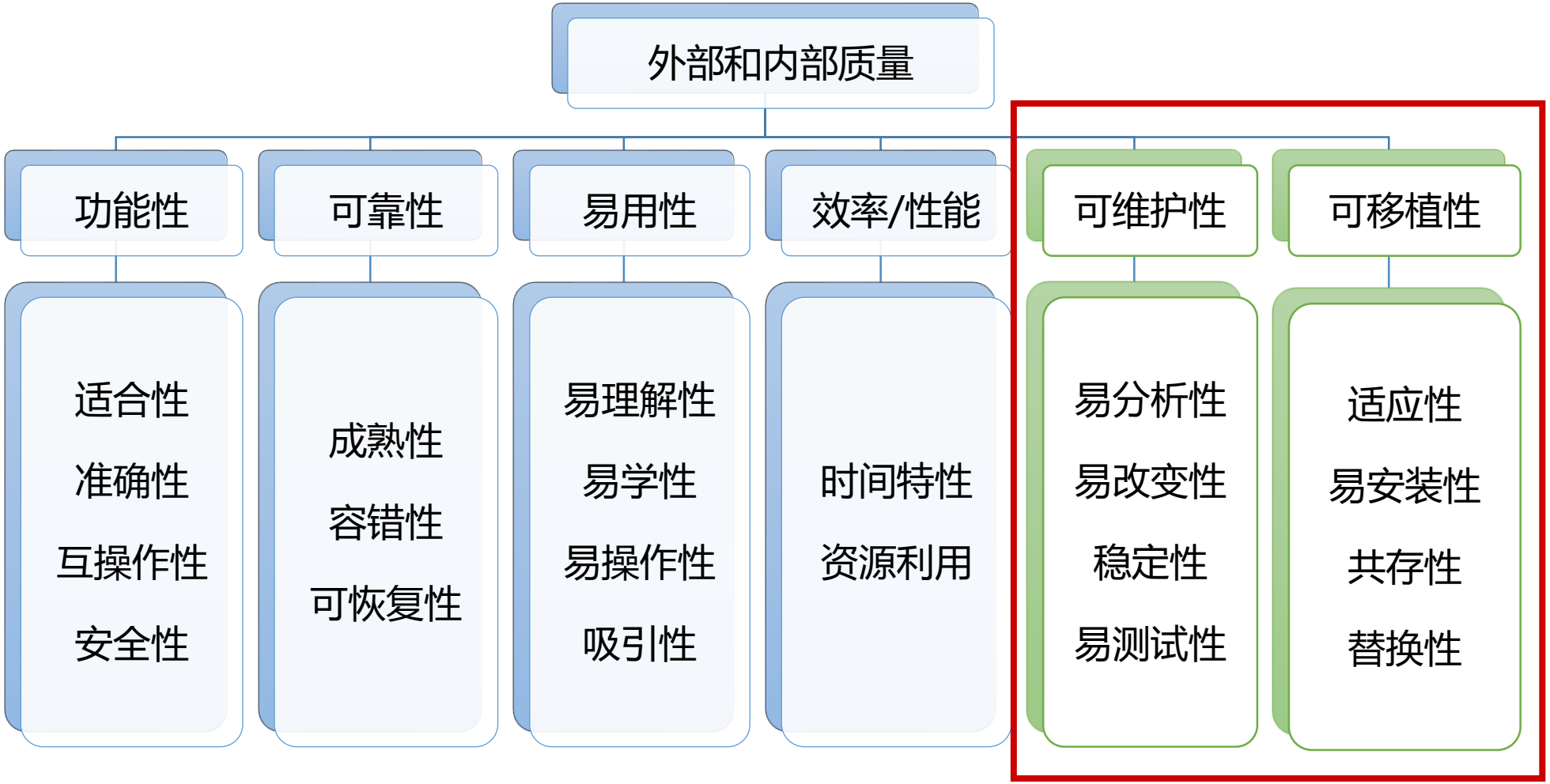
重构与设计模式

清华大学软件学院 刘强

教学提纲

1. 软件重构概述
2. 软件重构方法
3. 设计模式

回顾：软件质量


外部和内部质量
功能性
可靠性
易用性
效率/性能
可维护性
可移植性
适合性
准确性
互操作性
安全性
成熟性
容错性
可恢复性
易理解性
易学性
易操作性
吸引性
时间特性
资源利用
易分析性
易改变性
稳定性
易测试性
适应性
易安装性
共存性
替换性

编写优美的程序

什么是优美的程序？

编写优美的程序

任何一个傻瓜都能写出计算机可以理解的程序，
只有写出人类容易理解的程序才是优秀的程序员。

—— Martin Fowler

高效程序员不应该浪费很多时间用于程序调试，
他们应该一开始就不要把故障引入。

—— Edsger Dijkstra

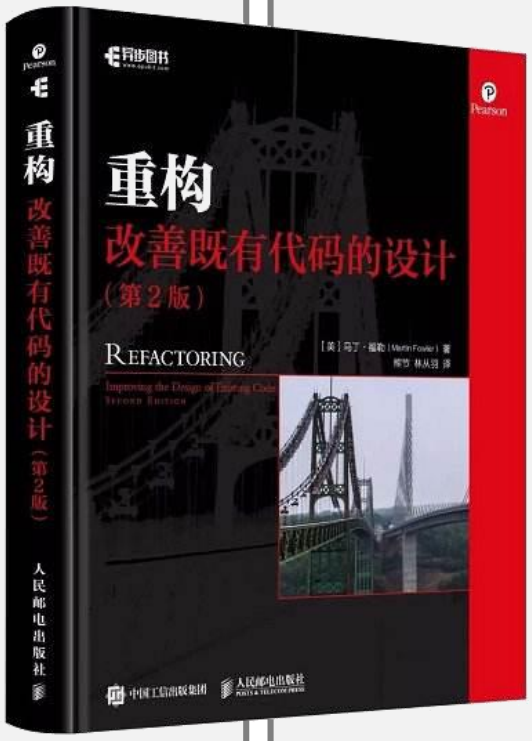
编写优美的程序

优美的代码可理解性高，修改成本低。不过优美的代码是很难一次写出的!

- 对代码的逻辑层次要有感觉
- 善于抽取算法代码
- 提取工具代码
- 提高抽象技巧

代码的坏味道

重复的代码
过长的方法
过大的类
过长的参数列表
发散式变化
霰弹式修改
依恋情结
数据泥团
基本类型偏执
Switch语句
平行继承关系



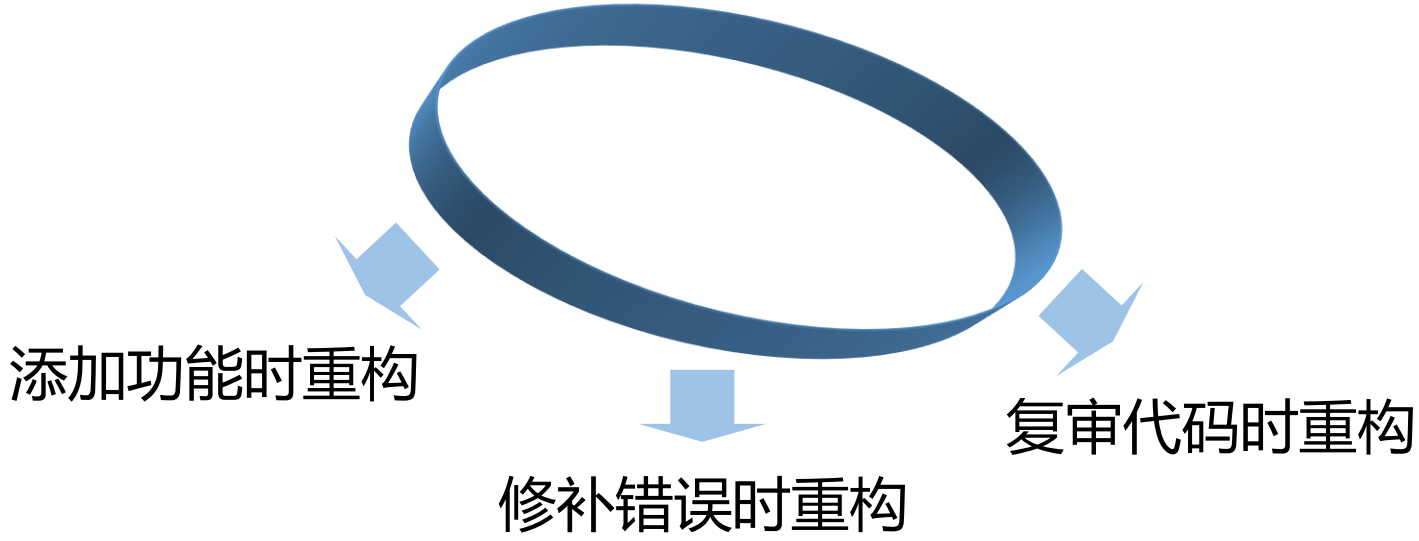
多余的类
不确定的一般性
临时字段
消息链
二传手
过度亲密
异曲同工的类
不完整的库类
纯稚的数据类
拒收的馈赠
过多的注释

重构的概念

重构（Refactoring）是对软件内部结构的一种调整，其目的是在不改变软件功能和外部行为的前提下，提高其可理解性、可扩展性和可重用性。

何时重构?

三次法则：第一次做某件事时只管去做；
第二次做类似的事会产生反感，但无论如何还是去做；
第三次再做类似的事就该重构。



什么时候不适合重构?

代码太混乱，设计完全错误

- 与其重构不如重新开始

明天是Deadline

- 永远不要做Last-Minute-Change；应推迟重构但不可忽略，即使进入Production的代码都正确地运行。

重构的工作量显著地影响估算

- 一个任务的估算时间是3天，如果为了重构，就需要更多的时间。
- 推迟重构但不忽略，可以把重构作为一个新任务，或者安排在重构的迭代周期中完成。

重构与添加新功能

添加新功能

- 添加新功能时，不应该修改既有代码，只管添加新功能

重构

- 重构时不再添加功能，只管改进程序结构

重构与添加新功能可交替进行

教学提纲

1. 软件重构概述
2. 软件重构方法
3. 设计模式

软件重构方法

具备重构的意识，在编程过程中培养重构的能力

- 把基础打牢固
- 多看点优秀的代码
- 避免复制粘贴，如果看见重复代码时应该有意识要消灭它
- 减少对代码生成器的依赖
- 在处理现有代码时尽量用重构代替重写，在重写之前一定要先重构
- 尽量让所有的方法都是可测试的

```
protected void queryBtn(object sender, EventArgs e) {
    //如果项目编号不为空时
    if (this.xmbhText.Text.ToString().Trim().Equals("") == false) {
        //对输入的查询条件—项目编号是否合法进行校验
        Byte[] MyBytes = System.Text.Encoding.Default.GetBytes(
                         this.xmbhText.Text.ToString().Trim());
        if (MyBytes.Length > 10) {
            MessageBox.Alert("项目编号不能超过10个字节！");
            this.xmbhText.Focus();
            return;
        }
    }
    //如果项目名称不为空时
    if (this.xmmcText.Text.ToString().Trim().Equals("") == false) {
        //对输入的查询条件—项目名称是否合法进行校验
        Byte[] xmmccheck = System.Text.Encoding.Default.GetBytes(
                           this.xmmcText.Text.ToString().Trim());
        if (xmmccheck.Length > 40) {
            MessageBox.Alert("项目名称不能超过40个字节！");
            this.xmmcText.Focus();
            return;
        }
    }
    //实现数据库绑定
    GridView1_DataBind();
}
```

重复的代码

症状：

- 容易形式：两个代码段看上去几乎相同
- 困难形式：两个代码段都拥有几乎相同的作用

措施：

- 抽取方法：如果同一个模块的两个函数存在重复代码，则将重复代码抽取成一个函数，在原有函数的对应处调用所抽取的函数。
- 抽取模块：如果两个毫不相关的模块存在重复代码，则将重复代码抽取成到一个独立的模块中或者某个模块中，其他模块引用这个模块。
- 替换算法：如果有些函数以不同的算法做相同的事，则选择一个比较清晰的函数，而将其他函数的算法替换掉。

```
protected void queryBtn(object sender, EventArgs e) {
    //如果项目编号不为空时
    if (this.xmbhText.Text.ToString().Trim().Equals("") == false) {
        //对输入的查询条件—项目编号是否合法进行校验
        Byte[] MyBytes = System.Text.Encoding.Default.GetBytes(
                         this.xmbhText.Text.ToString().Trim());
        if (MyBytes.Length > 10) {
            MessageBox.Alert("项目编号不能超过10个字节！");
            this.xmbhText.Focus();
            return;
        }
    }
    //如果项目名称不为空时
    if (this.xmmcText.Text.ToString().Trim().Equals("") == false) {
        //对输入的查询条件—项目名称是否合法进行校验
        Byte[] xmmccheck = System.Text.Encoding.Default.GetBytes(
                           this.xmmcText.Text.ToString().Trim());
        if (xmmccheck.Length > 40) {
            MessageBox.Alert("项目名称不能超过40个字节！");
            this.xmmcText.Focus();
            return;
        }
    }
    //实现数据库绑定
    GridView1_DataBind();
}
```

提取到一个单独的方法中

举例：清华人校友应用

校内学籍数据接入时需要对不规范的数据进行清洗，不同年代不同类型的学籍数据保存在不同的数据库中，清洗时每种数据的处理流程类似但也有少量的差别。

数据来源	记录数	覆盖时段	提供部门	存储形式	接入方式
本科生学籍表	78619	本1994至今	教务处	信息中心 ORACLE 数据库	信息中心 数据共享
本科生学籍异动表	11857				
本科生二学位表	7784				
本科生照片表	124768				
学历认证证书信息表	103060	本1977至今			
研究生学籍表	117328	硕1992至今 硕1989至今	研究生院		
研究生学籍异动表	19109				
研究生照片表	215818				
研究生院历史硕士学籍库	9135	硕1978-1992		研究生院 DBASE数据库	拷贝DBF文件
研究生院历史博士学籍库	1501	博1980-1991			
档案馆历史学籍库	145063	本硕博 1924-2000	档案馆	档案馆 单机数据库	拷贝EXCEL文件
档案馆历史照片库	139182				拷贝JPG文件

实际情况：

- 每次处理一种学籍，单独编写代码
- 类似的代码写很多遍，复制粘贴

举例：清华人校友应用

- 代码重构：共同的部分抽取为一个方法

文件	说明	行数变化
enrollment-cleansing.go	学籍数据清洗	增加 97
import-bksxj.go	本科生学籍	增加 29，删除 60
import-xlrz.go	学历认证	增加 21，删除 49
import-yjsxj.go	研究生学籍	增加 28，删除 65

- 重构后的好处：
  - 便于维护：需求变化时，只需要在一处代码修改，不必担心遗漏
  - 便于复用：新增类型时，直接调用公共的方法即可，复制粘贴再修改会改不干净

- 需要注意的是抽象的层次不能过低，也不宜过高

信息中心本科毕业证书-1978起	信息中心本科学籍-199x起	信息中心研究生学籍-199x起	档案馆-2000前	系统内维护
	XSLXM 学生类型码	XSLBM 学生类别码	学历	EnrollmentType
XH 学号	XH 学号	XH 学号	学号	StudentId
ID ID				
XM 姓名	XM 姓名	XM 姓名	姓名	Name
	YWXM 英文姓名	YWXM 英文姓名		
	XMPY 姓名拼音			
XB 性别	XBM 性别码	XBM 性别码	性别	Gender
CSRQ 出生日期	CSRQ 出生日期	CSRQ 出生日期	出生日期	Birthday
		SFZJLXM 身份证件类型码		
	SFZH 身份证号	ZJH 证件号	身份证号码	DocumentNumber
	KQH 考区号	JGSSM 籍贯省市码	籍贯	
		JGSX 籍贯市县		
	MZM 民族码	MZM 民族码	民族	
	ZZMMM 政治面貌码	ZZMMM 政治面貌码		
	GBM 国别码	GBM 国别码		Nationality
	SFLXS 是否留学生	SFLXS 是否留学生		
RXNY 入学年月	RXRQ 入学日期	RXNY 入学年月	入学日期	EnrollmentDate
XZ 学制	XZM 学制码		学制	
	RXNJM 入学年级码			
	SSNJM 所属年级码			
BYJ 毕业届				
XSM 系所名	DWNM 单位内码	DWNM 单位内码	院系编码、院系名称	DepartmentCode/Name
	XZDWNM 行政单位内码			
ZYM 专业名	ZYH 专业号	ZYH 专业号	专业代码、专业名称	MajorCode/Name
	PYDLH 培养大类号			
	FLFXM 分流方向码			

例如“学籍所属院系”

有的数据库中用代码表示，有的数据库中用名称表示，二者处理的逻辑路径不同。

若强行统一成一个方法，其内部还需要做条件判断，可维护性可能反而降低。

过长的函数

症状：

- 只要看到超过N（如10）行代码的方法，立即检查是否可以重构之。
- 长度是一个警告信号，并不表示一定有问题。

措施：

- 绝大多数场合下，可以采取抽取方法的重构手法来把方法变小。
- 每当需要以注释来说明点什么时，就可以把需要说明的东西写进一个独立方法中。
- 以其用途（而非实现方法）对抽取的方法命名。

说明：

- 对于现代开发工具，方法调用增多不会影响性能，但长方法难以理解。

举例：清华人校友应用

- GraphQL Schema 定义
  - 如果写死在 go 源码中，无法利用编辑器的语法高亮等功能，故单独放在一个文件中，服务启动时读取
  - 用 packr 打包，以便只交付单个可执行文件

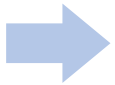
文件	行数
schemas-mp.gql	610

- 存在问题
  - 单个文件太长，需要修改时， Ctrl+F、上下滚动半天才能找到具体修改的位置
  - 可维护性较差

举例：清华人校友应用

- 重构：将文件拆分，在加载时合并，这样其他代码无需修改
- 效果：修改文件内容更容易了

文件	行数
schemas-mp.gql	610



文件	行数
schemas-mp-association.gql	45
schemas-mp-campus.gql	119
schemas-mp-contact.gql	85
schemas-mp-donation.gql	67
schemas-mp-find.gql	86
schemas-mp-legal.gql	57
schemas-mp.gql	164

```
http.Handle("/api",
 handlers.Logging(
  handlers.AttachDatabase(
   handlers.Authentication(
    &handlers.GraphQLHandler{
     Schema: graphql.MustParseSchema(
      loadSchema("schemas-mp.gql"),
      &resolver.RootResolver{})})))))
```

重构前

```
func loadSchema(filename string) string {
 box := packr.NewBox("./schema")
 content, err := box.FindString(filename)
 if err != nil {
  panic(err)
 }
 return content
}
```

重构后

```
http.Handle("/api",
 handlers.Logging(
  handlers.AttachDatabase(
   handlers.Authentication(
    &handlers.GraphQLHandler{
     Schema: graphql.MustParseSchema(
      loadSchema([]string{
       "schemas-mp.gql",
       "schemas-mp-association.gql",
       "schemas-mp-campus.gql",
       "schemas-mp-contact.gql",
       "schemas-mp-donation.gql",
       "schemas-mp-find.gql",
       "schemas-mp-legal.gql",
      }),
      &resolver.RootResolver{})})))))
```

```
func loadSchema(filenames []string) string {
 box := packr.NewBox("./schema")
 concatenated := ""
 for _, filename := range filenames {
  content, err := box.FindString(filename)
  if err != nil {
   panic(err)
  }
  concatenated += content
  concatenated += "\n"
 }
 return concatenated
}
```

发散式变化

症状：

- 某个模块因为不同的原因在不同的方向上发生变化

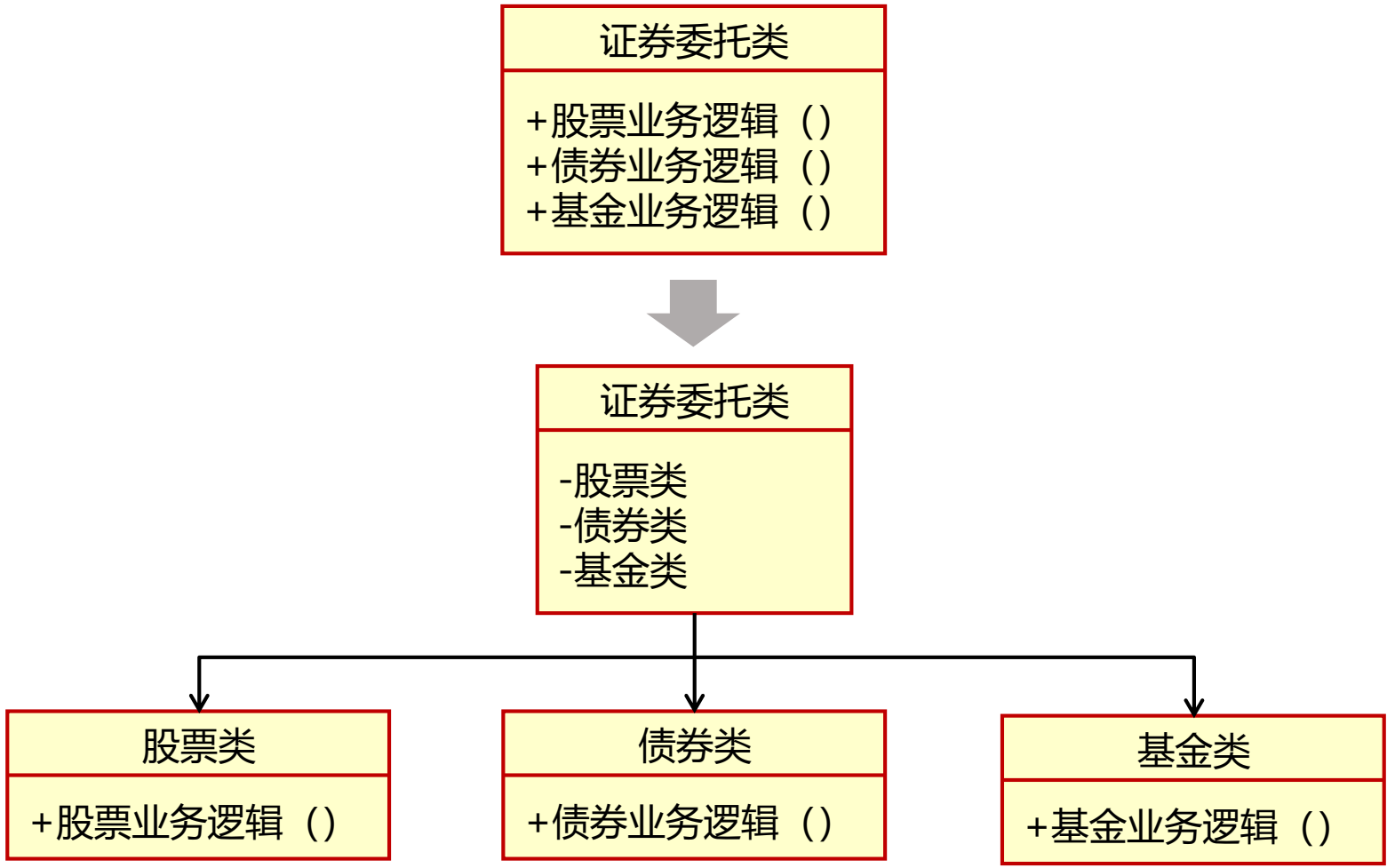
措施：

- 如果发生变化的两个方向自然形成了先后次序，则将二者拆分成两个阶段进行分离，二者之间通过一个清晰的数据结构进行沟通。
- 如果两个方向之间有更多的来回调用，则先创建适当模块，再用搬移函数把处理逻辑分开。
- 如果函数内部混合了两类处理逻辑，应该先用提炼函数将其分开，然后再进行搬移。

说明：

- 发散式变化指的是“一个模块受多个外界变化的影响”，其基本思想是把相对不变的和相对变化的相隔离，即封装变化。

发散式变化


证券委托类
+股票业务逻辑（）
+债券业务逻辑（）
+基金业务逻辑（）
证券委托类
-股票类
-债券类
-基金类
股票类
+股票业务逻辑（）
债券类
+债券业务逻辑（）
基金类
+基金业务逻辑（）

霰弹式修改

症状：

- 发生一次改变时，需要修改多个模块的多个地方

措施：

- 找出一个应对这些修改负责的模块，这可能是一个现有的模块，也可能需要通过抽取模块来创建一个新模块。
- 使用移动字段和移动方法等措施将功能置于所选的模块中，如果未选中模块足够简单，则可以使用内联类或内联函数将该模块除去。

说明：

- 散弹式修改指的是“一种变化引发多个模块的修改”，其基本思想是将变化率和变化内容相似的状态和行为放在同一个类中，即集中变化。

举例：清华人校友应用

- 在小程序中很多界面都要有提示性的文案，这些文字信息散落在程序各处，当提示信息修改时，就要找到相应的代码进行修改。
- 重构
  - 将界面文案的字符串资源集中管理
  - 便于根据需求快速修改文案
  - 便于日后支持多种语言的文案显示

mp-thu-alumni/utils/copywriting.js

```
@@ -0,0 +1,27 @@
+ const copywriting = {
+   register: {
+     'UPDATE_PHONE': '为了让您的账户更安全，请设置常用的联系方式',
+     'UPDATE_DOCUMENT': '* 请确认证件与本人一致，证件号码准确',
+     'UPDATE_DOCUMENT_FACE': '* 请确认证件与本人一致，证件号码准确\n
+     'UPDATE_DOCUMENT_PAY': '* 请确认证件与本人一致，证件号码准确\n即
+   },
+   campusInfo: {
+     'UPDATE_EXTRA': '您可以上传毕业证书等相关证件照片\n帮助我们更快的
+     'UPDATE_EXTRA_WAIT': '您可以上传毕业证书等相关证件照片\n帮助我们
+   },
+   endorse: {
+     'SAME_PERSON': '您不能帮助自己认证',
```

举例：清华人校友应用

mp-thu-alumni/pages/myCampusInfoExtra/myCampusInfoExtra.wxml

```
@@ -11,7 +11,7 @@
        </view>
        <view class="line"></view>
        <view class="tip text-small">
-         <text>您可以上传毕业证书等相关证件照片\n帮助我们更快的核实您的信息\n* 长按图片可以移除</text>
+         <text>{{copywriting.register['UPDATE_EXTRA']}}</text>
        </view>
      </view>
      <button class="main-btn" hover-class="btn-hover" bindtap="submit">提交</button>
@@ -25,7 +25,7 @@
        </view>
        <view class="line"></view>
        <view class="tip text-small">
-         <text>您的信息已提交审核，请耐心等待</text>
+         <text>{{copywriting.register['UPDATE_EXTRA_WAIT']}}</text>
        </view>
      </view>
    </block>
```

重构前的代码

重构后的代码

教学提纲

1. 软件重构概述
2. 软件重构方法
3. 设计模式

设计模式

回顾学过的数据结构

- 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?

答案是肯定的

- 计模式使我们可以重用已经成功的经验
- Pattern = Documented experience

设计模式

设计模式描述了软件系统设计过程中常见问题的一些解决方案，它是从大量的成功实践中总结出来的且被广泛公认的实践和知识。

设计模式的好处

- 使人们可以简便地重用已有的良好设计
- 提供了一套可供开发人员交流的语言
- 提升了人们看待问题的抽象程度
- 帮助设计人员更快更好地完成系统设计
- 模式是经过考验的思想，具有更好的可靠性和扩展性

设计模式

设计模式不是万能的

- 模式可以解决大多数问题，但不可能解决遇到的所有问题
- 应用一种模式一般会“有得有失”，切记不可盲目应用
- 滥用设计模式可能会造成过度设计，反而得不偿失

设计模式是有难度和风险的

- 一个好的设计模式是众多优秀软件设计师集体智慧的结晶
- 在设计过程中引入模式的成本是很高的
- 设计模式只适合于经验丰富的开发人员使用

设计模式的基本原则

单一职责原则

开放封闭原则

Liskov替换原则

依赖倒置原则

接口分离原则

单一职责原则

单一职责原则（Single Responsibility Principle，SRP）

> ***There should never be more than one reason for a class to change.***
>
> **R. Martin, 1996**

说明：

- 让一个类有且只有一种类型责任，当这个类需要承当其他类型的责任的时候，就需要分解这个类。
- 该原则可以看作是高内聚、低耦合在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。

单一职责原则

举例：下面的接口设计有问题吗?

```
interface Modem {
    public void dial(String pno);
    public void hangup();
    public void send(Char c);
    public char recv();
}
```

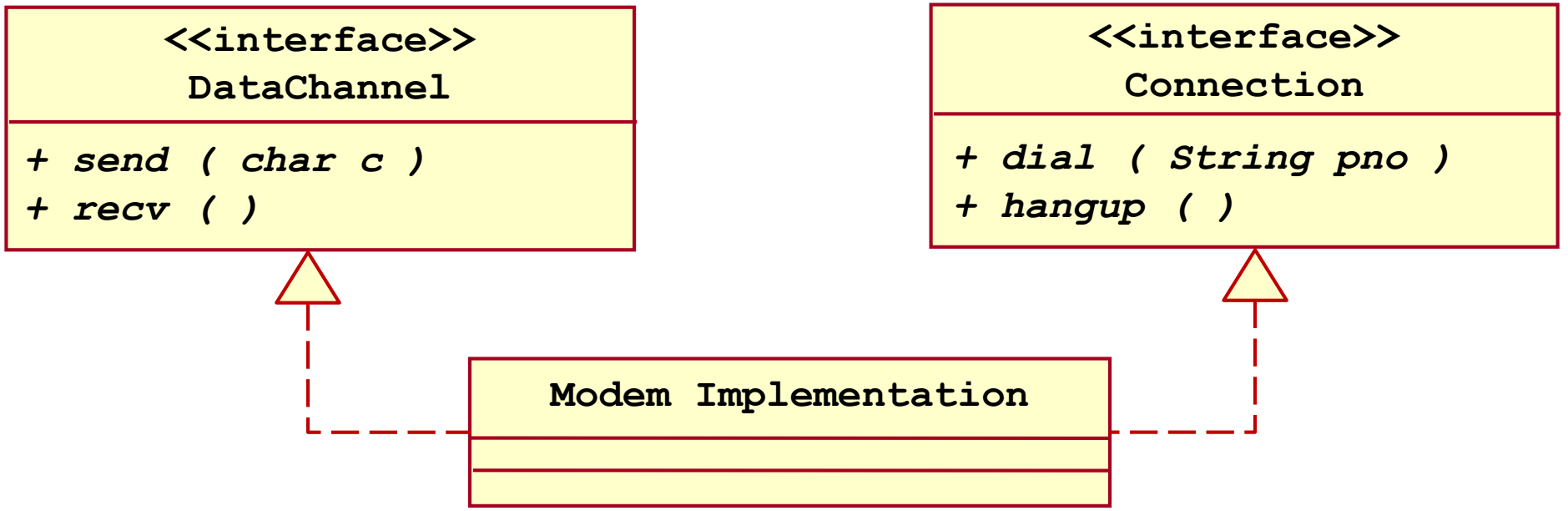
接口职责：

- 连接管理：dial 和 hangup 进行调制解调器的连接处理
- 数据通信：send 和 recv 进行数据通信

单一职责原则

Modem接口是否应该分离?

- 依赖于应用程序变化的方式，它是否影响连接函数的特征?
- 数据通信是经常变化的；连接管理是相对稳定的



单一职责原则

总结：

- 职责的划分：一个类的一个职责是引起该类变化的一个原因
- 单一职责原则分离了变与不变
  - 对象的细粒度：便于复用
  - 对象的单一性：利于稳定
- 不要创建功能齐全的类

单一职责原则是一个最简单的原则，也是最难正确应用的一个原则，因为我们会自然地将职责结合在一起。

开放封闭原则

开放封闭原则（Open-Closed Principle，OCP）

> ***Software entities should be open for extension, but closed for modification.***
>
> **B. Meyer, 1988 / quoted by R. Martin, 1996**

- Be open for extension

  模块的行为是可扩展的，即可以改变模块的功能。

- Be closed for modification

  对模块行为进行扩展时，不需要改动源代码或二进制代码。

- 问题：怎样可以在不改动源码的情况下改变模块行为？

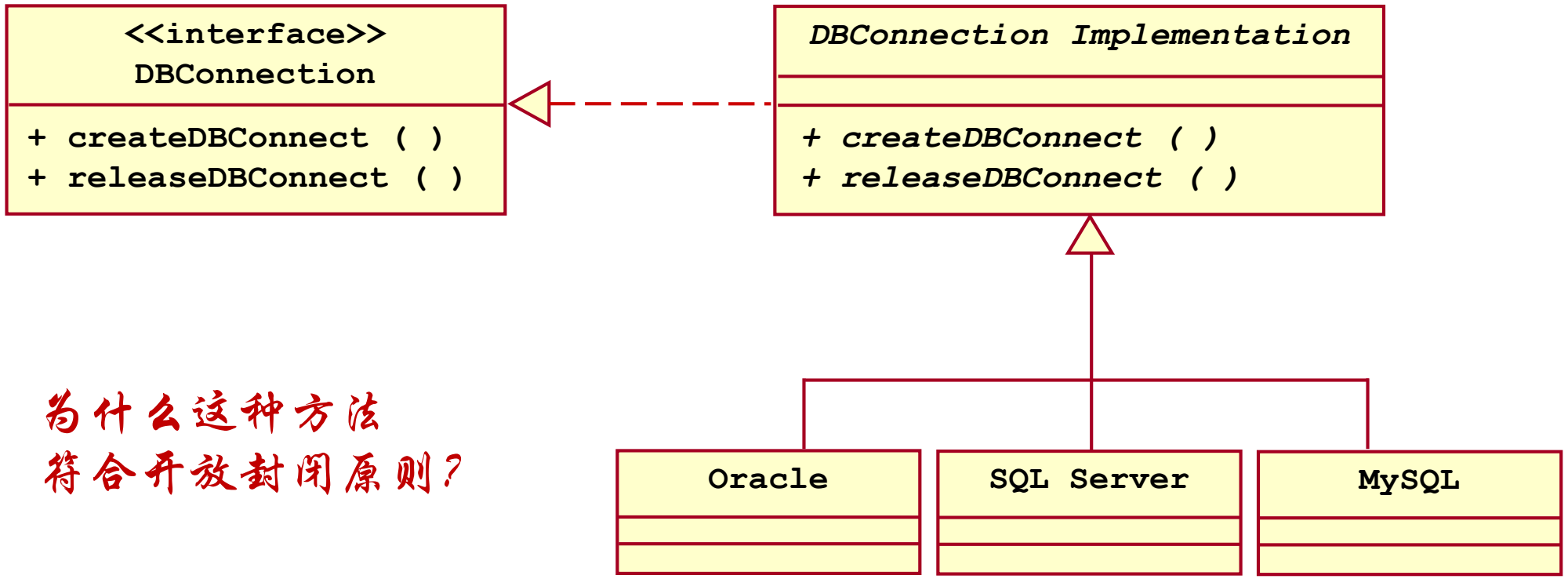
开放封闭原则

举例：如要增加新的数据库类型（如MySQL）怎么处理?

```
bool createDBConnect(...)
{
    ...
    switch (dbType)
        {
            case ORACLE:
                建立 Oracle 连接;
                break;
            case SQLSERVER:
                建立 SQL Server 连接;
                break;
    }
    ...
}
```

开放封闭原则

数据库连接的另一种修改方法：



为什么这种方法
符合开放封闭原则？

开放封闭原则

开放封闭原则：在设计一个软件模块（类、方法）时，应该在不修改原有代码（修改封闭）的基础上扩展其功能（扩张开放）。

实现方式

- 将那些不变的部分加以抽象成不变的接口，这些不变的接口可以应对未来的扩展；
- 接口的最小功能设计：原有的接口要么可以应对未来的扩展，不足的部分可以通过定义新的接口来实现；
- 模块之间的调用通过抽象接口来实现，即使实现层发生变化也无需修改调用方的代码。

开放封闭原则

关键是抽象

- 创建一个抽象基类，描述一组任意个可能行为，而具体的行为由派生类实现。
- 软件实体依赖于一个固定的抽象基类，这样它对于更改可以是关闭的，通过从这个抽象基类派生，也可以扩展该实体的行为。

程序不可能是100%封闭的

- 无论模块多么封闭，都会存在一些无法封闭的变化。
- 设计人员应该策略地对待这个问题，即先预测出最有可能发生变化的部分，再构造抽象来隔离这些变化。

开放封闭原则

- 开放封闭原则是面向对象设计的核心所在，遵循这个原则可以带来面向对象技术的巨大好处，即可维护性、可扩展性、可复用性和灵活性。

- 几乎所有的设计模式都是对不同的可变性进行封装，从而使系统在不同角度上达到开放封闭原则的要求。

- 开发人员应该仅对程序中呈现出频繁变化的那些部分做出抽象，然而切忌对应用程序中的每个部分都刻意地进行抽象，拒绝不成熟的抽象和抽象本身一样重要。

Liskov替换原则

Liskov替换原则（Liskov Substitution Principle，LSP）

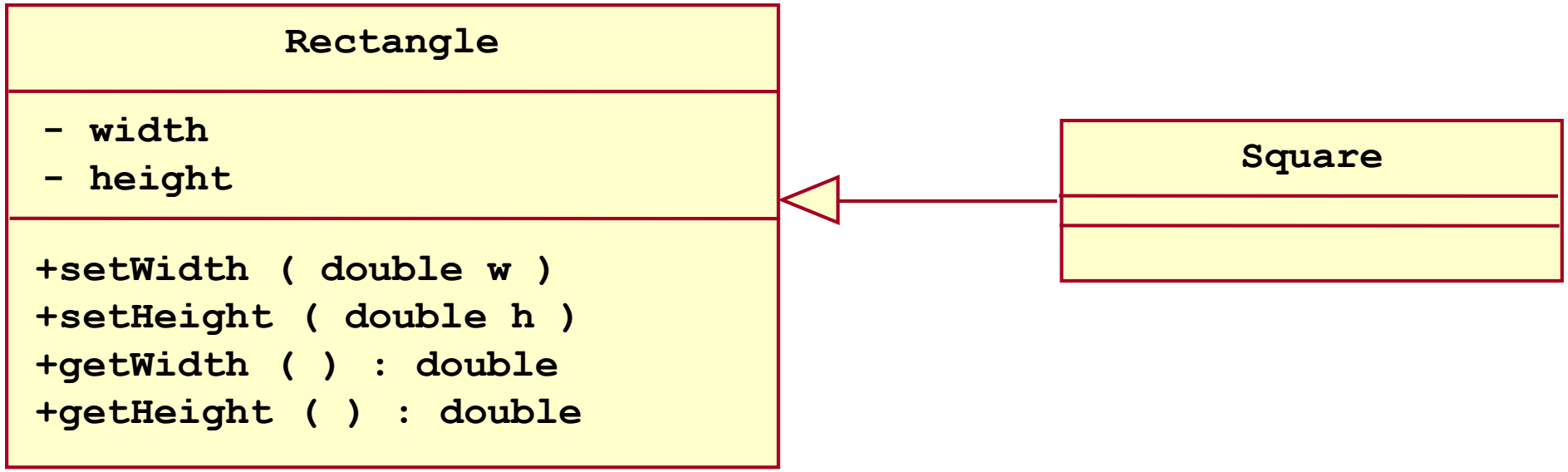
> ***Inheritance should ensure that any property proved about supertype objects also holds for subtype objects.***
>
> **B. Liskov, 1987**

> ***Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.***
>
> **R. Martin, 1996**

理解：子类型必须能够完全替换其父类型。

Liskov替换原则

举例：正方形应该从矩形继承吗？



你是否编写过类似的程序？有什么问题吗？

Liskov替换原则

覆写 Square 类成员函数

```
void Square::setWidth(double w)
{
     Rectangle::setWidth(w);
     Rectangle::setHeight(w);
}

void Square::setHeight(double h)
{
     Rectangle::setWidth(h);
     Rectangle::setHeight(h);
}
```

考虑下面的测试函数：

```
void g(Rectangle& r)
{
     r.setWidth(5);
     r.setHeight(4);
     assert(r.Area() == 20);
}
```

为什么不对？　　原因：从行为方式来看，Square 不是 Rectangle。

Liskov替换原则

契约式设计（Design by Contract）

- 契约式设计把类及其客户之间的关系看作是一个正式的协议，明确各方的权利和义务。
- 契约是通过每个方法声明的前置条件（preconditions）和后置条件（postconditions）来指定的。
- 一个方法得以执行，其前置条件必须为真；执行完毕后，该方法应保证后置条件为真。
- 规则：在重新声明派生类函数时，只能使用相等或者更弱的前置条件来替换原始前置条件，只能使用相等或者更强的后置条件来替换原始后置条件。

Liskov替换原则

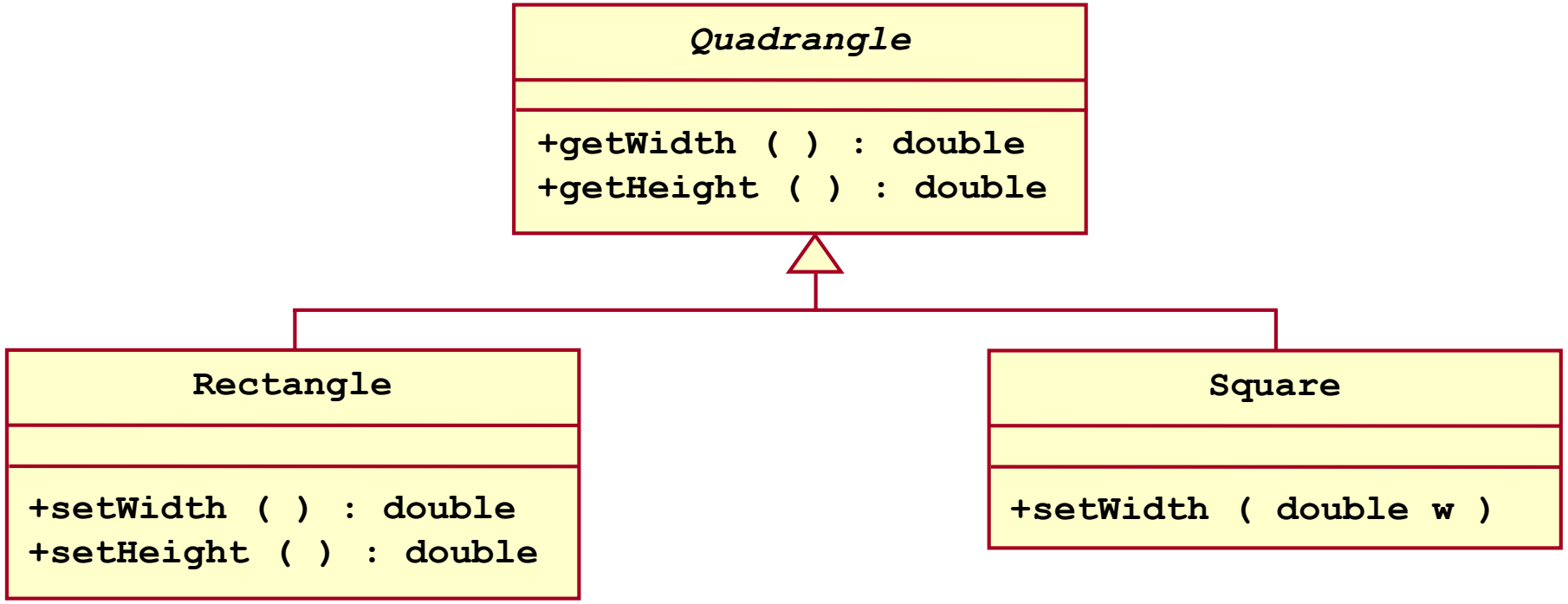
Square符合契约式设计规则吗?

- `Rectangle::setWidth(double w)`的后置条件
  `assert((width == w) && (height == oldHeight))`
- `Square::setWidth(double w)`的后置条件
  `assert((width == w) && (height == w))`
- 由此可见，`Square::setWidth(double w)`的后置条件不能遵从`Rectangle::setWidth(double w)`后置条件的所有约束，故前者的后置条件比后者的弱。

如何解决以上问题?

Liskov替换原则

重构方法：创建一个新的抽象类C，将其作为两个具体类的超类，将A、B的共同行为移动到C中来解决问题。



Liskov替换原则

- LSP可以保证系统或子系统有良好的扩展性。只有子类能够完全替换父类，才能保证系统或子系统在运行期内识别子类接口，因而使得系统或子系统具有良好的扩展性。

- LSP有利于实现契约式编程。契约式编程有利于系统的分析和设计，即定义好系统的接口，然后在编码时实现这些接口即可。在父类里定义好子类需要实现的功能，而子类只要实现这些功能即可。

- LSP是保证OCP的重要原则。它们在实现方法上有个共同点，使用中间接口层来达到类对象的低耦合，即抽象耦合。

依赖倒置原则

常见的设计问题

- 很难添加新功能，因为每一处改动都会影响其他很多部分。
- 当对某一处进行修改，系统中看似无关的其他部分出现了问题。
- 很难在别的应用程序中重用某个模块，因为不能将它从现有的应用程序中独立提取出来。

什么原因?

- 耦合关系：“高层模块”过分依赖于“低层模块”

一个良好的设计应该是系统的每一部分都是可替换的。

依赖倒置原则

依赖倒置原则（Dependency Inversion Principle，DIP）

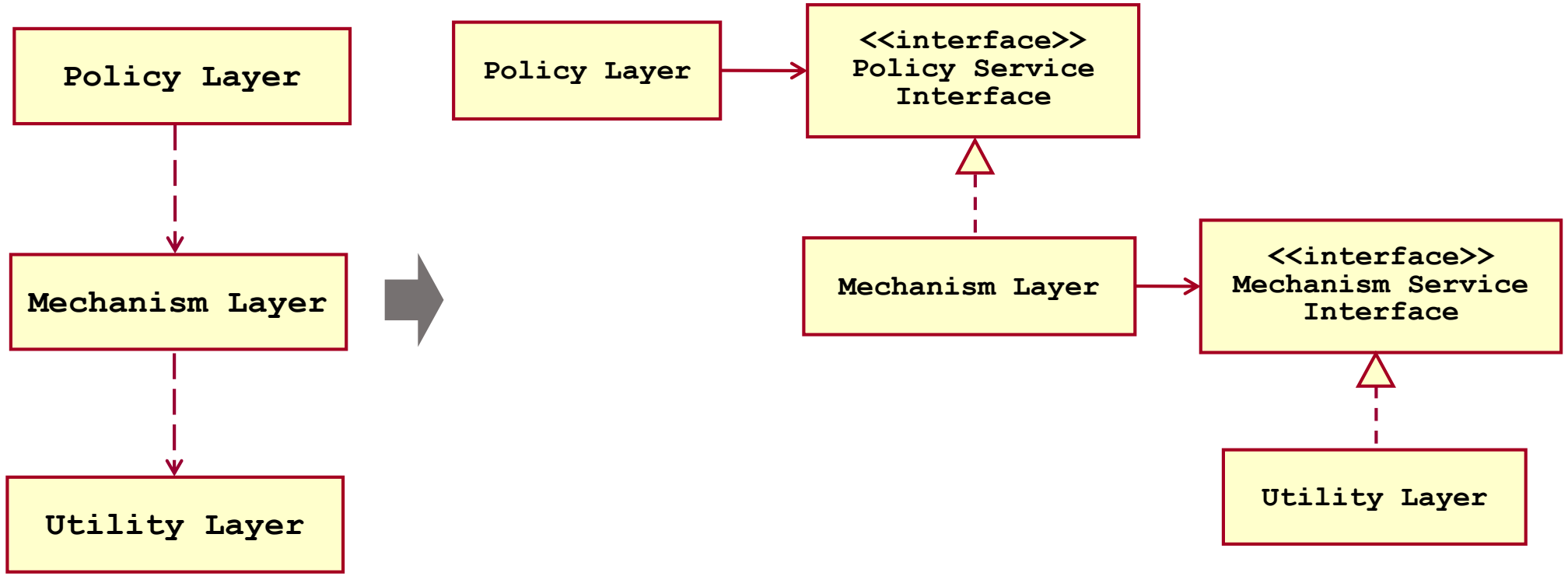
> ***I. High-level modules should not depend on low-level modules. Both should depend on abstractions.***
> ***II. Abstractions should not depend on details. Details should depend on abstractions.***
>
> **R. Martin, 1996**

依赖于抽象

- 任何变量都不应该持有一个指向具体类的指针或引用
- 任何类都不应该从具体类派生
- 任何方法都不应该覆写它的任何基类中已经实现的方法

依赖倒置原则

层次化：应避免较高层次直接使用较低层次。



依赖倒置原则

总结：

- 面向接口编程，不要针对实现编程
- 抽象不应该依赖于细节，细节应该依赖于抽象
- 高层模块和低层模块以及客户端模块和服务模块等都应该依赖于接口，而不是具体实现
- 从问题的具体细节中分离出抽象，以抽象方式对类进行耦合

依赖倒置原则有利于高层模块的复用，其正确应用对于创建可重用的框架是必须的。

接口分离原则

接口分离原则（Interface Segregation Principle，ISP）

> ***Clients should not be forced to depend upon interfaces that they do not use.***
>
> **R. Martin, 1996**

- 胖类会导致其客户程序之间产生不正常且有害的耦合关系。因此，客户程序应仅依赖于它们实际调用的方法。
- 将胖类的接口分解为多个特定于客户程序的接口，每个特定的接口仅声明其特定客户程序调用的那些函数，胖类则继承所有特定于客户程序的接口并实现之。

接口分离原则

举例：一个庞大的类

存在问题：

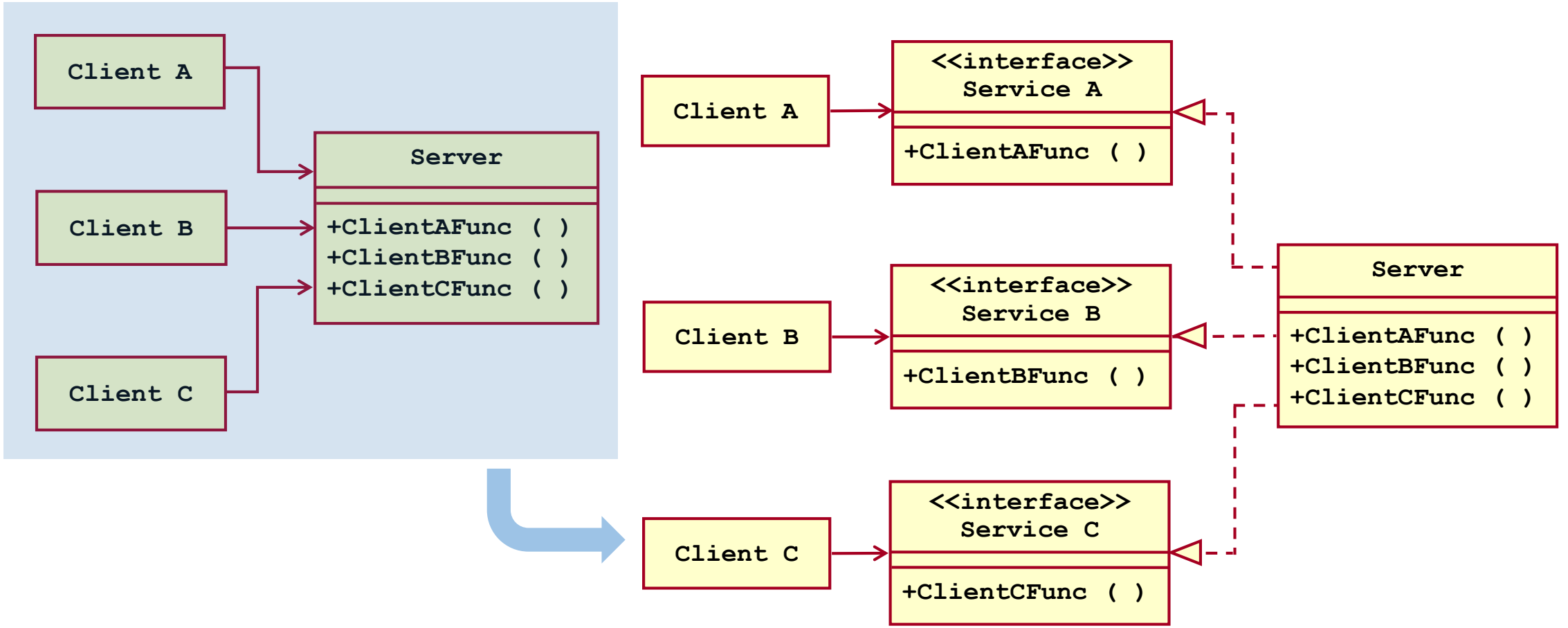
- 弄清楚需要修改什么以及修改会影响到什么是很困难的
- 可能会遇到多个开发人员同时进行不同修改的情况，这样会导致任务调度冲突
- 测试如此庞大的类是非常痛苦的

如何处理这种庞大的类?

ScheduledJob

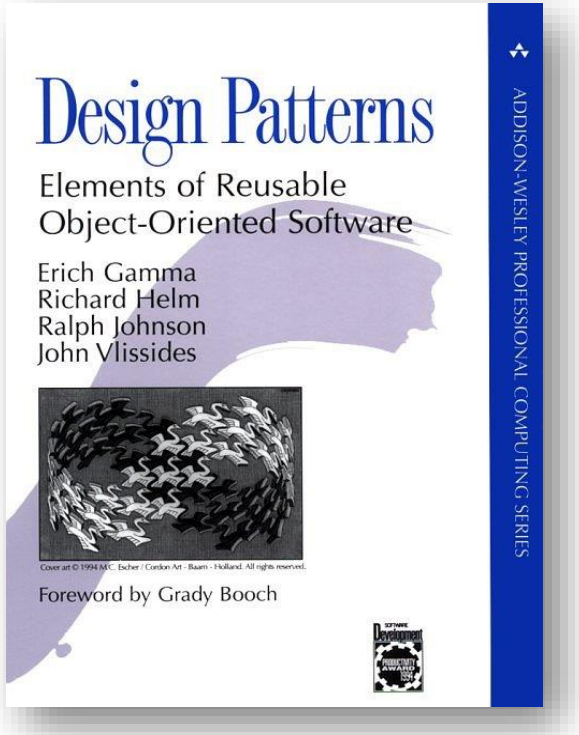
```
+addPredecessor ( ScheduledJob )
+addSuccessor ( ScheduledJob )
+getDuration( ) : int
+show ( )
+refresh ( )
+run ( )
+pause ( )
+resume ( )
+isRunning ( )
+postMessage ( ) : void
+isVisible ( ) :boolean
+isModified ( ) : boolean
+persist ( )
+acquireResources ( )
+releaseResources ( )
+getElapsedTime ( )
+getActivities ( )
……
```

接口分离原则


Client A
Client B
Client C
Server
+ClientAFunc ( )
+ClientBFunc ( )
+ClientCFunc ( )
Client A
<<interface>>
Service A
+ClientAFunc ( )
Client B
<<interface>>
Service B
+ClientBFunc ( )
Client C
<<interface>>
Service C
+ClientCFunc ( )
Server
+ClientAFunc ( )
+ClientBFunc ( )
+ClientCFunc ( )

Gamma 等人的设计模式

Gamma等四人提出的设计模式具有重大影响

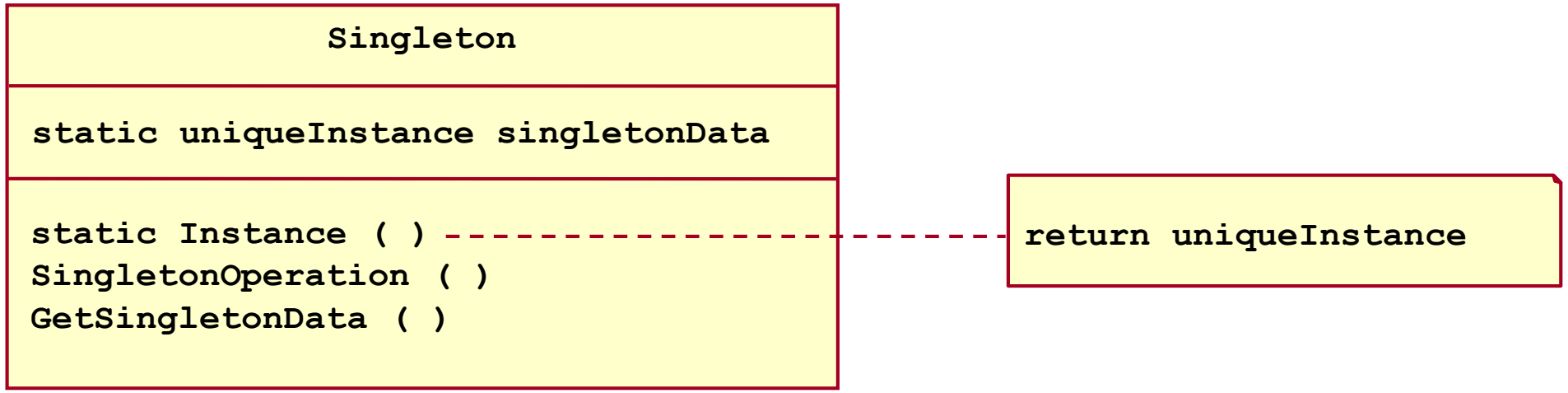


Gang of Four：
Gamma, Helm, Johnson, Vlissides

单件模式

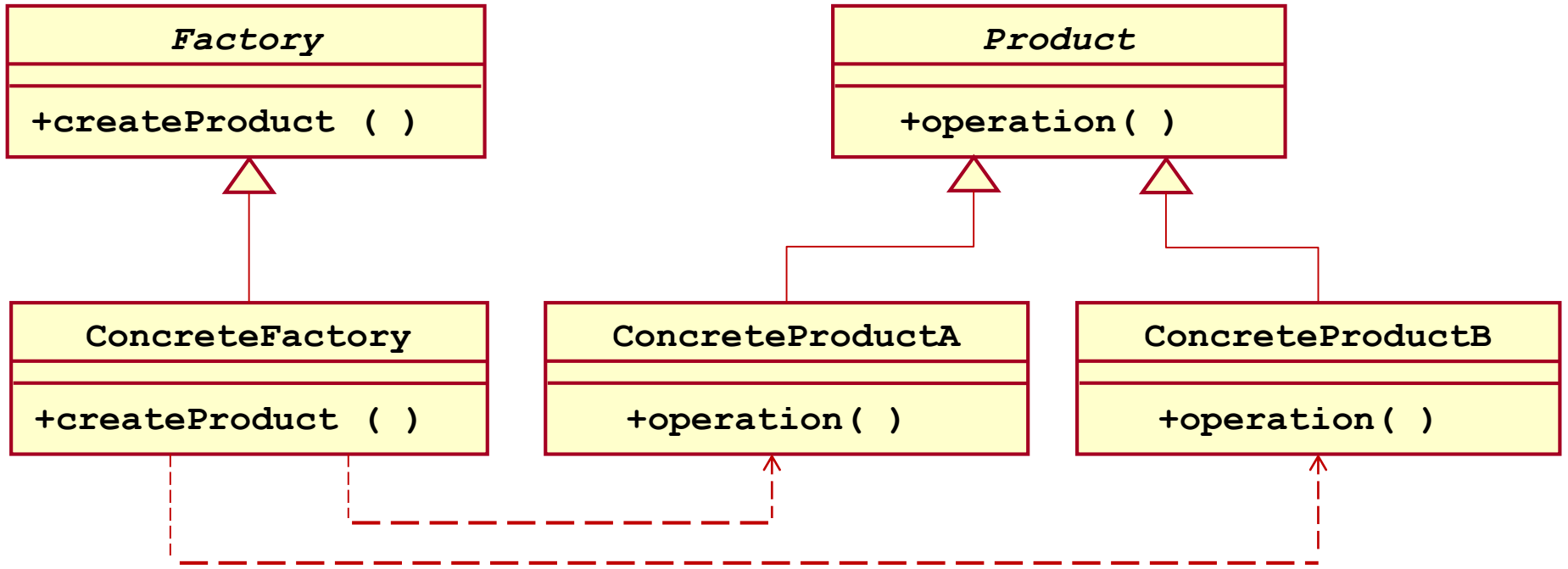
单件模式（Singleton）

- 用于确保整个应用程序中只有一个类实例且这个实例所占资源在整个应用程序中是共享时的程序设计方法
- 适用于需要控制一个类的实例数量且调用者可以从一个公共的访问点进行访问

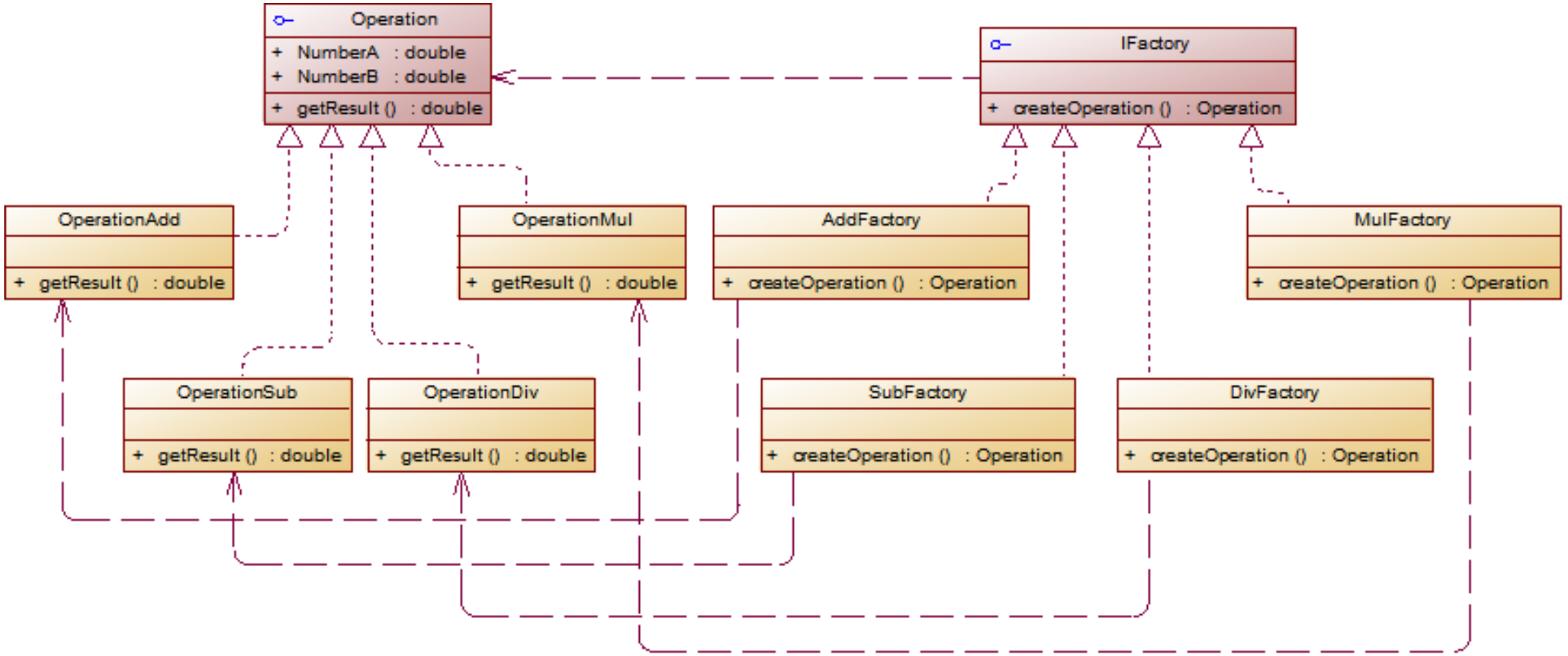


工厂方法（Factory Method）

工厂方法模式定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中，实现了开放封闭原则。

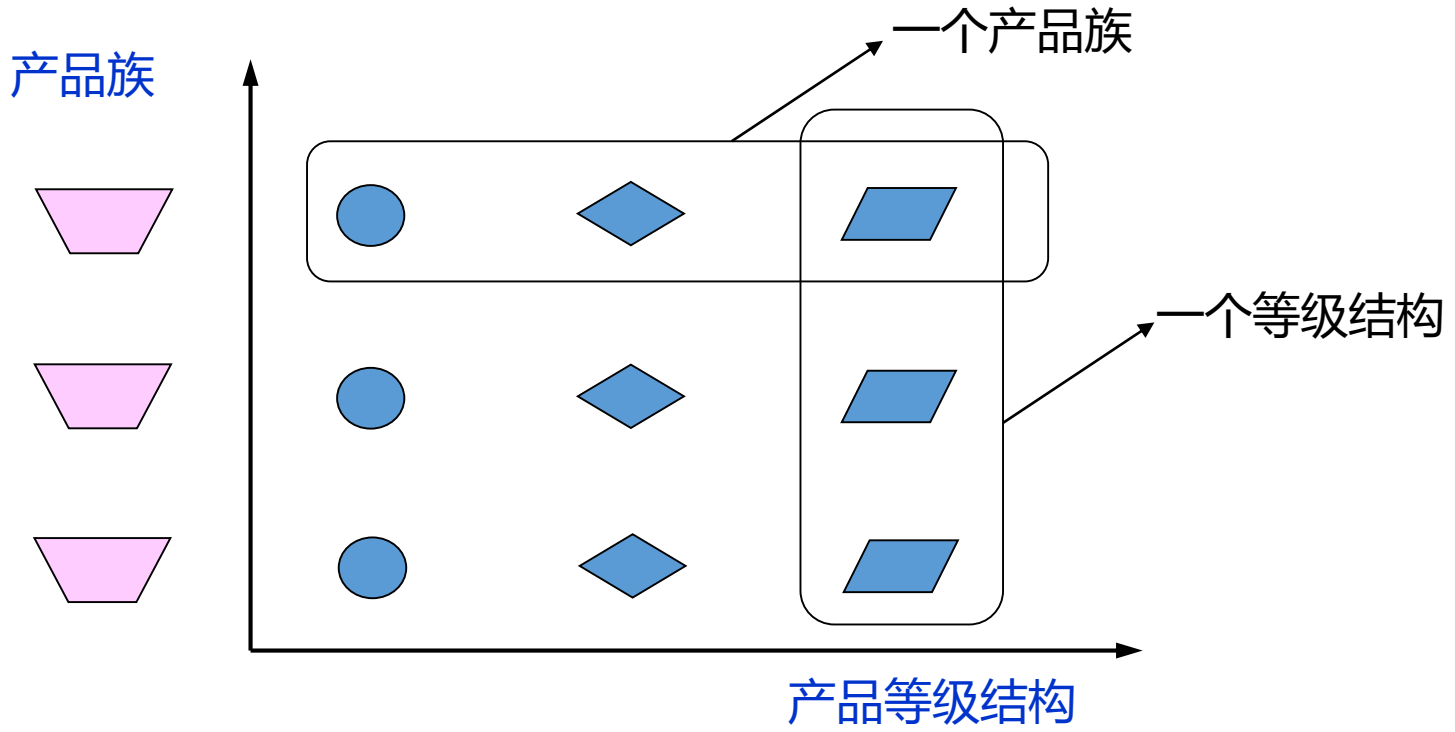


工厂方法（Factory Method）

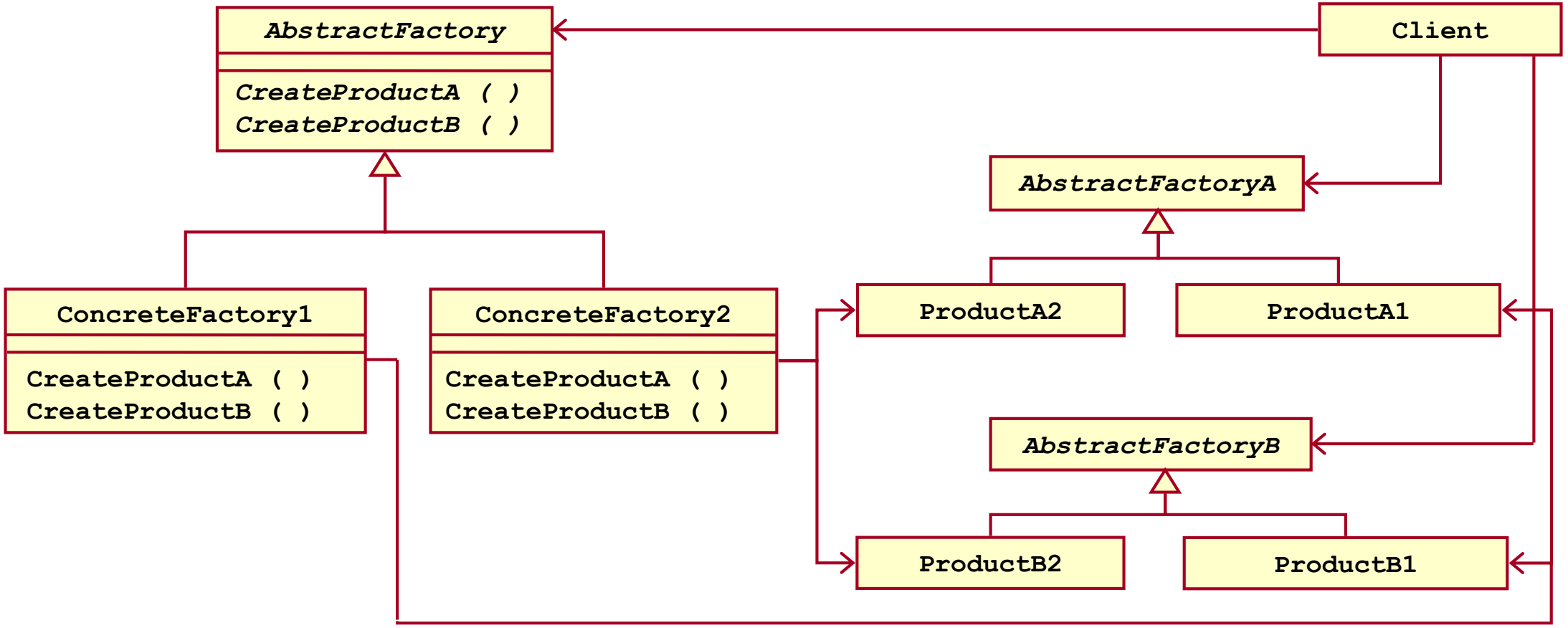


抽象工厂模式（Abstract Factory）

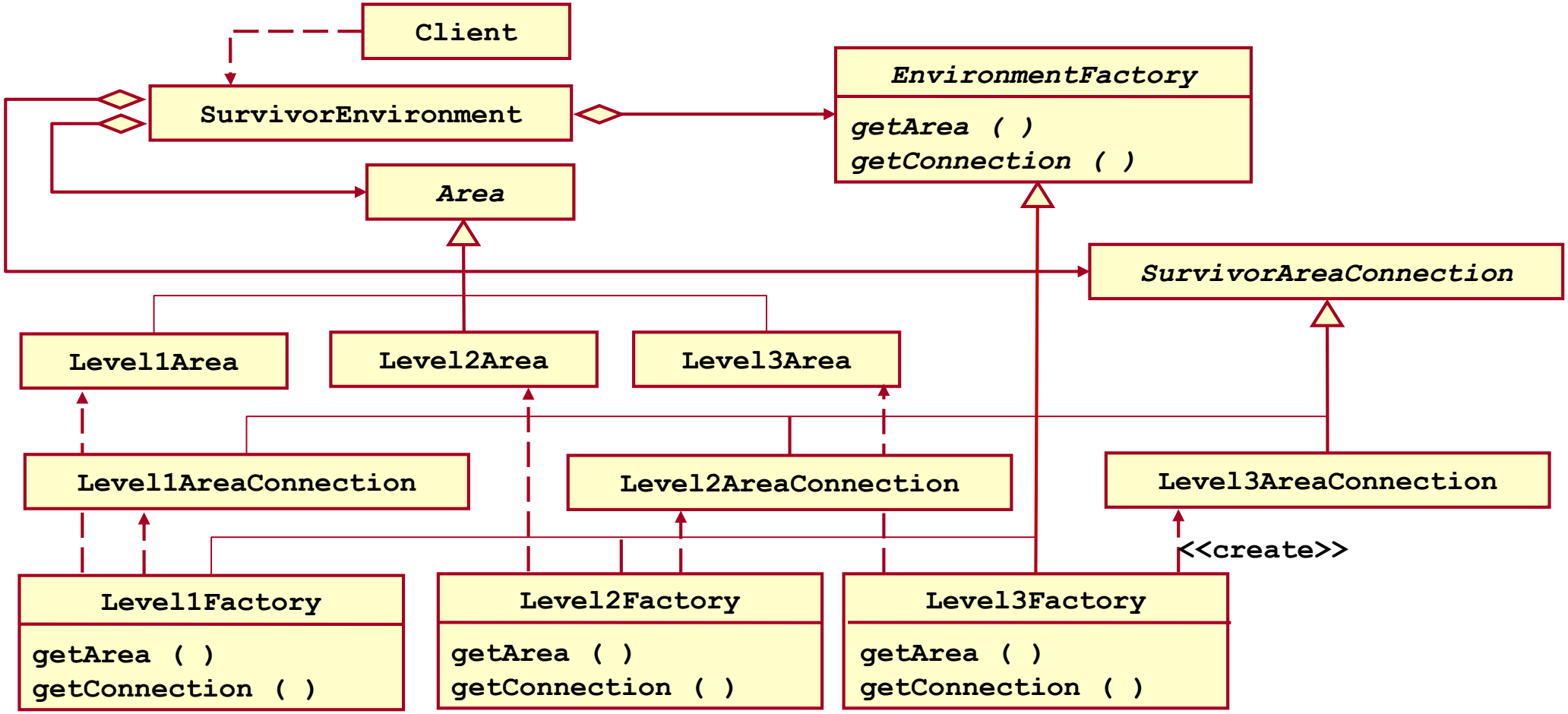
抽象工厂模式可以向客户端提供一个接口，使其在不必指定产品具体类型的情况下，创建多个产品族中的产品对象。



抽象工厂模式（Abstract Factory）

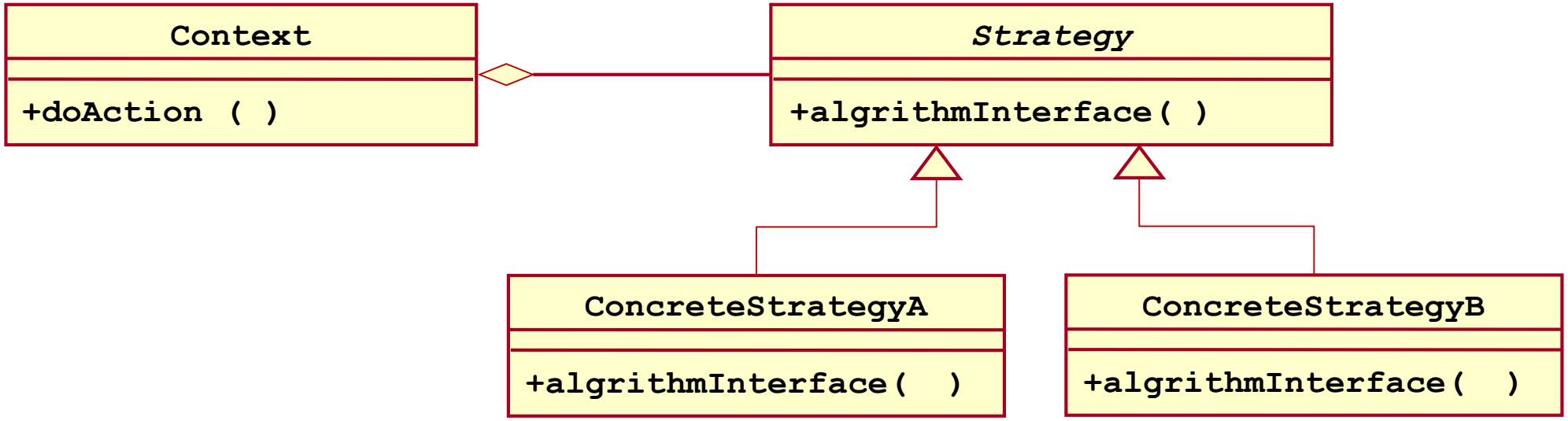


抽象工厂模式（Abstract Factory）

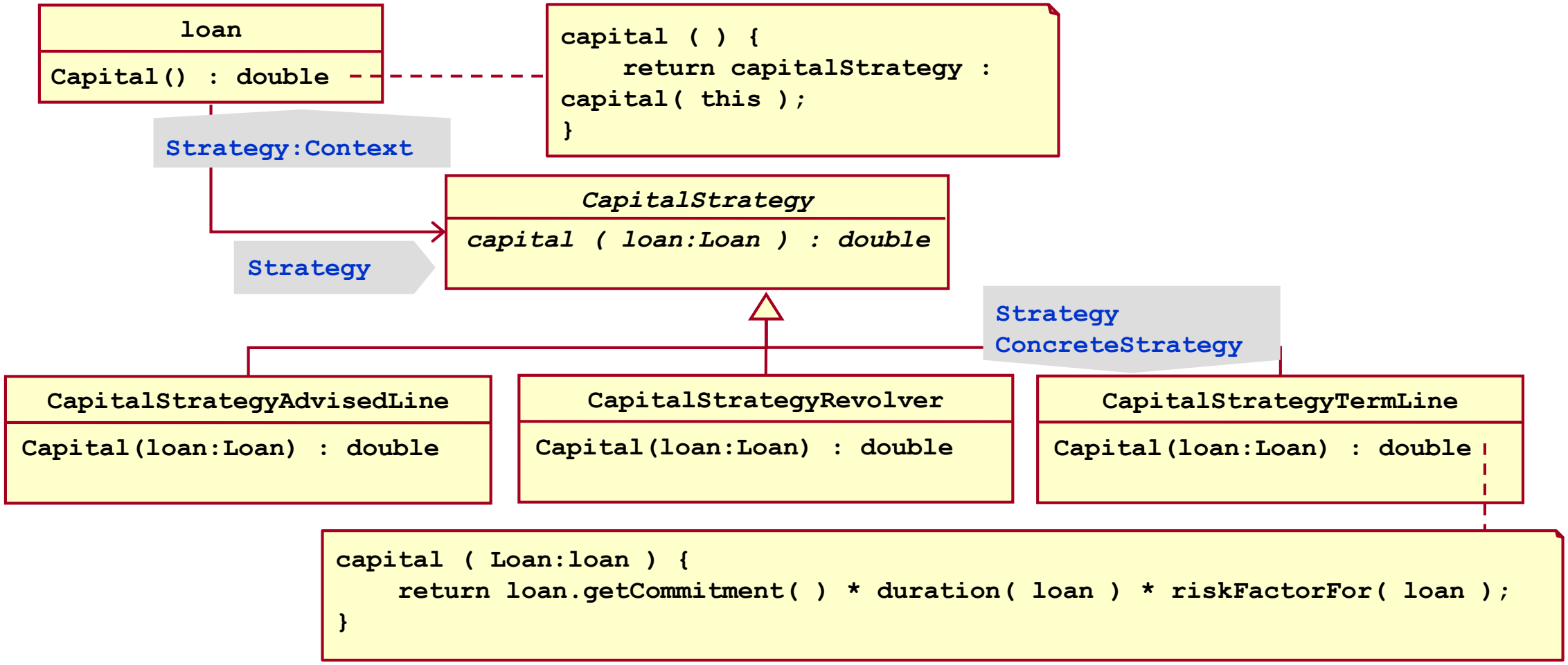


策略模式（Strategy）

策略模式（Strategy）可以有助于管理由于算法过多变体而产生的复杂度。

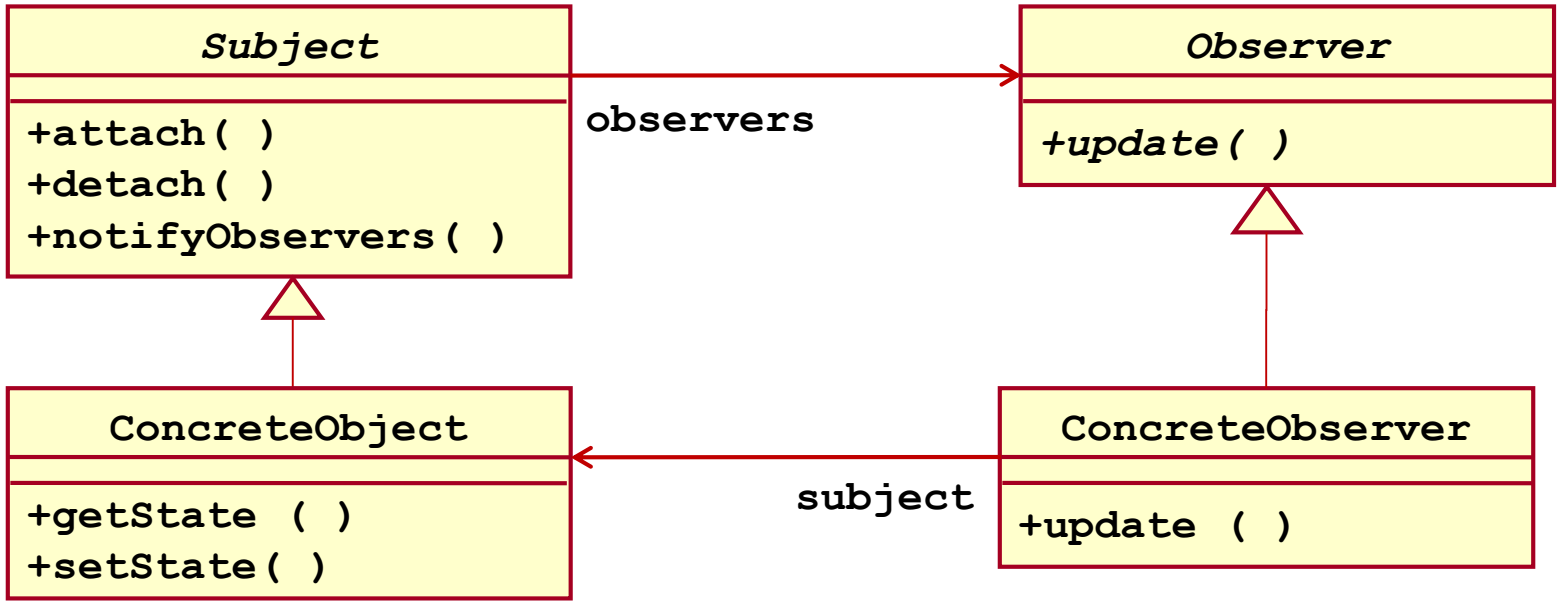


策略模式（Strategy）



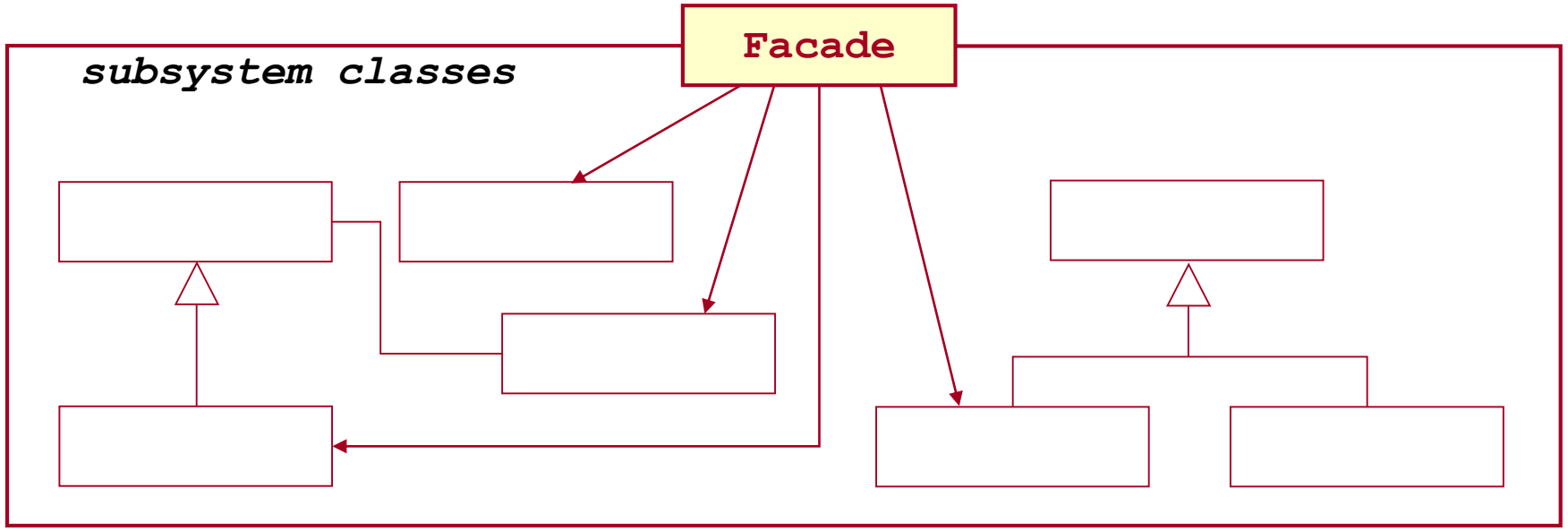
观察者模式（Observer）

观察者模式是发布—订阅体系结构风格的一种应用，其目的是定义一种一对多的依赖关系，当一个对象的状态发生变化时，所有依赖于它的对象都得到通知并被自动更新。

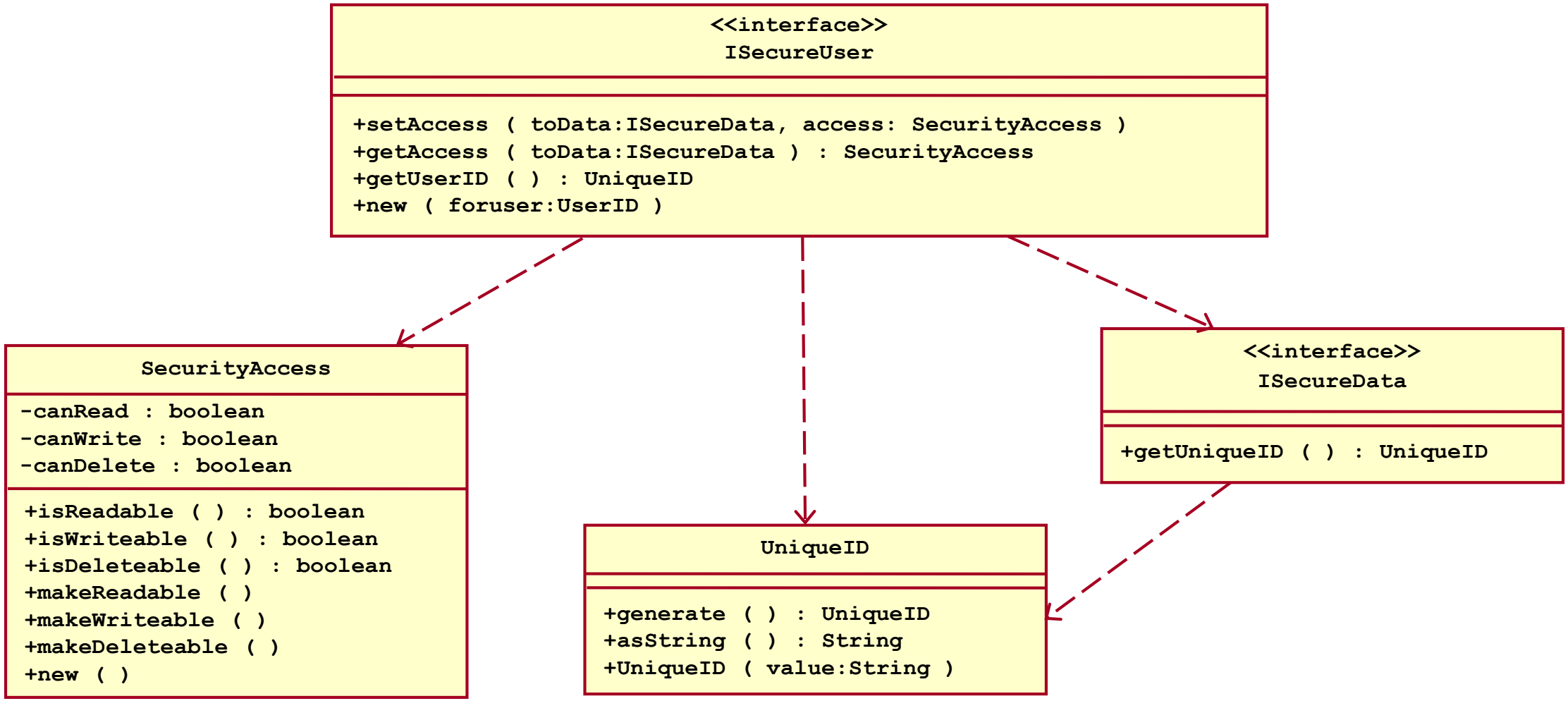


外观模式（Façade）

Façade模式用简单的统一接口封装子系统，从而降低类间的耦合性。单个Façade类通过调用低层类的方法，为子系统实现了高层接口。

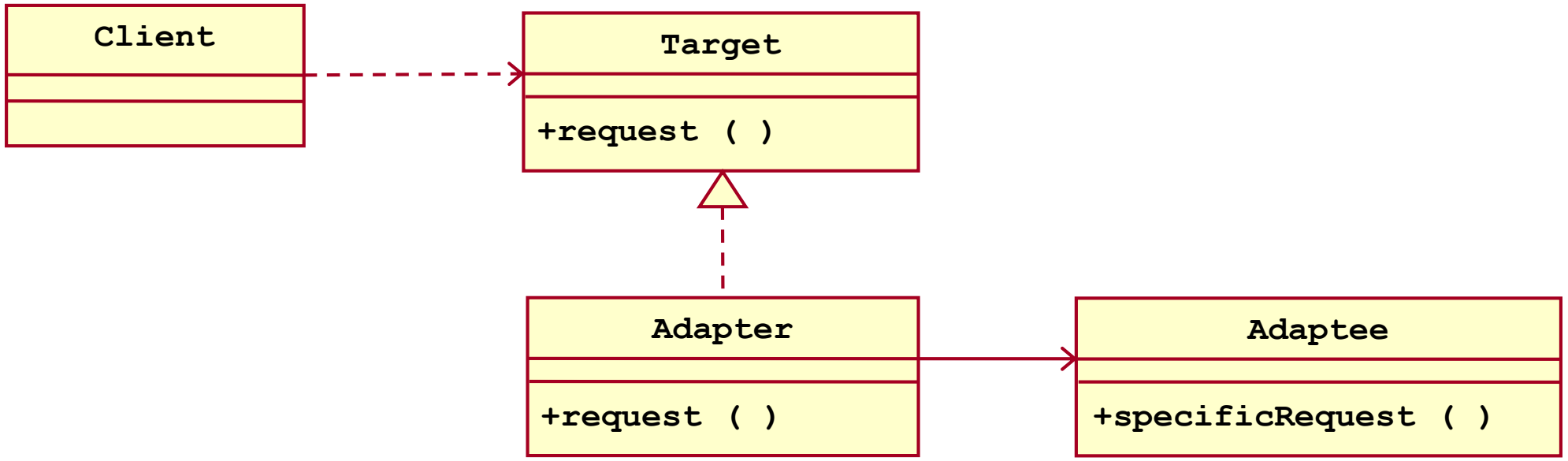


外观模式（Façade）

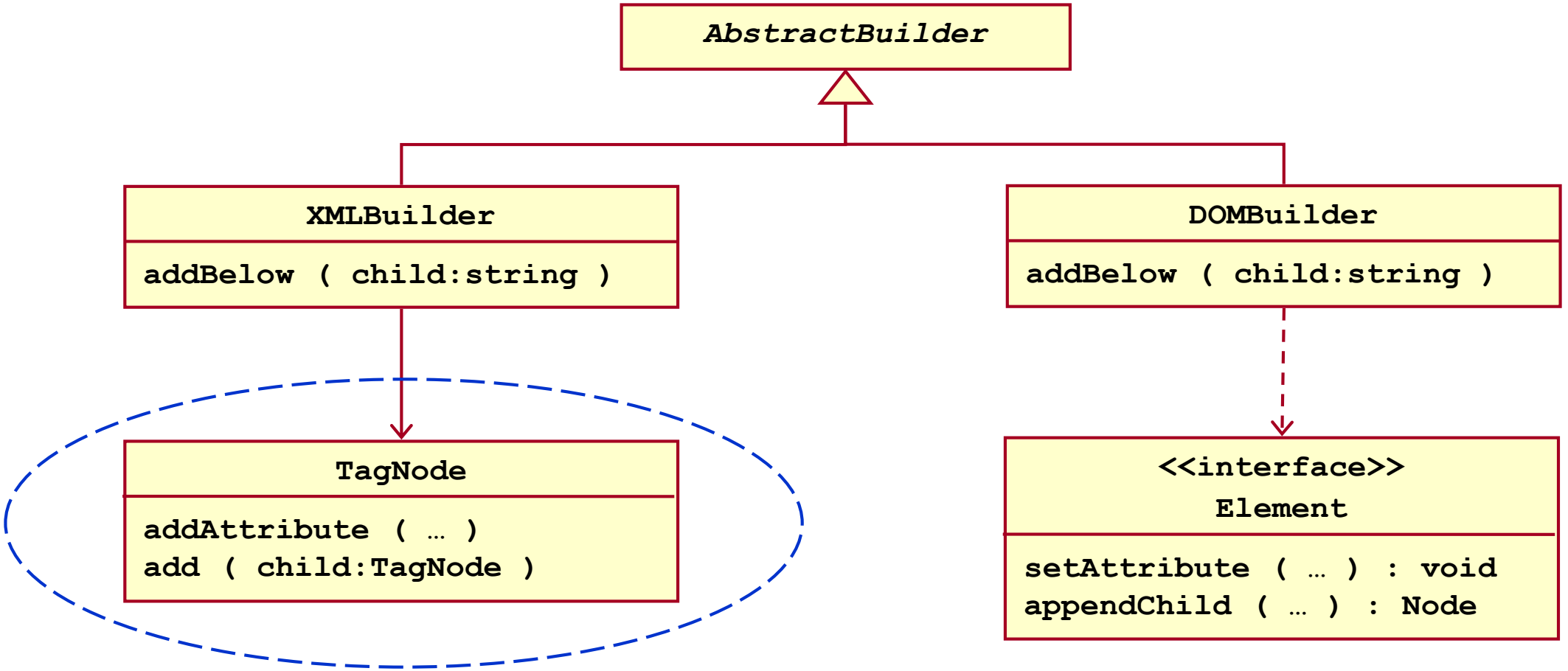

<<interface>>
ISecureUser
+setAccess (toData:ISecureData, access: SecurityAccess)
+getAccess (toData:ISecureData) : SecurityAccess
+getUserID ( ) : UniqueID
+new (foruser:UserID)
SecurityAccess
-canRead : boolean
-canWrite : boolean
-canDelete : boolean
+isReadable ( ) : boolean
+isWriteable ( ) : boolean
+isDeleteable ( ) : boolean
+makeReadable ( )
+makeWriteable ( )
+makeDeleteable ( )
+new ( )
UniqueID
+generate ( ) : UniqueID
+asString ( ) : String
+UniqueID (value:String)
<<interface>>
ISecureData
+getUniqueID ( ) : UniqueID

适配器模式（Adapter）

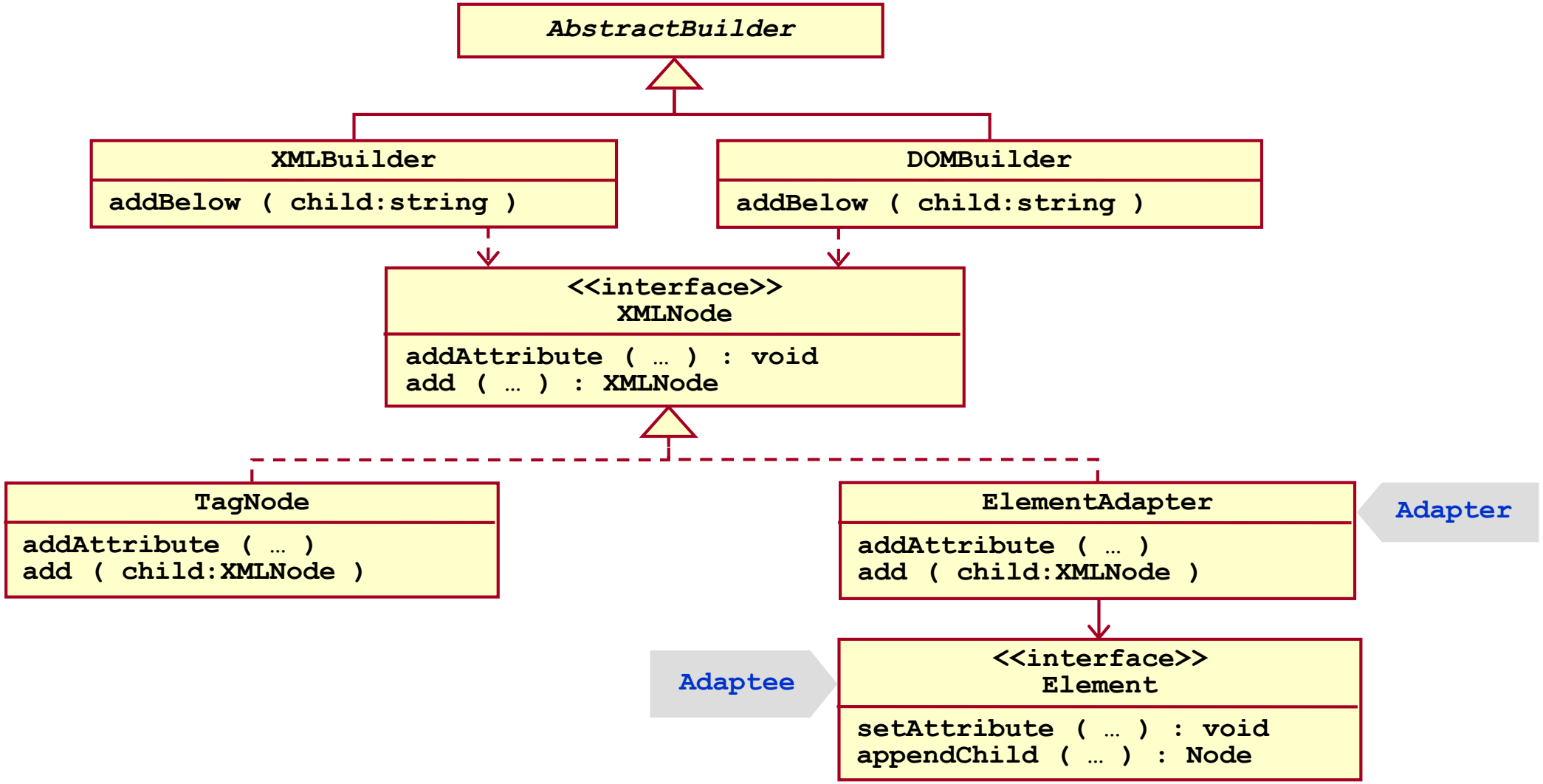
适配器模式将一个类的接口转换成客户希望的另一个接口，使原本由于接口不兼容而不能一起工作的类可以一起工作。



适配器模式（Adapter）



适配器模式（Adapter）



适配器模式（Adapter）

适用情形

- 两个类所做的事情相同或相似，但是具有不同的接口。
- 如果类共享同一个接口，客户代码会更简单、更直接、更紧凑。
- 无法轻易改变其中一个类的接口，它或是第三方类库的一部分，或是一个已经被其他客户代码广泛使用的框架的一部分，或无法获得源代码。

Adapter 与 Façade

- 二者的应用级别不同，其中Adapter模式用于适配对象，Façade模式用于适配子系统。

装饰模式（Decorator）

星巴克咖啡
订单管理系统

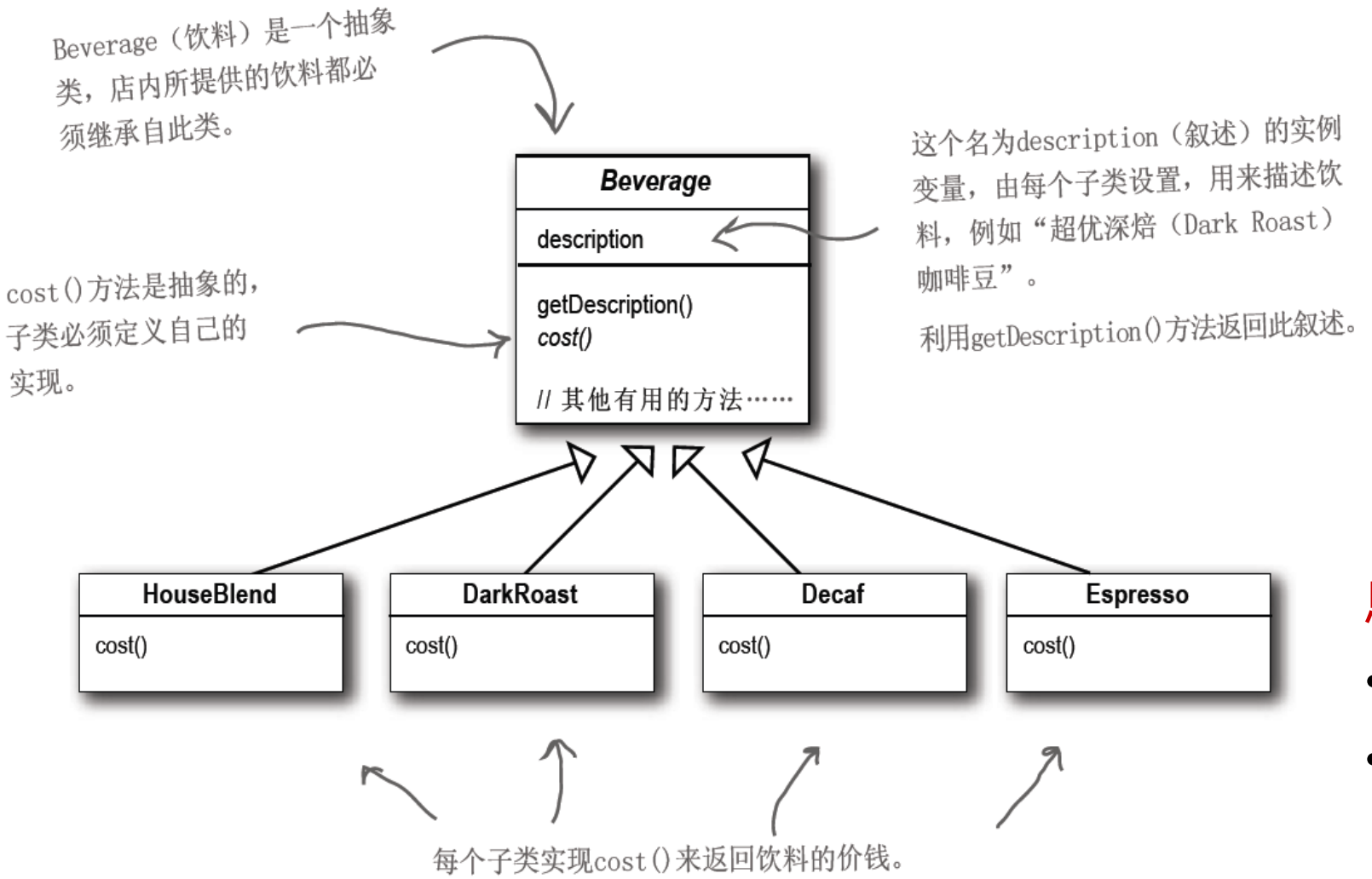
HouseBlend
家常咖啡

DarkRoast
深焙咖啡

Espresso
浓缩咖啡

Decaf
脱咖啡因咖啡

装饰模式（Decorator）

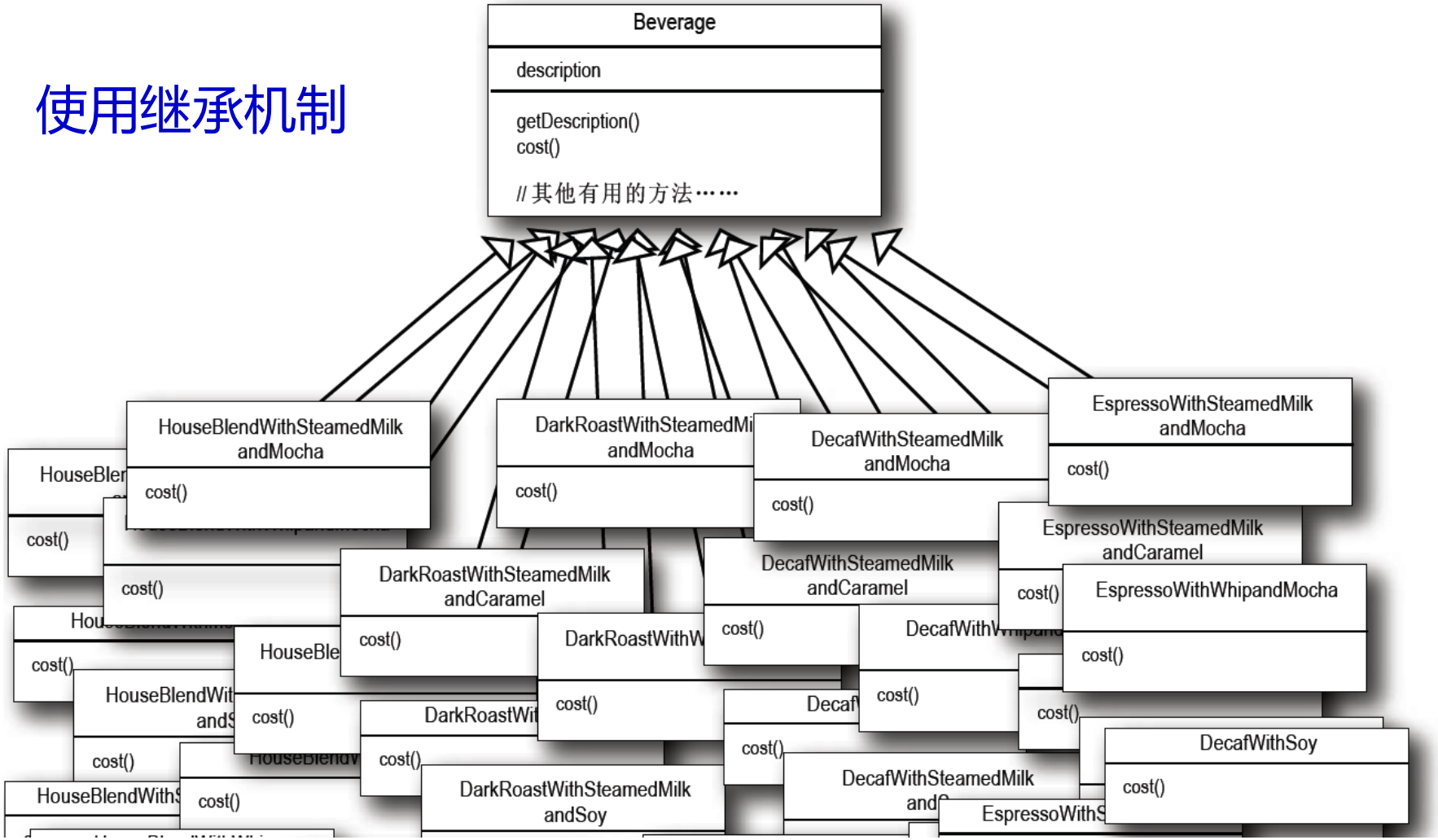


思考：

- 调料的增/删?
- 调料价格修改?

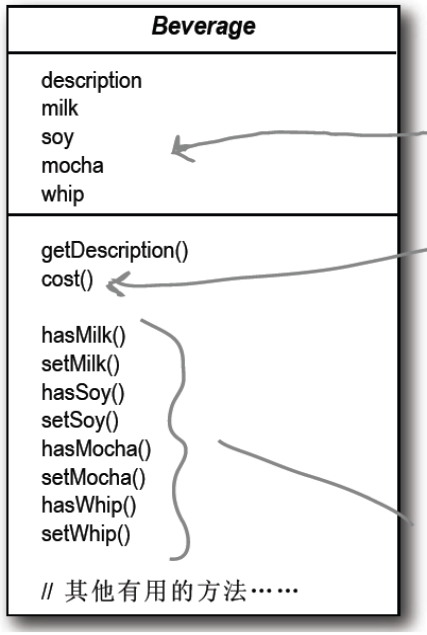
装饰模式（Decorator）

方法1：使用继承机制



装饰模式（Decorator）

方法2：使用实例变量和继承，跟踪这些调料



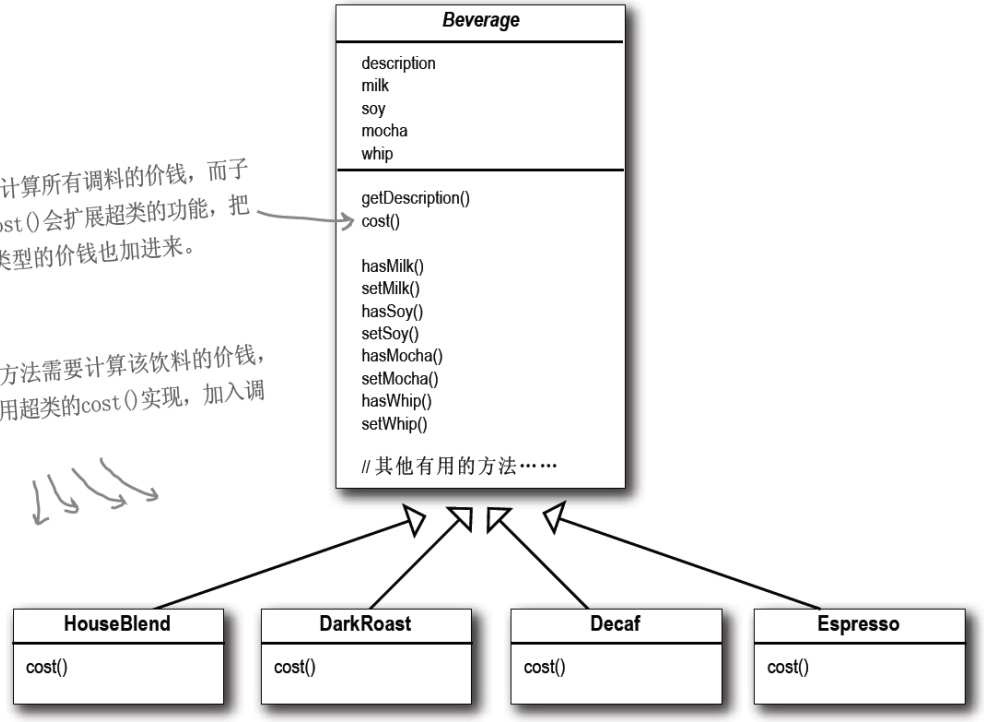
各种调料的新的布尔值

现在，Beverage类中的cost()不再是一个抽象方法，我们提供了cost()的实现，让它计算要加入各种饮料的调料价钱。子类仍将覆盖cost()，但是会调用超类的cost()，计算出基本饮料加上调料的价钱。

这些方法取得和设置调料的布尔值。

超类cost()将计算所有调料的价钱，而子类覆盖过的cost()会扩展超类的功能，把指定的饮料类型的价钱也加进来。

每个cost()方法需要计算该饮料的价钱，然后通过调用超类的cost()实现，加入调料的价钱。



装饰模式（Decorator）

方法2：使用实例变量和继承，跟踪这些调料

- 调料价钱的改变会使我们更改现有代码
- 一旦出现新的调料，我们就需要加上新的方法，并改变超类中的cost()方法
- 以后可能会开发出新饮料，对于这些饮料（冰茶）而言，某些调料可能并不适合，但是在这个设计方案中，Tea子类仍将继承那些不合适的方法，比如hasWhip()。
- 如何顾客想要双倍摩卡，怎么办?
- ……

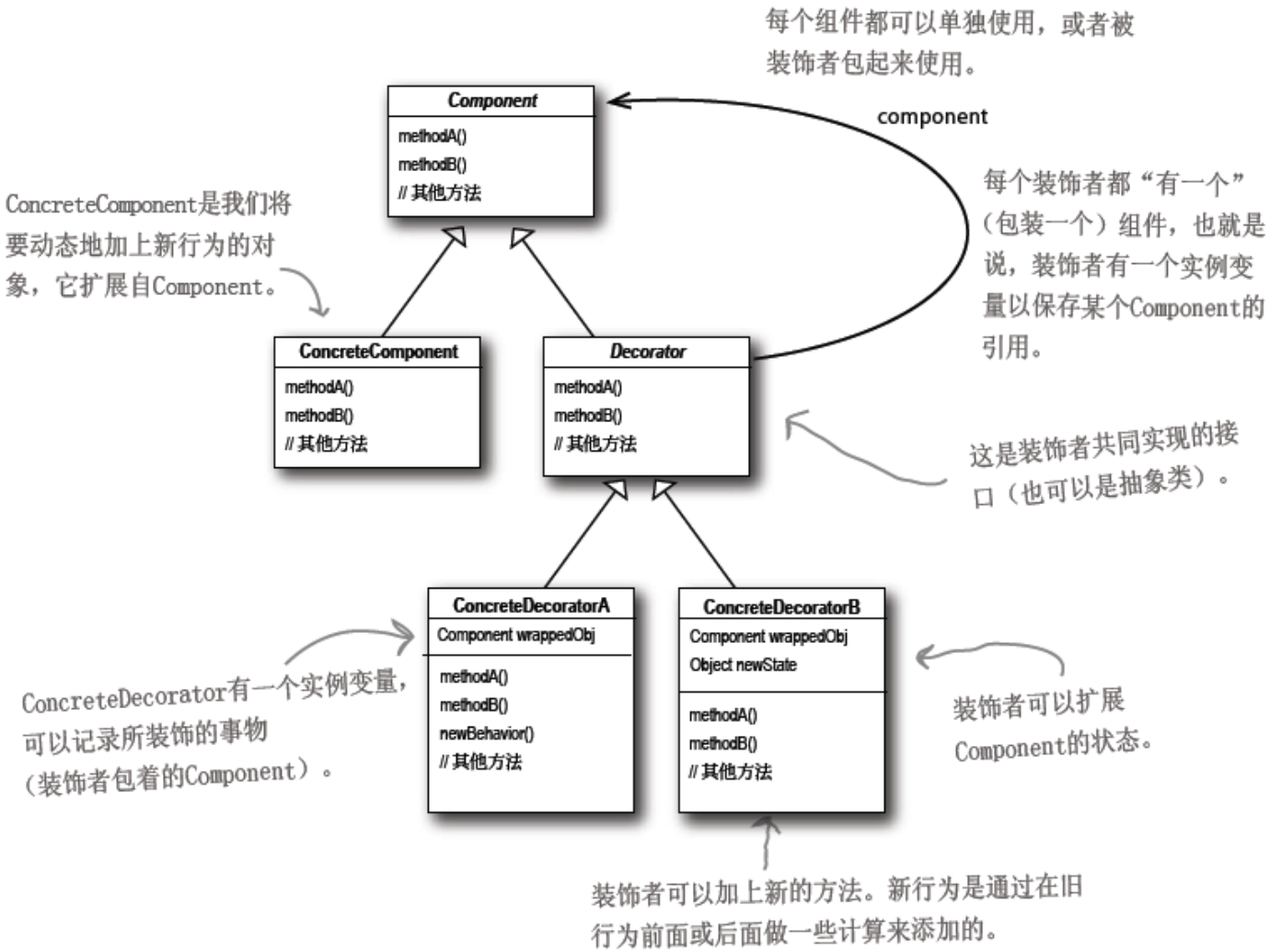
装饰模式（Decorator）

通常给一个类或对象增加行为的两种方式：

- 继承机制：通过继承一个现有类，可使子类在拥有自身方法的同时还拥有父类的方法。但是这种方法是静态的，用户不能控制增加行为的方式和时机。
- 关联机制：将一个类的对象嵌入另一个对象中，由另一个对象来决定是否调用嵌入对象的行为以便扩展自己的行为，这个嵌入对象称为装饰器（Decorator）。

装饰模式（Decorator）动态地给一个对象增加一些额外的职责，就增加对象功能来说，装饰模式比生成子类的实现更为灵活。

装饰模式（Decorator）

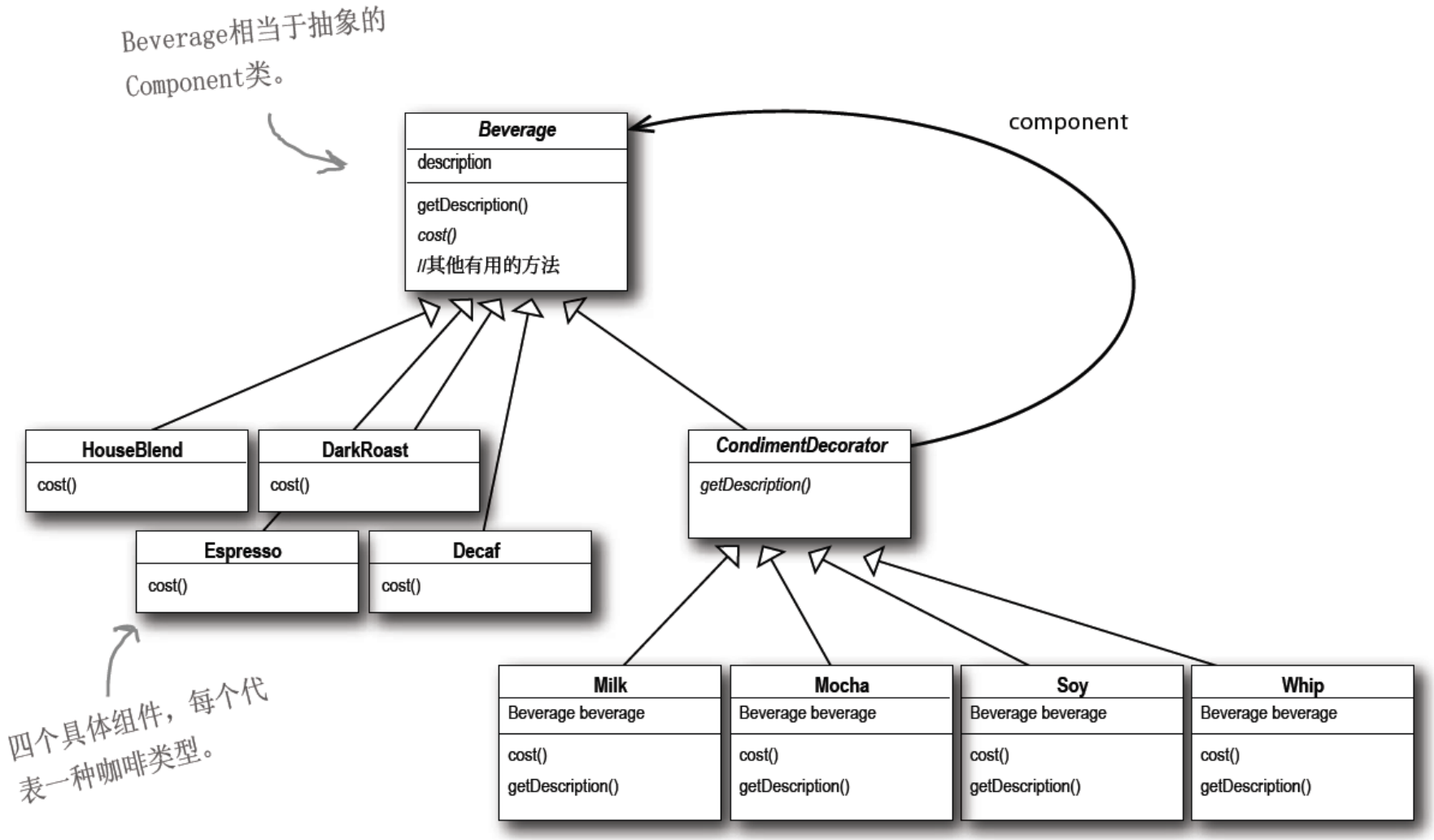


装饰模式（Decorator）

方法3：使用装饰模式

- 以饮料（Beverage）为主体，在运行时以调料（Condiment）来装饰饮料
- 举例：顾客想要摩卡和奶泡深焙咖啡
  - 取出一个深焙咖啡（DarkRoast）对象
  - 以摩卡（Mocha）对象装饰它
  - 以奶泡（Whip）对象装饰它
  - 调用cost方法，并依赖委托（delegrate）将调料的价格加上去

装饰模式（Decorator）



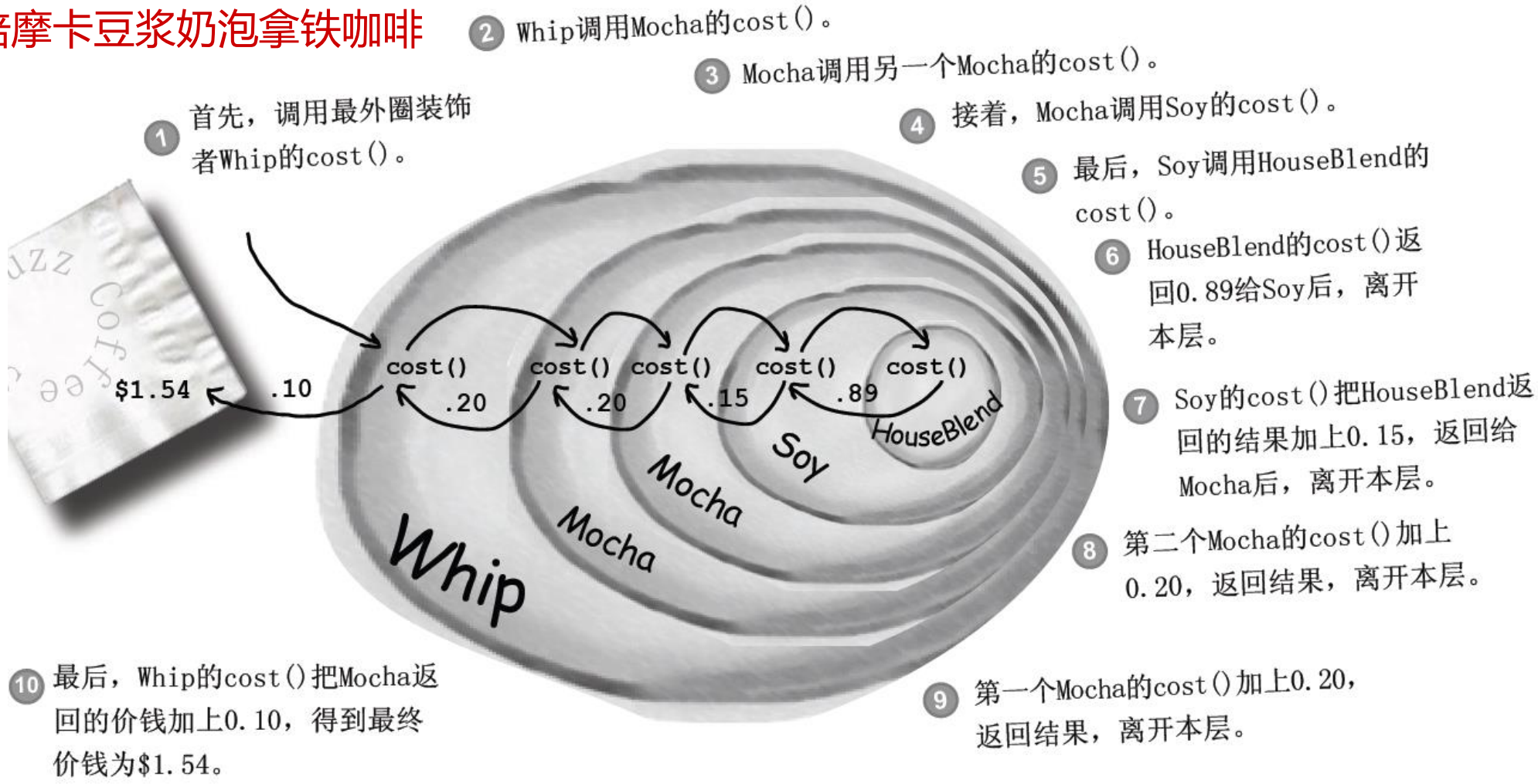
装饰模式（Decorator）

问题：为什么Decorate类扩展自Component类?

- 装饰者和被装饰者必须是一样的类型，在此使用继承达到“类型匹配”
- 类型匹配意味着装饰者和被装饰者具有相同接口，从而装饰者可以取代被装饰者
- 新的行为并不是继承自超类，而是由组合对象得到，即所有饮料和调料可以更有弹性地加以混合和匹配
- 可以在任何时候，实现新的装饰者增加新的行为；如果是依赖继承的话，每当需要新行为时就必须修改代码
- Component类型可以使用抽象类，也可以使用接口

装饰模式（Decorator）

订单：双倍摩卡豆浆奶泡拿铁咖啡



装饰模式（Decorator）

总结

- 开放封闭原则：应该允许行为可以被扩展，但无需修改现有代码
- 装饰者模式意味着一群装饰者类，这些类用来包装具体组件
- 装饰者反映出被装饰者的组件类型（具有相同的类型）
- 装饰者可以在被装饰者的行为前面或后面加上自己的行为，甚至将被装饰者的行为整个取代掉，而达到特定的目的
- 可以用无数个装饰者包装一个组件
- 装饰者一般对组件的客户是透明的，除非客户程序依赖于组件的具体类型
- 装饰者会导致设计中出现许多小对象，如果过度使用则会让程序变得复杂

谢谢大家！

THANKS