火龙果•整理
uml.org.cn

# 技术架构视图-设计原则与模式

胡协刚
软件架构师 **UML/RUP**专家

火龙果•整理
uml.org.cn

# 内容提要

- 面向对象设计原则
- 对象集合视角下的开闭原则
- 正方形悖论与Liskov替换原则
- GRASP模式

火龙果•整理
uml.org.cn

# 面向对象设计原则

火龙果•整理
uml.org.cn

# 软件系统开始坏死的症状

✓一个软件系统开始坏死时表现的症状有：

- 硬化Rigidity——系统变得越来越难以变更，修复或增添新功能的代价高昂;
- 脆弱Fragility——对系统的任何哪怕是微小的变更都可能造成四处（甚至是与变更处没有逻辑上的关联之处）崩溃;
- 绑死Immobility——抽取系统的任何部分用来复用都非常困难;
- 胶着Viscosity——以与原有设计保持一致的方式来对实施变更已经非常困难，诱使开发人员绕过它选择容易但有害的途径，其结果却使系统死的更快。

火龙果•整理
uml.org.cn

# 软件质量

✓软件系统最关键的质量特性有:

- 正确性correctness
- 健壮性robustness
- 可扩展性extensibility
- 可复用性reusability

✓软件系统其它重要的质量特性有:

- 兼容性compatibility
- 可移植性portability
- 易用性ease of use
- 高效性efficiency
- timeliness, economy and functionality

火龙果•整理
uml.org.cn

# 面向对象设计的基本原则

✓类的设计原则:

开闭原则、依赖倒置原则、 Liskov替换原则、单一职责原则、接口分离原则、组合复用原则、所知最少原则

✓包内聚原则:

发布与复用等价原则、共同封闭原则、共同复用原则

✓包耦合原则:

无循环的依赖原则、稳定的依赖原则、稳定的抽象原则

火龙果•整理
uml.org.cn

# 类的设计原则

- 开闭原则（Open Closed Principle）——对扩展开放而对变更封闭open for extension but closed for modification
- 依赖倒置原则（Dependency Inversion Principle）——依赖于抽象类而非具体类depend upon abstractions but concretions（客户代码依赖于具体类之上的基类的共性行为，相对于按顺序从下面依赖而言是一种倒置）
- Liskov替换原则（Liskov Substitution Principle）——子类应当能完全替代其基类（的行为）subclasses should be substitutable for their base classes

火龙果•整理
uml.org.cn

# 类的设计原则

- 单一职责原则（Single Responsibility Principle）——一个类只应当承担单一和集中的职责，这样引发类进行变更的原因只有一个A class should have one, and only one, reason to change
- 接口分离原则（Interface Segregation Principle）——为客户client提供多个特定的接口好过一个多种用途集于一身的接口，即客户不被强制依赖于其不需要（不使用）的操作many client specific interfaces are better than one general purpose interface

火龙果•整理
uml.org.cn

# 类的设计原则

- 组合复用原则（Composite Reuse Principle）——尽可能地使用对象的多态组合而非继承（来实现复用）favor polymorphic composition of objects over inheritance
- 所知最少原则（Principle of Least Knowledge）——一个类的操作实现中，只应调用下列对象的操作：它自己、作为参数传入的对象、它创建的对象、它包含的对象for an operation on a class, only operations on the following objects should be called: itself, its parameters, object it creates, or its contained instance objects

火龙果•整理 uml.org.cn

# 包内聚原则

- ✓发布与复用等价原则（Release Reuse Equivalency Principle）——软件复用的粒度与发布包等价 the granule of reuse is the granule of release
- ✓共同封闭原则（Common Closure Principle）——一道发生变更的类应当属于同一个包classes that change together, belong together
- ✓共同复用原则（Common Reuse Principle）——那些不被一起复用的类不应当分组到同一个包中 classes that aren't reused together should not be grouped together

火龙果•整理
uml.org.cn

# 包耦合原则

- ✓无循环的依赖原则（Acyclic Dependencies Principle）——包与包之间的依赖关系不能造成循环 the dependencies between packages must not form cycles
- ✓稳定依赖原则（Stable Dependencies Principle）——包依赖的方向应当指向其它更稳定的包depend in the direction of stability
- ✓稳定抽象原则（Stable Abstractions Principle）——系统中最稳定的包应当是抽象包 stable package should be abstract packages

火龙果•整理
uml.org.cn

# 对象集合视角下的开闭原则

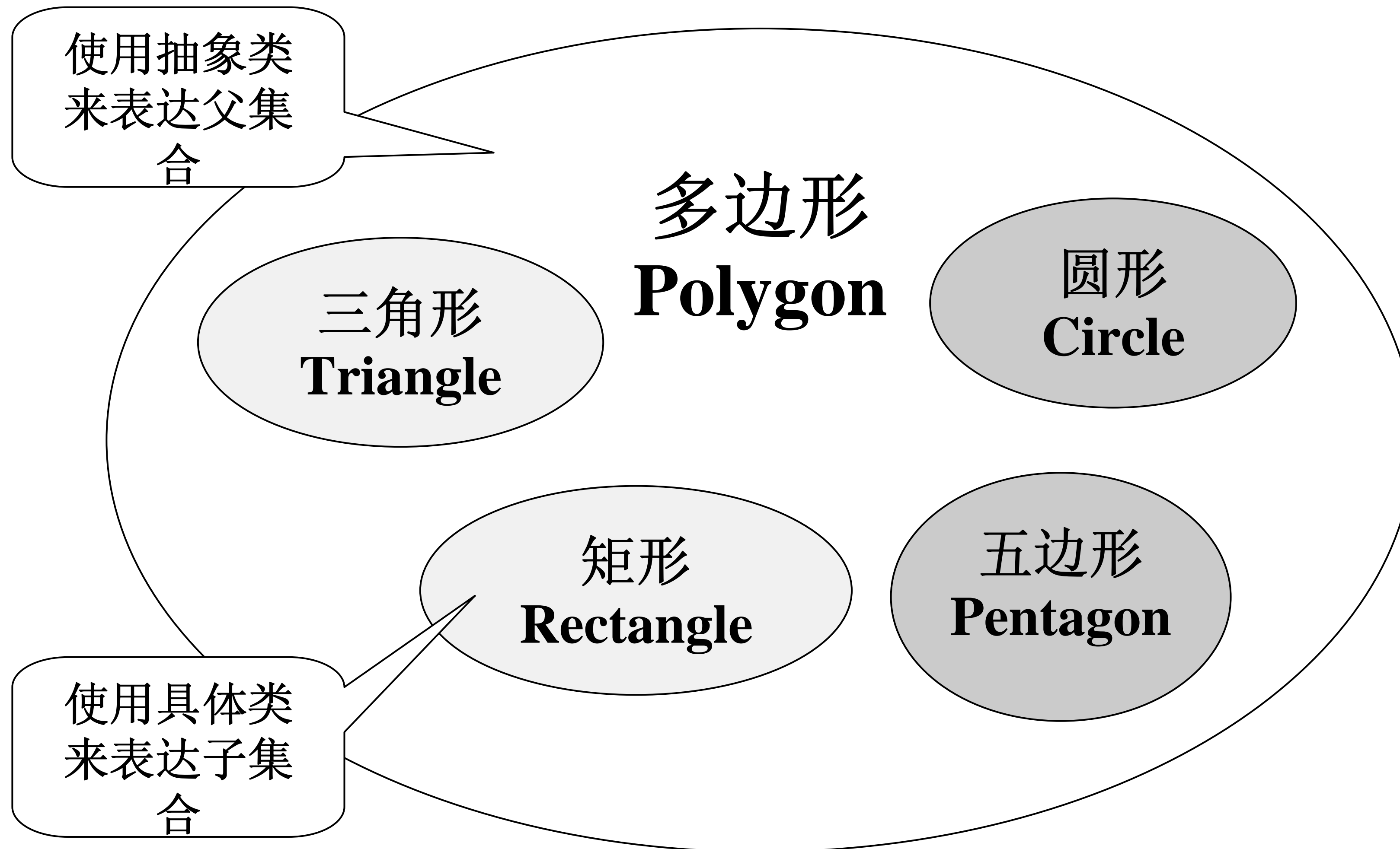
火龙果•整理
uml.org.cn

# 实现开闭原则

- 利用依赖倒置原则来实现对变更封闭——客户代码依赖于抽象类而非具体类，而抽象类代表了父集合的共性，它通常是比较稳定的（不会变化），因而可以在具体子类变化时，保证客户代码不受影响，实现了对变更封闭
- 利用Liskov替换原则来保证对扩展开放——只要能做到子类可以完全替代其基类的行为，那么新增的具体子类在重载父类时，不会对客户代码带来任何不良影响（即保证原来的抽象类仍代表了父集合的共性），因而实现了对扩展开放

火龙果●整理
uml.org.cn

# 对象集合与开闭原则



火龙果•整理
uml.org.cn

# 满足开闭原则的Java实例

```
//多边形编辑类只依赖多边形抽象类，不依赖于三角形、矩形等具体类
public class PolygonsEditor {
  Set set = new HashSet();            //保持一批多边形对象实例
  public void doScale(float rate) {
        for (Iterator iter = set.iterator(); iter.hasNext();) {
              Polygon plg = (Polygon) iter.next();
              plg.scale(rate);  //对每个多边形对象实例执行缩放操作

        }
  }
}
//现在增添一个圆形具体类，多边形编辑类和三角形等其它具体类均不受
  影响
public class Circle extends Polygon {
  float radius;
  public void scale(float rate) {
        radius = radius * rate;
  }
}
```

火龙果•整理
uml.org.cn

# 思考：支持开闭原则的背后

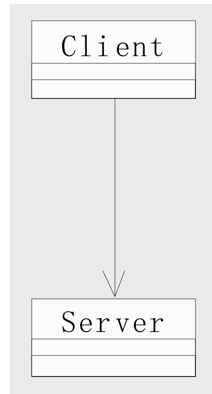
- 软件所面对问题域中包含的对象（具体类）种类繁多，不可穷尽；通常只能在当前所要解决的问题范围内，对涉及到的部分对象进行处理；然而随着需求的变化，问题域范围改变或扩大，新的对象将要被加入，旧有系统面临被打破的危险；
- 从类的集合定义角度，我们要处理的实际上是代表对象集合的类，当我们直接使用代表较小集合的具体类时（例如三角形、矩形等），必然要面对开发大量不同具体类的压力，否则今后一旦需求变更，出现新的具体类（例如圆形）时，使用它们的客户代码将被迫更改；

火龙果•整理
uml.org.cn

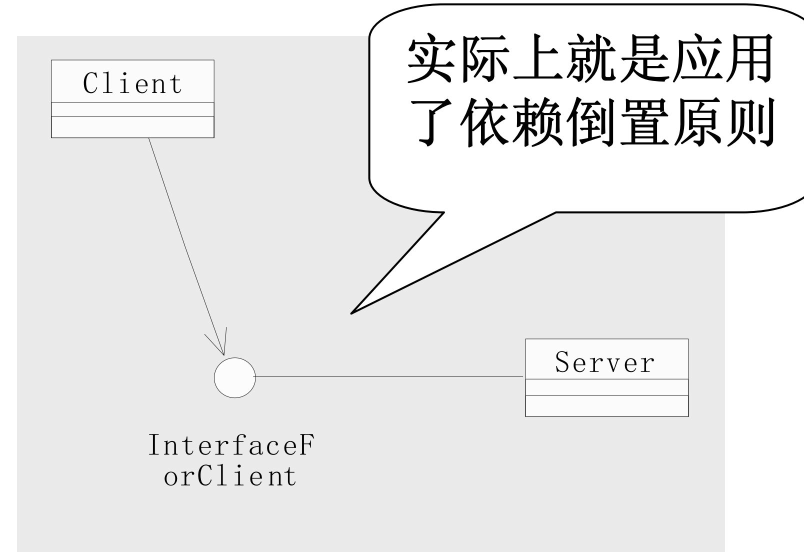
# 思考：支持开闭原则的背后

- 然而我们可以利用代表更大集合的抽象类或接口（例如多边形），其客户代码只依赖抽象类或接口中代表大集合中所有对象的共性行为，这样只需开发少量的具体类以满足当前的需求，而未来出现新的具体类时，其它代码部分却不受影响。

# 示例：实现开闭原则的途径

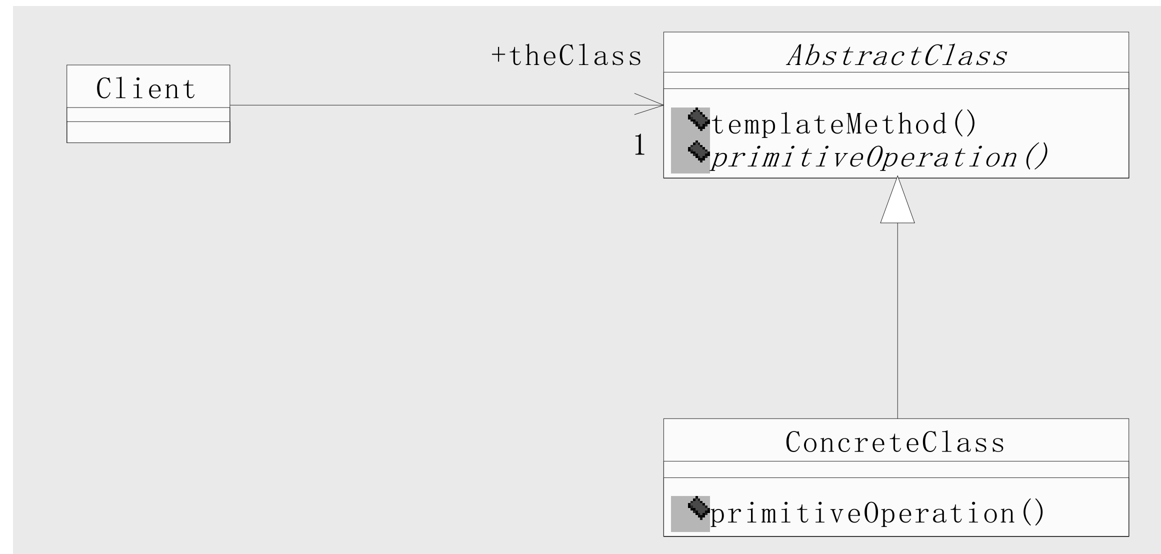



使用接口



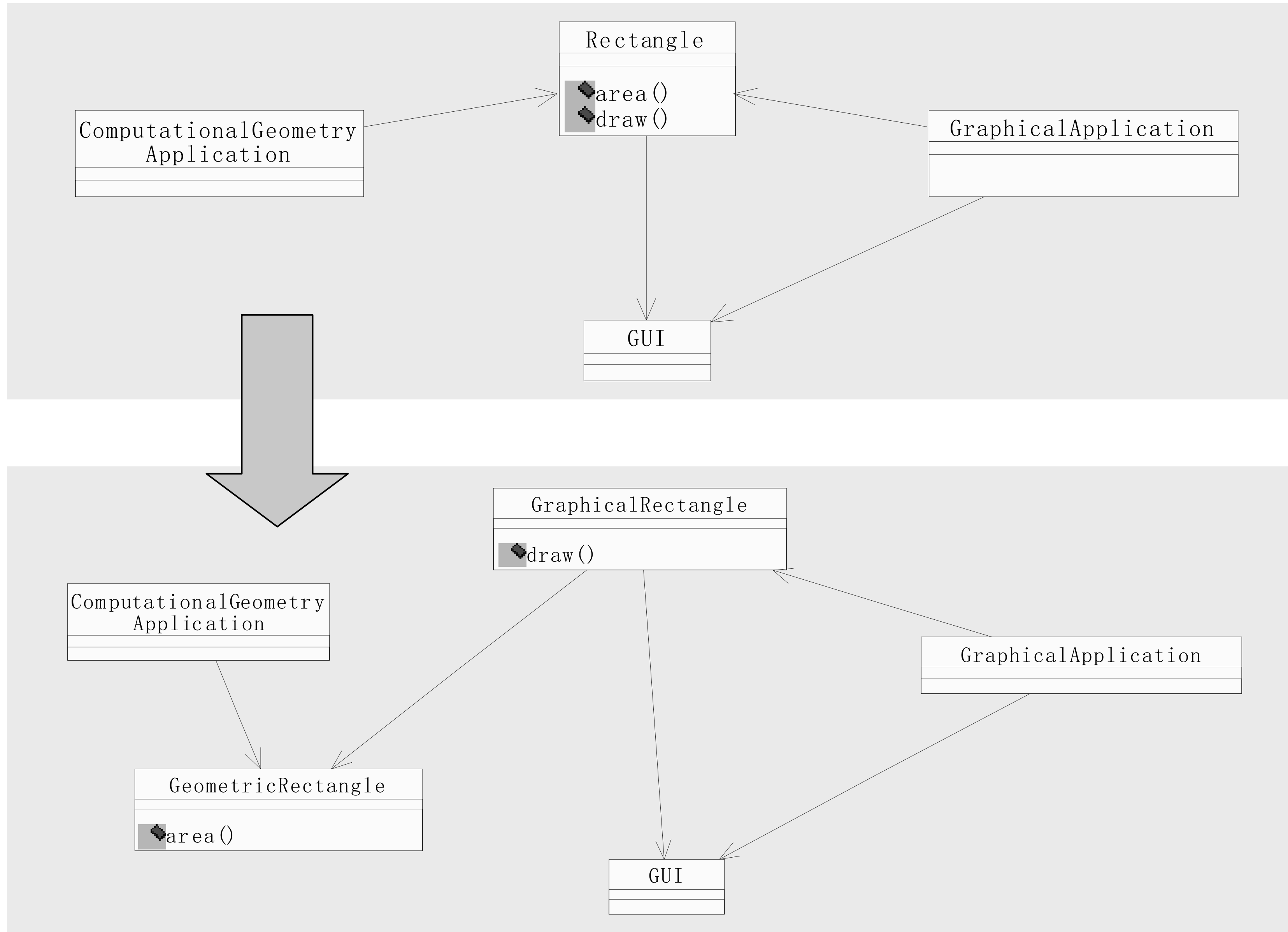

实际上就是应用了依赖倒置原则

使用模板方法模式



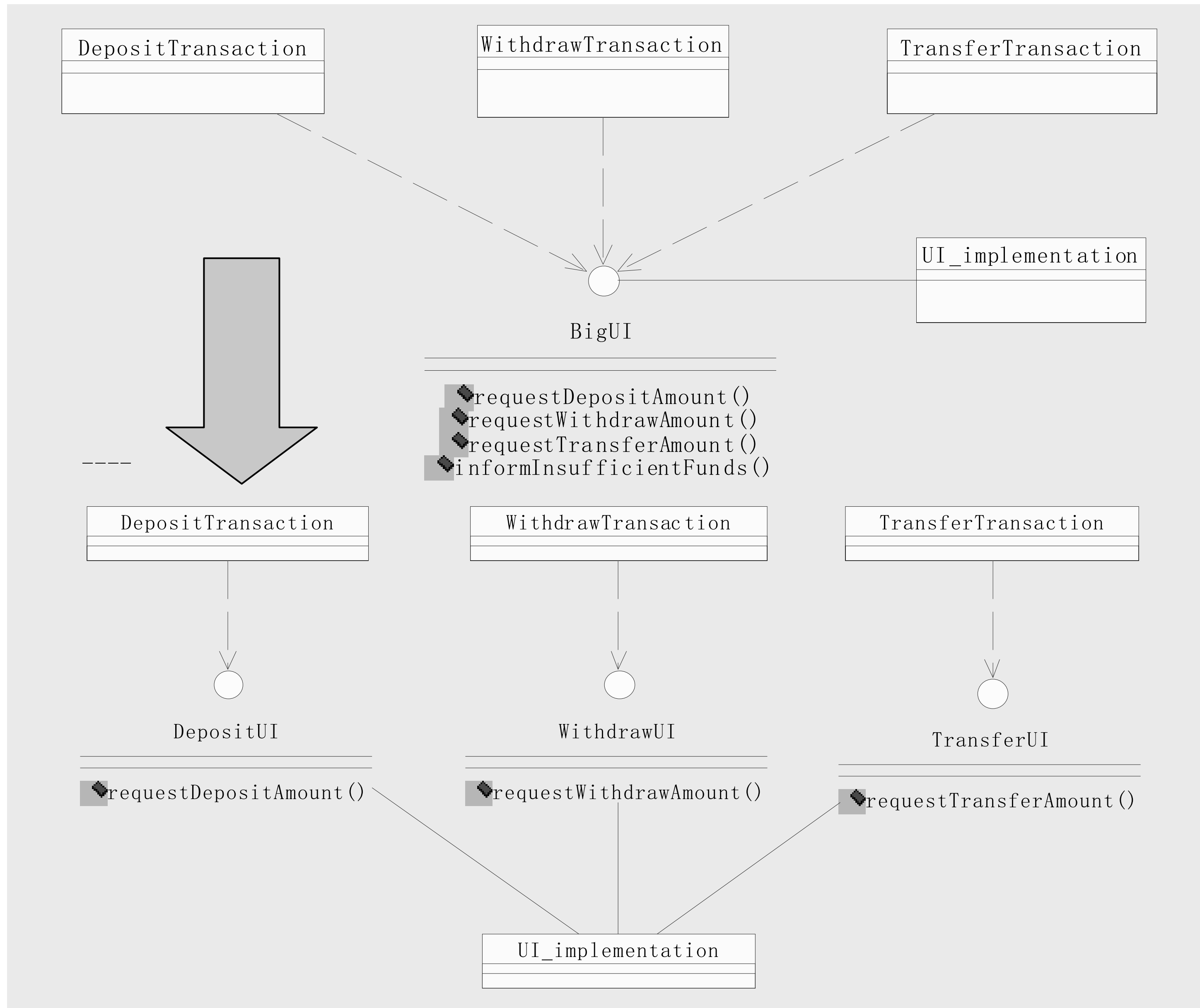
火龙果•整理
uml.org.cn

# 示例：确保单一职责原则



火龙果•整理
uml.org.cn

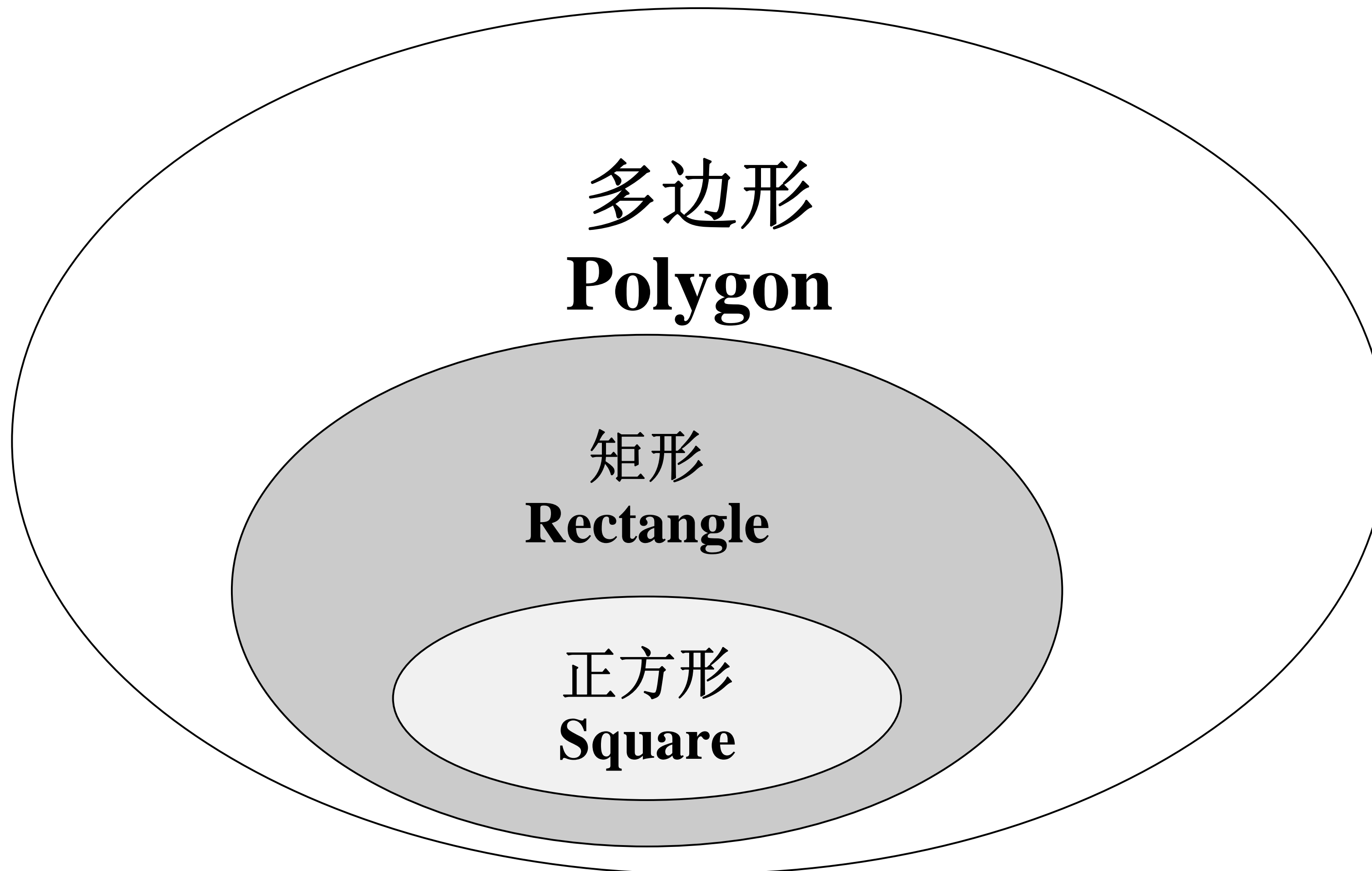
# 示例：确保接口分离原则



火龙果•整理
uml.org.cn

# 正方形悖论与Liskov替换原则

# 正方形悖论—集合视图



火龙果•整理
uml.org.cn

# 正方形悖论—继承关系

```
//矩形类具有两个方法用来设置其长宽两边
public class Rectangle{
   float width = 0;            //矩形的宽
   float length = 0;           //矩形的长
   public void setWidth(float x) {
            width = x;
            }
   public void setLength(float y) {
            length = y;
            }
}
//正方形类应当继承矩形类设置长宽两边的两个方法
public class Square extends Rectangle {
   public void setSide(float l) {
            width = l;
            length = l;
   }
}
```

火龙果●整理
uml.org.cn

# 正方形悖论—测试代码

```
//下面是一段测试代码
//创建矩形、正方形等实例
List rectangleList = new LinkedList();
rectangleList.add(new Rectangle(10,20));
rectangleList.add(new Square(10));
//对一批矩形实例进行缩放操作
for (Iterator iter = rectangleList.iterator(); iter.hasNext();) {
   Rectangle theRectangle = (Rectangle) iter.next();
   theRectangle.scale(2);  //执行矩形的缩放操作
}
//对一批矩形实例进行长宽设置操作
for (Iterator iter = rectangleList.iterator(); iter.hasNext();) {
   Rectangle theRectangle = (Rectangle) iter.next();
   theRectangle.setWidth(30);  //执行矩形的宽设置操作
   theRectangle.setLength(20);  //执行矩形的长设置操作
   assertEquals(theRectangle.getWidth(), 30);
   assertEquals(theRectangle.getLength(), 20);
   assertTrue(theRectangle.validateItself());
}
```

正方形将不再是正方形，它的两边不再相等，不能通过自我确认测试

火龙果•整理
uml.org.cn

# 正方形悖论—思考

```
//有人提出修改矩形类设置长宽两边的方法，并在正方形中重载它
//矩形类用一个方法用来同时设置其长宽两边
public class Rectangle{
  float width = 0;          //矩形的宽
  float length = 0;         //矩形的长
  public void setSide(float x, float y) {
        width = x;
        length = y;
  }
}
//正方形类重载设置长宽两边的方法
public class Square extends Rectangle {
  public void setSide(float x, float y) {
        width = x;
        length = x;
}
```

# 正方形悖论—测试代码

```
//下面是一段根据新的方法调整后的测试代码
//对一批矩形实例进行缩放操作
for (Iterator iter = rectangleList.iterator(); iter.hasNext();) {
    Rectangle theRectangle = (Rectangle) iter.next();
    theRectangle.scale(2);  //执行矩形的缩放操作
}
//对一批矩形实例进行长宽设置操作
for (Iterator iter = rectangleList.iterator(); iter.hasNext();) {
    Rectangle theRectangle = (Rectangle) iter.next();
    theRectangle.setSide(30, 20);  //执行矩形的长宽设置操作
    assertEquals(theRectangle.getWidth(), 30);
    assertEquals(theRectangle.getLength(), 20);
    assertTrue((theRectangle.validateItself());
}
```

正方形将不能通过对矩形长宽设置操作结果的测试

火龙果•整理
uml.org.cn

# 正方形悖论—结论

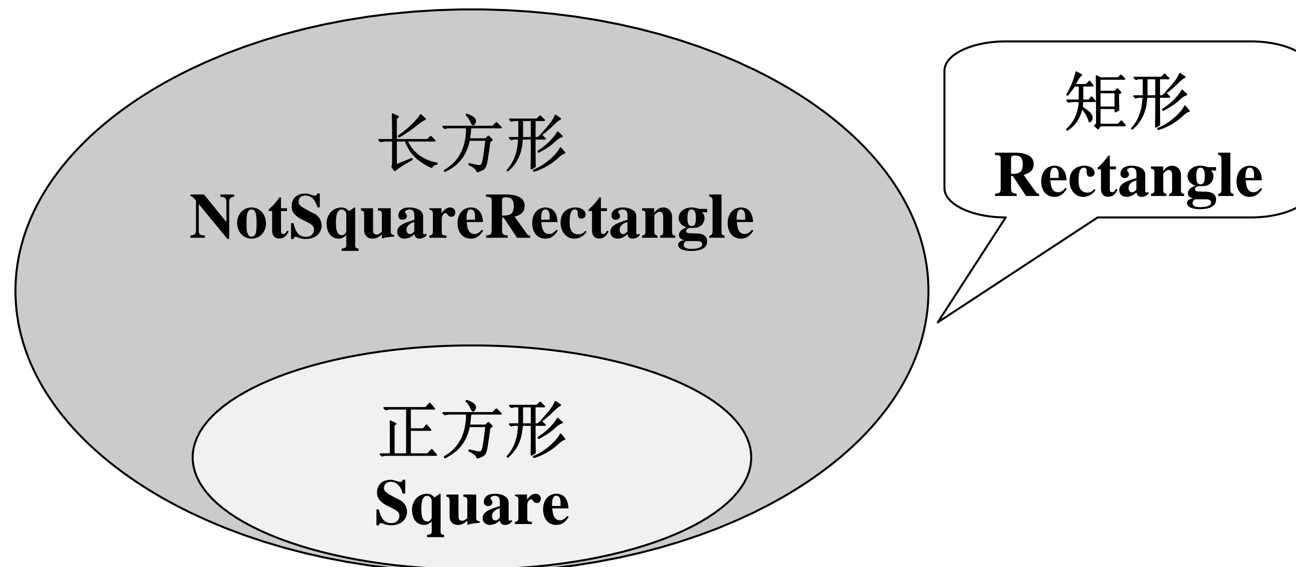
- 设置长宽操作只是长方形（即非正方形矩形）才拥有的行为；对于正方形而言，它四边相等，没有长宽的差别
- 所谓矩形，严格地讲，应当包含正方形和长方形两类；它的行为应当是正方形和长方形所共有的行为，例如缩放操作、旋转等；
- 设置长宽操作并不是正方形和长方形都拥有的共性行为，所以不能在矩形Rectangle中定义它
- 悖论的发生在于抽象类的定义有误，即将长方形当作了矩形（赋予基类其子类才能拥有的行为）
- “正方形悖论的实质是违背了Liskov替换原则”

火龙果•整理 uml.org.cn

# 正方形悖论—结论

●参考的定义——

**public class Rectangle**
**public class Square extends Rectangle**
**public class NotSquareRectangle extends Rectangle**



火龙果•整理
uml.org.cn

# GRASP模式

火龙果•整理
uml.org.cn

# 什么是GRASP模式

- General Responsibility Assignment Software Patterns——通用职责分配软件模式
- 它介于分析和设计之间，是在类被识别之后，如何向类分配职责的最基本模式
- 通常在进行用例分析与设计时，应用GRASP模式来确定类在协作中所要承担的职责，这些职责包括:
  - Knowing——掌握、或通过计算引申得到某些信息，以及相关的其它对象
  - Doing——自己做、或触发其它对象做某些事，以及控制和协调其它对象的行为
- GRASP模式的主要依据是面向对象的封装、抽象原理，和低耦合、高内聚设计原则等

火龙果•整理
uml.org.cn

# GRASP模式列表

| 模式 | 简述 |
| --- | --- |
| 信息专家 | 将职责分配给信息专家（类），它掌握了为履行职责所必需的信息（数据） |
| 创建者 | 如满足以下条件之一，将创建类A实例的职责指派给类B:<br>1）B包含了A；2）B聚合了A；3）B具有初始化A实例所需要的数据；4）B记录了A的实例；5）B紧密地使用A |
| 高内聚 | 分配职责时应保持高聚合度 |
| 低耦合 | 分配职责时应保持低耦合度 |
| 控制者 | 将处理系统事件的职责分配给如下的类:<br>1）代表整个系统、设备、子系统的类；2）代表一个用例场景、一次会话的协调者 |

火龙果•整理
uml.org.cn

# GRASP模式列表

| 模式 | 简述 |
|---|---|
| 多态 | 当相关选择或行为随着类型变化而变化时，将行为的职责—利用多态操作—指派给行为发生变化的类型 |
| 间接 | 将职责指派给一个中间对象，用来在其它构件或服务之间居中协调，以避免它们之间直接耦合 |
| 纯虚构 | 将一组高内聚的职责指派给一个虚构类，它不代表问题域中的任何概念，仅仅用来提供支持高聚合、低耦合与重用的行为 |
| 保护变量 | 识别预计的变量和常量，分配对应的职责以创建围绕它们的稳定接口，从而保护它们不对其它元素造成不期望的影响 |

火龙果•整理
uml.org.cn

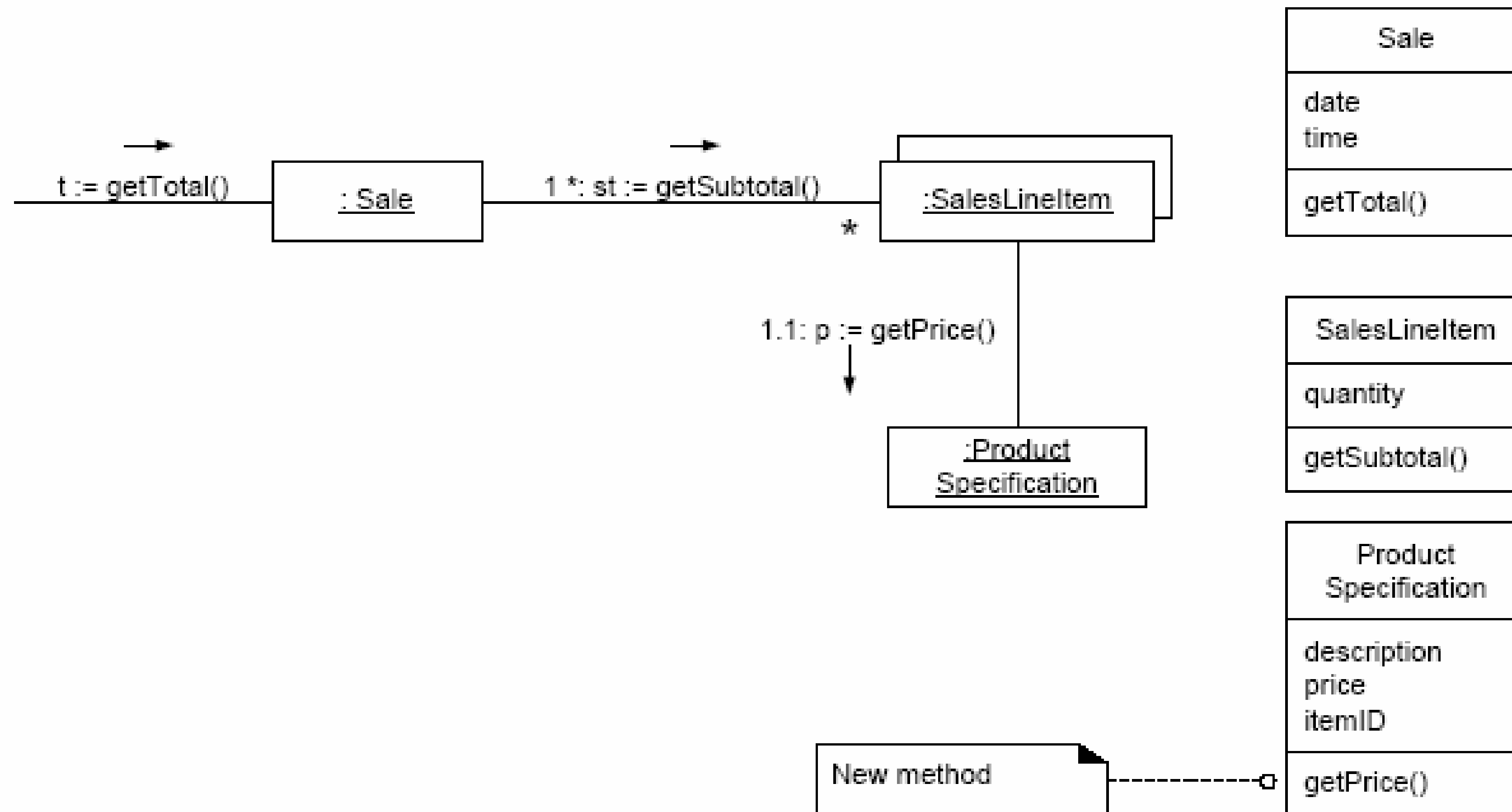
# 信息专家模式

- 将一个职责分配给信息专家——即掌握了为履行职责所必需的信息的类
- 对象只使用它们自己所包含的信息来完成任务，从而保持封装性，支持低耦合度
- 系统行为只分布于那些具有所需信息的类中，从而支持高聚合度

火龙果•整理
uml.org.cn

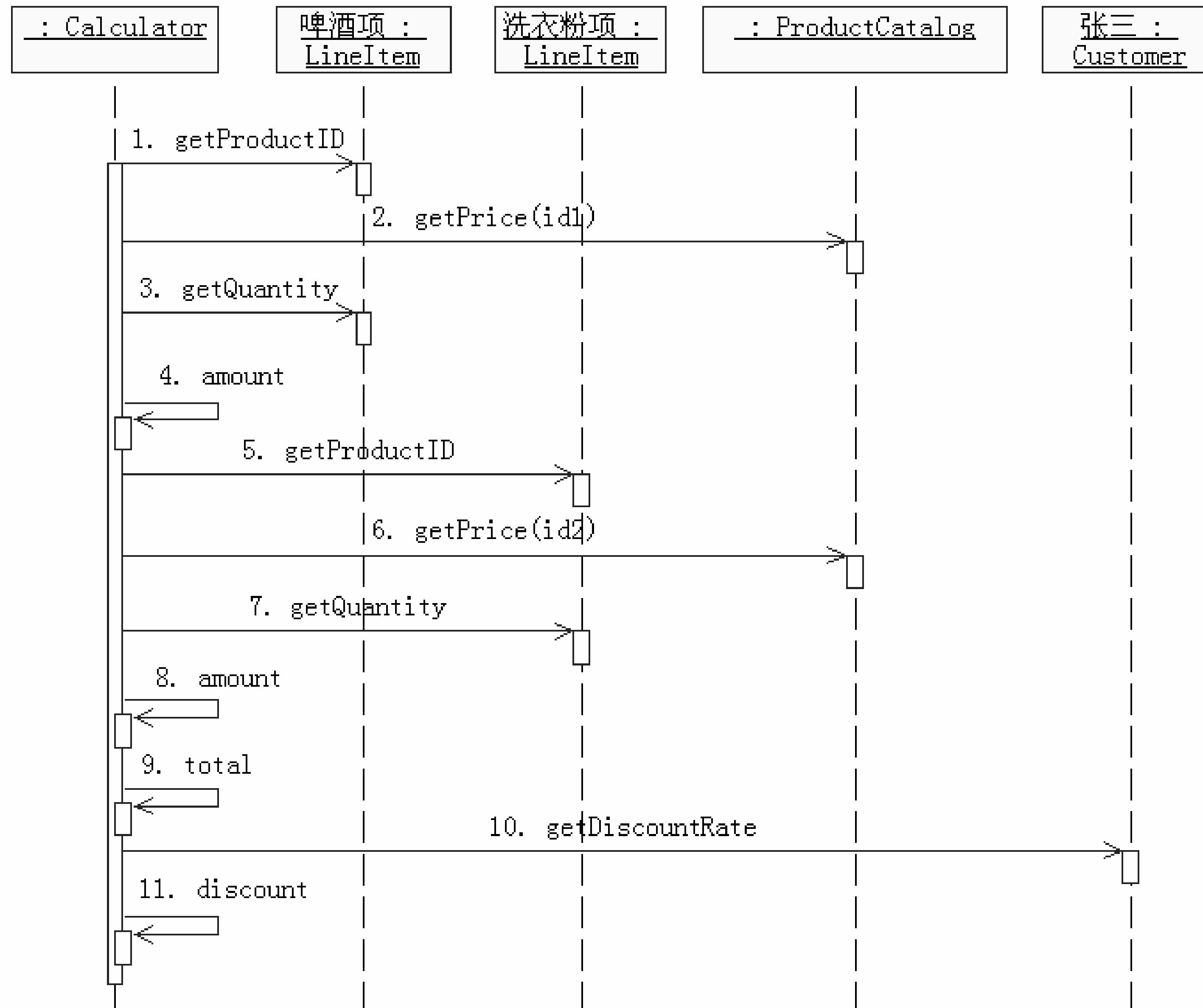
# 示例：信息专家模式

- 所有的类承担的职责都只依赖其本身所掌握的信息



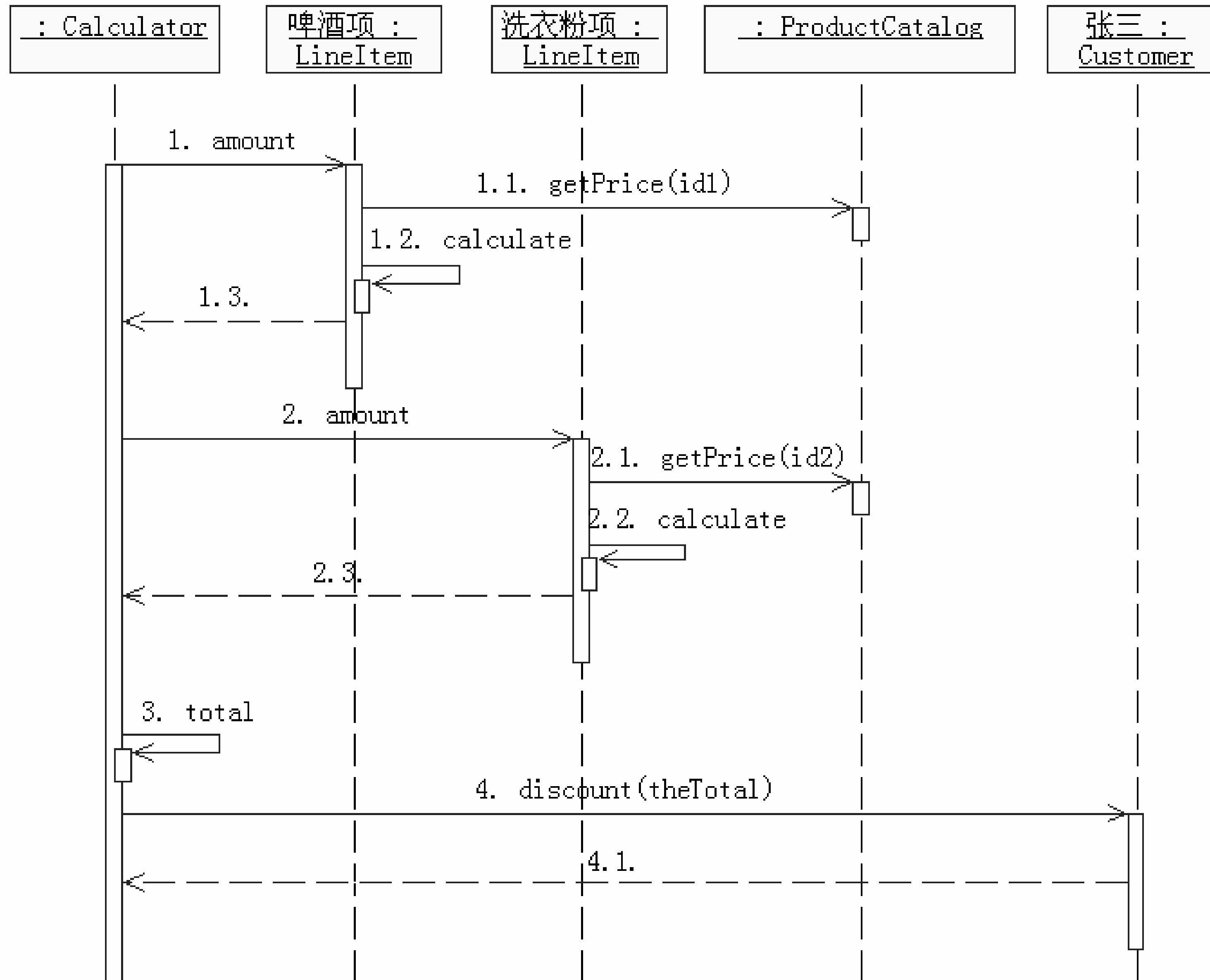
火龙果•整理
uml.org.cn

# 实例：违反信息专家模式



火龙果•整理
uml.org.cn

# 实例：应用信息专家模式



火龙果•整理
uml.org.cn

# 信息专家模式的不足

- 在某些场合下，为类分配职责时如果仅仅只考虑其掌握的信息是否满足要求，往往会得到一个并不优雅的设计
- 考察前面Sale类的例子，我们需要对其进行持久化操作（将Sale对象实例保存到数据库中），那么按照信息专家模式，我们会在Sale类上直接添加一个Save方法；然而这样做，使得Sale类严重依赖于数据库存取的细节，破坏了职责的内聚性，同时也违反了separation of concerns隔离关注面的原则
- 在GRASP等模式之上，实际上存在更为重要的东西——设计原则

火龙果•整理
uml.org.cn

# 创建者模式

- 将创建类A实例的职责指派给类B实例，如果满足以下条件之一的话：

1）B包含了A;

2）B聚合了A;

3）B具有初始化A实例所需要的数据（即B是创建A的实例这项任务的信息专家）;

4）B记录了A的实例;

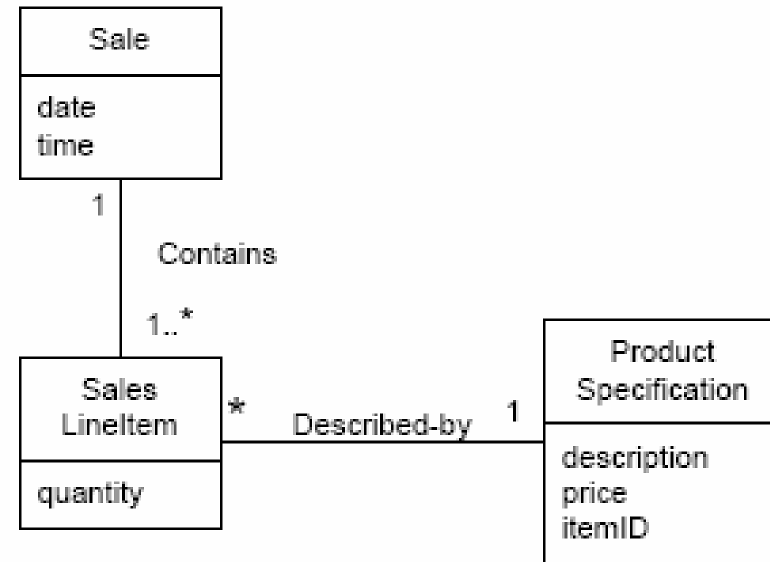
5）B紧密地使用A

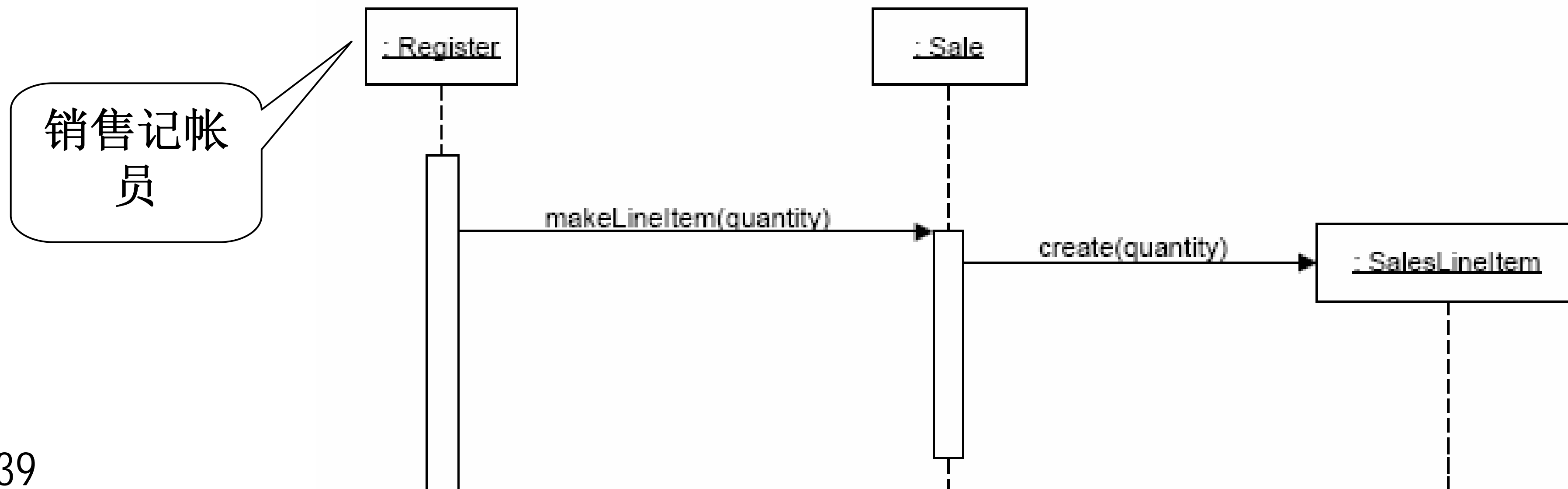
- 如果满足两条以上条件的话，可以让B同时还聚合A

火龙果•整理
uml.org.cn

# 示例：创建者模式

Sale实例将包含（记录）SalesLineItem实例，因此将创建类SalesLineItem实例的职责指派给类Sale实例



火龙果•整理
uml.org.cn

# 低耦合模式

- 在为类分配职责时应保持低耦合度
- 耦合度——是对一个类与其他类关联、知道其他类的信息或者依赖其他类的强弱程度的度量
- 高耦合——过度依赖于多个其它类；它带来的问题有：相关类的变更会迫使自身也发生改变；难以独立地被理解；难以重用（需要其它被依赖的类同时使用）

火龙果•整理
uml.org.cn

# 低耦合模式

- 如果不以重用为目标，耦合就不那么重要了
- 如果类等元素是稳定的，耦合在此并不会带来负面问题；在类之间保持中等程度的耦合是正常和必要的，否则类的协作就无从谈起

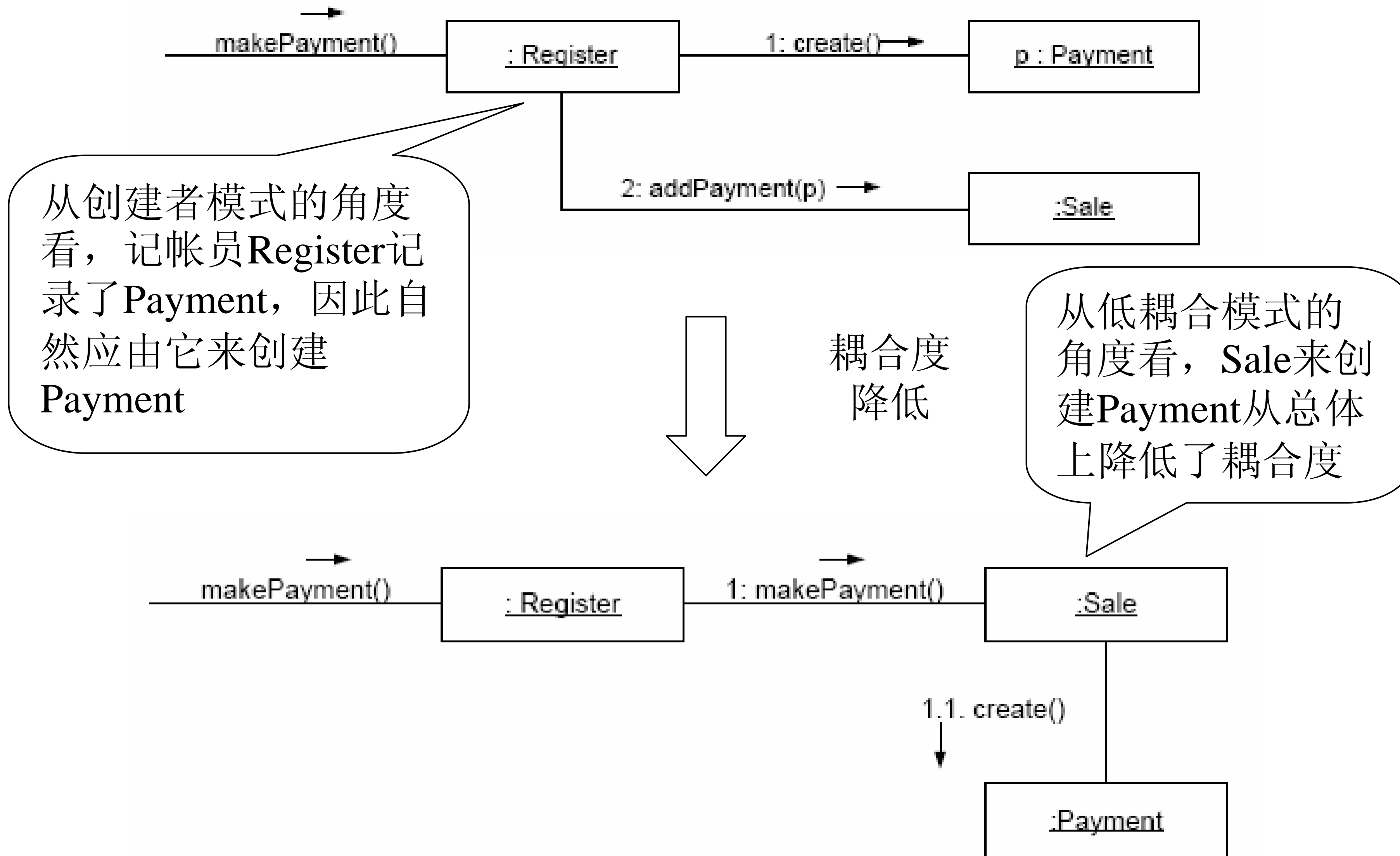
火龙果•整理
uml.org.cn

# 耦合的各种场景

- 类型X有一个属性（数据成员或实例变量）引用了类型Y的实例或类型Y本身;
- 类型X有一个方法以各种方式引用了类型Y的实例或类型Y本身；例如，类型X引用了类型Y的局部变量或参数，或者从一个消息返回的对象类型是类型Y的实例
- 类型X是类型Y的一个直接或间接的子类
- 类型Y是一个接口，类型X实现了这个接口

火龙果•整理
uml.org.cn

# 示例：低耦合模式



火龙果•整理
uml.org.cn

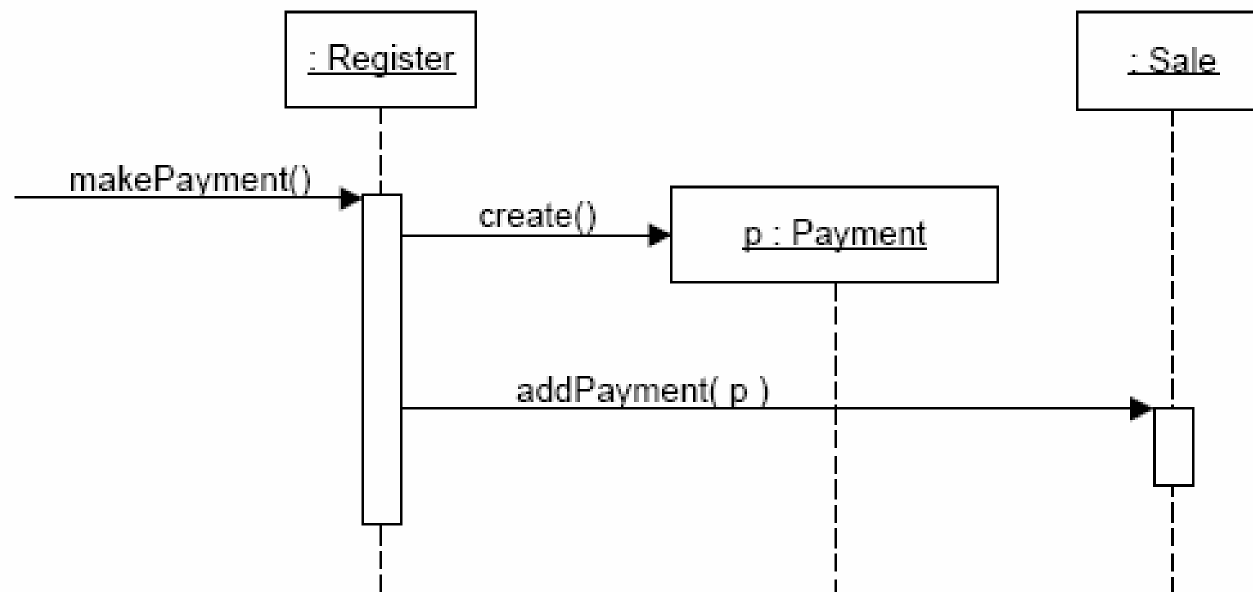
# 高内聚模式

- 在为类分配职责时应保持高内聚度
- （功能）聚合度——是对一个类中的各个职责之间相关程度和集中程度的度量
- 高内聚：一个类只（在一个功能领域内）承担高度相关的职责，并且不至于工作过度
- 低内聚：一个类做了许多不相关的工作，或者工作过度；它带来的问题有：难以理解；难以重用；难以维护；脆弱、易受变更的影响

火龙果•整理
uml.org.cn

# 示例：高内聚模式

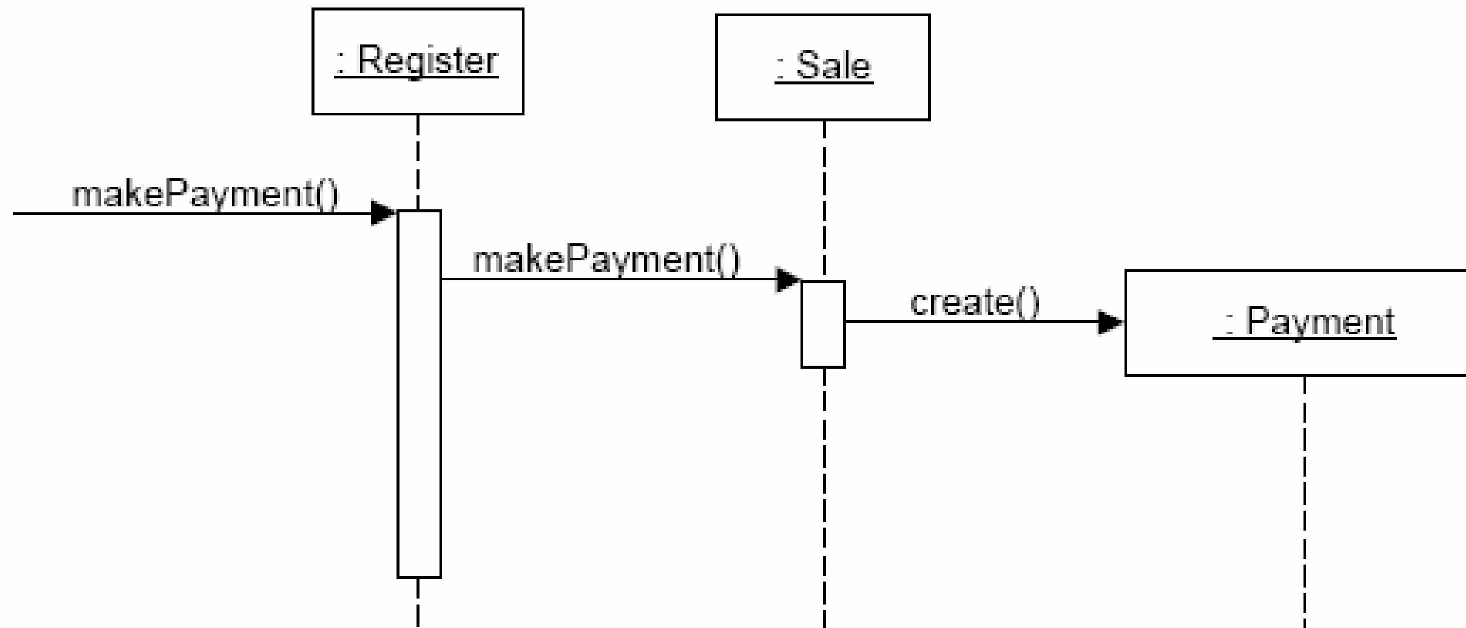
- 随着Register类增添其它的职责，让它继续来承担创建Payment的职责，将显著消弱Register行为的内聚度



火龙果•整理
uml.org.cn

# 示例：高内聚模式

- 显然由Sale来创建Payment，不但从总体上降低了耦合度，同时也提高了内聚度



火龙果•整理
uml.org.cn

# 控制者模式

- 控制者是处理系统事件的非边界类对象
- 将处理系统事件消息的职责分配给:

-代表整个系统行为的类（Façade controller 门面控制者）

-代表真实世界中自主地参与交互的类（Role Controller角色控制者，实质上对应到分析模型中的服务型控制类）

-代表一个用例中所有事件的处理者的类（Use case controller用例控制者，即对应到分析模型中的普通控制类）

火龙果•整理
uml.org.cn

# 控制者模式

- 推论:

外部接口对象和表示层（如窗口、Applet、Application、视图、文档等）不能用作控制者来承担处理系统事件的职责

使用同一个控制类处理同一个用例中的所有事件

- 本模式与RUP的分析类概念一脉相承，即与控制类的职责非常类似

火龙果•整理
uml.org.cn

# 系统操作与控制者

- 系统事件是由外部使用者发起的高层事件，是一个外部输入事件。
- 这些事件将对应于系统操作：系统为响应这些事件而执行的操作。
- 系统操作反应了企业或领域过程中的主要行为，应该由领域层的对象（即控制者）来处理，而不是在系统的接口层、表示层或应用层中被处理。

火龙果•整理
uml.org.cn

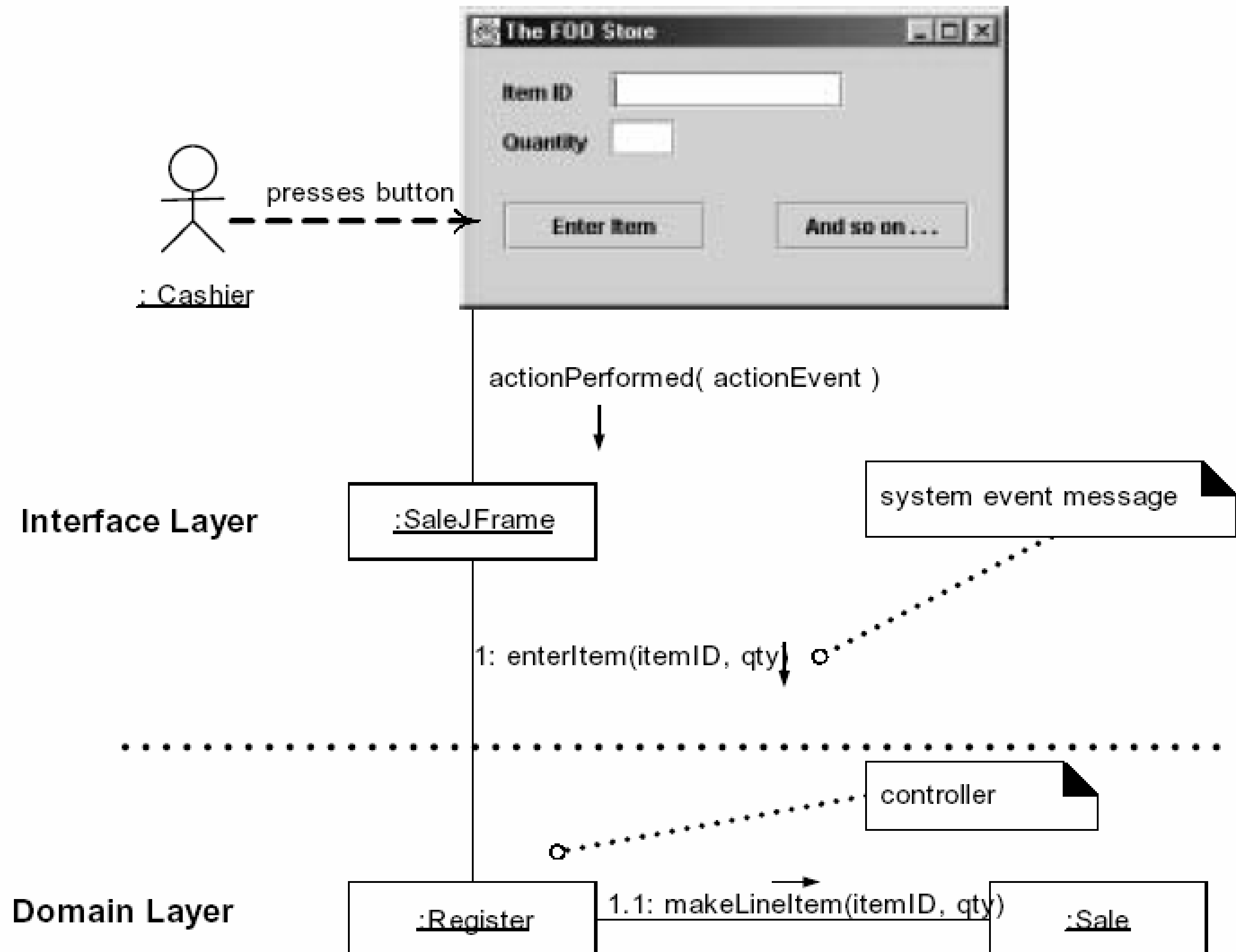
# 用例控制者

- 如果当把职责分配给其他种类的控制者时，将会导致低内聚或高耦合，或者当许多系统事件跨越了不同的过程时，可以选择使用用例控制者
- 增加了构件的可重用性
- 能够把握用例的状态
- 要警惕肥控制者，让它承担过多的职责

火龙果•整理
uml.org.cn

# 示例：控制者模式

- Register是用例控制者，承担了处理系统事件的职责



火龙果•整理
uml.org.cn

# 思考：用例行为分解与复用

- ✓ 软件复用的努力通常集中在设计层面上
- ✓ 用例的结构实质上是系统行为的结构，它提供了在行为层面实现复用的机会
  - – 调用包含用例，子用例对行为的继承，通过扩展用例扩展不同的行为组合，它们至少在描述上实现了复用
  - – 将用例结构在行为描述上的复用转化为实施上的复用，将把软件复用度提到更高的水平（类似于老的功能复用）

火龙果•整理
uml.org.cn

# 思考：用例行为分解与复用

- 这意味着系统的架构将反映用例的结构，当用例表达的需求发生变更时，其对构架的影响将被封闭在对应的局部范围内（针对于被包含用例、父用例、子用例、扩展用例、基用例等的实现部分）
- 可以使用UseCase Controller模式来实现这个目标:
  - 每个被复用的用例都有对应的独立UseCase Controller类，这个类将在系统中依照用例结构的关系来被复用;
  - 泛化关系中的子用例UseCase Controller类将继承对应父类的行为，可以在此应用Template Method模板方法模式。

火龙果•整理
uml.org.cn

# 思考：用例备选流实现

- ✓ 用例中的备选事件流通常会成为需求变更发生的主要场所
- ✓ 在用例详述中分别描述基本流和备选流，首先在需求描述工作上减少了需求变更的影响
- ✓ 如果能够在实施层面也能分隔基本流和备选流，则系统应对需求变更的能力将更强
  – 为基本流和多个备选流分别开发各自的UseCase Controller，通过面向对象的动态绑定功能，在运行时刻依据条件逻辑判断来替换Controller对象实例，从而实现用例行为的分支执行

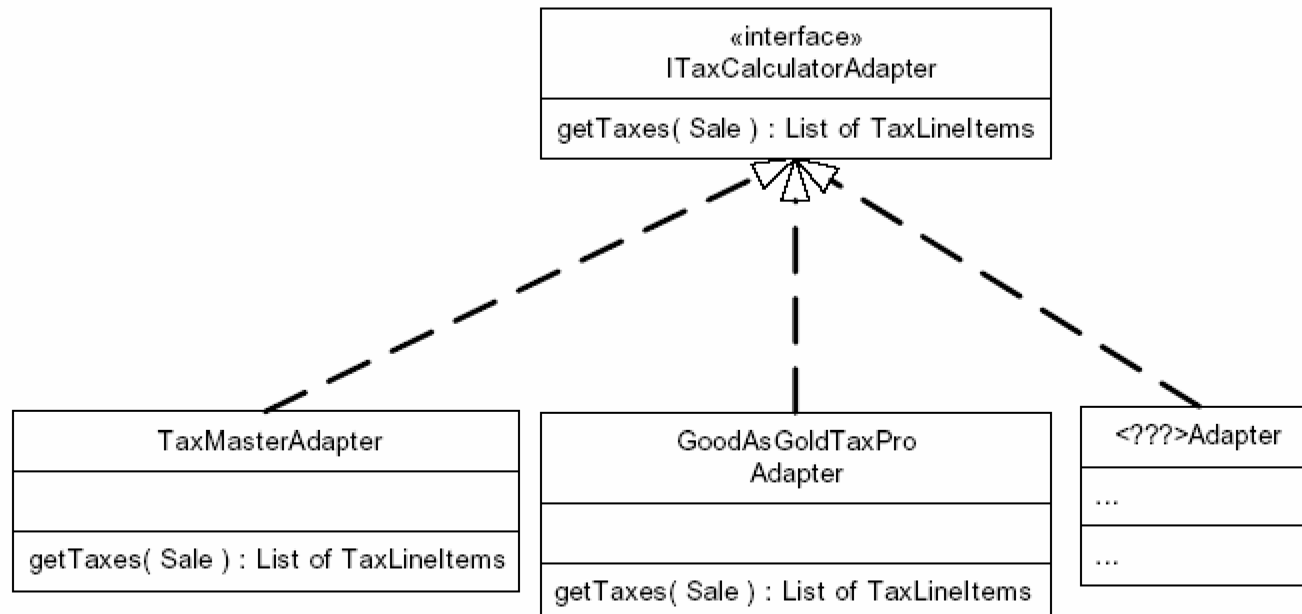
火龙果•整理
uml.org.cn

# 多态模式

- 当相关选择或行为随着类型变化而变化时，将行为的职责——利用多态操作——指派给行为发生变化的类型
- 推论：不要通过测试对象的类型，并使用条件逻辑判断来选择基于类型的可选执行分支
- 通过条件逻辑判断来选择可选执行分支，是程序中常见的处理场景，但这种方式在新的条件分支出现时引起的变动太大
- 一个基于通过多态来分配职责的设计可以很容易地被扩展以处理新的变化（包括上述新的执行分支）

# 示例：多态模式

- 不同的计税类实现了getTaxes()接口方法，利用多态操作，提供了不同的计税途径；而当新的计税法出现时，客户代码不受影响






火龙果•整理
uml.org.cn

火龙果•整理
uml.org.cn

# 纯虚构模式

- 将一组高内聚的职责指派给一个虚构类，它不代表问题域中的任何概念，仅仅用来提供支持高聚合、低耦合与重用的行为
- 类的设计通常通过两种途径:

（针对概念的）表示分解representational decomposition

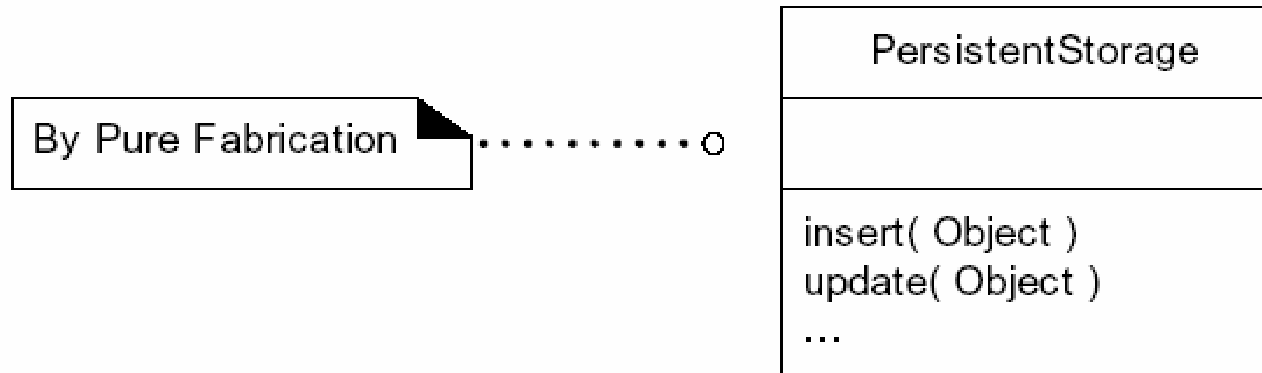
（纯粹的）行为分解behavioral decomposition

- 一个纯虚构通常基于行为分解，是一种以功能为中心的对象，而与问题域概念无关，通常为架构中高层元素提供服务

火龙果•整理
uml.org.cn

# 示例：纯虚构模式

- PersistentStorage纯粹是虚构类，它用来提供持久化的服务支持，从而帮助需要持久化的Sale类与持久化机制去耦合



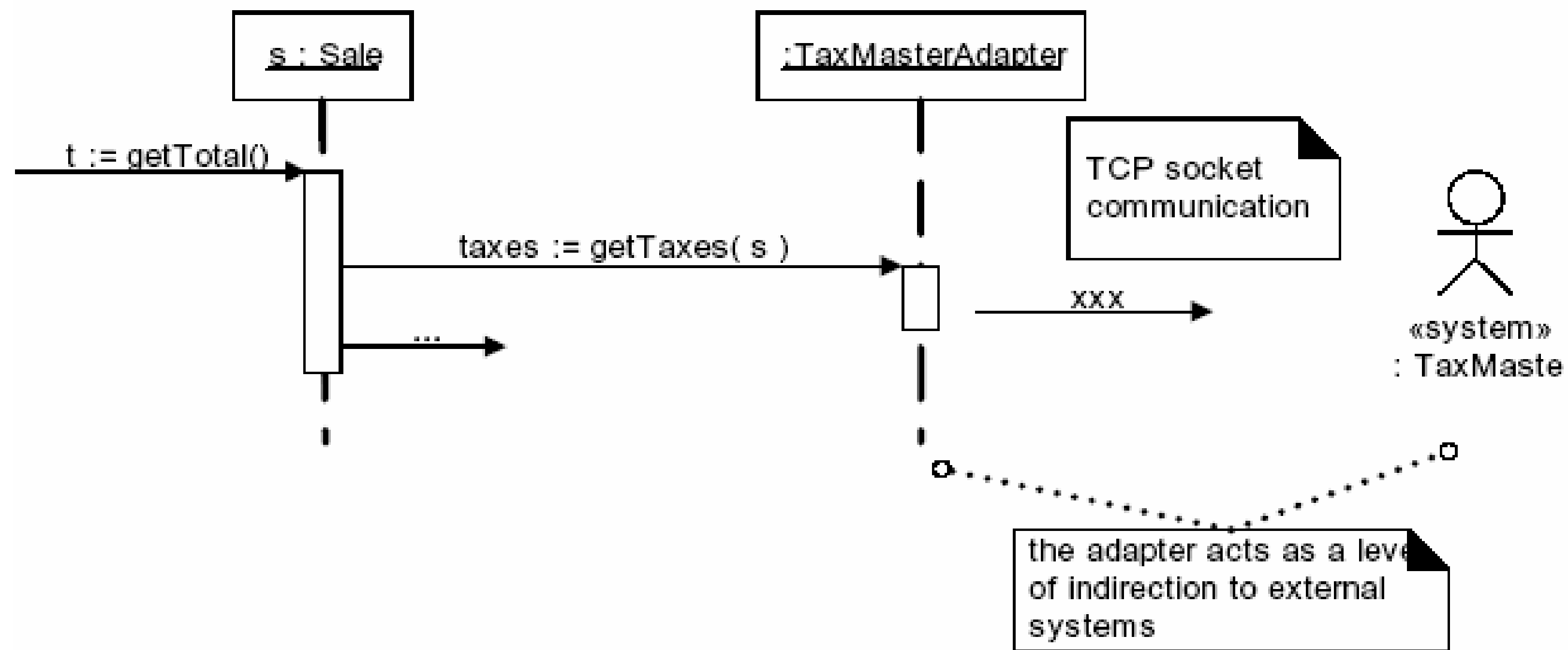
火龙果•整理
uml.org.cn

# 间接模式

- 当将职责指派给一个中间对象，用来在其它构件或服务之间居中协调，以避免它们之间直接耦合
- 实例：发布-订阅，观察者，适配器等

# 示例：间接模式

- TaxCalculatorAdapter充当了Sale与外部计税器之间的一个中间对象，TaxMasterAdapter通过多态重载getTaxes()接口实现，在此成为具体的中介者



火龙果•整理
uml.org.cn

# 保护变量模式

- 识别预计的变量和常量，分配对应的职责以创建围绕它们的稳定接口，从而保护它们不对其它元素造成不期望的影响
- 封装、接口、多态、间接等原则和模式是实现保护变量模式的基本途径
- 数据驱动设计、规则引擎、解释执行、反射、元数据设计等则是实现保护变量模式的更高级手段
- 不要与陌生人交谈模式是保护变量模式的一个特例

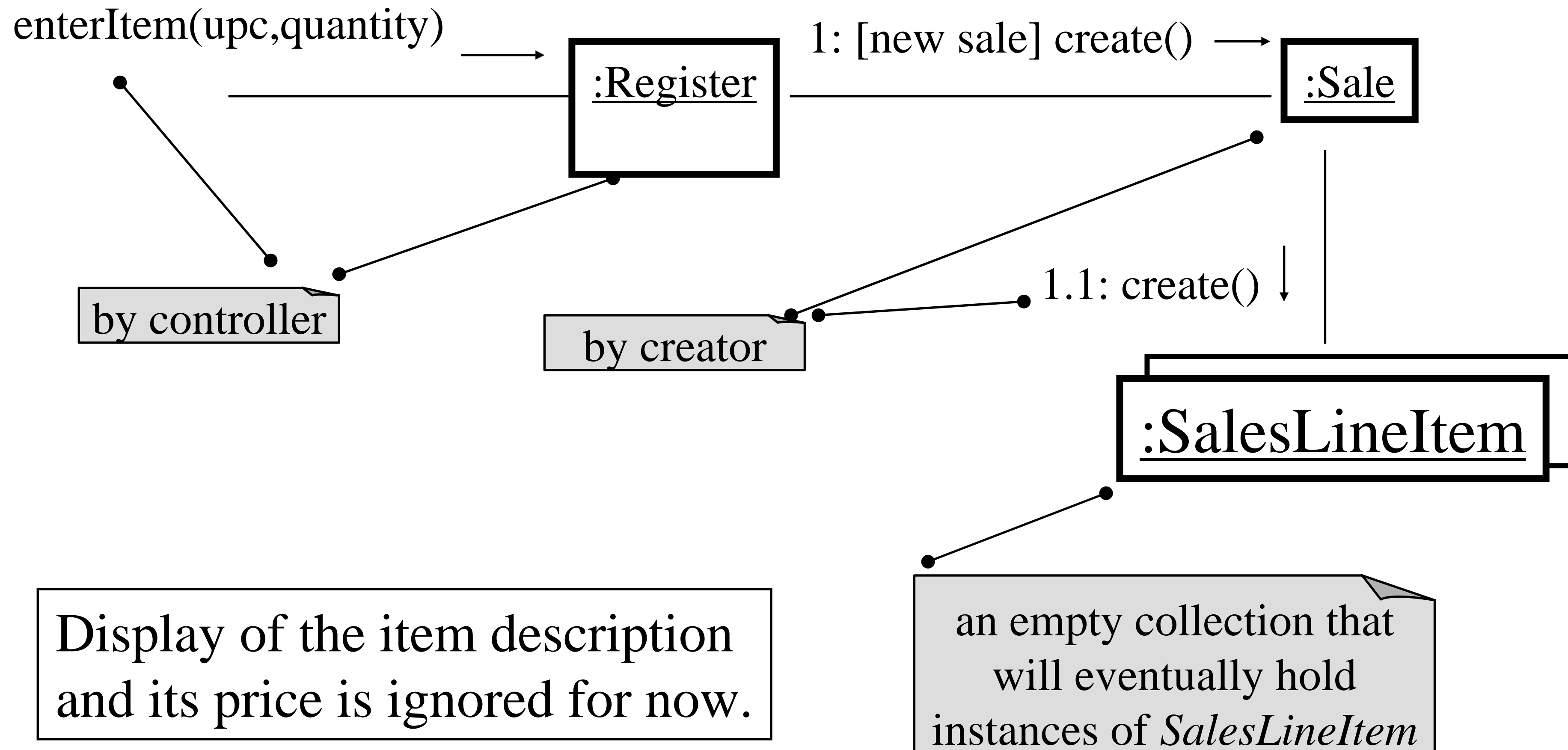
火龙果•整理
uml.org.cn

# 应用GRASP模式精化用例分析

- 首先应用控制者模式选择适当的controller来协调用例的协作过程，并识别其它类型的控制者（role controller）来分担对应职责
- 然后应用创建者模式来识别对象实例的创建关系，用例分析时可以忽略边界类对象、控制类对象的创建细节，但领域对象（实体）的创建关系应当被明确地定义
- 接下来，应用信息专家模式，找出所有的信息专家experts，并据此分配对应的职责
- 此后，应用耦合模式来对对象职责的分配进行适当的调整以便降低系统的耦合度
- 最后，检查整个协作过程中的问题，并予以解决

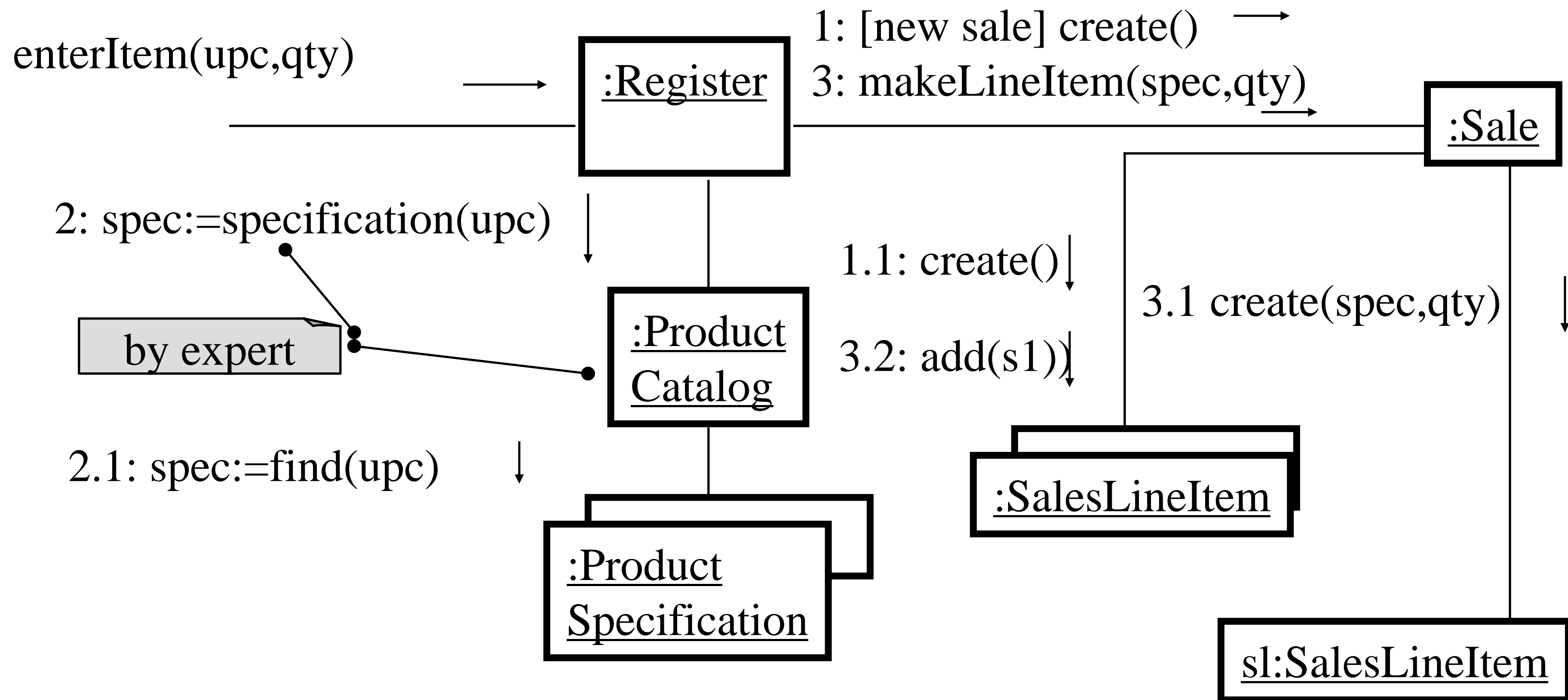
火龙果•整理
uml.org.cn

# 示例：精化用例分析



火龙果•整理
uml.org.cn

# 示例：精化用例分析



火龙果•整理
uml.org.cn

# 参考资料

<UML Distilled: A Brief Guide to the Standard Object Modeling Language>

<Applying uml and patterns 2nd>