第9章 面向对象设计

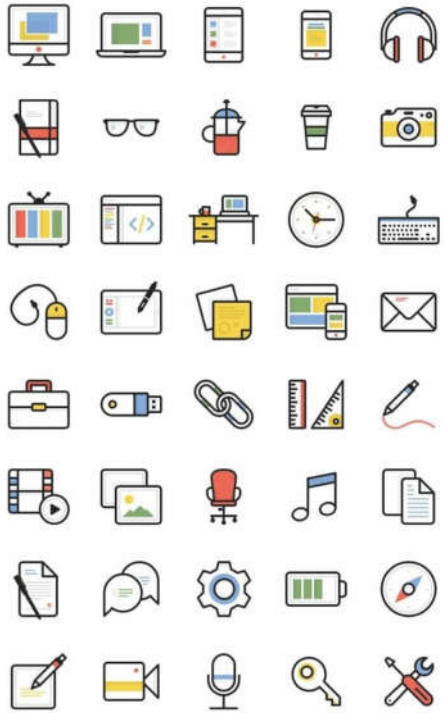
- 面向对象设计概述
- 精化类及类间关系
- 数据设计
- 设计模式简介
- 面向对象的测试

设计模式简介

回顾常用的数据结构：

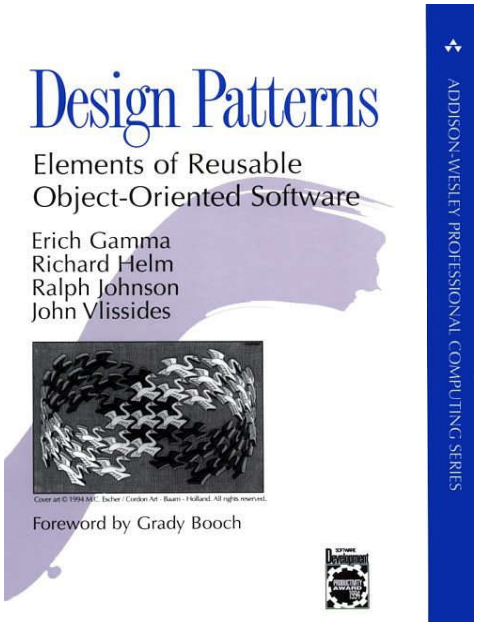
- **Trees，Stacks，Queues，……**
- 它们给软件开发带来了什么？

问题：在软件设计中是否存在一些可重用的解决方案？



设计模式简介

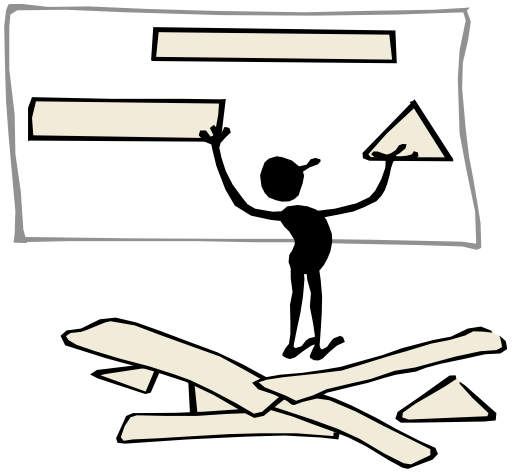
Gamma等四人提出的设计模式具有重大影响



Gamma, Helm, Johnson, Vlissides

设计模式简介

设计模式是指一套经过规范定义的、有针对性的、能被重复应用的解决方案的总结。使用设计模式是为了更有效地重用原有代码，使得代码重用有章可循，增加软件结构和代码的可理解性，增强代码的可靠性。



设计模式（**Design Pattern**）为**OOD**在如下两方面提供了可行的解决方式。

- 静态特征：设计所得的模型要尽可能复用原有的类和模型；
- 动态变化：设计所得的模型需要适应将来新的需求。

设计模式简介

设计模式的优势

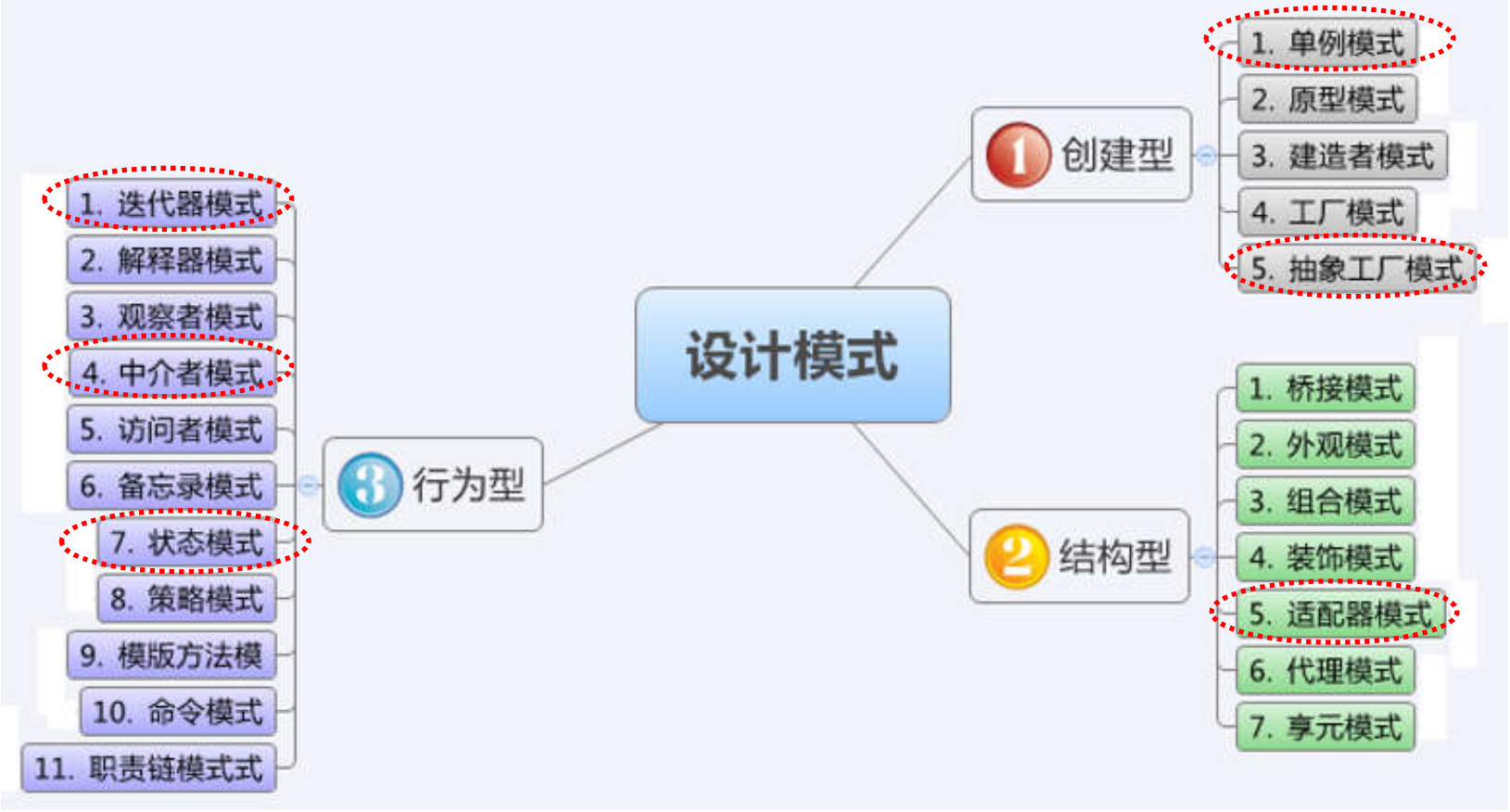
- 模式记录了专家的经验，并让非专家也能理解。
- 模式自身具有各自一套描述，利于开发者更好的交流。
- 模式帮助人们快速理解一个系统——只要这个系统是用该模式来描述。
- 模式帮助系统今后的重组变得容易，具有更好的可靠性和扩展性。

但设计模式绝不是设计的全部

- 误解：过分的关注程序结构的灵活性。这样容易仅从设计的灵活性角度来进行系统设计的重构，并最终可能造成系统的复杂、混乱、不易理解、难以维护，从而导致项目失败。
- 一个好的程序结构，根本不在于其中使用了多少设计模型、多么具有弹性，而主要在于该结构是否能清晰地展现设计者的意图。

设计模式简介

设计模式分类图



设计模式简介

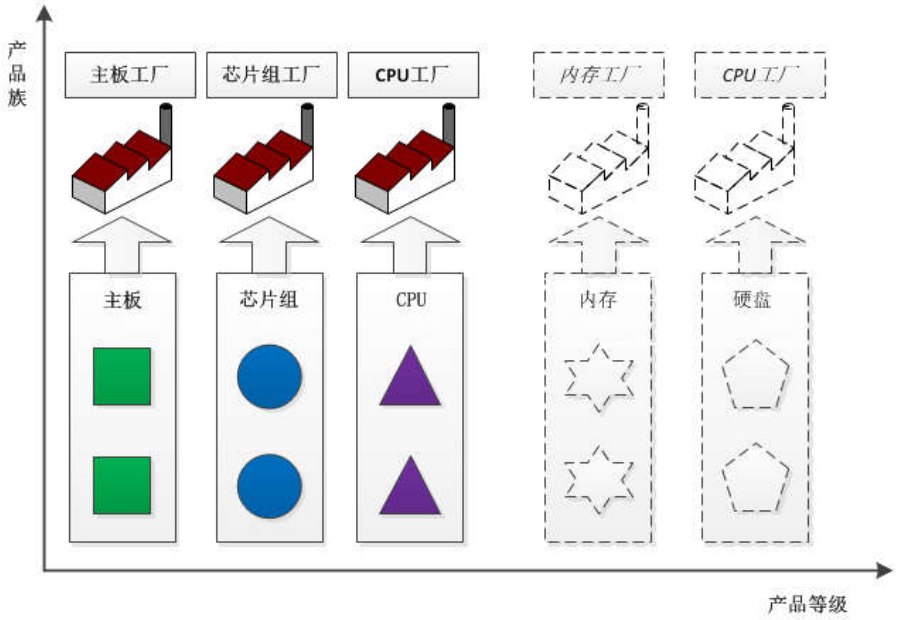
1. Singleton模式

- 在一些应用场景下，有时只需要产生一个系统实例或一个对象实例。
- **Singleton**模式使得系统运行时仅有一个受约束的实例存在，降低系统控制的复杂度，避免由于产生多个对象而造成的混乱。

设计模式要素	说 明
模式名称	**Singleton**
目的	一个类仅提供一个实例，并且该实例贯穿于整个应用系统的生存期。
问题描述	只需要对类实例化出一个对象。
解决方案	为了确保一个类只有一个对象，定义静态成员数据和静态成员函数，以得到控制访问的唯一实例。
参与者	包括一个静态成员数据，它是对该类访问的唯一实例；获取该静态成员数据的静态成员函数，它使得能从外部访问类的唯一实例。
结构	用实例描述的示例图，如图**9-15**所示。

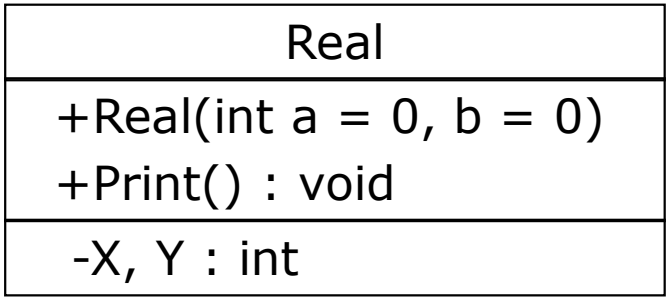
2. Abstract Factory模式

在一些应用场景下，需要用不同的对象操作统一的接口。Abstract Factory模式在增强系统功能扩展灵活性的同时，把对类的修改而造成的对系统的影响降到最低。



设计模式要素	说 明
模式名称	Abstract Factory
目的	提供一个获得不同类的对象的方法。
问题描述	在一个类中能实例化出不同类型的对象。
解决方案	定义类的成员函数，该函数能得到不同类的对象实例。
参与者	抽象工厂类，得到不同类的实例；需生成对象的类及统一的访问接口。
结构	用实例描述的示例图，如图9-16所示。

2. Abstract Factory模式



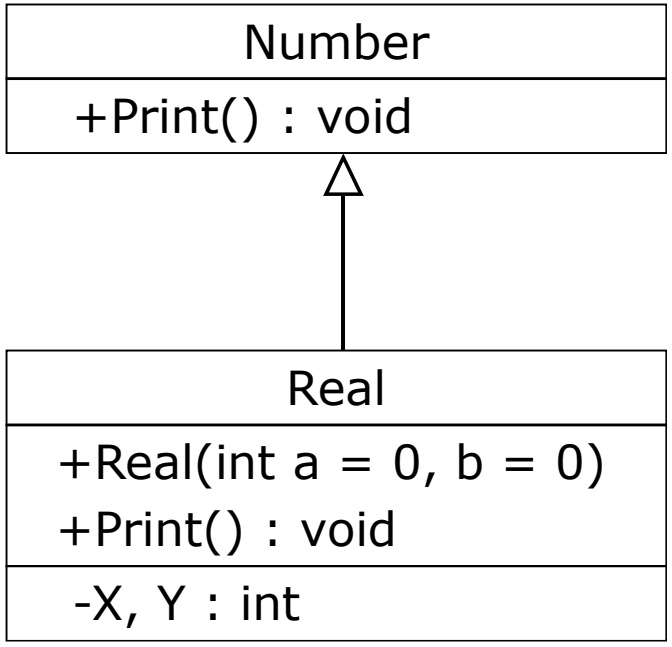

```
void main()
{
        Real  R(2, 5);
        R.Print();
}
```

分析：

main函数直接依赖于类**Real**，如果改为显示其它数据类型，则必须修改两句代码。

2. Abstract Factory模式

使用抽象类



- 改进一

更改main函数代码

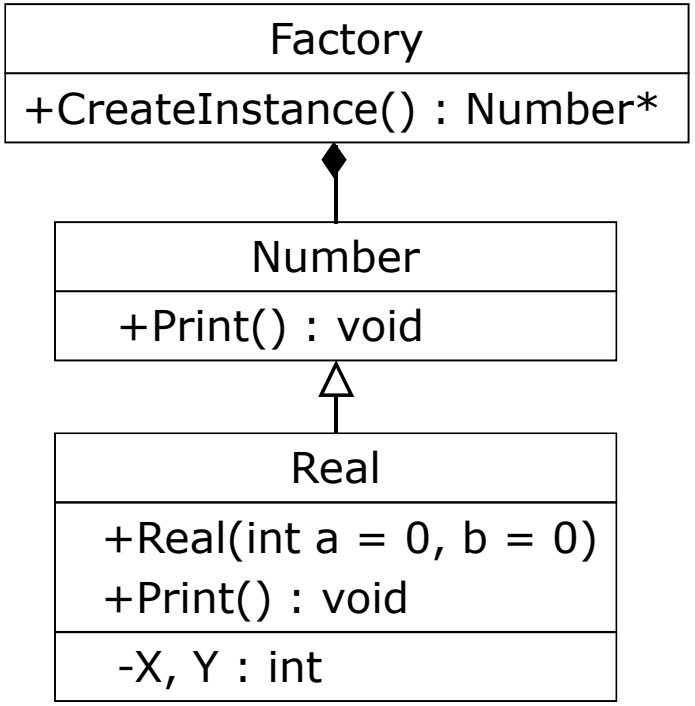
```
void main()
{
        Number *N=new Real(2, 5);
        N->Print();
}
```

分析：

现在只有一句用到了类**Real**，耦合度降低了！但仍然与**Real**有关。

2. Abstract Factory模式

定义数据类工厂



- 改进二

```
void main()
{
    Factory F;
    Number * N = F.CreateInstance();
    N->Print();
}
```

分析：

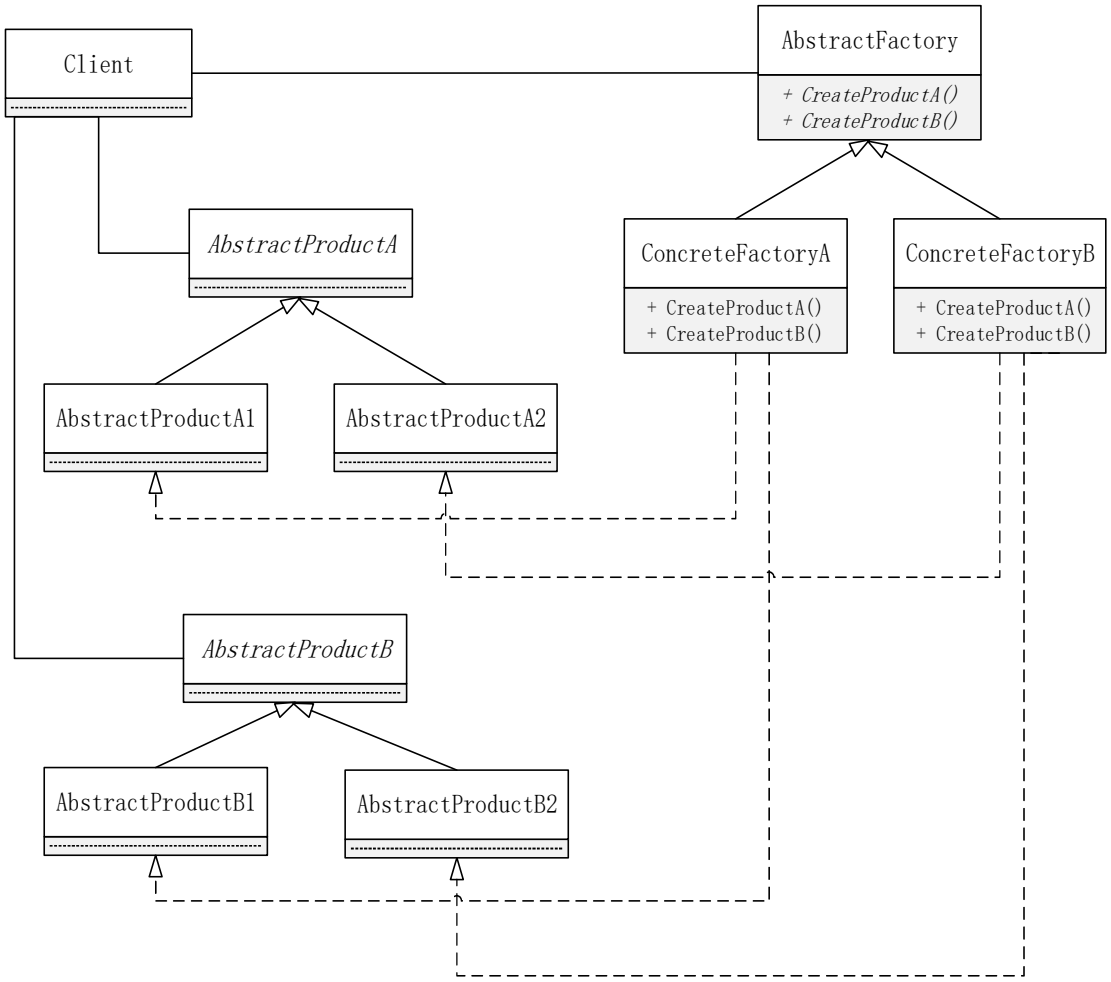
抽象化程度提高，完全与数据类型**Real**无关。

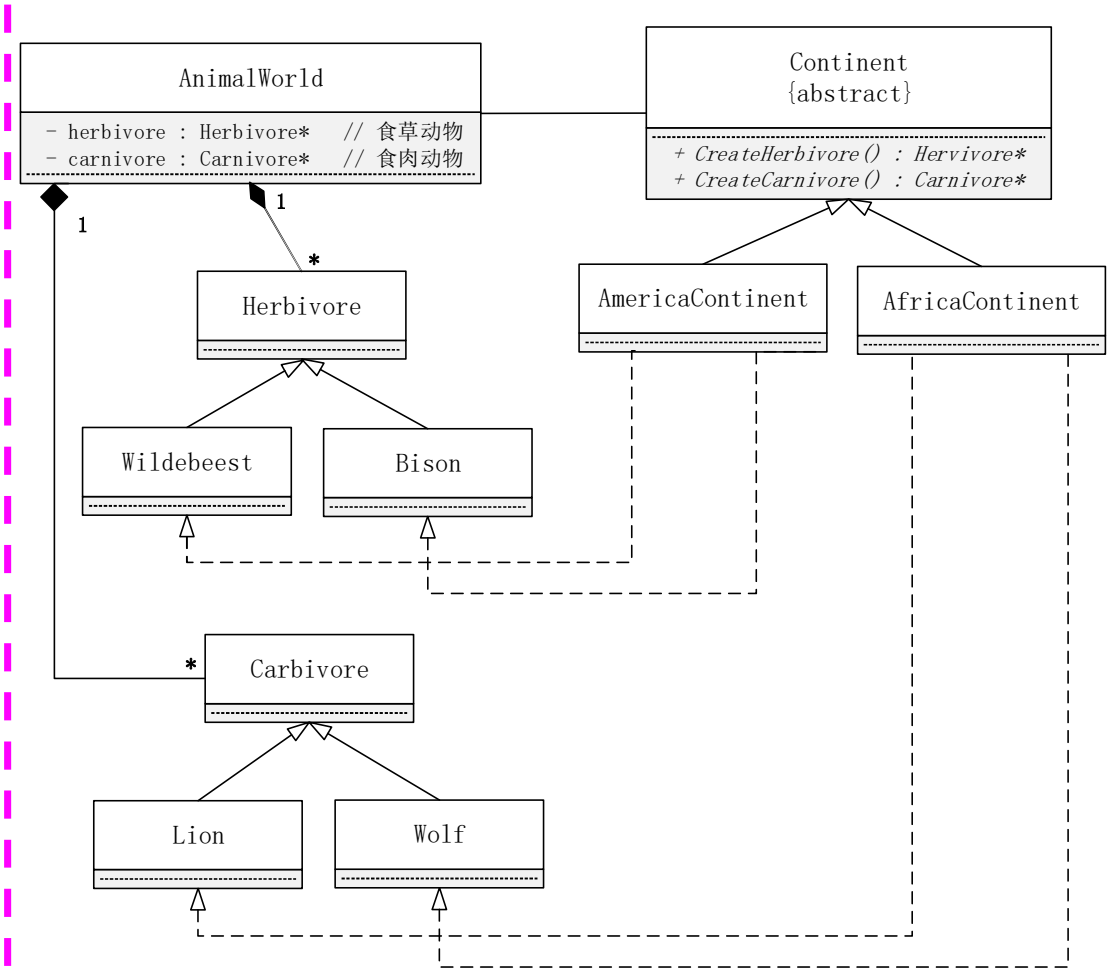
设计模式简介

思考

- 如果再添加一个新数据类型：Complex。如何修改前述程序？
- 过分的抽象、灵活会带来程序设计、理解的复杂性。
- 模式是把双刃剑，要用得适当！

完整的Abstract Factory 模式





3. Mediator模式

类与类之间有相互关系，如果类间的相互关系比较复杂，可以定义中介类专门处理这些关系。这样处理的优势在于：

一 是降低类间的耦合度；

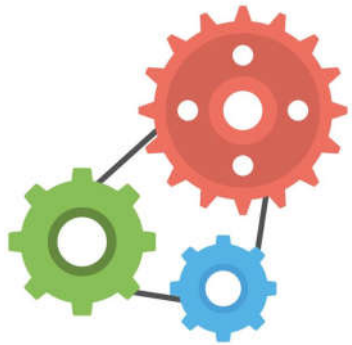
二 是使得类的设计集中于自身功能的实现，以提高类的内聚性；

三 是由于中介类的存在，使得如果类间关系发生改变时，主要的修改发生在中介类，而对类自身的影响降到最小。

设计模式要素	说 明
模式名称	**Mediator**
目的	通过定义中介类的方式处理类间复杂的交互关系，也能消除类间多对多的关联关系。
问题描述	降低类间关联的复杂度，希望能灵活地改变类间的关联。
解决方案	定义中介类，依次降低类间耦合度，提高类的内聚性。
参与者	发生关联的双方（类），以及中介类。
结构	用实例描述的示例图，如图9-17所示。

设计模式简介

3. Mediator模式——“供货商”与“商品”



- 一个供应方可以有多种产品
- 要进行以货易货必须有两个供应方
- 具体的供应商可能有多个

以货易货这一基本流程所涉及的双方是 多对多的关系。

系统一定会扩充：

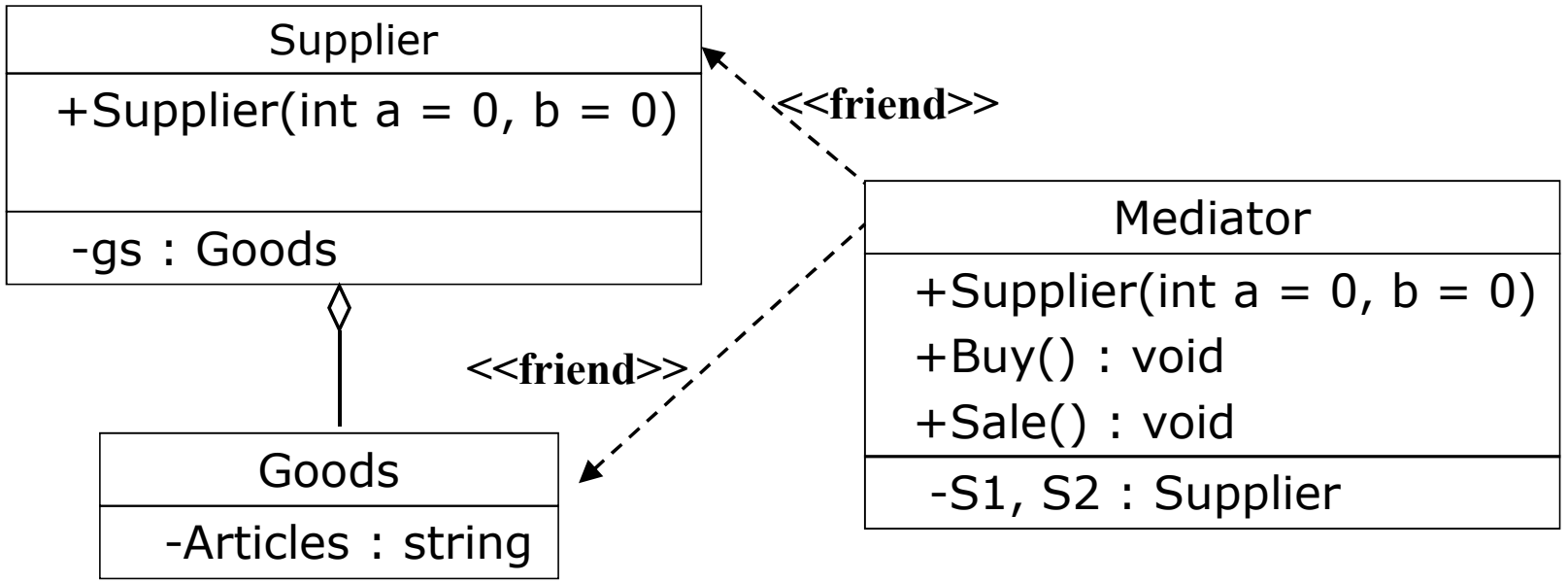
1. 只买，只卖，预订，期货交易等等
2. 每个交易一般都需要满足一定的条件：比如以货易货要求价值相等

系统必须支持功能不断地扩充。

3. Mediator模式——“供货商”与“商品”

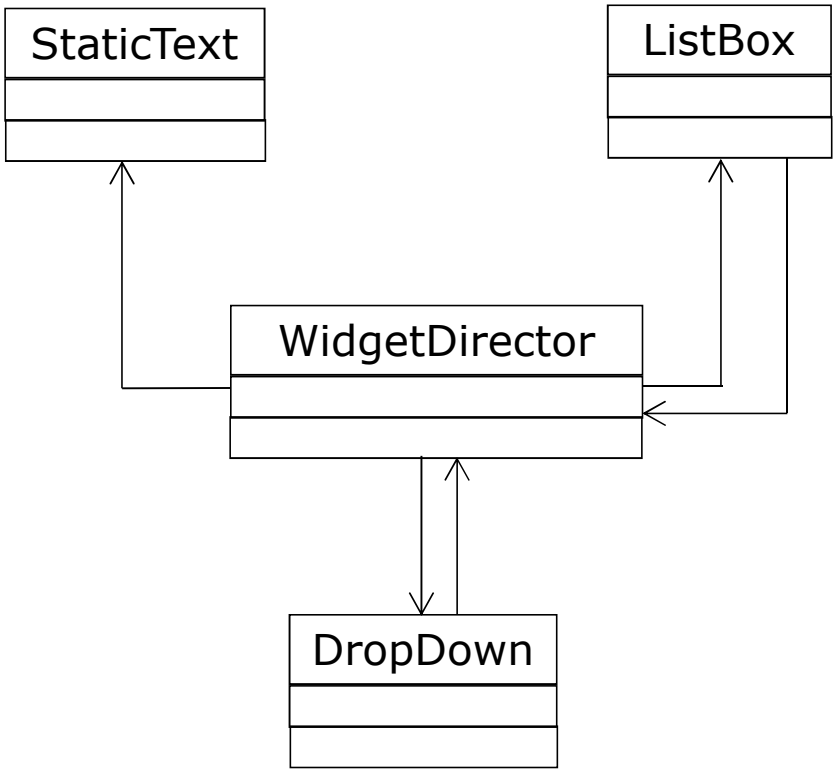
创建一个供应商基类：

- 使用Goods这一数据结构来表达以货易货交易信息
- 利用中介者集中处理交易
- 每个供应商就是这个类的一个实例。



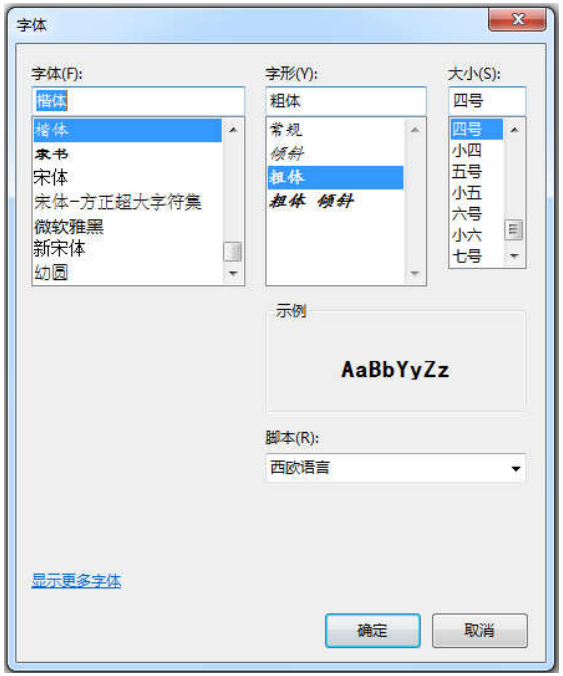
设计模式简介

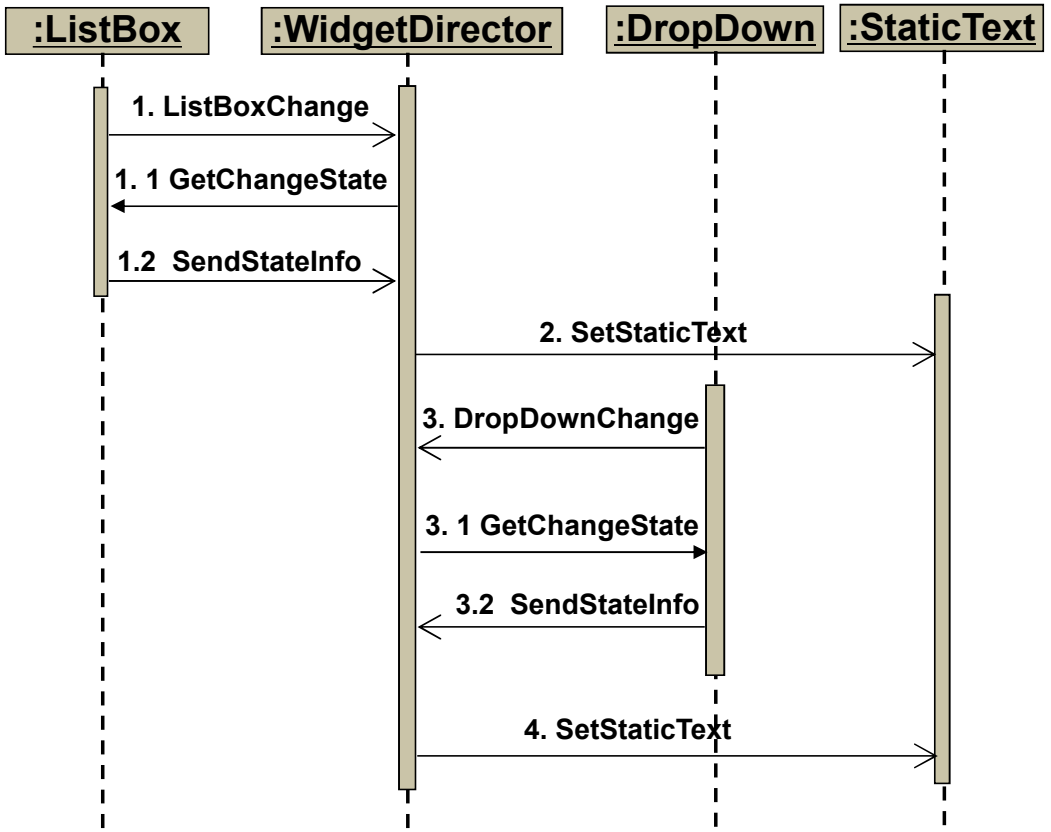
3. Mediator模式——类图



设计模式简介

3. Mediator模式——顺序图





4. Adapter模式

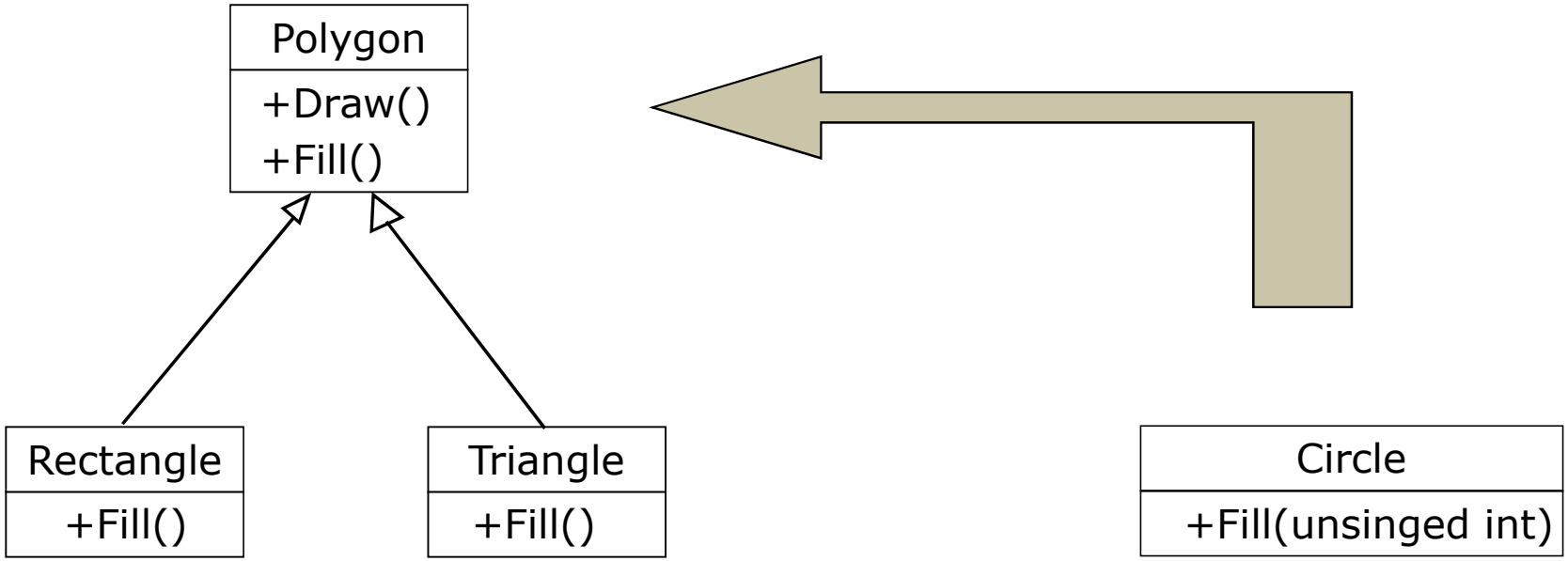
为了适应不同类的接口，常常需要修改各自的接口。但修改类的接口，会影响其它已使用该接口的代码。**Adapter**模式采用定义一个适应不同接口的接口类是解决此类问题。该模式的特点是：

- 避免直接修改接口带来的副作用；
- 增加不同类间接口应用的灵活性。

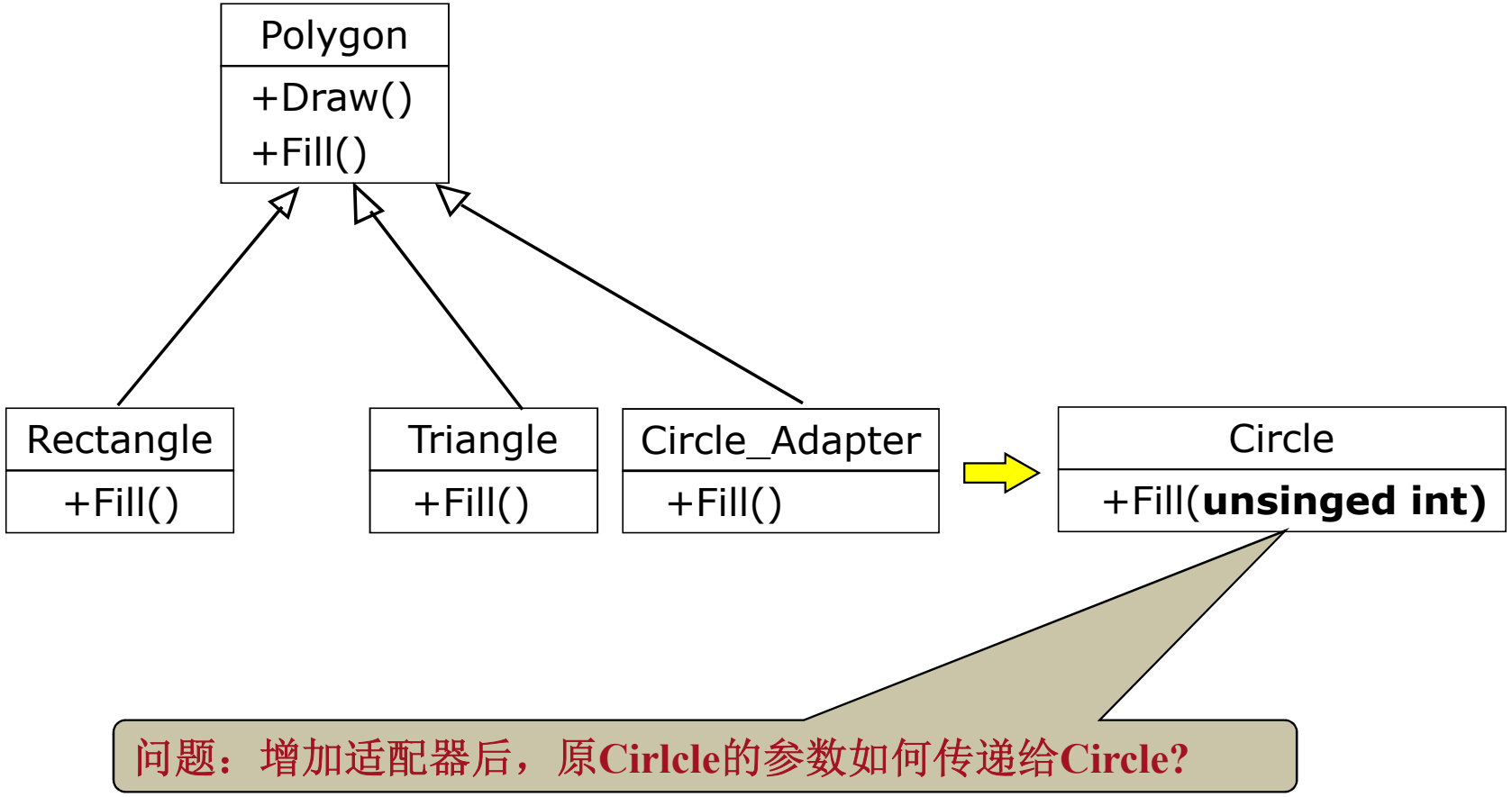
设计模式要素	说 明
模式名称	**Adapter**
目的	解决类间接口不匹配的问题。
问题描述	将类的接口转换为所希望的另一种接口。
解决方案	定义**Adapter**类，用其转换类的接口。
参与者	接口不匹配的类，**Adapter**类。
结构	用实例描述的示例图，如图**9-18**所示。

4. Adapter模式

适配器（Adapter）模式是用于将类的一个接口转换为另一个类所需接口，以使由于接口不兼容的类能够一起协作。



4. Adapter模式


Polygon
+Draw()
+Fill()
Rectangle
+Fill()
Triangle
+Fill()
Circle_Adapter
+Fill()
Circle
+Fill(unsinged int)
问题：增加适配器后，原Cirlcle的参数如何传递给Circle?

5. Iterator模式

对于保存数据的类来说，需要提供一种便于顺序访问类中的数据，同时又无需暴露类中数据存储形式的方法。将保存数据的类称为容器，访问类中数据的算法类（Iterator）称之为迭代器。这样，当需要遍历容器中数据时，只需要通过Iterator类而无须知道数据的存放结构，从而迭代器成为数据访问与容器之间的一座桥梁。

设计模式要素	说 明
模式名称	Iterator
目的	提供一种便于顺序访问类中数据，同时又无需暴露类中数据的存储形式。
问题描述	对类中数据的顺序访问，需要知道容器内部的数据结构情况，并难以统一访问形式。
解决方案	定义Iterator类，对某类型的容器统一提供Iterator模式的访问。
参与者	容器类与迭代类Iterator。
结构	用实例描述的示例图，如图9-19所示。

5. Iterator模式

类图

面向对象测试

面向对象测试概述

面向对象提供的封装性、继承性、多态性机制为OOP带来灵活性的同时，也使得原有的测试技术必须有所改变。

面向对象技术所独有的多态，继承，封装等新特点，产生了传统语言设计所不存在的错误可能性，或者使得传统软件测试中的重点不再显得突出，或者使原来测试经验认为和实践证明的次要方面成为了主要问题。

在OOP中，如何分析与测试表达式：

Y = fun(X);

面向对象测试

1. 确保属性的封装性约束

- 封装性通过访问权限，明确地限制了属性和方法的访问权限，减少结构化测试中对成员函数非法调用的测试，但需要考虑测试对成员数据是否满足封装性要求。

有两类问题需要注意：

- 当类的属性中定义有指针、引用或数组时（除了考虑访问之外，还有深拷贝/浅拷贝等问题）；
- 当类的成员函数返回指针或引用，而该指针或引用指向类的私有属性时。

面向对象测试

2. 派生类对基类成员函数的测试

对父类中已经测试过的成员函数，有两类情况需要在子类中再次进行测试：

a）继承性：父类的成员函数在子类中做了修改；

b）多态性：成员函数调用了改动过的另一个成员函数。

- 继承性在使得代码重用效率提高的同时，也使得原有代码中的错误得到传播，并增加了派生类的测试工作。
- 多态性增强继承中对基类成员函数的覆盖，也使得在类继承体系的类家族中，对同一接口的函数（虚函数）操作更为复杂，测试策略的设计需要更为仔细。

面向对象测试

3. 对抽象类的测试

由于抽象类不能直接定义对象，因而对抽象类的测试，主要是通过对其派生的测试来进行的。

对抽象类测试的基本过程：

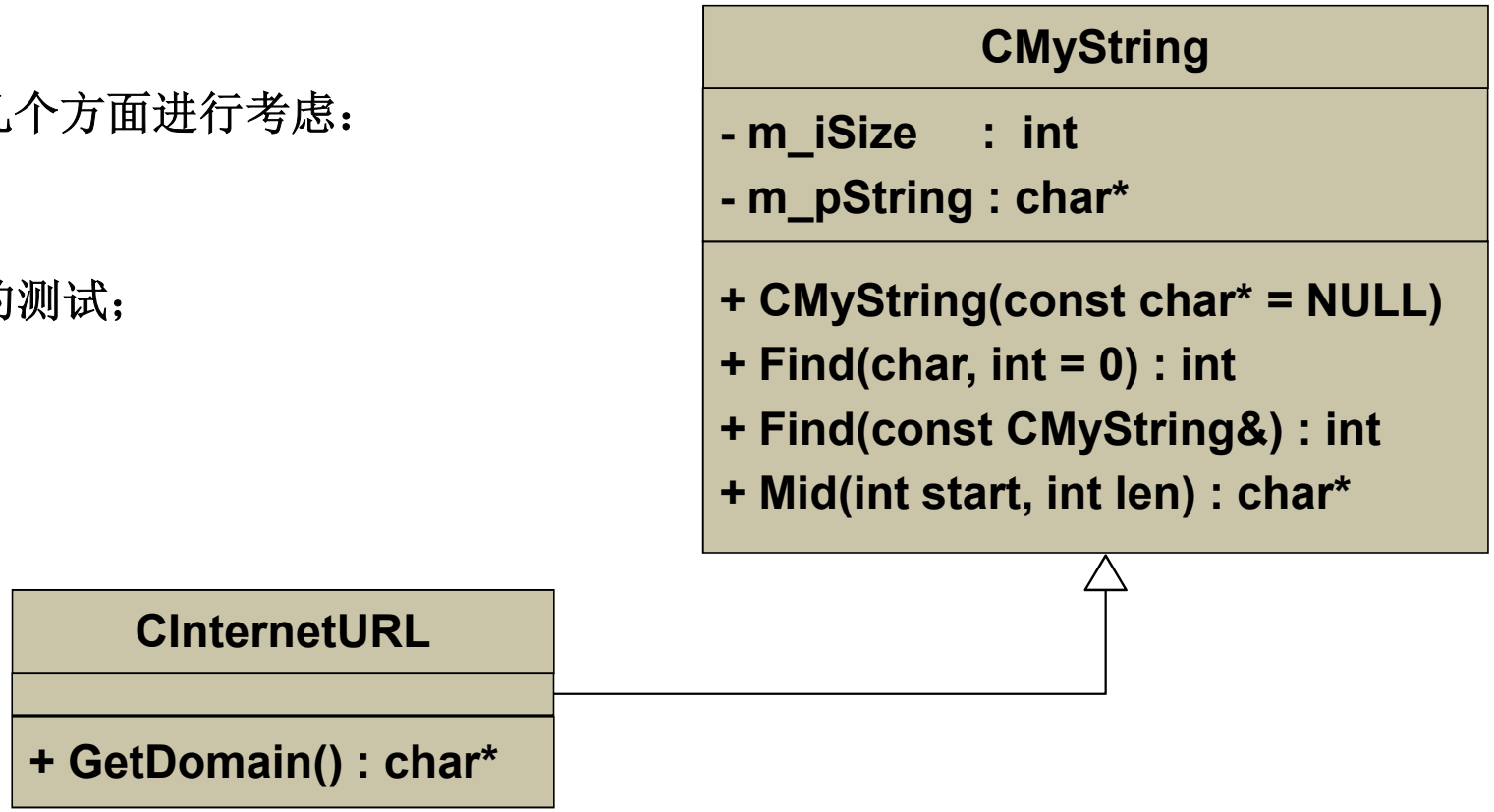
- 测试派生类自身的功能与性能；
- 测试派生类与抽象类（基类）的关系；
- 测试派生类与其他类间的关系。

面向对象测试

1. 面向对象的测试过程——单元测试（类测试）

对类进行测试需要从以下几个方面进行考虑：

- 确保属性的封装性约束；
- 派生类对基类成员函数的测试；
- 对抽象类的测试 。



面向对象测试

2. 面向对象的测试过程——集成测试

面向对象的集成测试主要是两个方面：

- 类的过程测试；
- 类的独立性测试，特别是当类作为部件发布时，更需要测试类的跨平台的适应性。

2. 面向对象的测试过程——集成测试

例：一个银行信用卡的应用中，设计了如下类：**Account**。

Account
-AccountID : string
+Account(int) +Login() : bool +Deposit() : bool +Withdraw() : bool +Balance() : double +Summarize() : List<string> +Quit() : bool

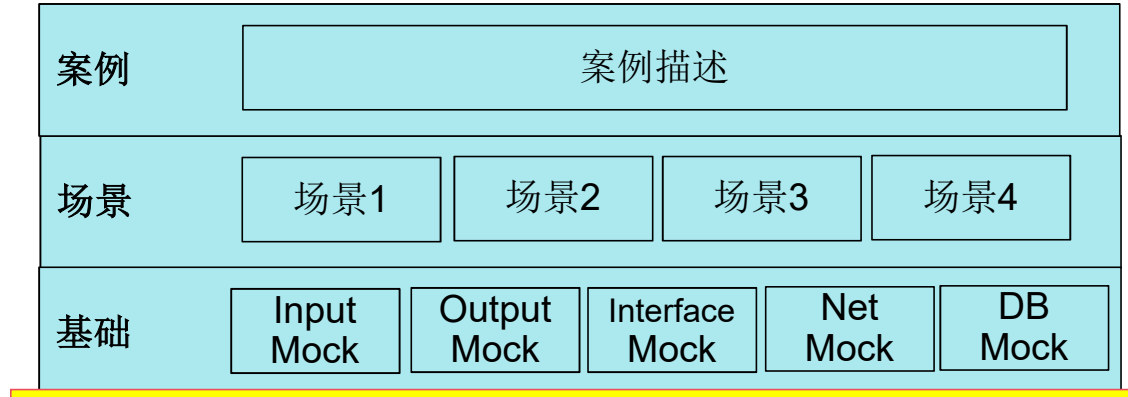
- *一个类中待测试的内容太多怎么办？*
- *测试结构较复杂怎么办？*
- *缺失数据怎么办？*

基本准则：

- 基于过程的测试
- 覆盖类中的所有成员

面向对象测试

2. 面向对象的测试过程——集成测试



基本准则：

- 基于过程的测试
- 覆盖类中的所有成员

案例集：Login+[Deposit | Withdraw | Balance | Summarize]+Quit