代码重构与模式

清华大学软件学院 刘强

教学提纲

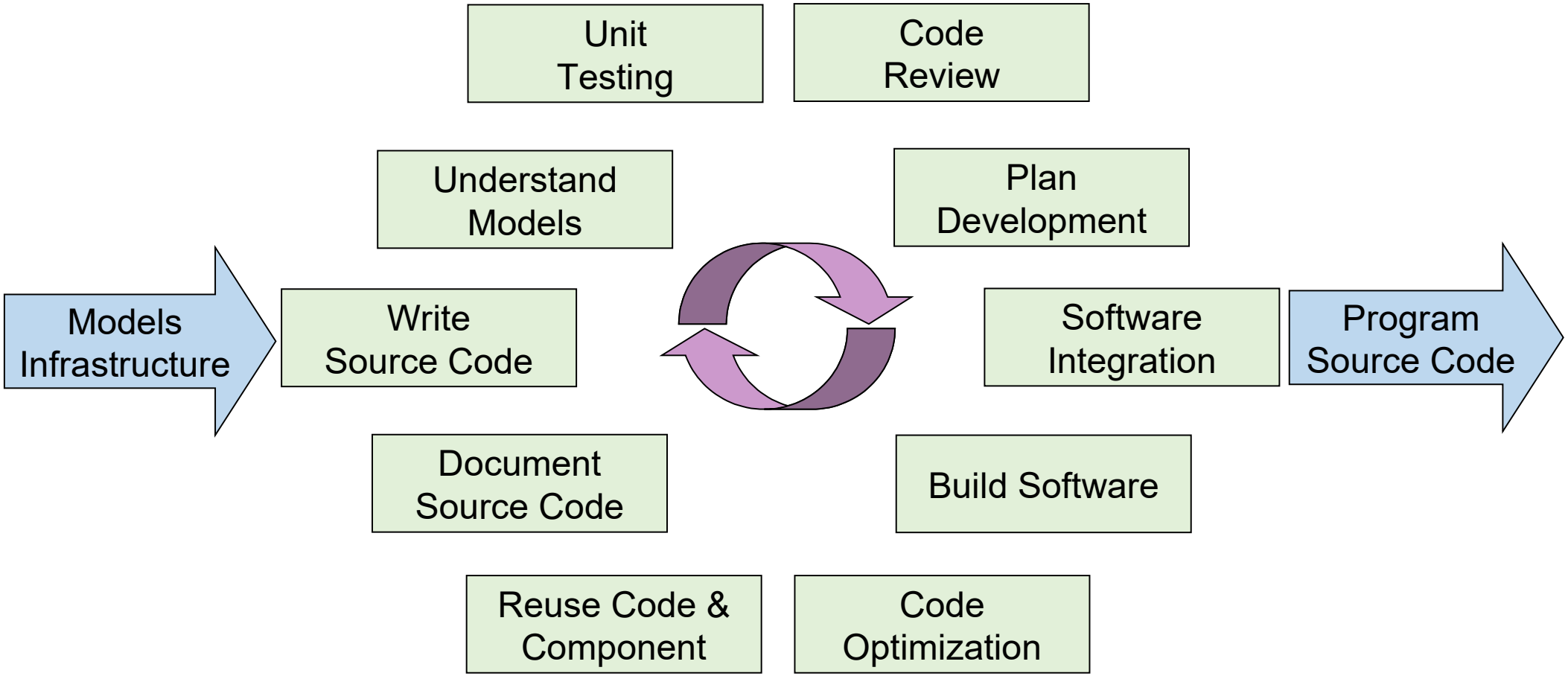
1. 编程原则与实践
2. 软件重构
3. 设计模式

教学提纲

1 □□□□□□□

- 软件编码工作
- 软件编码规范
- 代码走查
- 良好的编码实践

软件编码工作


Unit Testing
Code Review
Understand Models
Plan Development
Models Infrastructure
Write Source Code
Software Integration
Program Source Code
Document Source Code
Build Software
Reuse Code & Component
Code Optimization

软件编码工作

程序设计

- 理解软件的需求说明和设计模型
- 补充遗漏的或剩余的详细设计
- 设计程序代码的结构

设计审查

- 检查设计结果
- 记录发现的设计缺陷（类型、来源、严重性）

编写代码

- 应用编码规范进行代码编写
- 所编写代码应该是易验证的

软件编码工作

代码走查

- 确认所写代码完成了所要求的工作
- 记录发现的代码缺陷（类型、来源、严重性）

编译代码

- 修改代码的语法错误

测试所写代码

- 对代码进行单元测试
- 调试代码修改错误

软件编码规范

软件编码规范是与特定语言相关的描写如何编写代码的规则集合。

目的

- 提高编码质量，避免不必要的程序错误
- 增强程序代码的可读性、可重用性和可移植性

□□□□□□ **https://code.google.com/p/google-styleguide/**

代码检查

□□□□□Code Review□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□

□□□□□□□

- □□□□□□□□
- □□ Backup
- □□□□□□□□□□
- □□□□□□□
- □□□□□□“□□□”

代码检查

非正式 正式

桌面检查 代码走查 代码审查

由一个程序员人工阅读代码，通过对源程序代码的分析和检验来发现程序中的错误。

设计或编程人员组成一个走查小组，通过阅读一段文档或代码并进行提问和讨论，发现可能存在的问题。

若干编程人员和测试人员组成一个审查小组，以会议形式通过阅读、讨论和争议对程序进行静态分析。

缺陷检查表

□□□□

•□□

□□□□□□

•□□□□□□□□□□□

•□□□□□□□□□□□□□

•□□□□□□□□□□□

□□□□

•□□

•□ Hashtable □ Vector □□□□□□□□□□□□□□□□□□

•□□□□ String □□□□□

•□□□□□□□□□□□□□□□□□□□□□□□□□□

•□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□

•□□□□□□□□□□□□□□□□□□□

•□□□□□□□□□□

•I/O □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ buffer □□□□□□

•□□□□□□□□□□□□□□□□□□□

•□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

•□□□□□□□□□□□□□□□□□□□□□□

•□□□□□□□□□□□□□□□□□□□□ profile □□

□□□□□□

•□□□□□□□□□□□□□□□□□□□□□□ C/C++ □

•□□□□□□□□□□□□□□□□□□□□□□ Socket □□□□

•□□□□□□□□□□□□□□ C/C++ □

•□□□□□□□□□□□□□□□

缺陷检查表

□□□□

- □□□□□□□□□□
- □□□□□□□□□□□
- □□□□□□□□□□□□□□□□□□□□□□

□□□□

- □□□□□□□□□□□□□□□□□□
- □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
- □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
- □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ A □□□ 1 □□□□ 2 □□□ B □□□ 2 □□□□ 1
- □□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□

- □□□□□□ SQL □□□□□□□□□□□□□□□□□□□□
- □□□□□□□□□□□□□□

□□□□□□□□

- □□□□□□□□□□□□□□□□□□□□□□□
- □□□□□□□□□□□□□
- □□□□□□□□□□□□□
- □□□□ PreparedStatement □□□□□

□□□□

- Socket □□□□□□□□□□□□
- □□□□□□□□□□□□□□□□
- Socket □□□□□□□□□
- □□□□□□□□□□□

JAVA □□□□

- □□□□□□□□□□□□□□□□□□□□□□ null □□□□
- □□□□□□□□□□□□□□□□□□ clone □□□□□□□□
- □□□□□□□□□□□□□
- □□□□□□□□□□□□□□□□
- □□□□□□□□□□□□□□□□□□□□□□

缺陷检查表

[illegible] • [illegible] • [illegible] • [illegible] • [illegible] • [illegible] • [illegible] [illegible] • [illegible] • [illegible]	[illegible] • [illegible] • [illegible] • [illegible] API [illegible] [illegible] • [illegible] •WEB [illegible] • [illegible] • [illegible] [illegible] • [illegible] • [illegible]

良好的编程实践

不要编写需要外部文档支持的代码，这样的代码是脆弱的，要确保你的代码本身读起来就很清晰。

应该如何做到这一点？

说说你的做法。。。

良好的编程实践

编写自文档化的代码

- 唯一能完整并正确地描述代码的文档是代码本身
- 编写可以阅读的代码，其本身简单易懂

□□□□□□□□□

练习中简单函数的编写质量如何？

如何修改成自文档化代码？

良好的编程实践

学会只编写够用的注释，过犹不及，重视质量而不是数量。应该把时间花在编写不需要大量注释支持的代码上，即让代码自文档化。

- 好的注释解释为什么，而不是怎么样
- 不要在注释中重复描述代码
- 当你发现自己在编写密密麻麻的注释来解释代码时，需要停下来看是否存在更大的问题
- 想一想在注释中写什么，不要不动脑筋就输入，写完之后还要在代码的上下文中回顾一下这些注释，它们是否包含正确的信息?
- 当修改代码时，维护代码周围的所有注释

良好的编程实践

函数编写的第一条规则是短小，第二条规则是更短小。
函数应该做一件事，做好这件事，并且只做这件事。

编编编编编编编编编编

示例程序做了几件事？

如何改写上述程序？

良好的编程实践

[illegible]

- [illegible] main [illegible]
- [illegible]
- [illegible]

练习：依赖注入

良好的编程实践

浮点运算

- [illegible]
- [illegible]
- [illegible]
- [illegible]

  0.1_{10} = 0.1000000000000000055511151231257827021181583404541015625

- [illegible] 2 [illegible] 2 [illegible] 4 [illegible] 8 [illegible] 16 [illegible]…[illegible]

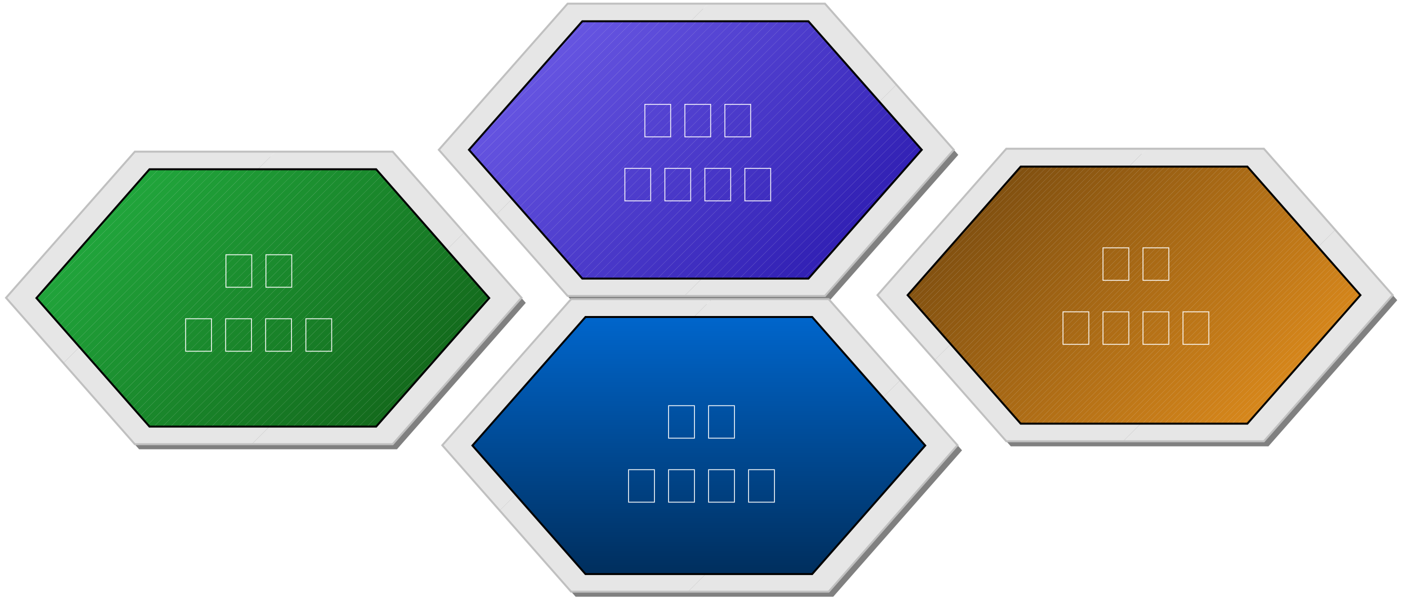
教学提纲

2 □□□□

- 软件重构概述
- 代码的坏味道
- 软件重构方法

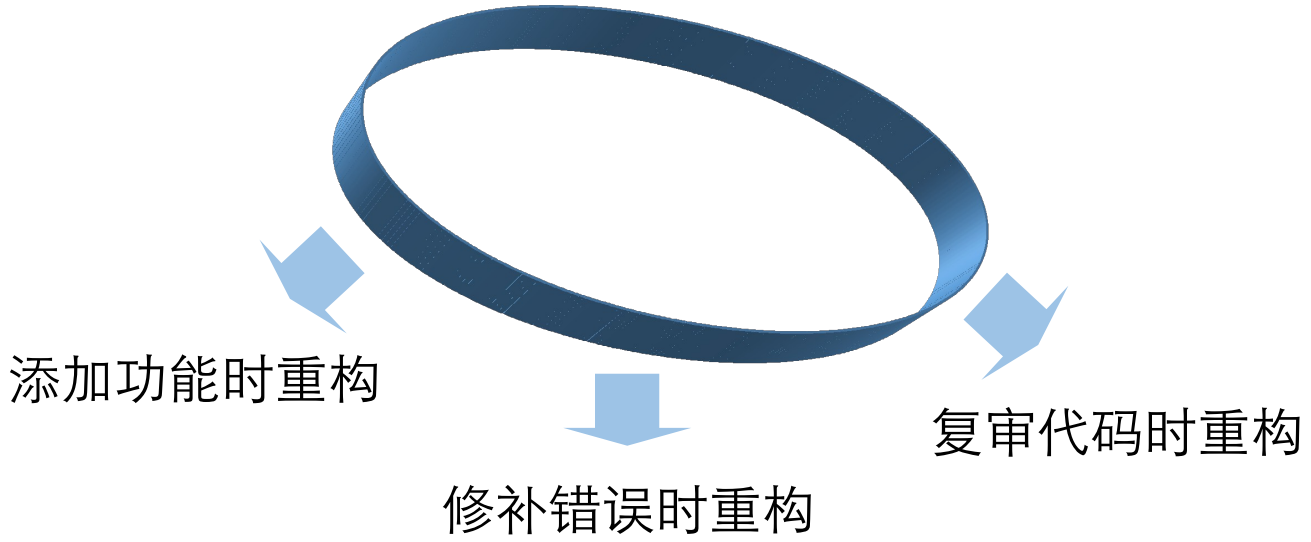
重构的概念

[illegible] Refactoring [illegible]



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？

□□□□□□□□□□□□

- □□□□□□□□□□

□□□ Deadline

- □□□□□ Last-Minute-Change □□□□□□□□□□□□□□□□ Production □□□□□□□□□□

□□□□□□□□□□□□□

- □□□□□□□□□□ 3 □□□□□□□□□□□□□□□□□□
- □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

重构与添加新功能

添加新功能

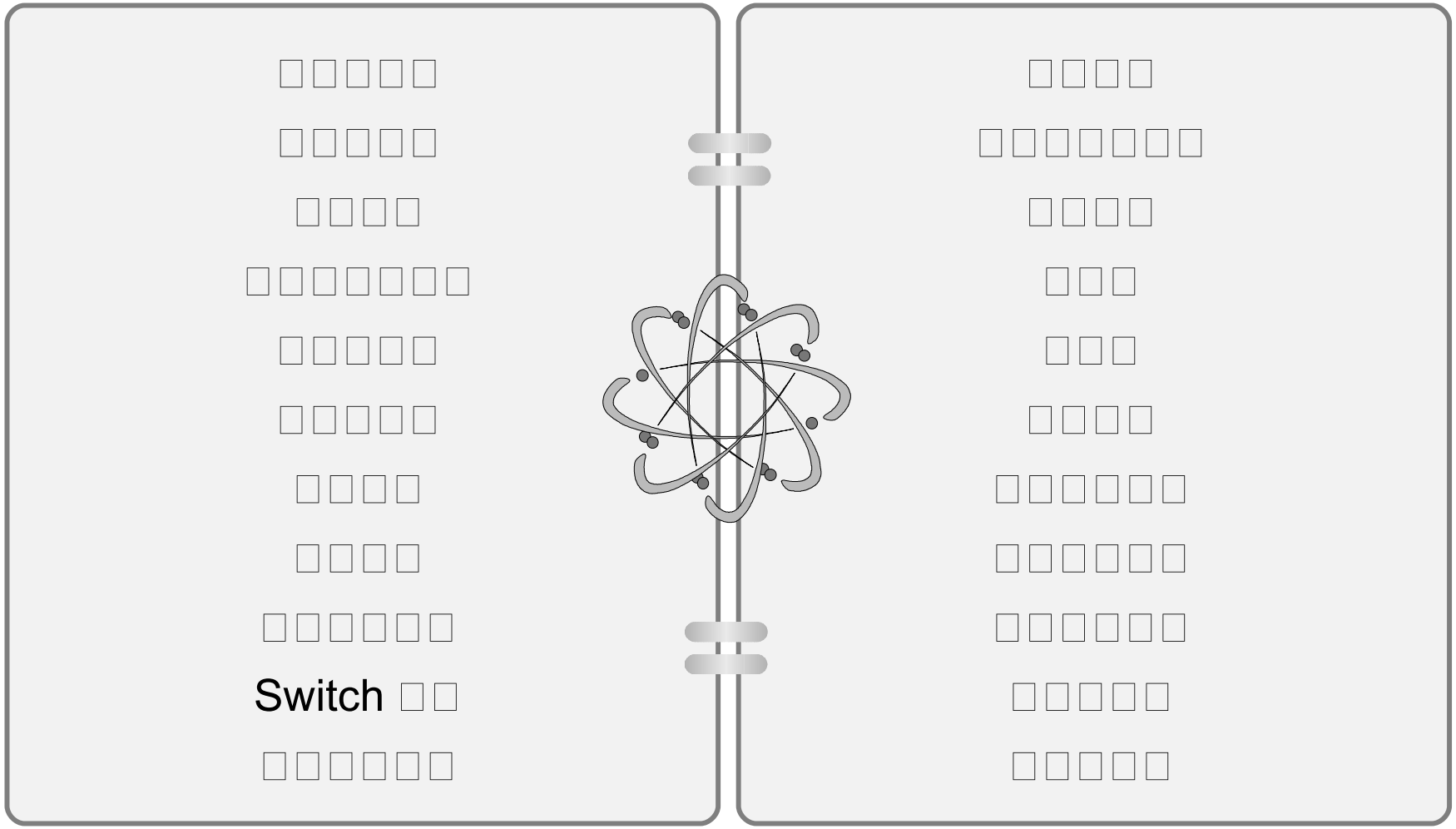
- 添加新功能时，不应该修改既有代码，只管添加新功能

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

代码的坏味道



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：如果同一个类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。
- 抽取类：如果两个毫不相关的类存在重复代码，则将重复代码抽取成到一个独立的类中或者某个类中，其他类引用这个类。
- 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。

重复的代码

```
protected void queryBtn(object sender, EventArgs e) {
      // □□□□□□□□□□
        if (this.xmbhText.Text.ToString().Trim().Equals("") == false) {
        // □□□□□□□□□——□□□□□□□□□□□□□
                Byte[] MyBytes = System.Text.Encoding.Default.GetBytes(
                        this.xmbhText.Text.ToString().Trim());
        if (MyBytes.Length > 10) {
            MessageBox.Alert(" □□□□□□□□ 10 □□□□ ");
            this.xmbhText.Focus();
            return;
        }
    }
    // □□□□□□□□□□
    if (this.xmmcText.Text.ToString().Trim().Equals("") == false) {
        // □□□□□□□□□——□□□□□□□□□□□□□
                Byte[] xmmccheck = System.Text.Encoding.Default.GetBytes(
                        this.xmmcText.Text.ToString().Trim());
        if (xmmccheck.Length > 40) {
                MessageBox.Alert(" □□□□□□□□ 40 □□□□ ");
                this.xmmcText.Focus();
                return;
        }
    }
    // □□□□□□□
        GridView1_DataBind();
}
```

过长的函数

[illegible]

- [illegible]N[illegible]10[illegible]
- [illegible]

[illegible]

- [illegible]
- [illegible]
- [illegible]

[illegible]

- [illegible]

发散式变化

症状：

•某个类因为不同的原因在不同的方向上发生变化

措施：

•如果类既要找到对象，又要对其做某些处理，则让调用者查找对象，并将该对象传入，或者让类返回调用者所用的值。

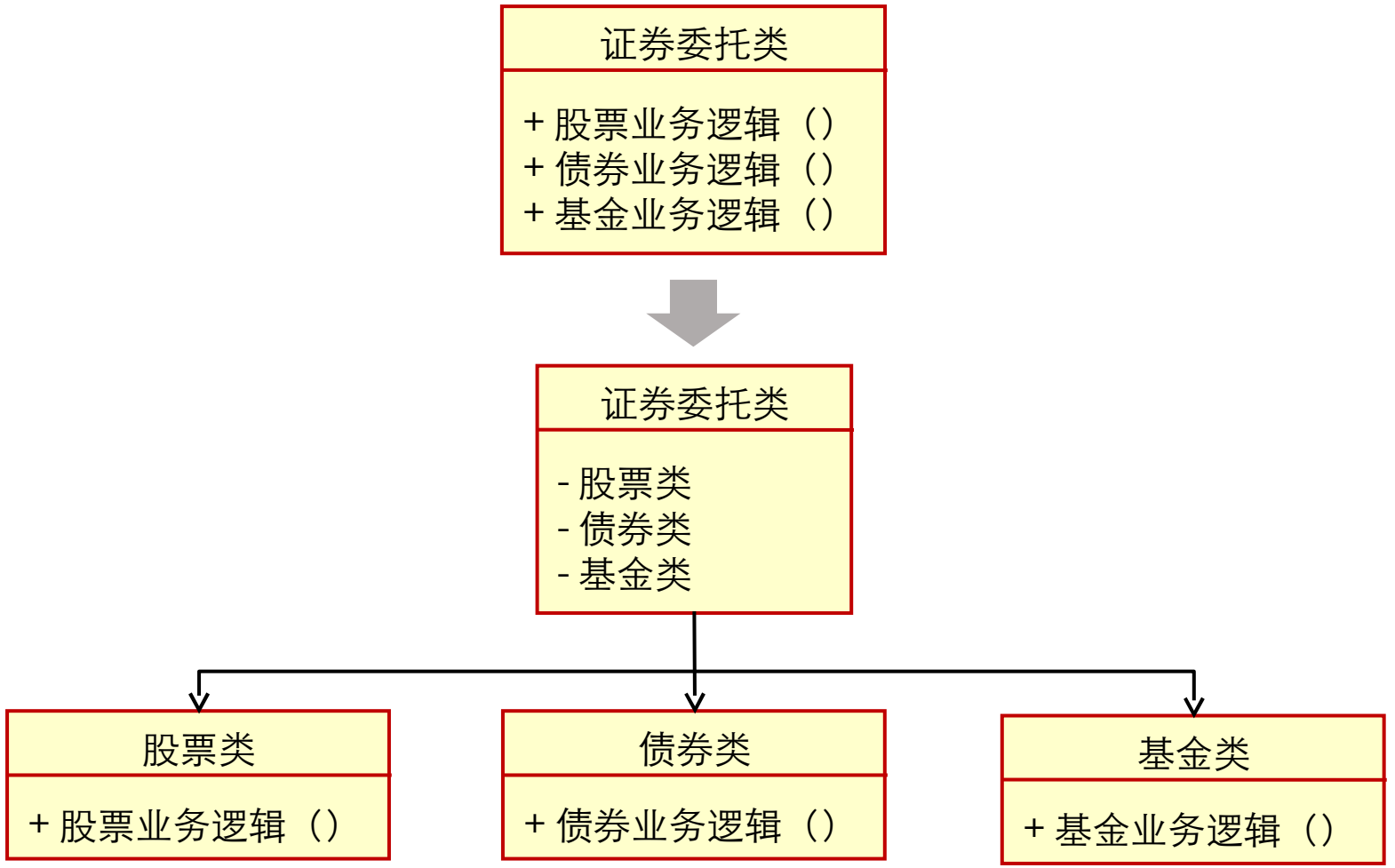
•采用抽取类的方法，为不同的决策抽取不同的类。

•如果多个类共享相同类型的决策，则可以合并这些新类。至少这些类可以构成一层。

说明：

•发散式变化指的是“一个类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，需要修改多个类的多个地方

措施：

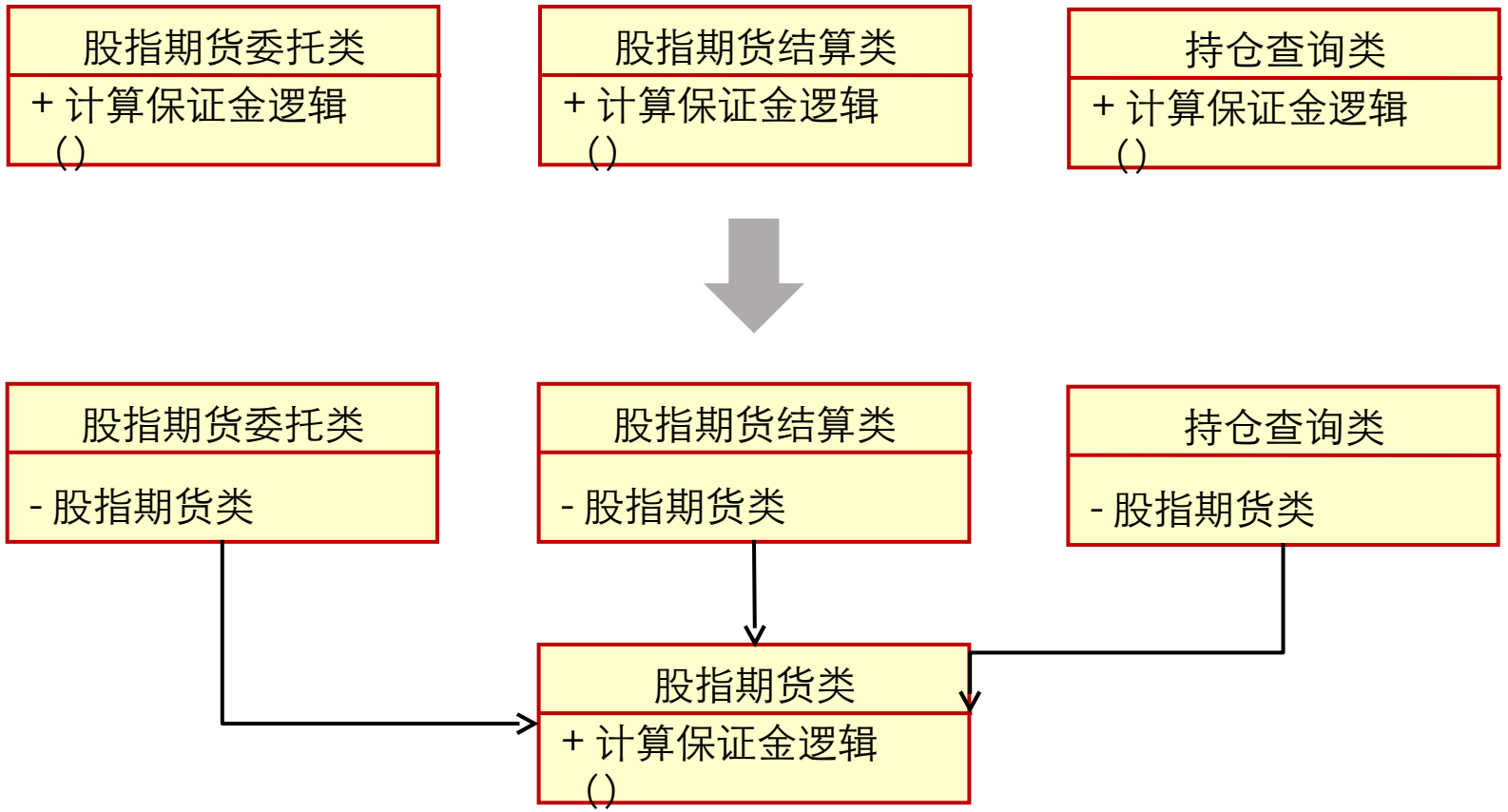
•找出一个应对这些修改负责的类，这可能是一个现有的类，也可能需要通过抽取类来创建一个新类。

•使用移动字段和移动方法等措施将功能置于所选的类中，如果未选中类足够简单，则可以使用内联类将该类除去。

说明：

•散弹式修改指的是“一种变化引发多个类的修改”，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。

霰弹式修改


股指期货委托类
+ 计算保证金逻辑
()
股指期货结算类
+ 计算保证金逻辑
()
持仓查询类
+ 计算保证金逻辑
()
股指期货委托类
- 股指期货类
股指期货结算类
- 股指期货类
持仓查询类
- 股指期货类
股指期货类
+ 计算保证金逻辑
()

数据泥团

症状：

•同样的两至三项数据频繁地一起出现在类和参数表中。

•代码声明了某些字段，并声明了处理这些字段的方法，然后又声明了更多的字段和更多的方法，如此继续。

•各组字段名以类似的子串开头或结束。

措施：

•如果项是类中的字段，则使用抽取类将其取至一个新类中；

•如果值共同出现在方法的签名中，则使用引入参数对象的重构方法以抽取新对象；

•查看这些项的使用：通常可以利用移动方法等重构技术，从而将这些使用移至新的对象中。

纯稚的数据类

症状：

•类仅有字段构成，或者只有简单的赋值方法和取值方法构成。

措施：

•采用封装字段阻止对字段直接访问（仅允许通过赋值方法和取值方法）

•对可能的方法尽量采用移除设置方法进行重构。

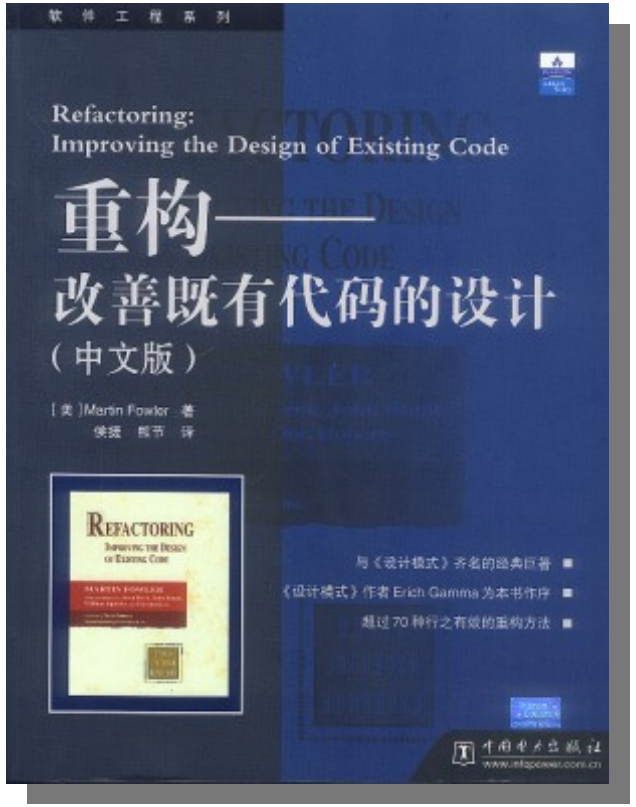
•采用封装集合方法去除对所有集合类型字段的直接访问。

•查看对象的各个客户，如果客户试图对数据做同样的工作，则对客户采用抽取方法，然后将方法移到该类中。

•在完成上述工作后，可能发现类中存在多出相似的方法，使用相应的重构技术，以协调签名并消除重复。

•对字段的大多数访问都不再需要，因为所移动的方法涵盖了其实际应用，因此可以使用隐藏方法来消除对赋值方法和取值方法的访问。

软件重构方法



- 重新组织函数
- 在对象之间移动特性
- 重新组织数据
- 简化条件表达式
- 简化函数调用
- 处理概括关系
- 大型重构

练习：抽取方法

下面的代码有什么不足？你认为应该如何重构？

```
double getPrice()
{
    int basePrice = _quatity * _itemPrice;
    double discountFactor;

    if (basePrice > 1000)
        discountFactor = 0.95;
    else
        discountFactor = 0.98;

    return basePrice * discountFactor;
}
```

练习：方法内联化

下面的代码有什么不足？你认为应该如何重构？

```
int getRating()
{
    return (moreThanFiveLateDeliveries()) ? 2 : 1;
}


boolean moreThanFiveLateDeliveries()
{
    return _numberOfLateDeliveries > 5;
}
```

练习：引入解释性变量

重构前

```
double price()
{
    //price is base Price - quantity Discount + shipping
    return _quantity*_itemPrice -
           Math.max(0,_quantity-500)*_itemPrice*0.05 +
           Math.min(_quantity*_itemPrice*0.1,100.0);
}
```

重构后

```
double price()
{
    final double basePrice = _quantity*_itemPrice;
    final double quantityDiscount =
                 Math.max(0,_quantity-500)*_itemPrice*0.05;
    final double shipping = Math.min(basePrice*0.1,100.0);
    return basePrice-quantityDiscount+shipping;
}
```

练习：简化条件表达式

下面的代码有什么不足？你认为应该如何重构？

```
double getPayAmount()
{
    double result;

    if (_isDead) result = deadAmount();
    else {
          if (_isSeparated) result = separatedAmount();
          else {
                if (_isRetired) result = retiredAmount();
                else result = normalPayAmount();
               }
          }
    return result;
}
```

练习：移动字段

下面的代码有什么不足？你认为应该如何重构？

```
class Account {
    ...
    private AccountType _type;
    private double _interestRate;

    double interestForAmount_days (double amount, int days) {
        return _interestRate * amount * days / 365;
    }
    ...
}
```

练习：移动方法

下面的代码有什么不足？你认为应该如何重构？

```
class Account...
    double overdraftCharge() {
        if (_type.isPremium()) {
            double result=10;
            if (_daysOverdrawn>7) result+=(_daysOverdrawn-7)*0.85;
            return result;
        }
        else return _daysOverdrawn*1.75;
    }
    double bankCharge() {
        double result=4.5;
        if(_daysOverdrawn>0) result+=overdraftCharge();
        return result;
    }
    private AccountType _type;
    private int _daysOverdrawn;
```

说明：假设有几种账户，每一种都有自己的“透支金额计算规则”。

练习：提炼类

下面的代码有什么不足？你认为应该如何重构？

```
Class Person {
    private string _name;
    private string _officeAreaCode;
    private string _officeNumber;
    public string getName() {
        return _name;
    }
    public string getTelephoneNumber() {
        return ("("+_officeAreaCode+")"+_officeNumber); // □□□□(□□)□□□□
    }
    string getOfficeAreaCode() {
        return _officeAreaCode;
    }
    void setOfficeAreaCode(string arg) {
        _officeAreaCode=arg;
    }
    string getOfficeNumber() {
        return _officeNumber;
    }
    void setOfficeNumber(string arg) {
        _officeNumber=arg;
    }
}
```

教学提纲

3 □□□□

- □□□□□ Singleton □
- □□□□□□□ Abstract Factory □
- □□□□□ Strategy □
- □□□□□□ Observer □

设计模式

□□□□□□□□□

- Trees, Stacks, Queues
- □□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□

- □□□□□□□□□□□□□□□□□
- Pattern □ Documented experience

设计模式

[illegible]

[illegible]

- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]

设计模式

设计模式不是万能的

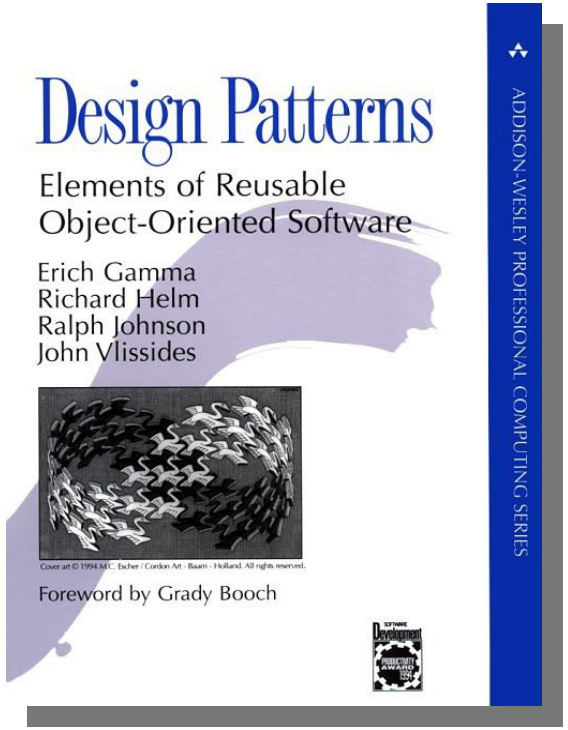
- 模式可以解决大多数问题，但不可能解决遇到的所有问题
- 应用一种模式一般会“有得有失”，切记不可盲目应用
- 滥用设计模式可能会造成过度设计，反而得不偿失

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

Gamma □□□□□□□

Gamma □□□□□□□□□□□□□□□□□



Gang of Four □

Gamma, Helm, Johnson, Vlissides

创建型模式

- 创建型模式描述了实例化对象的相关技术，使用继承来改变被实例化的类，而一个对象创建型模式将实例化委托给另一个对象。

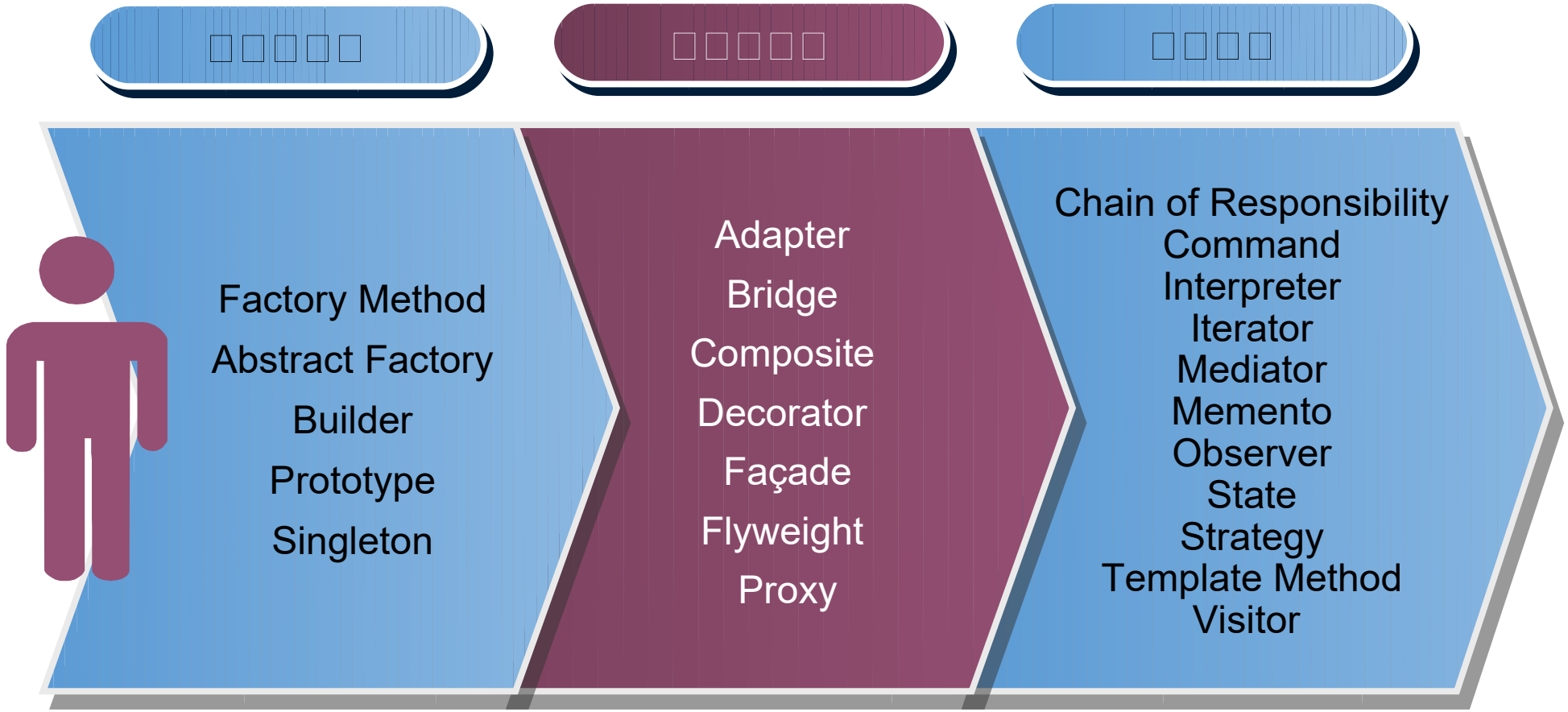
结构型模式

- 结构型模式描述了在软件系统中组织类和对象的常用方法，采用继承机制来组合接口或实现。

行为模式

- 行为模式负责分配对象的职责，使用继承机制在类件分配行为。

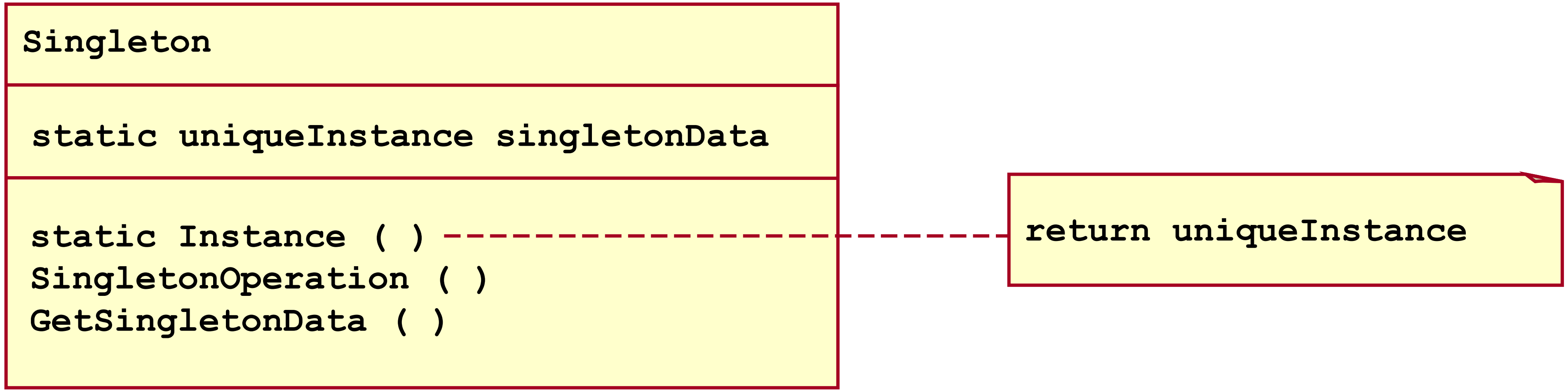
Gamma □□□□□□□


□□□□□
□□□□□
□□□□
Factory Method
Abstract Factory
Builder
Prototype
Singleton
Adapter
Bridge
Composite
Decorator
Façade
Flyweight
Proxy
Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

单例模式

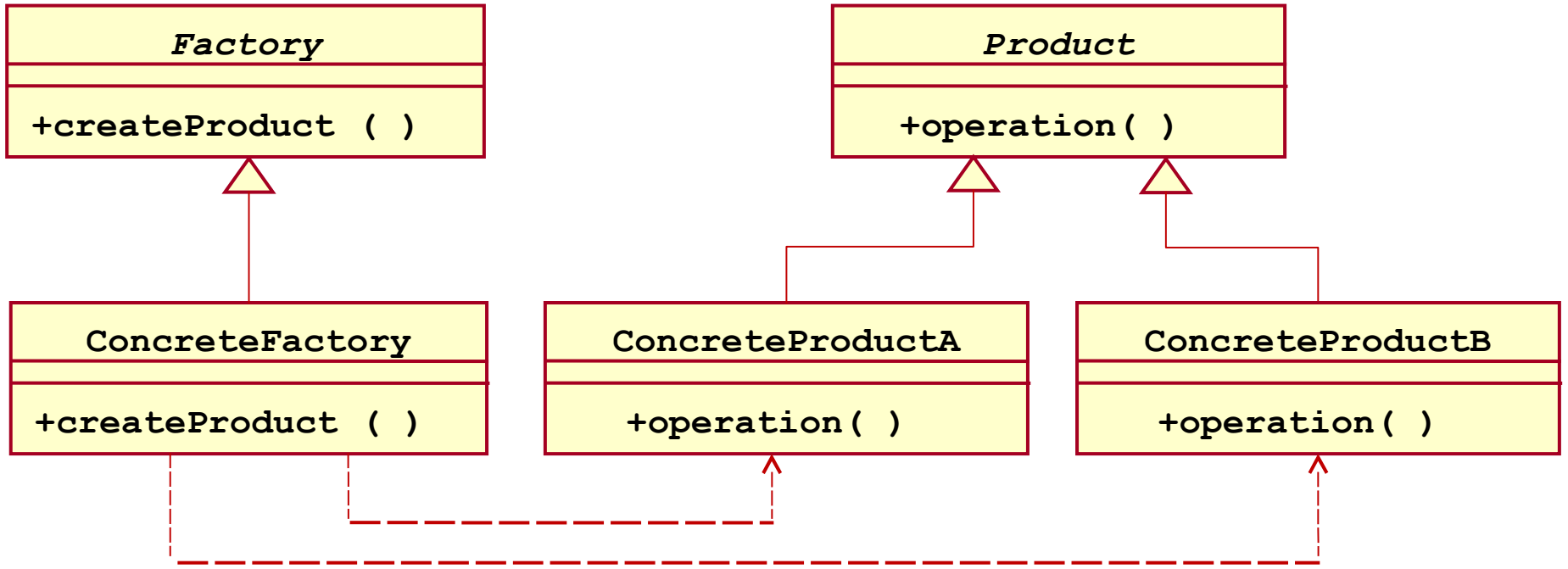
□□□□□ Singleton □

- □□
- □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

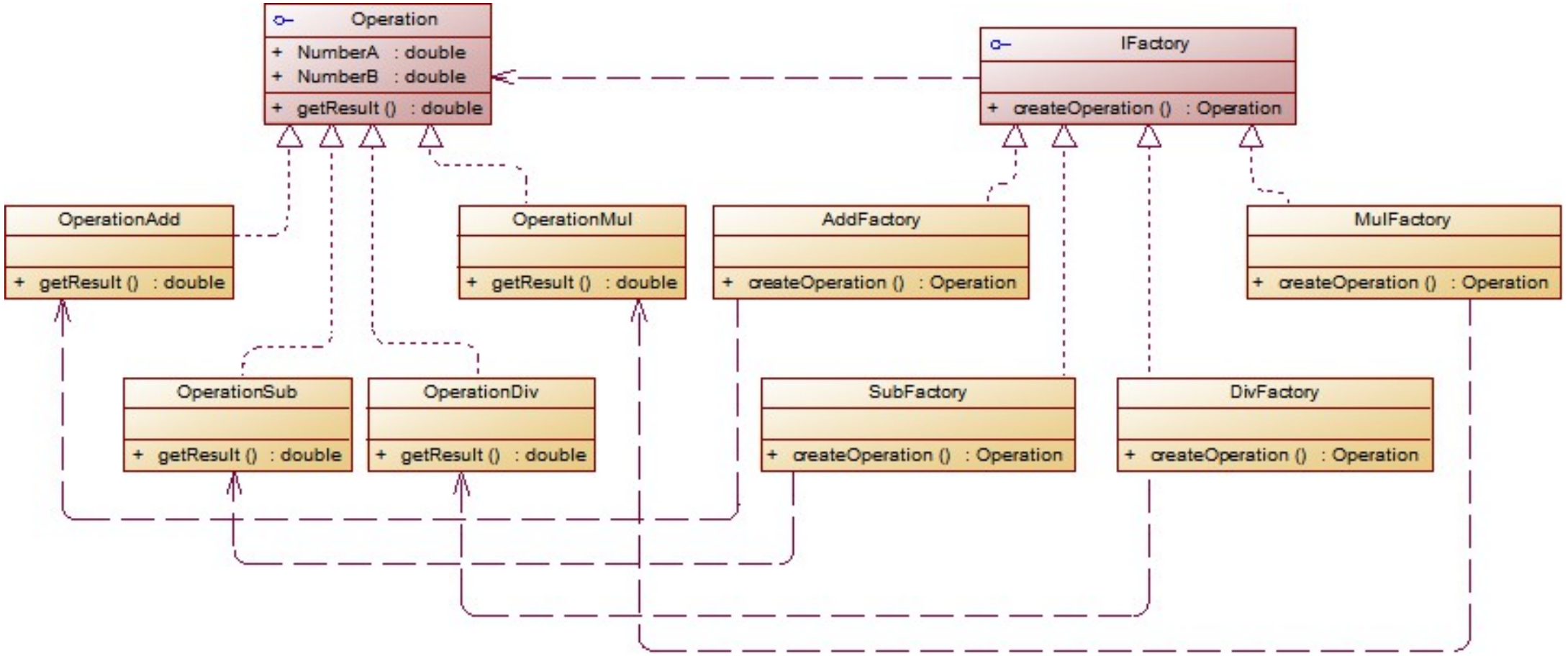


□□□□□ Factory Method □

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。

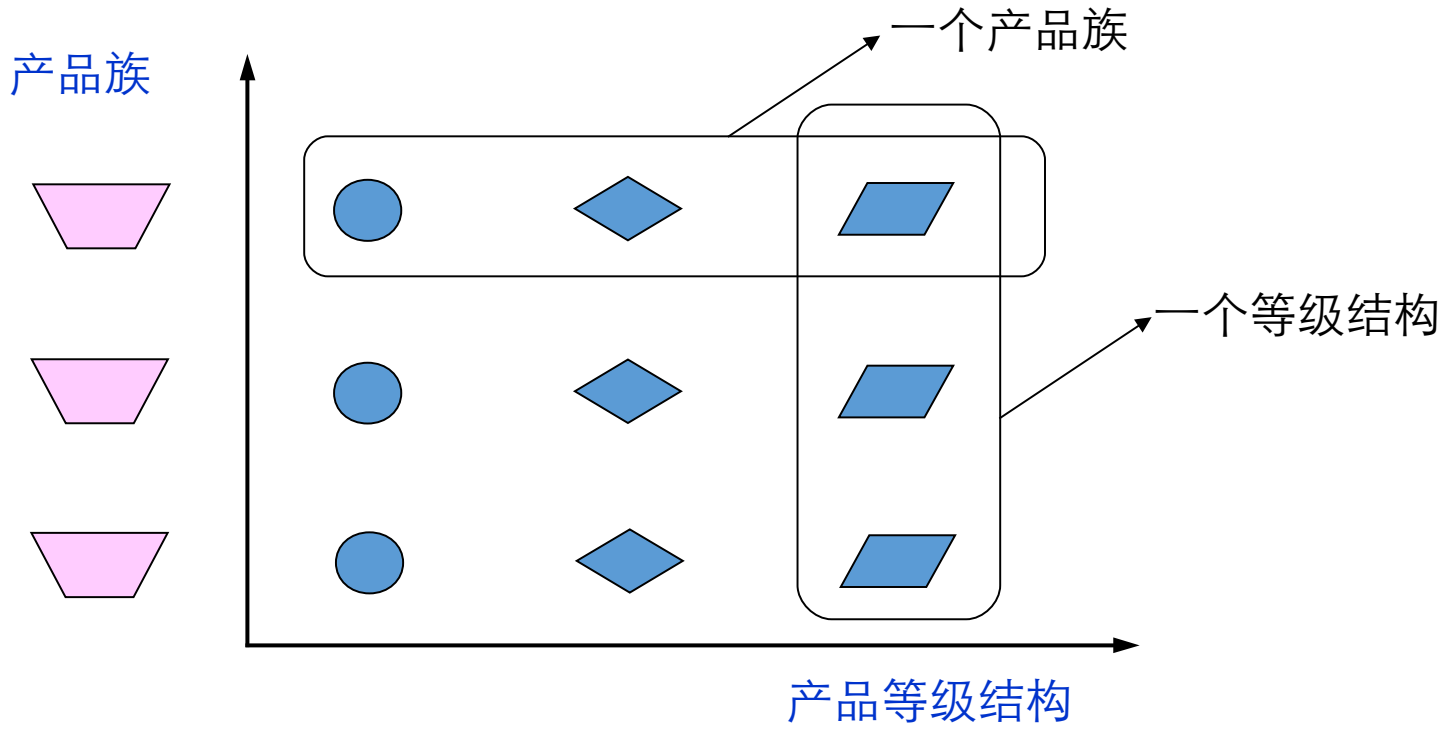


□□□□□ Factory Method □

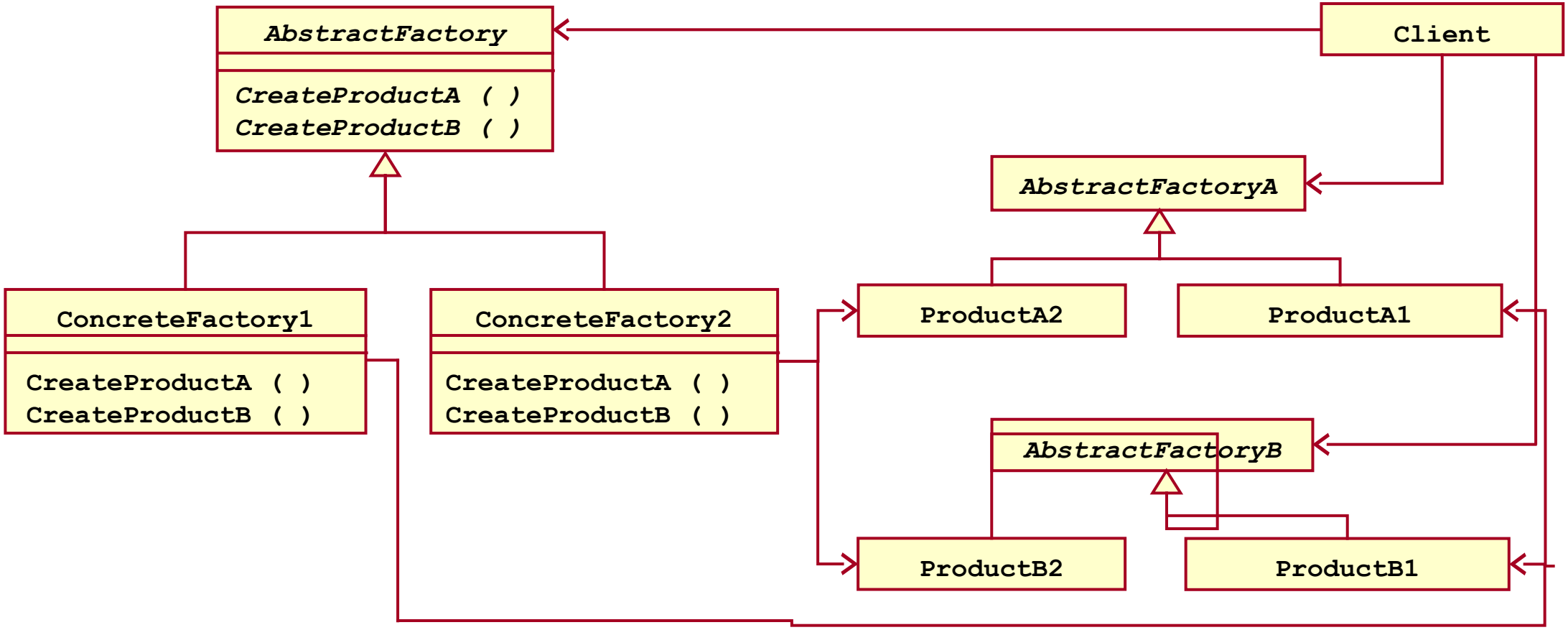


□□□□□□□ Abstract Factory □

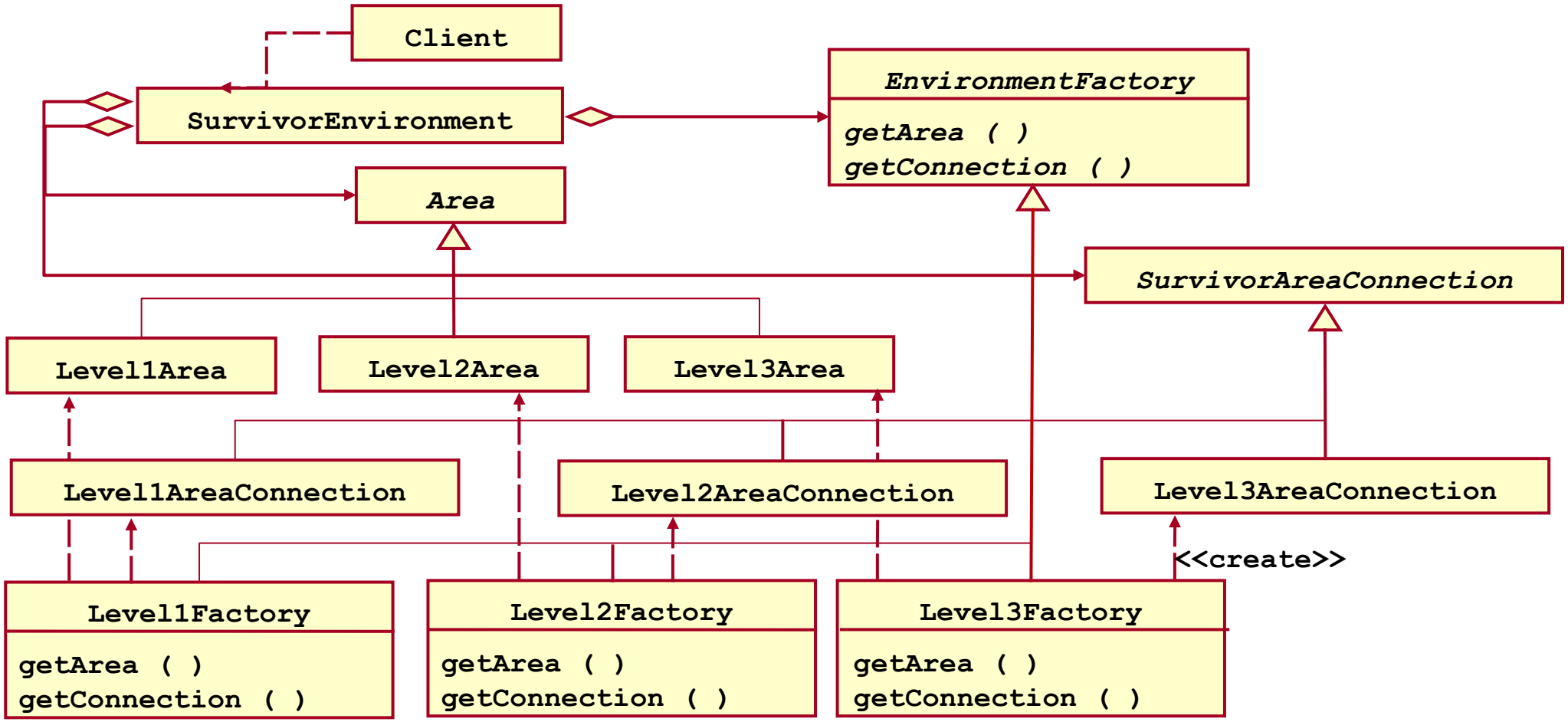
抽象工厂模式可以向客户端提供一个接口，使其在不必指定产品具体类型的情况下，创建多个产品族中的产品对象。



□□□□□□□ Abstract Factory □

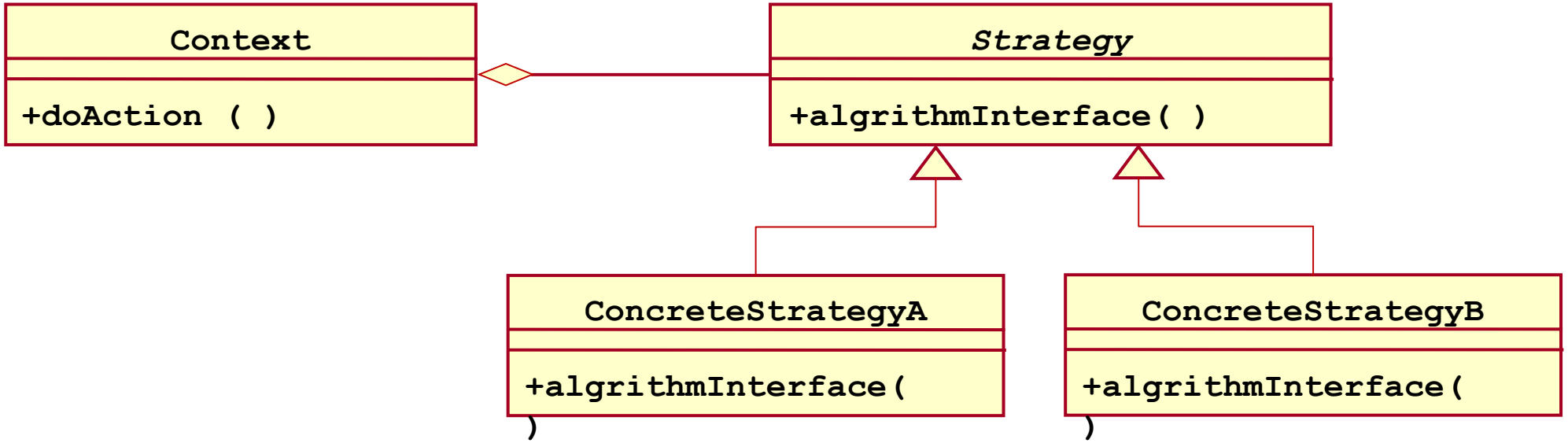


□□□□□□□ Abstract Factory □

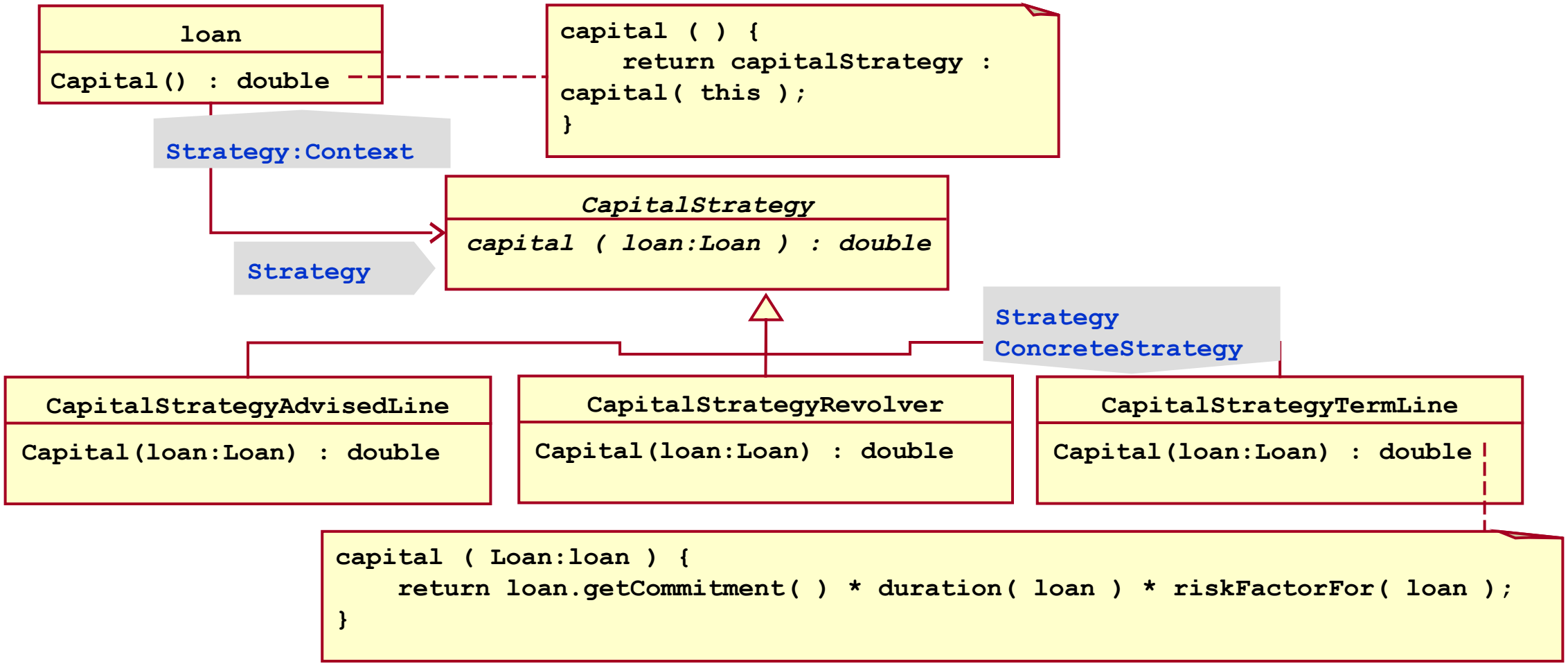


□□□□□ Strategy □

□□□□□ Strategy □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

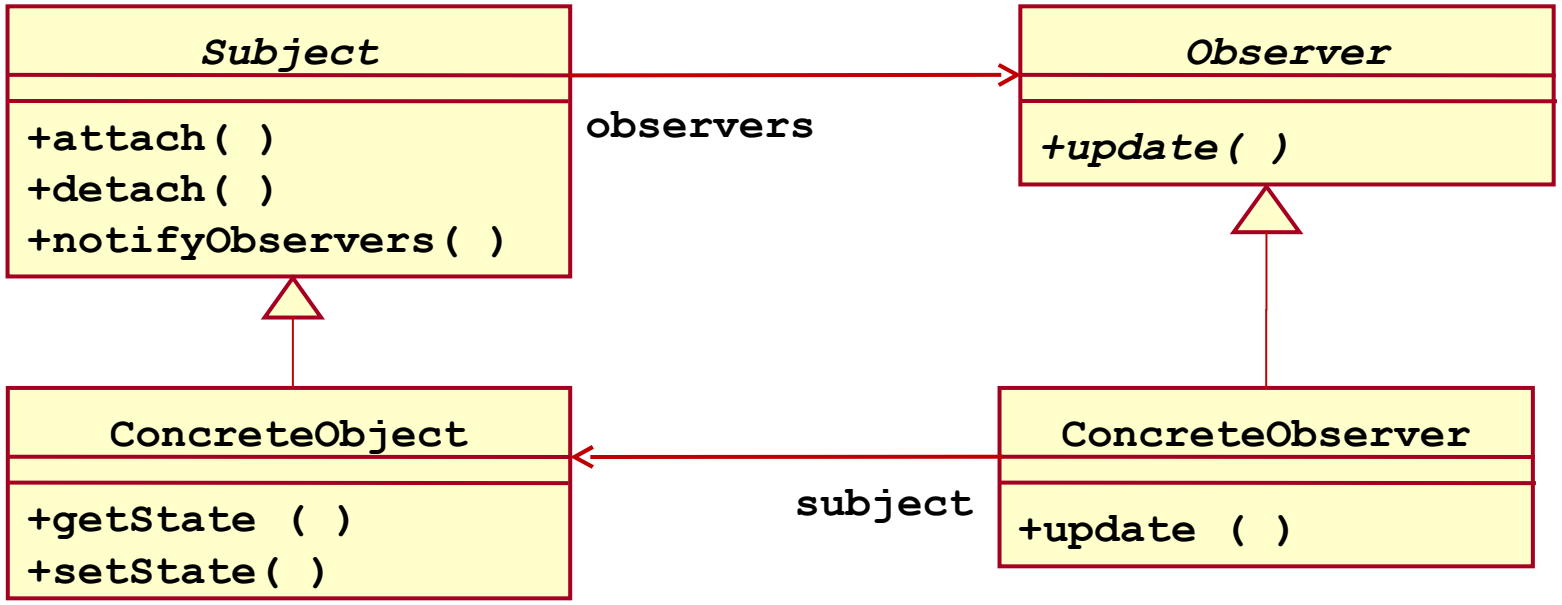


□□□□□ Strategy □


loan
Capital() : double
capital ( ) {
return capitalStrategy :
capital(this);
}
Strategy:Context
CapitalStrategy
capital (loan:Loan) : double
Strategy
Strategy
ConcreteStrategy
CapitalStrategyAdvisedLine
Capital(loan:Loan) : double
CapitalStrategyRevolver
Capital(loan:Loan) : double
CapitalStrategyTermLine
Capital(loan:Loan) : double
capital (Loan:loan) {
return loan.getCommitment( ) * duration(loan) * riskFactorFor(loan);
}

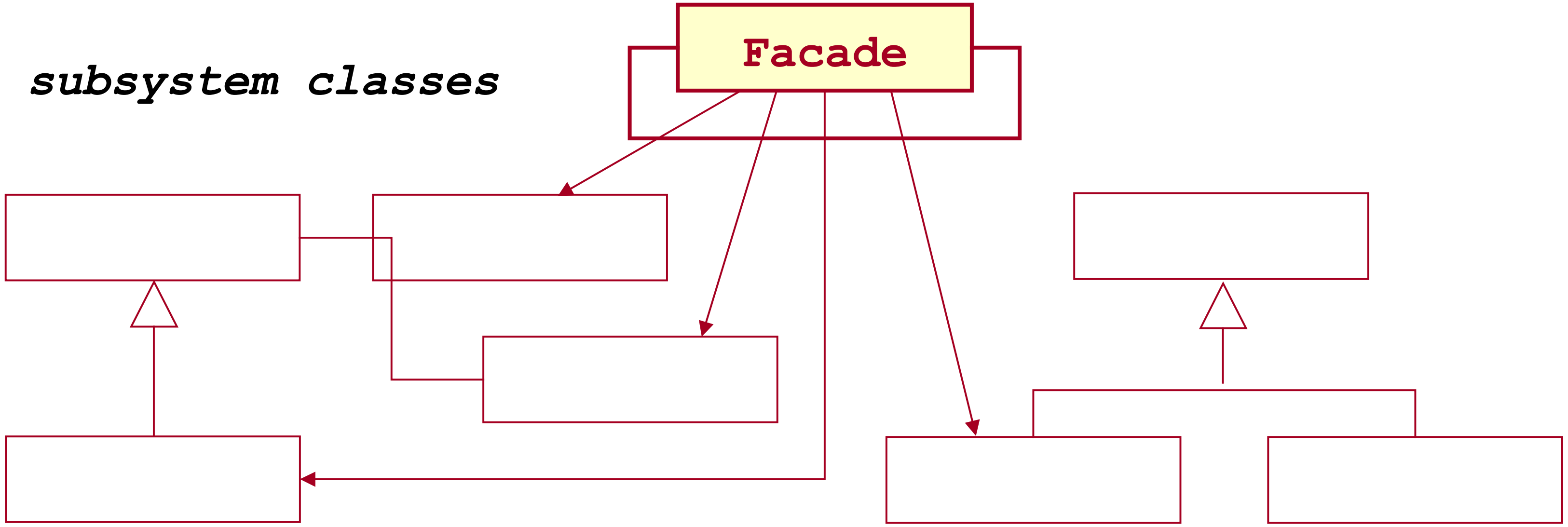
□□□□□□ Observer □

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。

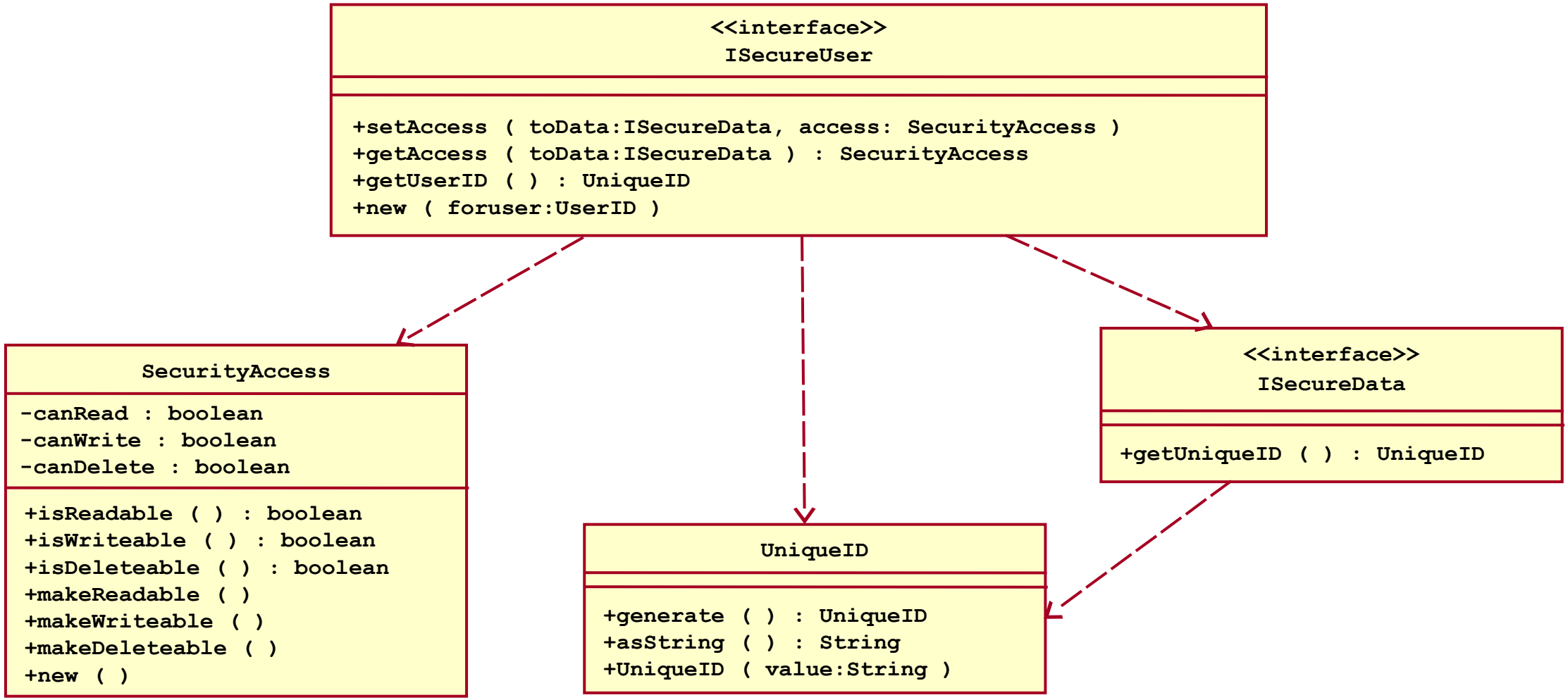


□□□□□ Façade □

Façade □□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Façade □□□□□□□□□
□□□□□□□□□□□□□□□

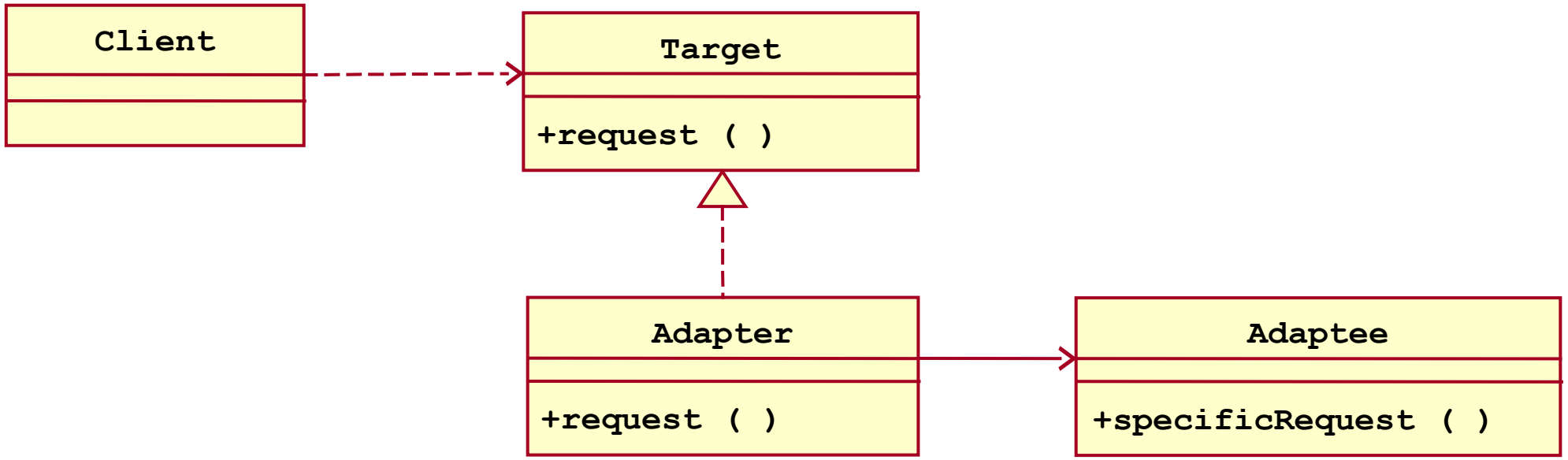


□□□□□ Façade □

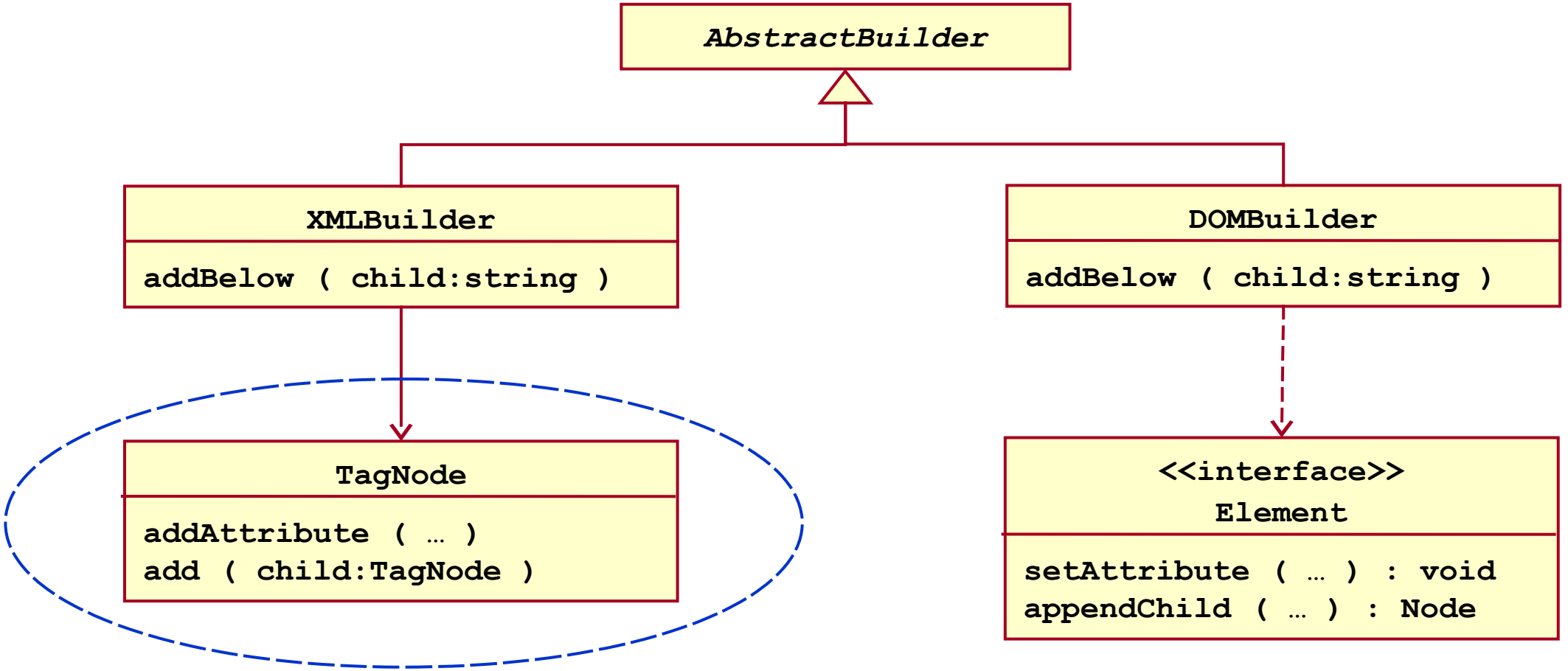

<<interface>>
ISecureUser
+setAccess (toData:ISecureData, access: SecurityAccess)
+getAccess (toData:ISecureData) : SecurityAccess
+getUserID ( ) : UniqueID
+new (foruser:UserID)
SecurityAccess
-canRead : boolean
-canWrite : boolean
-canDelete : boolean
+isReadable ( ) : boolean
+isWriteable ( ) : boolean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

□□□□□□ Adapter □

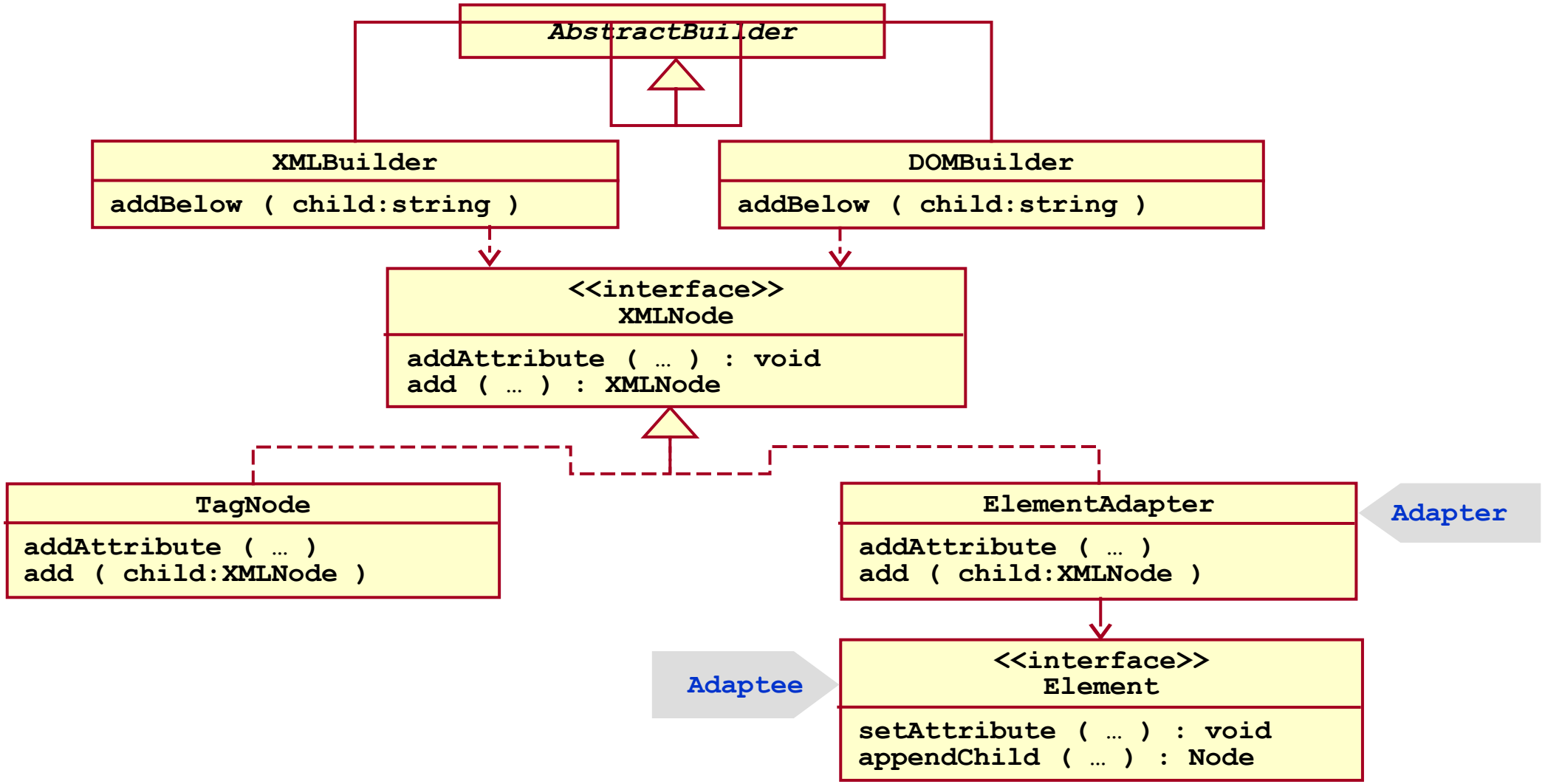
适配器模式将一个类的接口转换成客户希望的另一个接口，使原本由于接口不兼容而不能一起工作的类可以一起工作。



□□□□□□ Adapter □


AbstractBuilder
XMLBuilder
addBelow (child:string)
DOMBuilder
addBelow (child:string)
TagNode
addAttribute (…)
add (child:TagNode)
<<interface>>
Element
setAttribute (…) : void
appendChild (…) : Node

□□□□□□ Adapter □


AbstractBuilder
XMLBuilder
addBelow (child:string)
DOMBuilder
addBelow (child:string)
<<interface>>
XMLNode
addAttribute (…) : void
add (…) : XMLNode
TagNode
addAttribute (…)
add (child:XMLNode)
ElementAdapter
addAttribute (…)
add (child:XMLNode)
Adapter
Adaptee
<<interface>>
Element
setAttribute (…) : void
appendChild (…) : Node

□□□□□□ Adapter □

□□□□

- □□□□□□□□□□□□□□□□□□□□□□□□
- □□□□□□□□□□□□□□□□□□□□□□□□□□□□
- □□□
□□□□□□□□□□□□□

Adapter □ Façade

- □□□□□□□□□□□□ Adapter □□□□□□□□□ Façade □□□□□□□□□□

谢谢大家！

THANKS