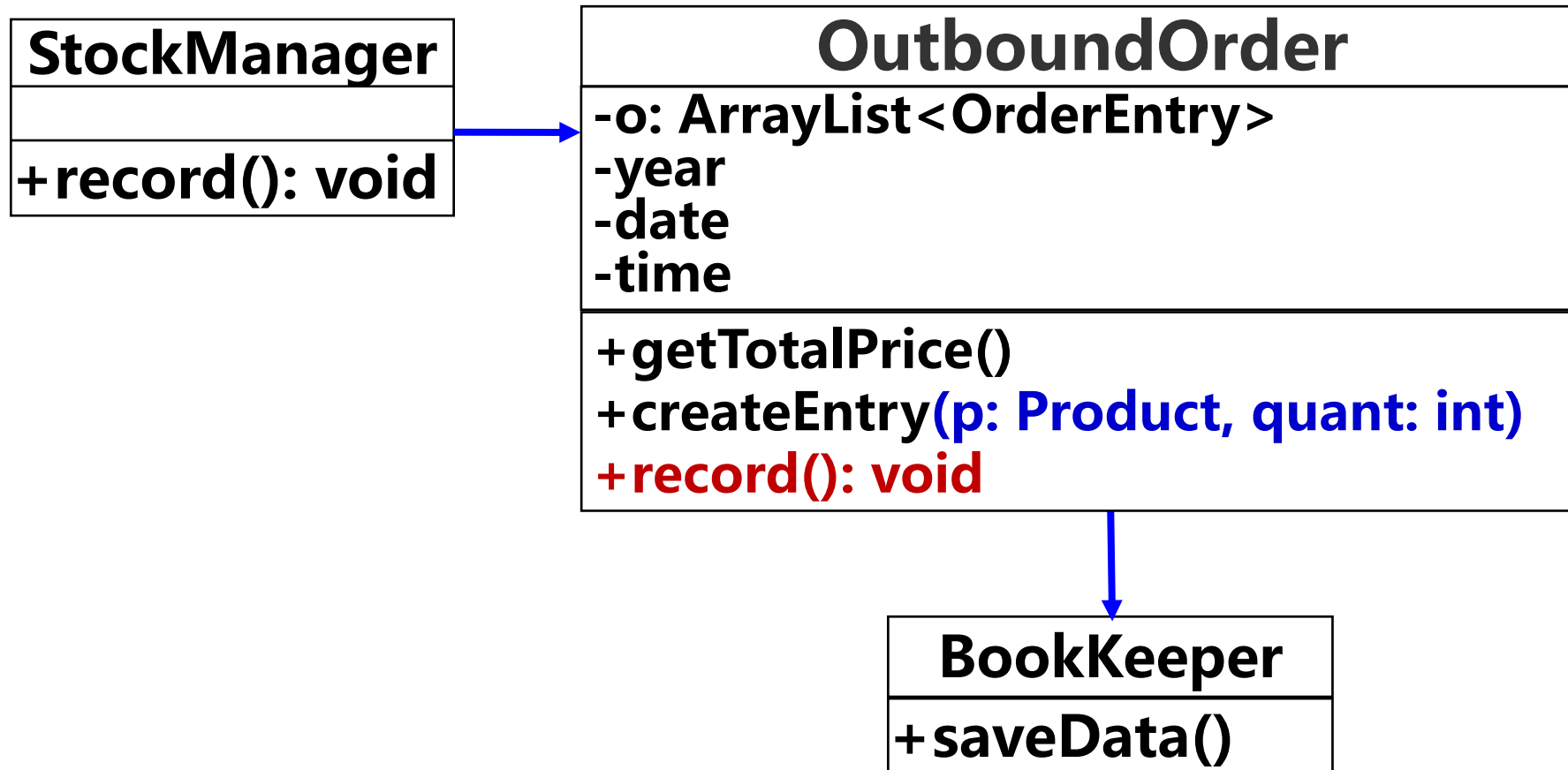
- **缺点:** 可扩展性不好: 如果要增加一种新的珍禽, 则需要 chooseFeed()方法中增加一个新的与类型相关的条件语句
- **解决方案:** 使用多态模式重新设计

- 改善设计如下



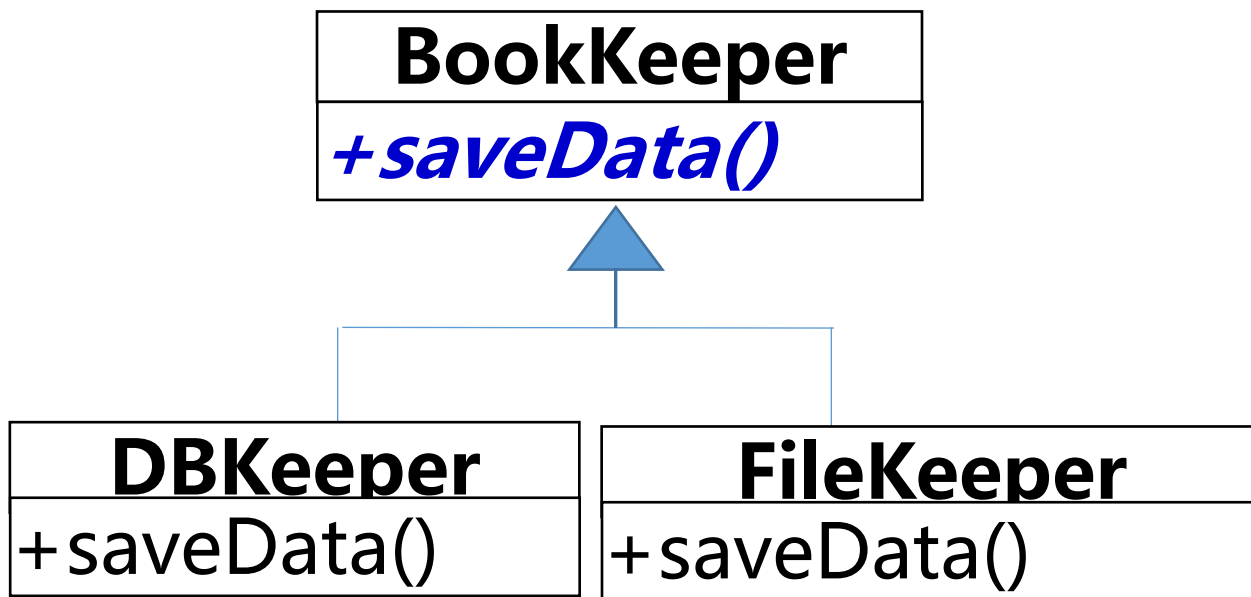
优点：新设计中，Ostrich类中的setFeed()方法不会包含条件语句，而是直接设定一种鸵鸟饲料。

例12. 出库单的例子（继续）

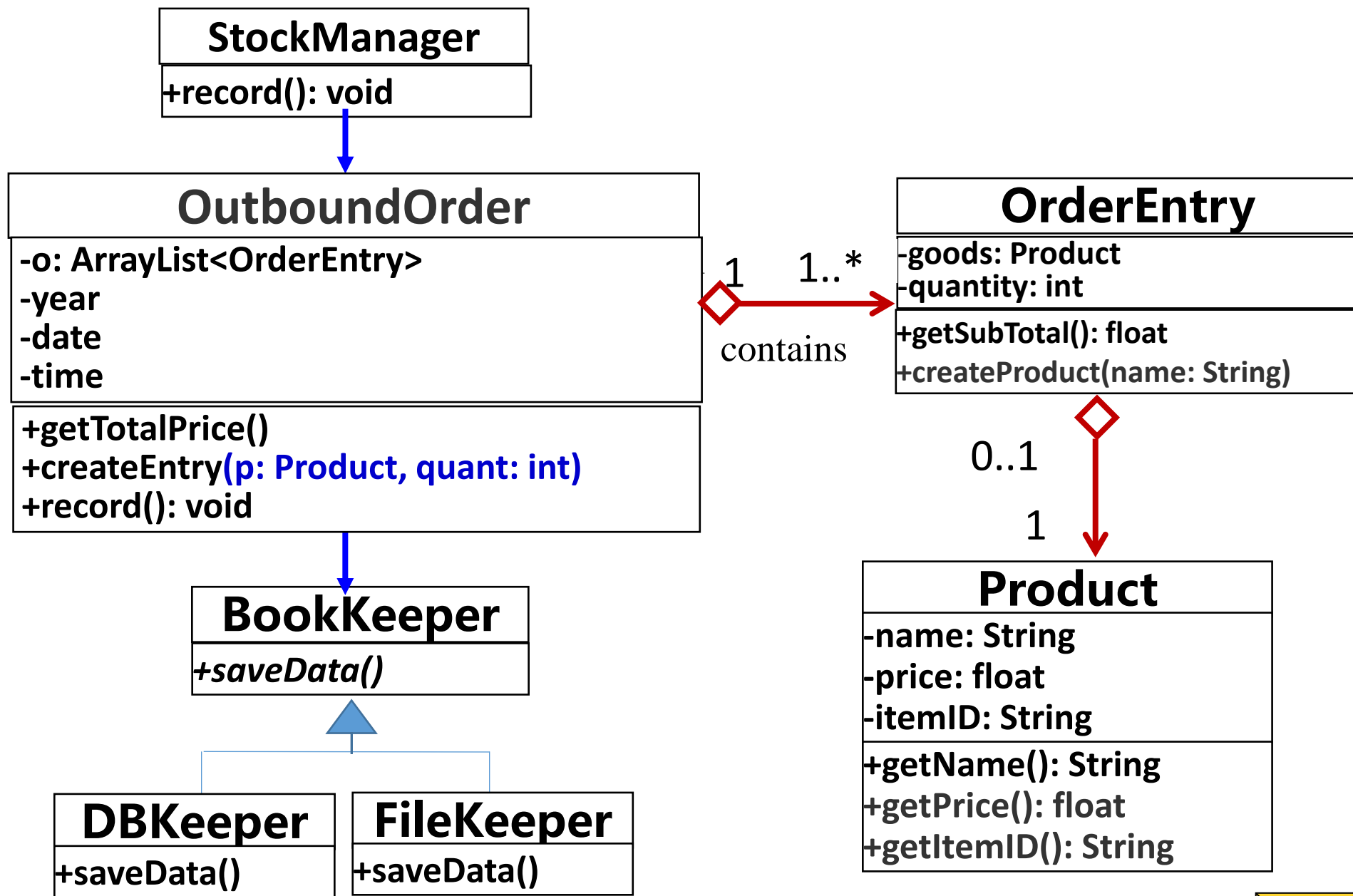


想法：如果能让BookKeeper类的saveData()方法除了将数据写入数据库以外，还要将数据写入数据文件。saveData()方法将会出现条件语句。

根据多态模式，将BookKeeper类改为层次类：



BookKeeper层次类的设计



电子商务出库单领域类图的最新设计

Back

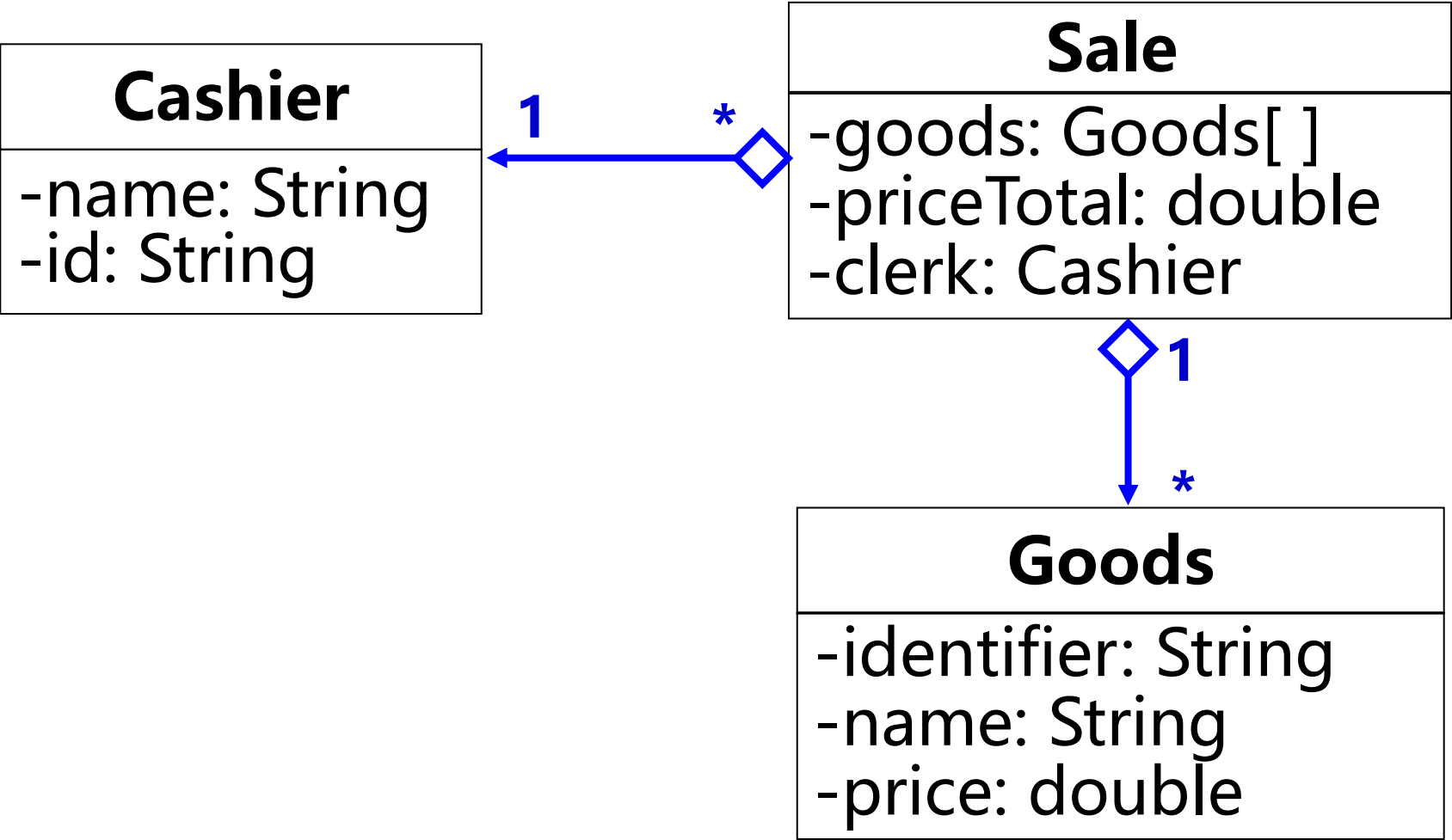
8. Pure Fabrication Pattern

纯虚构模式

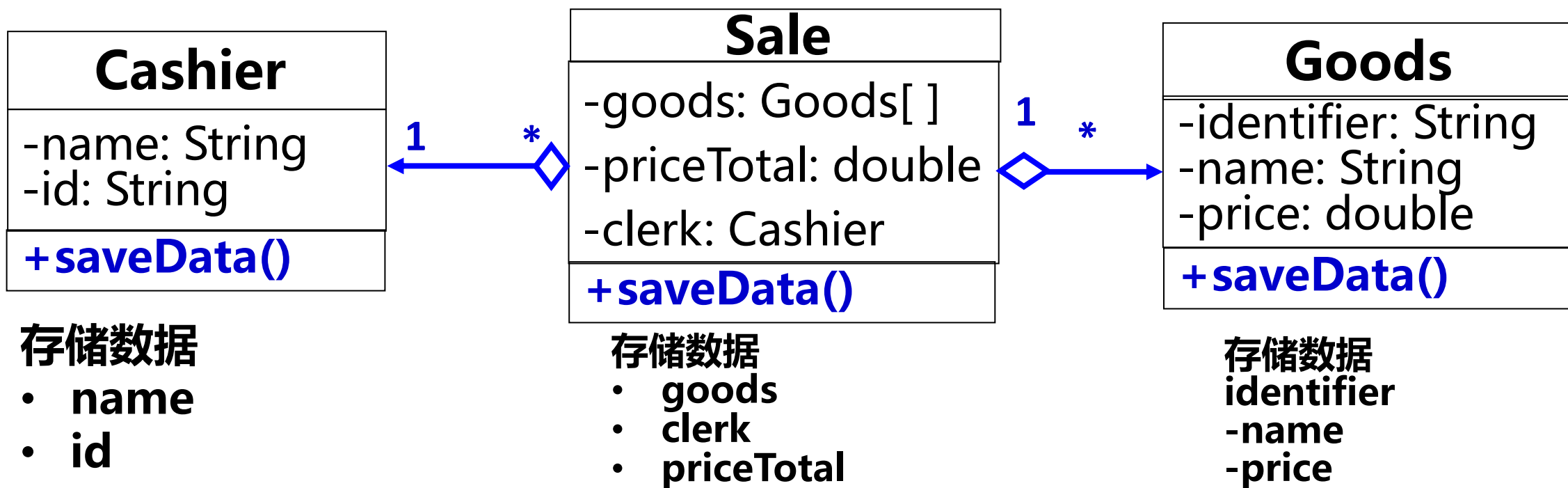
8. Pure Fabrication Pattern

- **问题：专家模式失灵的情况**
- **在OO设计中，有许多情况下，（根据专家模式）将责任分配给域层软件类会导致以下方面的问题**
 - **内聚性或耦合性差，或**
 - **低复用的可能性**

例13. 超市结账系统的设计



- **想法**：将对象Cashier, Sale 和 Goods 中所包含的数据存入数据库，因此要给每个类设计一个**saveData()**方法
- **使用专家模式**. 这样的操作应该分别包含在 Cashier, Sale 和 Goods 类之中，既然这些类里面有需要存储的永久数据。



- **设计缺点：导致较低的内聚**

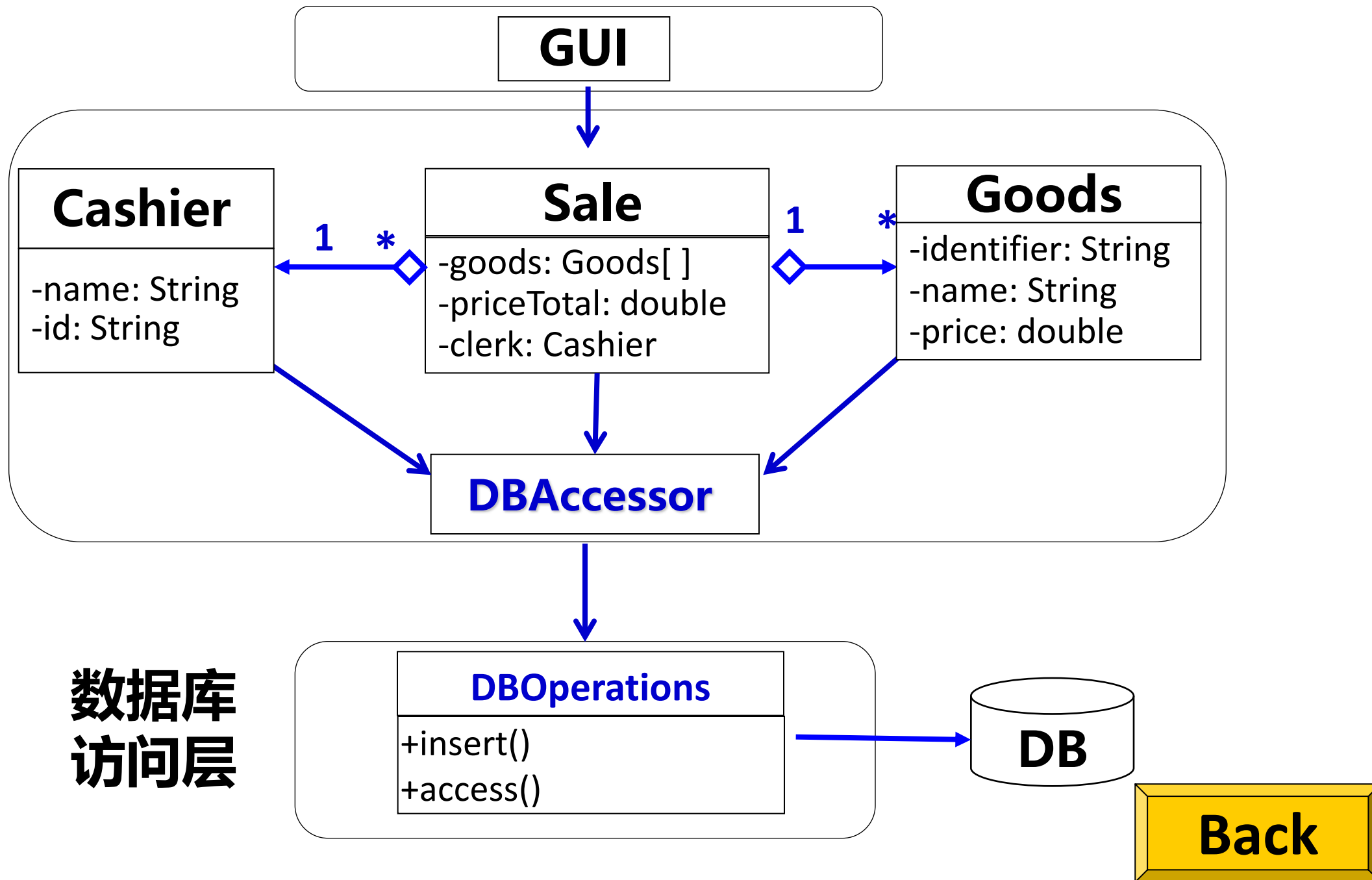
- **这种设计不好：**

- **数据库操作方法与这3个类没有概念上的关系，**
- **如果数据库操作包含在3个类中，那么您将得到一个低内聚的设计(导致低内聚的设计)；**
- **这样做将生成一个与数据库接口（例如JDBC）紧密耦合的类图**

- **尴尬局面：**

- **一方面，按照信息专家模式，三个领域类都应该拥有数据库访问的职责，而**
- **另一方面，如果这样做，会导致低内聚，高耦合与可重用性差的设计。**

- 纯虚构模式的定义：将一组高内聚的责任分配给一个不代表领域类概念的虚构的类。该虚构的类的存在支持高内聚，低耦合与复用
- **Pure Fabrication pattern:** Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept-
 - something made up, to support
 - high cohesion,
 - low coupling, and
 - reuse.
- Such a class is *a fabrication* of the imagination (这样的类叫做纯虚构类).
- Ideally, the responsibilities assigned to this fabrication support high cohesion and low coupling, so that the design of the fabrication is very clean, *or pure*—hence a pure fabrication.



9. Indirection Pattern

间接模式

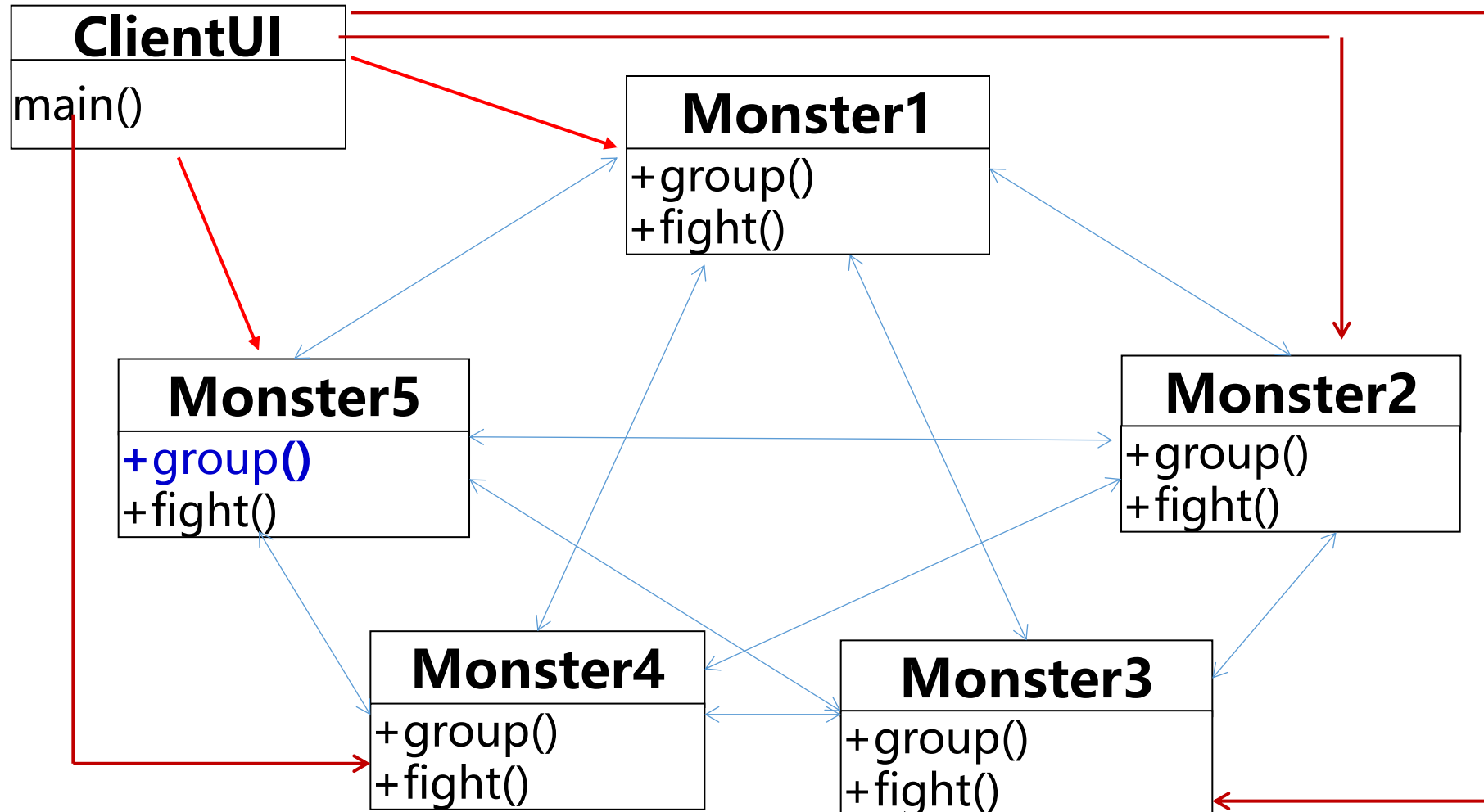
9. Indirection Pattern

Problem:

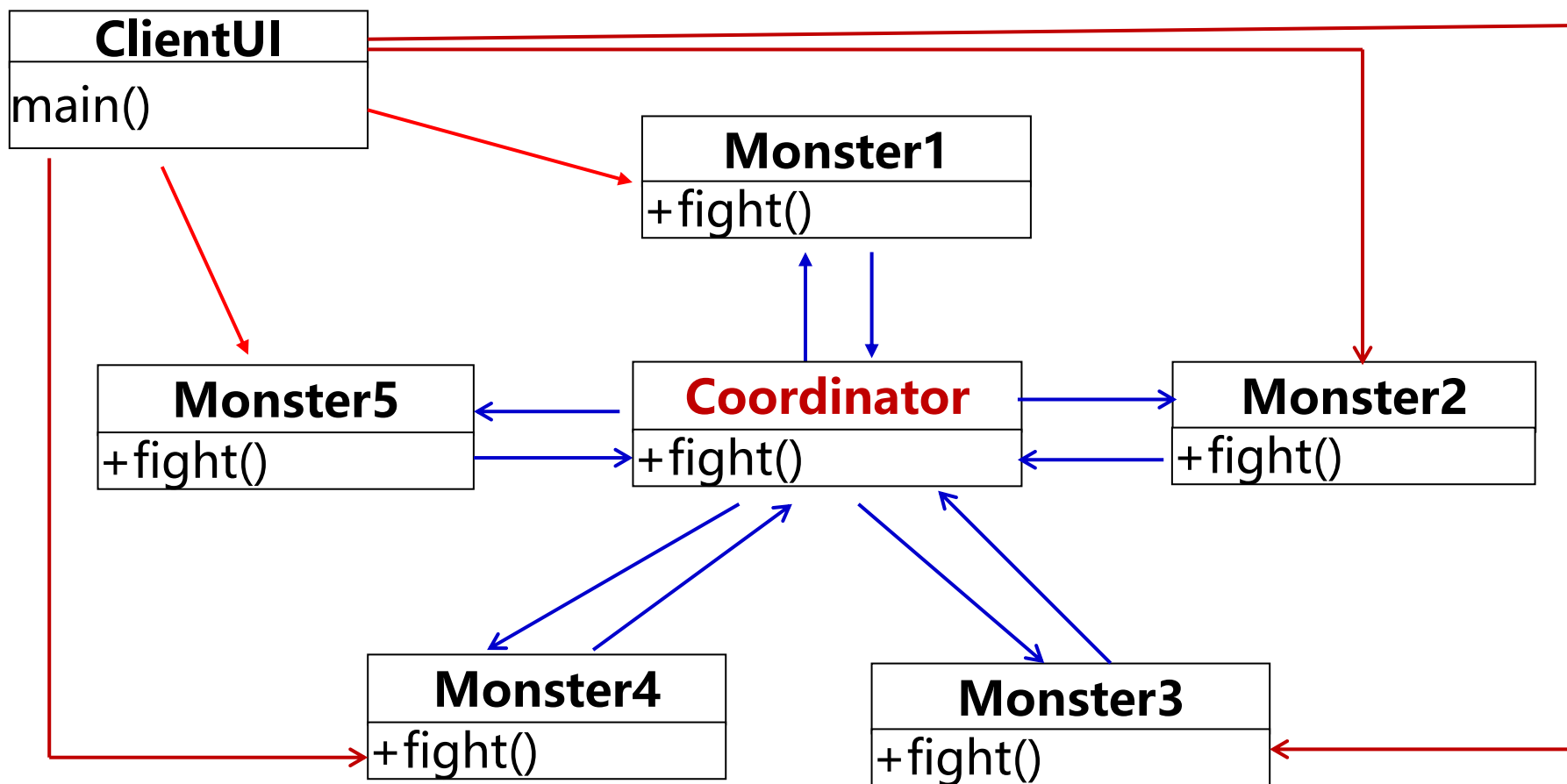
- **用什么办法能避免多个类的直接耦合**. Where to assign a responsibility, to avoid direct coupling between two (or more) things?
- **怎样给高耦合的类解耦**. How to de-couple objects so that low coupling is supported and reuse potential remains higher?

- **间接模式：将责任分配给中介对象，以便在其它组件之间进行协调，以使它们不直接耦合。**
- **Indirection pattern:** Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.
- **The intermediary creates an *indirection* between the other components.**

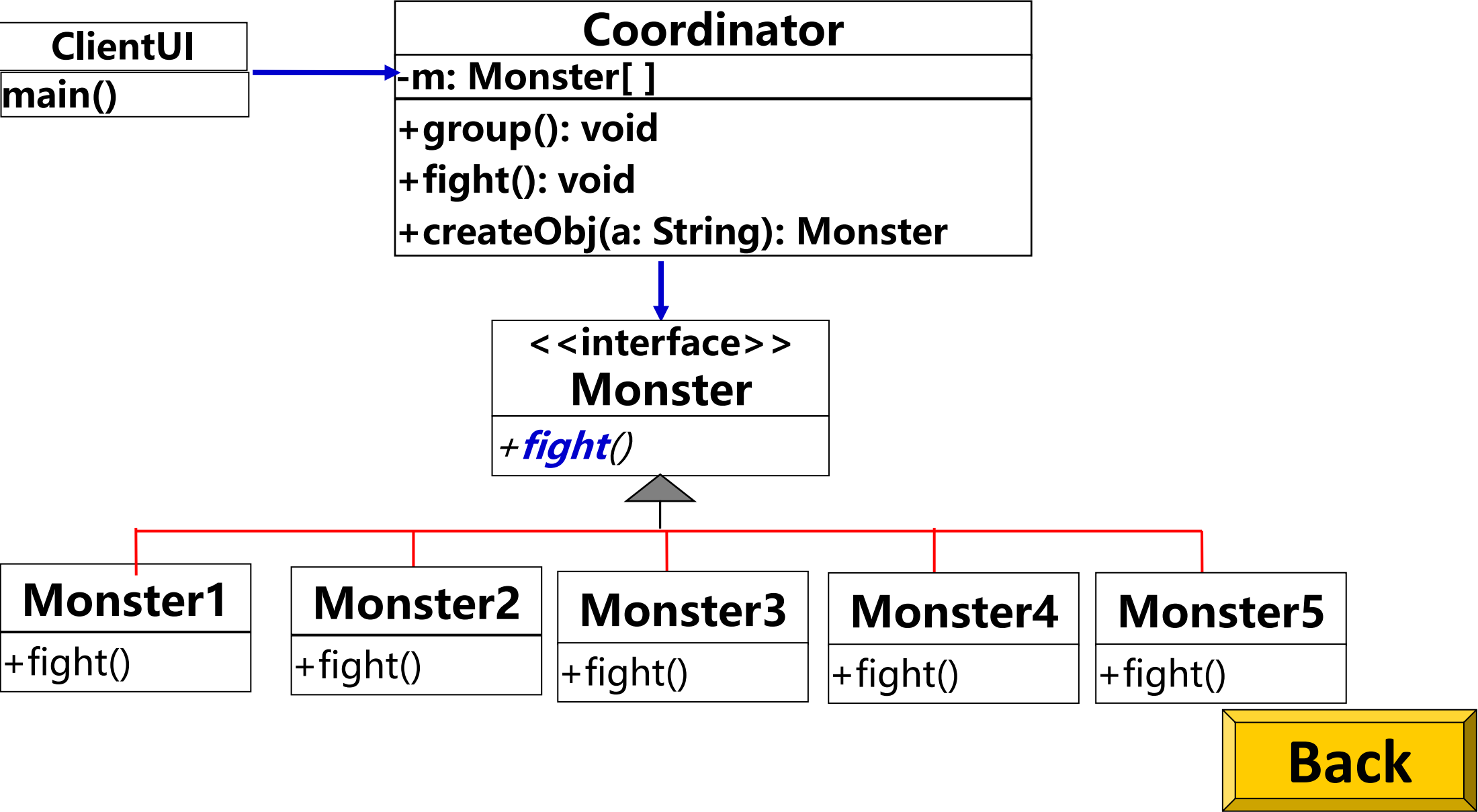
例14. 假如一个格斗类游戏软件的面向对象设计如图所示，其中，包含了5种类型的Monster类，分别为Monster1， Monster2， Monster3， Monster4， Monster5



- **问题：**五个Monster对象之间互相两两存在高度耦合。怎样改善设计以便取消高度的耦合呢？
- **解决方案：**可引入一个间接类，称为Coordinator。利用该类取消所有Monster类之间的互相调用. Coordinator对象负责与Monster们交互。



将Monster类设计为层次类之后，类图如下。



10. Protected Variations

受保护模式

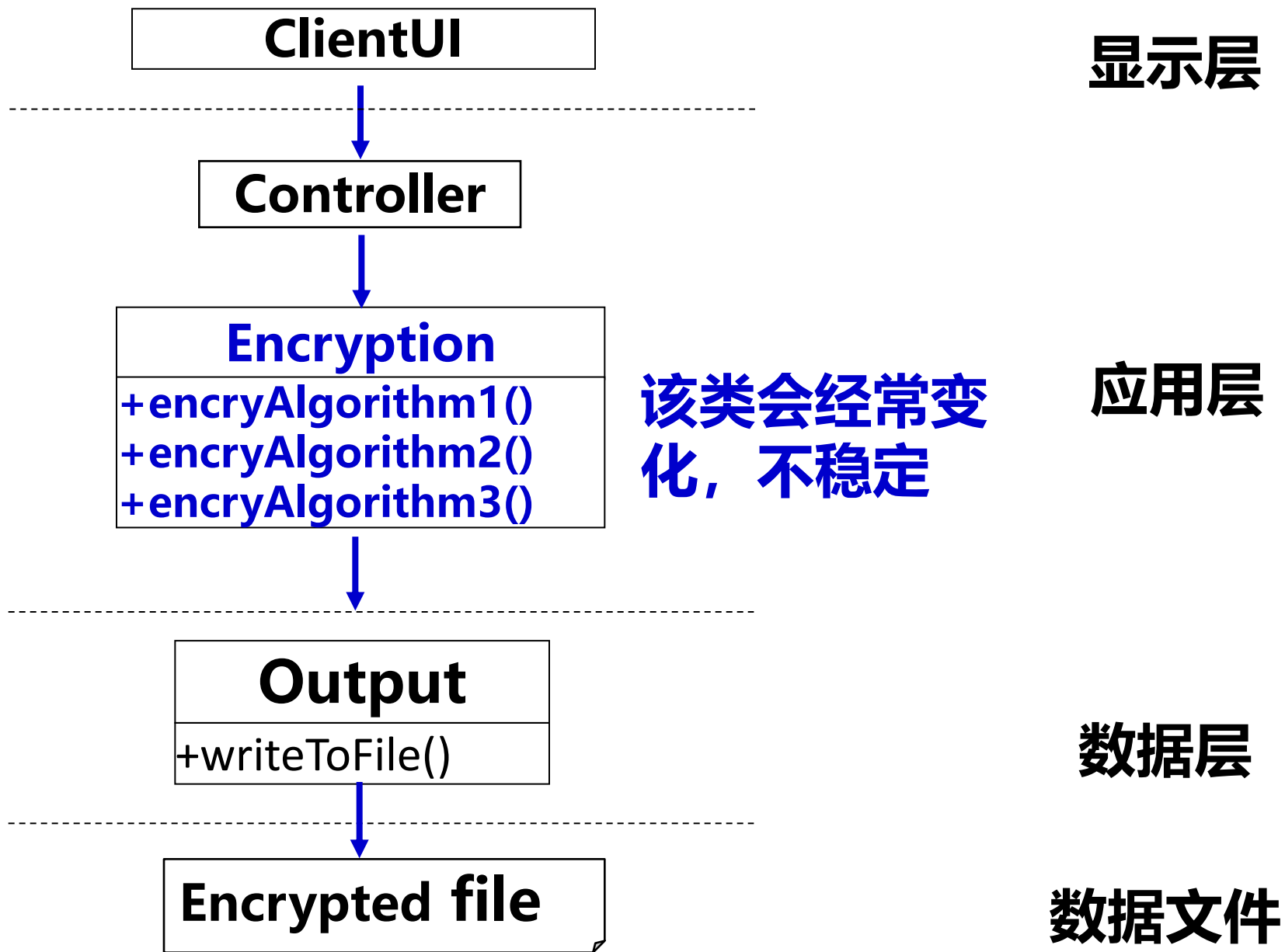
10. Protected Variations

- 问题：如何设计对象、子系统和系统，以使这些元素中的变化或不稳定对其他元素不会产生不良影响？
- **Problem:** How to design objects, subsystems, and systems so that the **variations or instability** in these elements does not have an undesirable impact on other elements?
- **受保护变化模式的定义**
- 确定能预测到的（类型）变化或不稳定点；分配责任以建立一个稳定的接口
- **Protected Variations pattern:** Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them

GRASP：受保护变化

- 识别可预知的变化或不稳定点；通过分配职责来创建围绕它们的稳定接口
 - PV是大多数编程和设计的机制和模式的基本动机之一，它使得系统能适应和隔离变化
 - PV与开放-封闭原则思想是一样的，只是侧重点稍异
- 优点
 - 低耦合

例15. 一位教授要求你设计一个用于测试加密算法的软件平台，以便研究生们可以使用这个平台来测试他们开发的加密算法。程序片段设计如下。



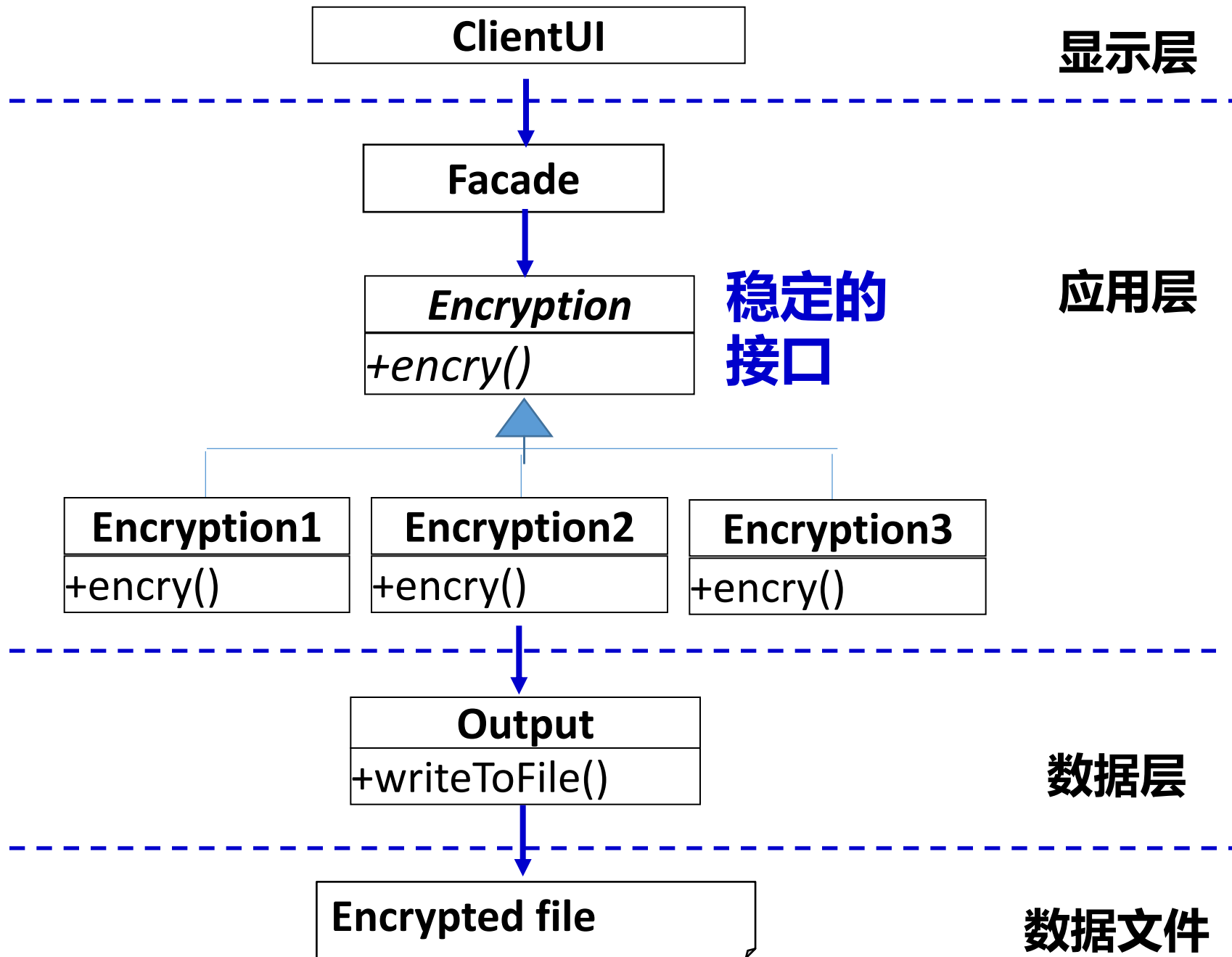
文字加密软件平台软件体系结构图-单独Encryption类的情况

- **该设计的缺点：**

- 修改某一个算法，需要重新编译整个类
- 添加某一个算法，需要重新编译整个类

- **重新设计：**

- 因此，Encryption类是不稳定的。为了改善可扩展性，按照受保护变化模式的原则，只需将该类进行重构，
 - 引入接口超类Encryption，在其中声明一个抽象方法encrypt()，然后按照加密算法设置三个类Encrypt1, Encrypt2和Encrypt3。
- 每个实现类都按照相应的算法写encrypt()方法的代码。设计类图如下图所示。



- **新设计的优点：**

- **容易维护.** Whenever you need to modify an algorithm, then you only need to modify a subclass of Encryption
- **容易扩展.** When you need to add a new algorithm, you only need to add a new subclass to the Encryption class hierarchy
- **本设计符合开闭原则.** This design follows the open closed principle



Back

11. Law of Demeter

Demeter定律

- 将期望的行为分配到各个类中的标准
- 对于给定的我们所需要的行为，我们必须决定将其放到哪个类之中呢。
- 标准 (Criteria):
 - **重用性 (Reusability)**: 此行为是否可重用
 - **复杂性 (Complexity)**: 实现此行为的复杂度(将一个方法放到一个类中可能比放在另外一个类中更容易实现)?
 - **可应用性 (Applicability)**: 此行为和将要把这个行为放置进的这个类 (型) 的相关度如何?
 - **实现知识 (Implementation knowledge)**: 该行为实现是否依赖于某个类(型)的内部数据细节?

- **Demeter定律：**类的方法不应该依赖于其它的类的结构，而只能依赖于自己所在的类或者其最靠近的超类。(Demeter是一个项目的名称；该定律有很多引申与解释，例如：不要和陌生人讲话)
- **Demeter Law:** One useful guideline in choosing the relationships among objects is called the Law of Demeter, which states that **the methods of a class should not depend in any way on the structure of any class, except the immediate (top-level) structure of their own class.**

例16: 银行账户类

Account
<ul style="list-style-type: none">-name: String-id: string-acctNum: String-balance: float
<ul style="list-style-type: none">+getName(): String+getId(): String+getAcctNum(): String+getBalance(): float+setName(n: String): void+setId(i: String)+setAcctNum(a: String): void+setBalance(b: int): void+deposit(a: float): void+withdraw(a: float): void+transfer(a: float): void

Account
类的结构

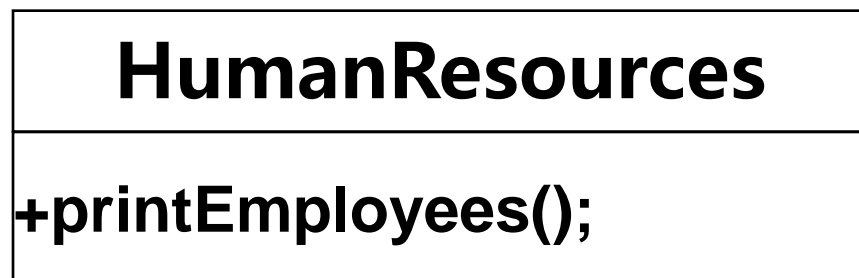
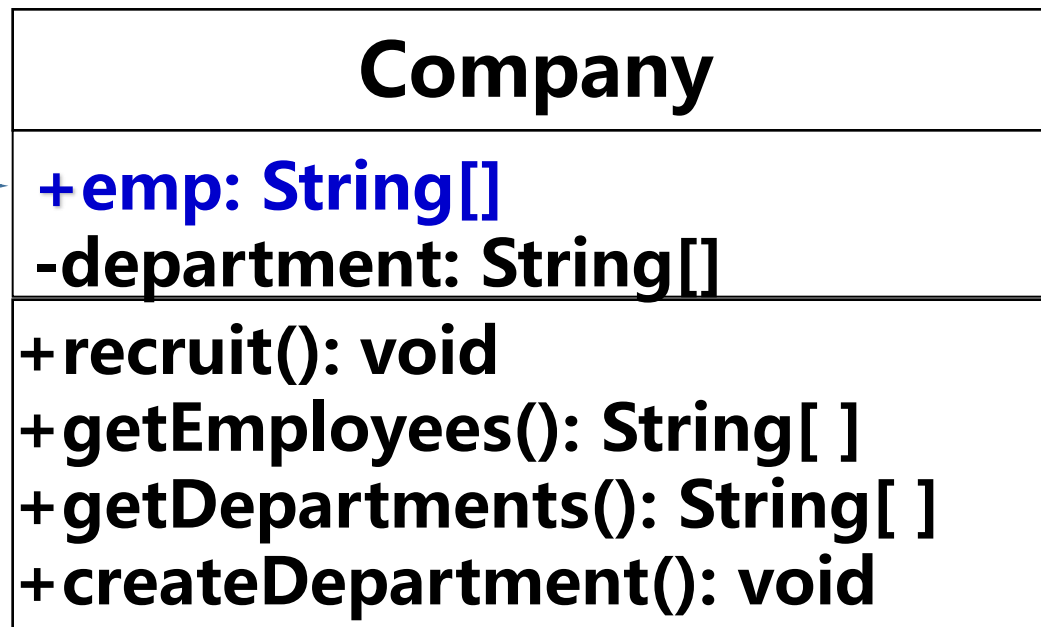
```
Public void deposit(amt: int){  
    balance = balance + amt;  
}
```

deposit (a)
withdraw(a)方法
依赖于balance

例17: 某个类的方法依赖于其它的类的结构 的例子

原始设计

全局
变量



```
public void printEmployees(){  
    for(int k=0; k<Company.emp.length(); k++){  
        System.out.print(Company.emp[k]);  
    }  
}
```

缺点: 1) 违反了面向对象的封装原则

2) **HumanResources类的方法**
依赖于Company类的结构:
emp; 违反了Demeter定律

重新设计1

HumanResources
+ printEmployees();

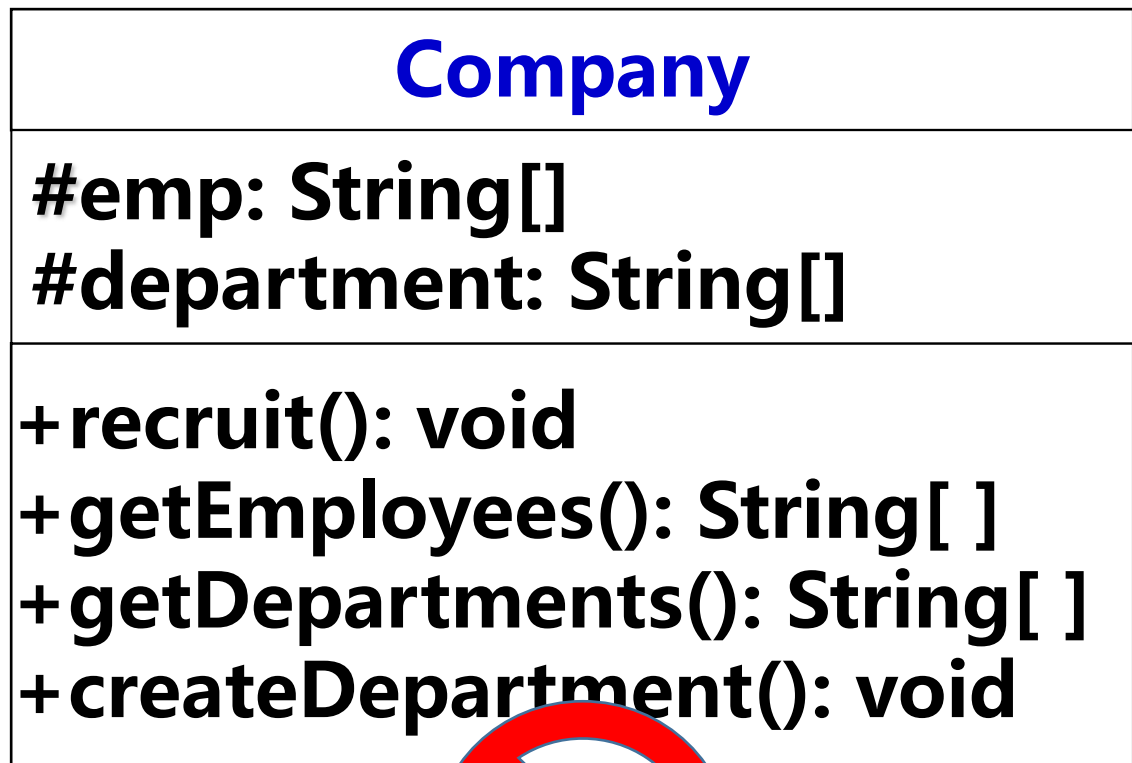
Company
-emp: String[] -department: String[]
+recruit(): void + getEmployees(): String[] +getDepartments(): String[] +createDepartment(): void

```
public void printEmployees(){  
    Company c =new Company();  
    String[] emp=c.getEmployees();  
    for(int k=0; k<emp.length(); k++){  
        System.out.print(emp[k]);  
    }  
}
```

HumanResources类的
printEmployees()方法中，
创建了Company类的对象，
然后，调用getEmployees()
方法，合法获得员工名单emp

此设计仍然不够好，仍
然违反Demeter定律

重新设计2

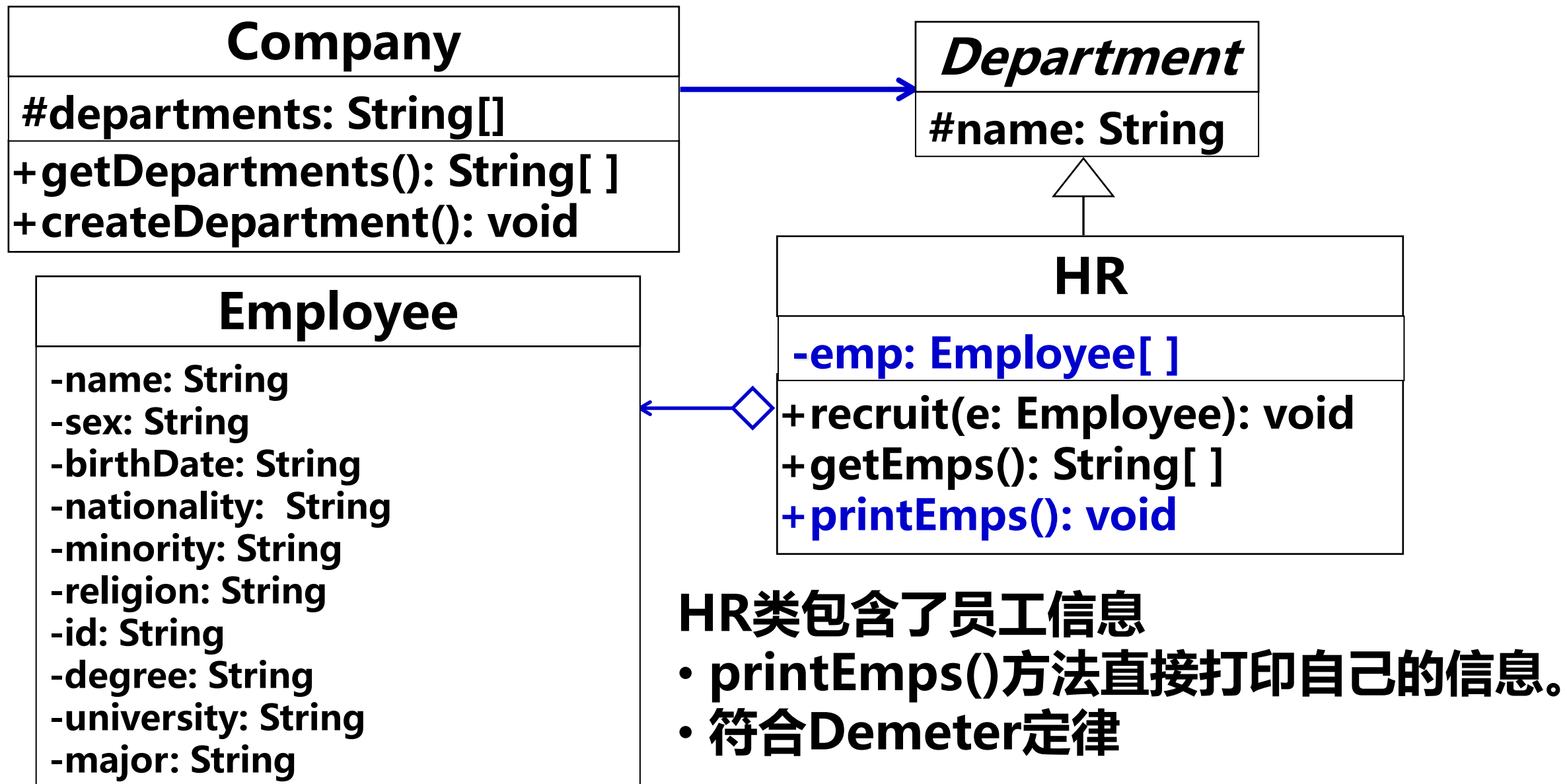


可以直接使用
超类中的emp,
将其内容打印
出来。



此设计不符合分
类法!
违反了继承的is-
a关系

重新设计3



- **应用Demeter定律的基本效果是可以创建松散耦合的类，其实现秘密被封装了；因此，为了理解一个类的意图，不必理解许多其它类的细节。**
- The basic effect of applying this law is the creation of loosely coupled classes, whose implementation secrets are encapsulated.
 - So that to understand the meaning of one class, you need not understand the details of many other classes






Back

课堂作业

- **需求描述：**为大型停车场创建一个信息显示系统
- 这个系统仅有一项功能：将停车位信息显示到屏幕
 - 需要系统能够显示每层停车场的车位占用情况
 - 哪个车位已经被占用？
 - 哪个车位是空位？
- 停车场示意图如下图所示。
 - 假设停车场的每一层都有标注；
 - 每个停车位都有标号。
- **意义：**该停车位显示系统可以使得用户方便地在停车场入口就能看见哪层有空车位，从而能方便地停车；不至于浪费时间逐层地开着车寻找车位。





Parking lot (停车场)

1st floor



P1	P2	P3	P4	P5	P6	P7	P8
							

Parking slot (停车位)

2nd floor

P1	P2	P3	P4	P5	P6	P7	P8
							

3rd floor

P1	P2	P3	P4	P5	P6	P7	P8
							

你的任务： 停车场信息显示系统的领域模型图设计如下。假设现在进入系统分析阶段，需要给领域模型图中的类增加方法。请按照系统功能描述，给各个类增加合适的方法。

