对象设计

Chapter 10

北京航空航天大學
SCHOOL OF SOFTWARE BEIHANG UNIVERSITY 软件学院

设计类

- 设计类
  - 设计模型的构造块
  - 设计类是已经完成了规格说明并且达到能够被实现程度的类
  - 来源于问题域和解域
    - 通过分析类的精化得到的问题域—添加实现细节
    - 解域，提供了能够实现系统的技术工具

设计类剖析

- 在分析中，只要尽量捕获系统需要的行为，而完全不必考虑如何去实现这些行为
- 在设计中，则必须准确地说明类是如何履行它们的职责
  - 完整的属性集合，包括详细说明的名称、类型、可视性和一些默认值
  - 将分析类指定的职责转化成一个或多个操作的完整集合

类设计的主要内容

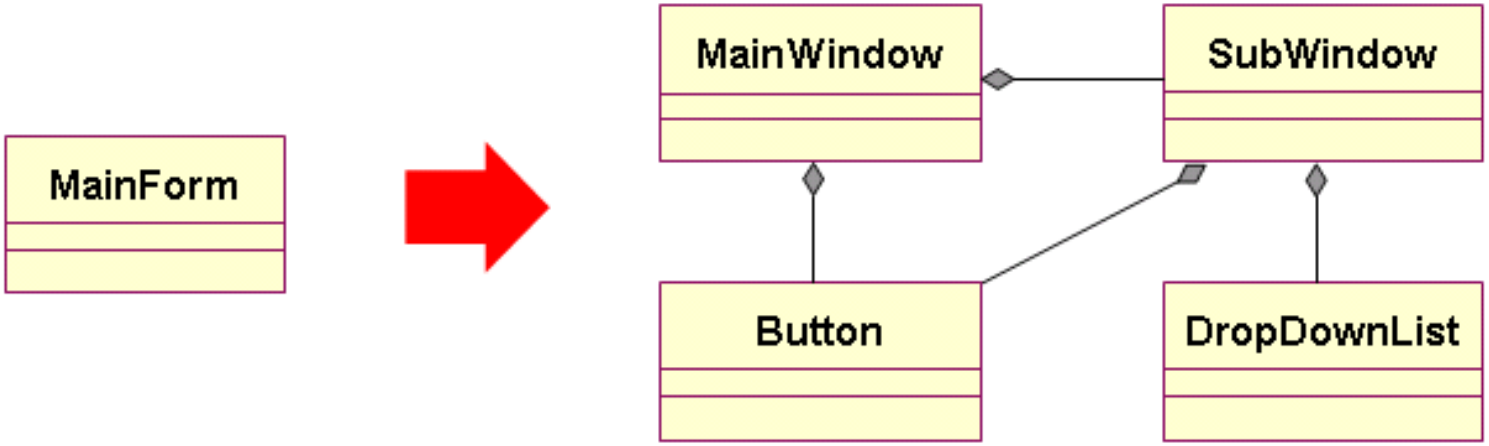
- 1.创建初始设计类
- 2.定义操作
- 3.定义方法和状态
- 4.定义属性
- 5.定义关系

1.创建初始设计类

- 创建初始设计类，需要考虑
  - 类构造型
    - 边界
    - 控制
    - 实体
  - 架构相关技术的应用
    - 架构框架
    - 数据库访问
    - 安全、性能等非功能需求
  - 设计原则和可适用的设计模式
    - 可扩展性、可复用性的设计质量属性

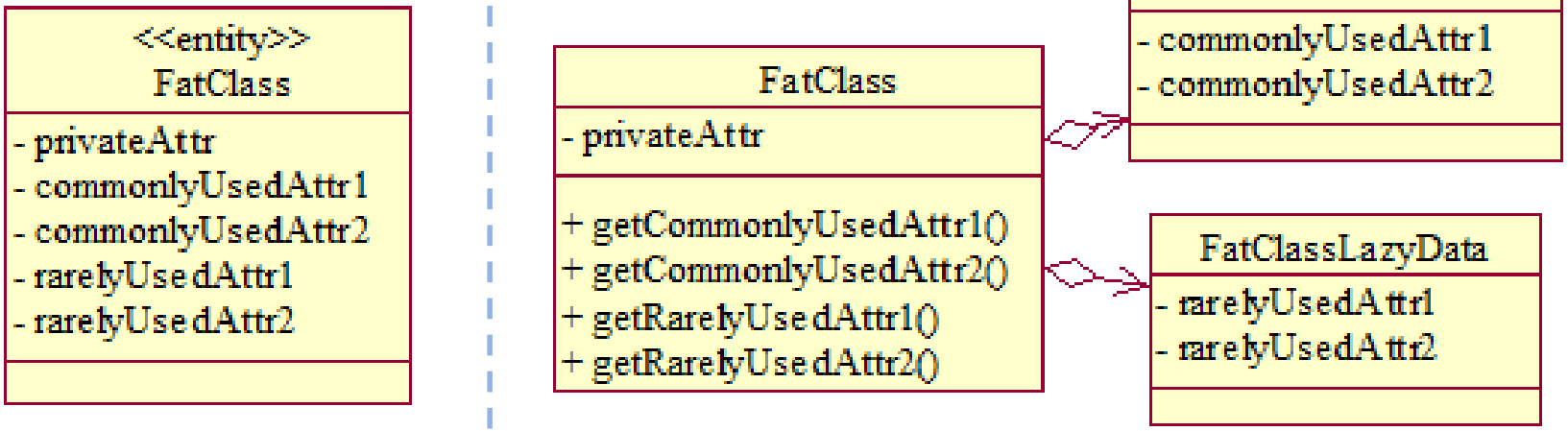
边界类的设计策略

- 用户界面（UI）边界类
  - 使用什么用户界面开发工具
  - 哪些界面可以用开发工具直接创建
- 外部系统接口边界类
  - 通常建模为独立的接口和具体类单独设计



实体类的设计策略

- 实体对象通常是被动的和持久性的
- 性能需求可能要对实体类进行重构
- 数据架构、数据库等技术方案影响实体类



控制类的设计策略

- 分层架构会影响到系统控制类的设计
- 如何处理控制类
  - 是否真正地需要它们?
  - 它们应当被分开吗?
- 下列情况下，控制类可能变为真正的设计类
  - 封装非常重要的控制流行为
  - 封装的行为很可能变化
  - 必须跨越多个进程或处理器进行分布
  - 封装的行为要求一些事务管理

调整控制类

- 调整控制类的基本策略
  - 提供公共控制类：多个用例若有同样活动的控制类，将其整合起来，把相同部分作为一个新的控制类
  - 分解复杂控制类：用例的控制流程过于复杂，则可以考虑根据不同的控制业务分解成多个控制类

2. 定义操作

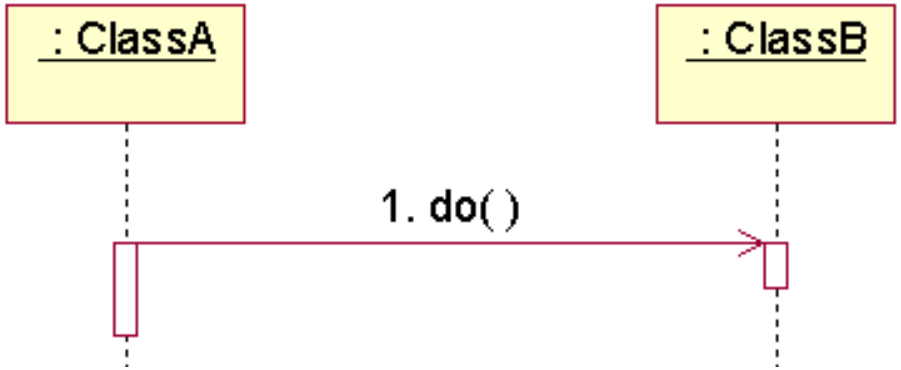
- 操作是类的行为特征，描述了该类对于特定请求做出应答的规范
  - 同一个类的每个操作都具有唯一签名，通过描述操作的签名完成类操作的定义

可见性 操作名(参数方向 参数名称:参数类型[多重性]=缺省值, ...):
返回类型[多重性]

  - UML中的四种可见性
    - 公有（+）、私有（-）、保护（#）和包（~）

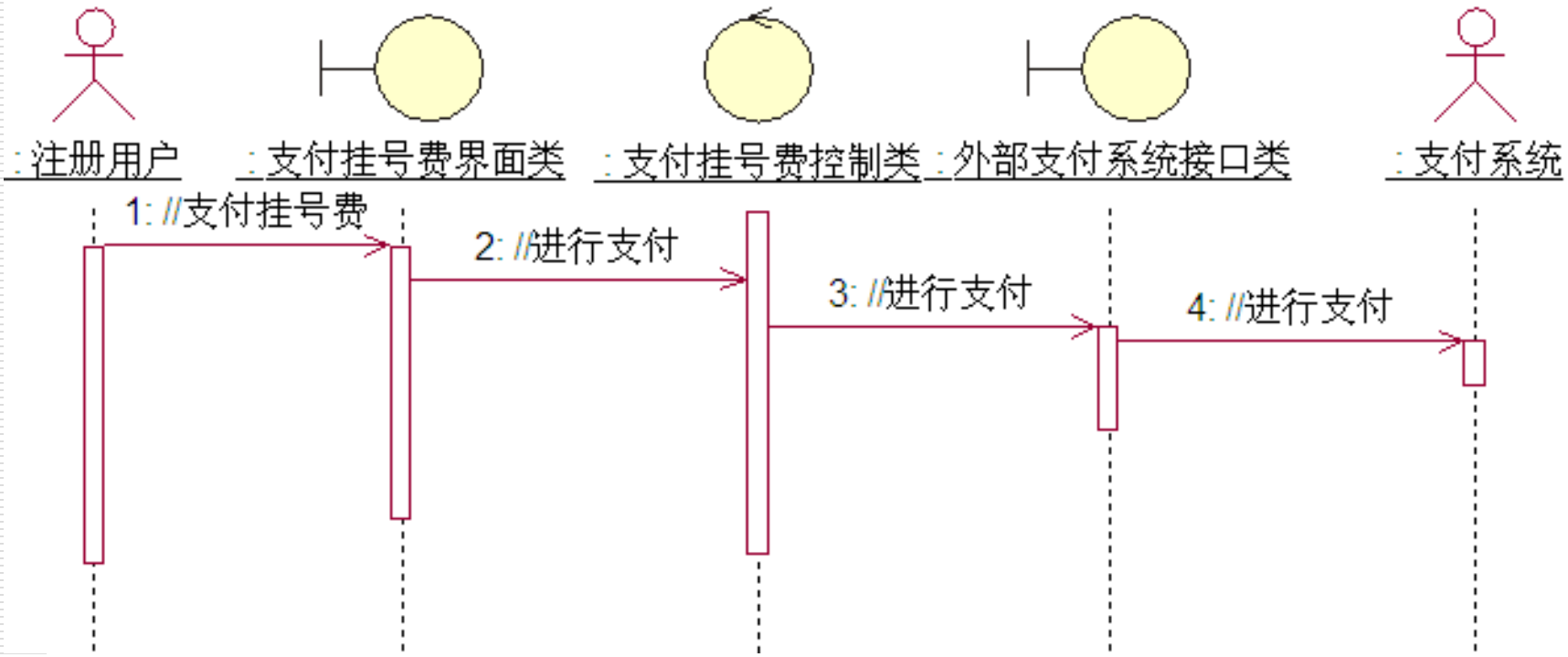
发现操作

- 显示在交互图中的消息

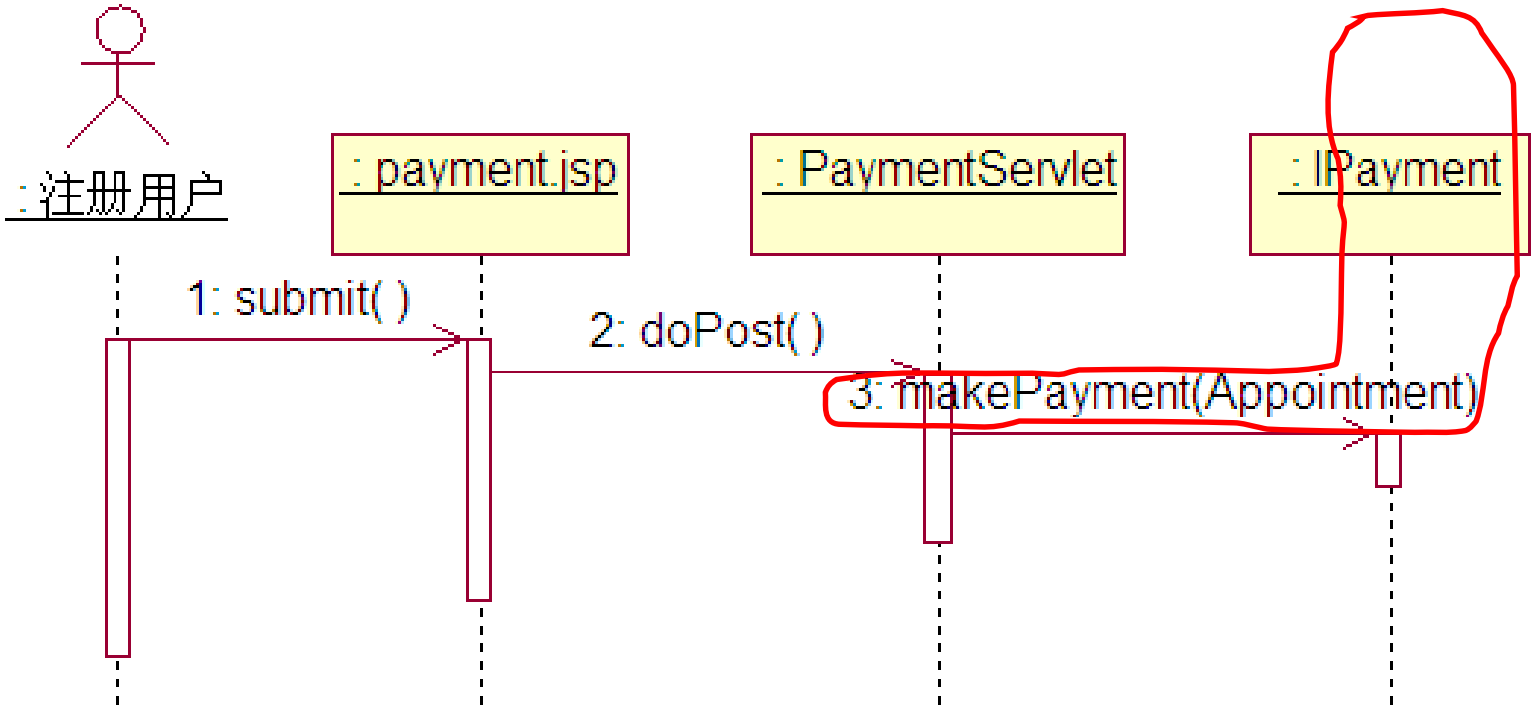


- 其它独立功能的实施
  - 自身的管理功能(构造、析构等)
  - 类复制的需要(测试类是否相等，创建类副本等)
  - 其它操作机制的需要(垃圾收集、测试等)

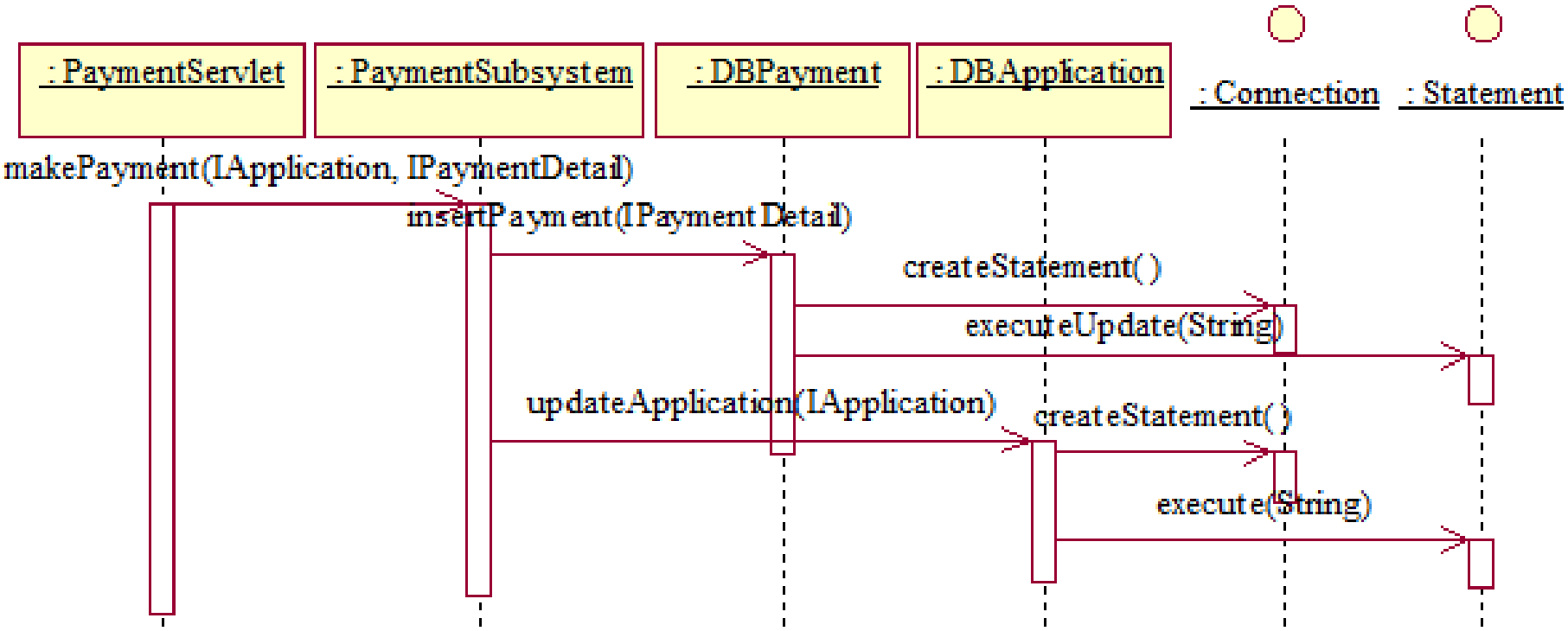
实例：引入子系统接口之前(分析)



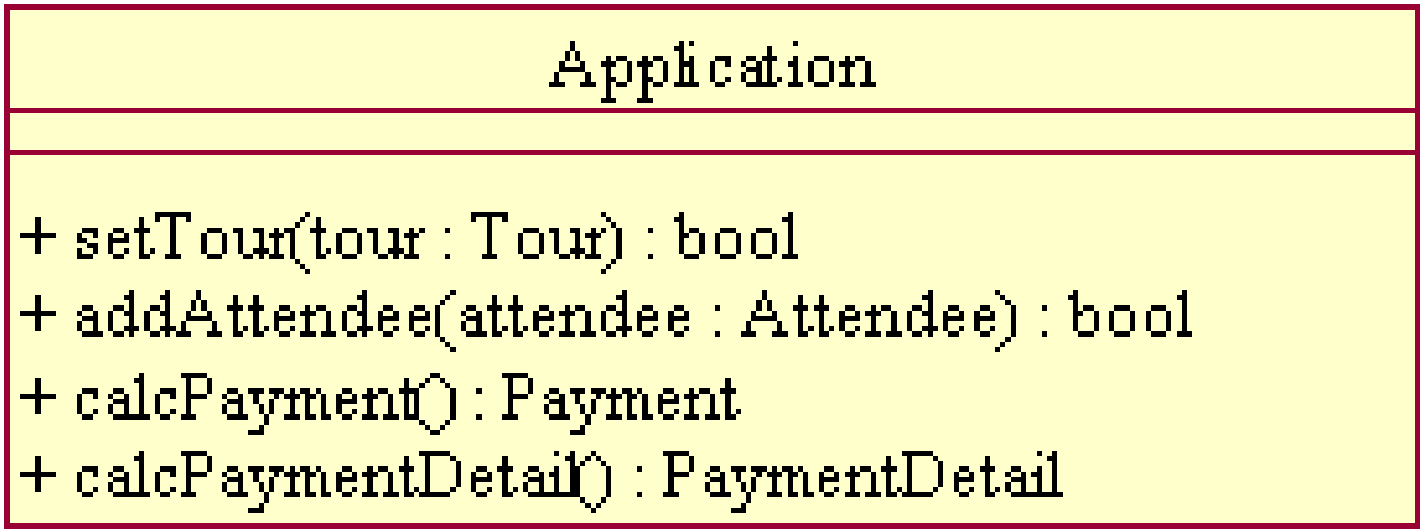
实例：设计中将边界类定义为接口



通过交互图描述接口操作的实现



示例：从交互图中定义类的操作

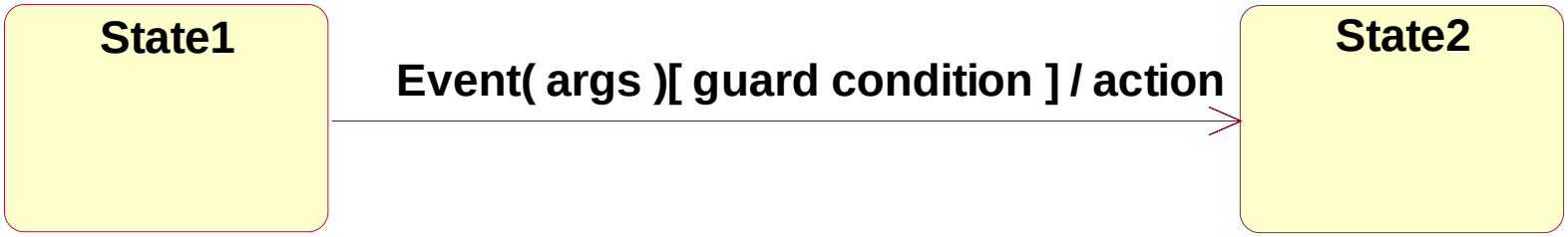


3.定义方法和状态

- 方法(Method)是指操作的具体实现算法
  - 详细说明操作实现的细节
  - 可采用UML活动图对方法进行建模
- 考虑的内容
  - 特殊算法
  - 要使用到其它对象和操作
  - 属性和参数如何实现和使用
  - 关系如何实现和使用
- 方法的实现往往受对象的状态影响，采用状态机图建模

状态机图

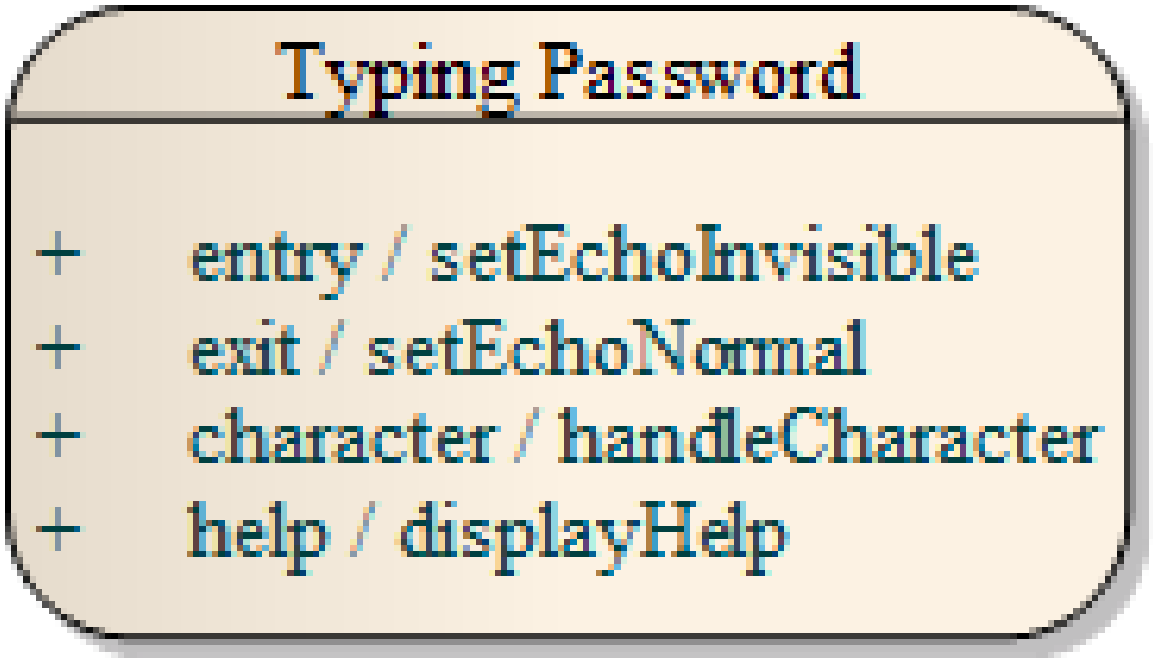
- 状态机（state machine）是一种行为，说明对象在它的生命周期中响应事件所经历的状态变化过程以及对那些事件的响应
- 状态机图（state machine diagram）是描述状态机的一种图，是由状态和转移组成的有向图
  - 描述了一个对象的发展历史



状态

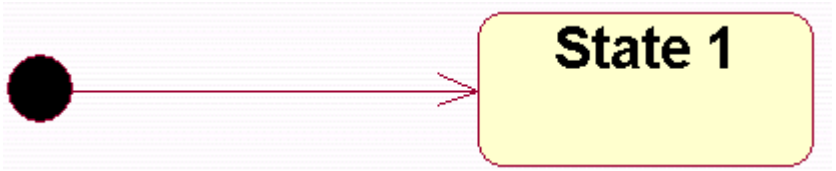
- 状态（state）描述了对象的生命周期中所处的某种条件或状况
  - 在此期间对象将满足某些条件、执行某些活动或等待某些事件的发生
- 复杂状态的内部结构
  - 入口动作、出口动作
  - 状态活动、内部转移
  - 延迟事件
  - 子状态机

状态


Typing Password
+ entry / setEchoInvisible
+ exit / setEchoNormal
+ character / handleCharacter
+ help / displayHelp

两个特殊状态

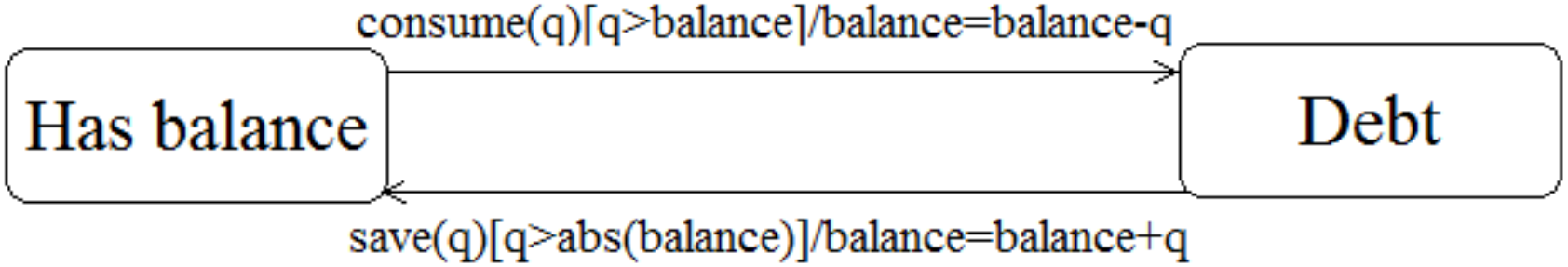
- 初始状态
  - 当一个对象创建时所进入的状态
  - 必须的
  - 只能有一个初始状态



- 最终状态
  - 显示一个对象生命的结束
  - 可选的
  - 可能有多个

转移

- 转移（transition）是从一个源状态到一个目标状态之间的一个有向关系，包括三个要素
  - 事件(event)：事件发生时转移才有可能发生
  - 守卫条件(guard condition)：当事件发生时，守卫条件为真，则发生转移；否则忽略该事件
  - 动作(action)：当转移发生时所执行的动作，该动作应当是原子操作。



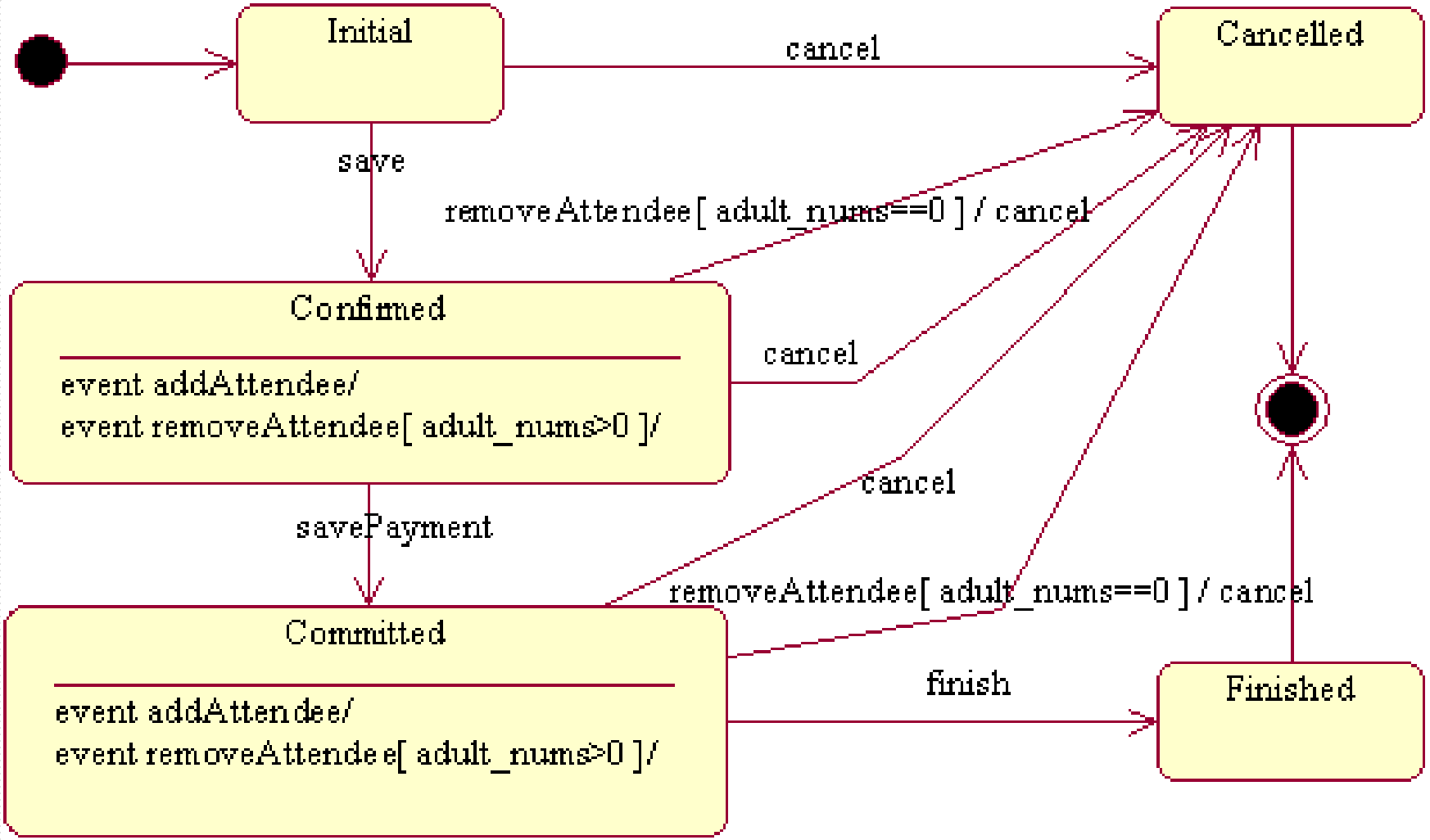
状态建模

- 状态建模可以针对一个完整的系统（或子系统），也可以针对单个类对象或用例（或用例的某个交互片段），
  - 目标是关注在内部哪些事件导致状态改变
  - 在类设计期间，针对那些受状态影响的对象进行状态建模
- 需要考虑的问题
  - 哪些对象有重要的状态
  - 如何确定一个对象可能的状态
  - 如何将状态机图映射到模型的其它部分

哪些对象需要进行状态建模

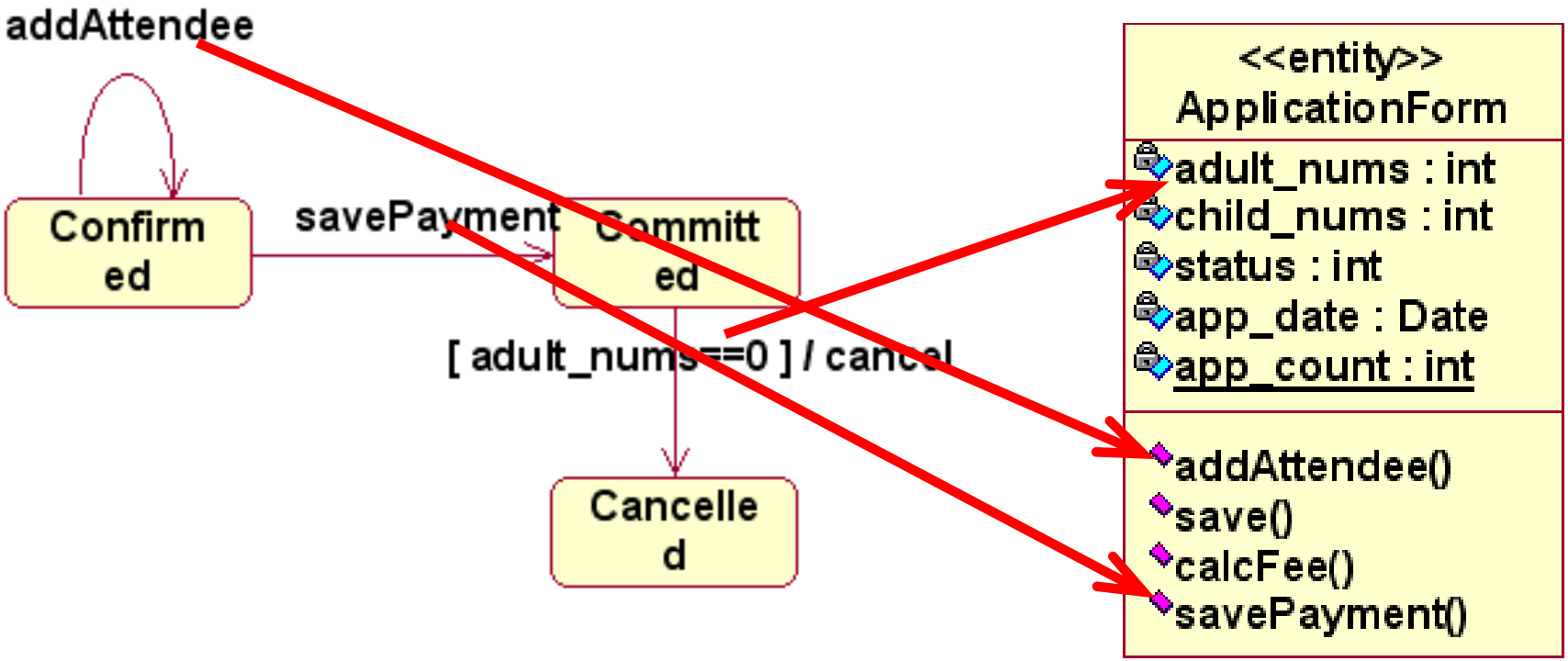
- 需要进行状态建模的对象
  - 其职责由状态转移所阐明的对象
  - 复杂的状态受控的用例
- 以下对象不需要进行状态建模
  - 直接映射到实现的对象
  - 非状态受控的对象
  - 只有一个状态的对象

实例：申请对象的状态机图



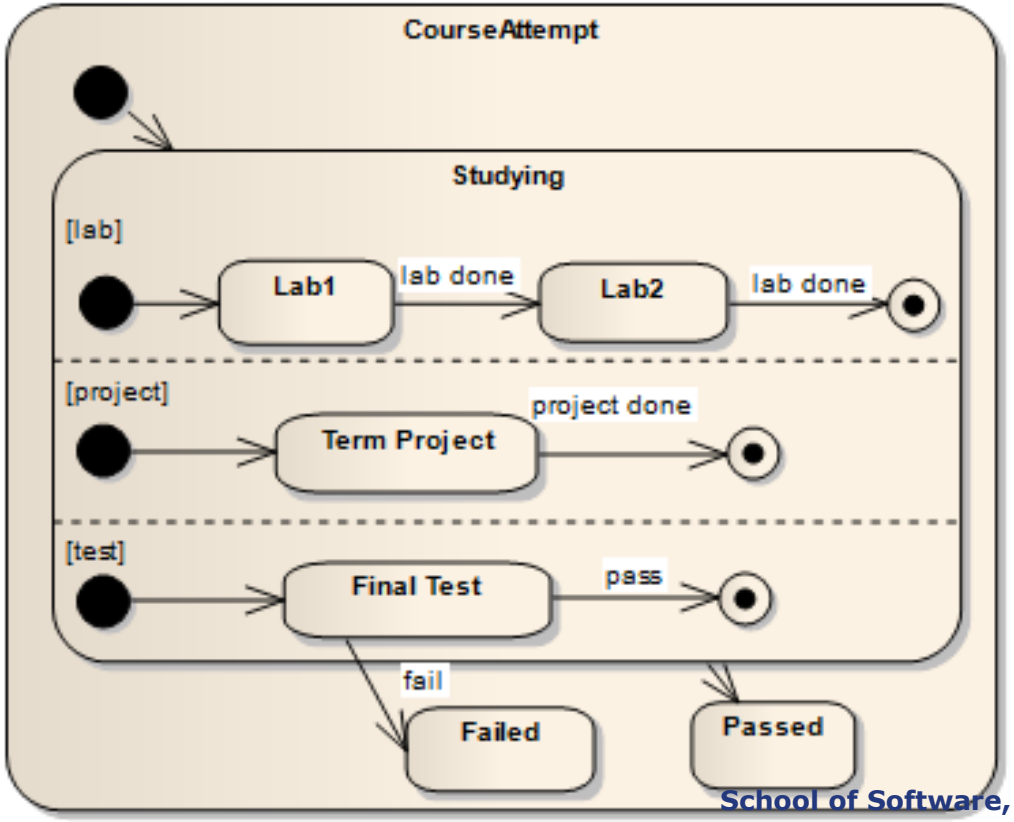
状态机图影射到模型的其它部分

- 事件可以映射到操作
  - 方法应当使用状态特定信息来更新
- 状态常常使用属性来表示



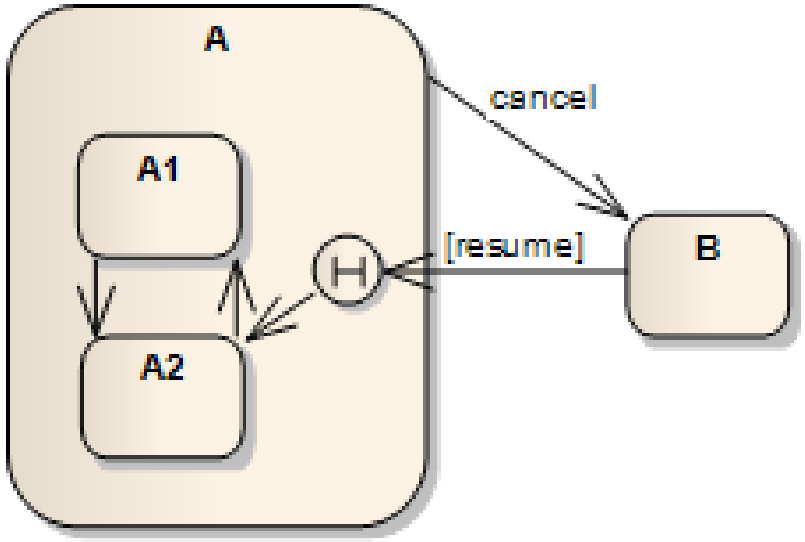
高级状态机：复合状态

- 复合状态是含有一组子状态的状态
  - 复合状态可以是单个区间，也可以包含多个区间，每个区间包含一组状态和相关的转移



高级状态机：伪状态

- 伪状态（pseudo state）是状态机图中的一类特殊顶点，每种伪状态都提供一种抽象操作
  - 初态和终态
  - 分叉和合并
  - 选择、接合
  - 历史状态
  - 入口点和出口点
  - 终结



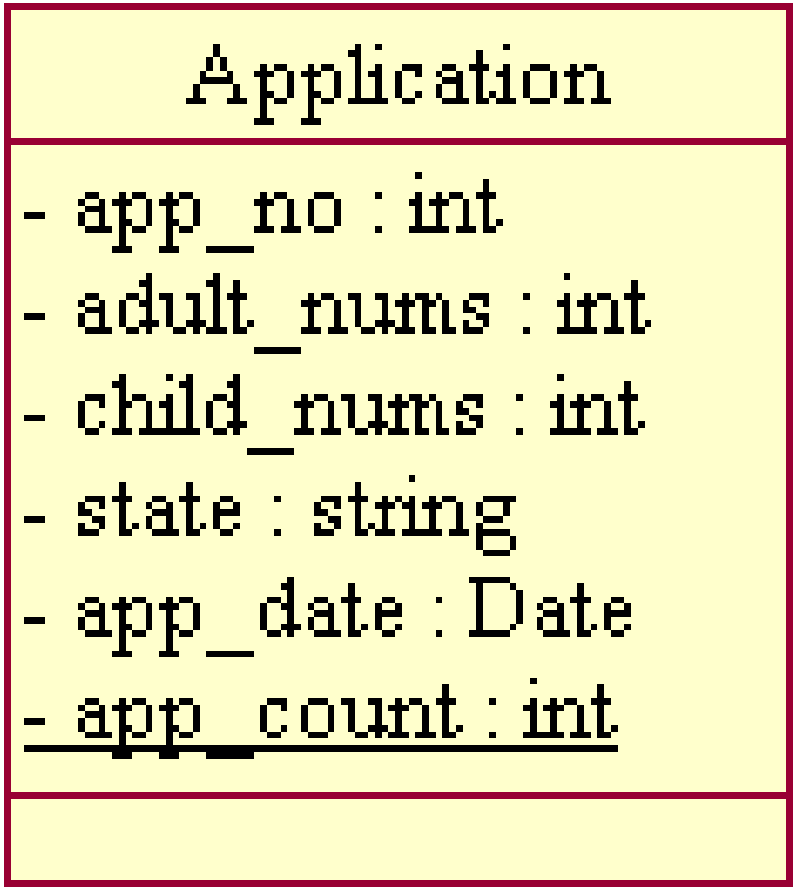
4. 定义属性

- 指定名字、类型、可见性和可选的缺省值
  - visibility attributeName : Type = Default
  - 类型应当是编程语言支持的数据类型
- 发现属性(attributes)
  - 检查类自身需要维护的所有信息
  - 检查方法和状态

属性和操作的范围

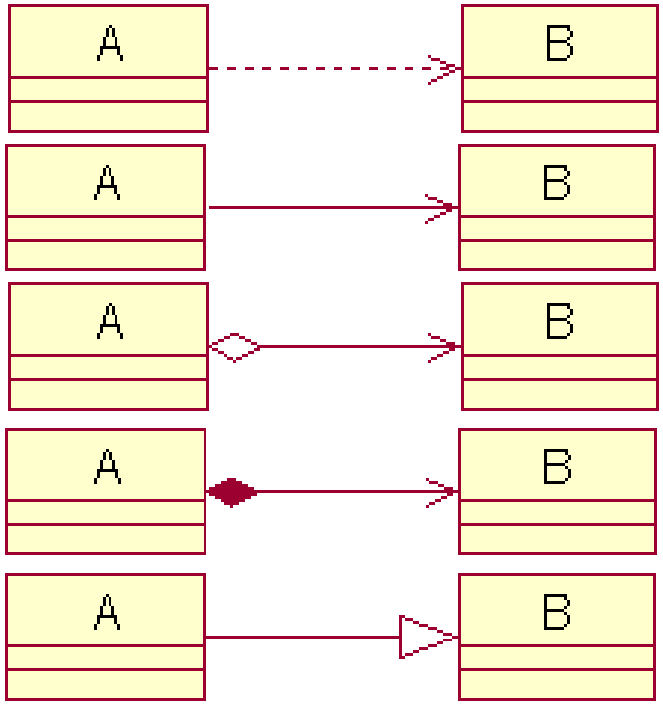
- 范围(Scope)定义了属性/操作的实例数量
  - 实例(instance)范围：每个类实例(对象)创建一个实例
  - 类(classifier)范围：所有类实例共享一个实例(静态成员)
- 类范围的表示
  - 属性/操作名加下划线
  - 使用构造型<<class>>

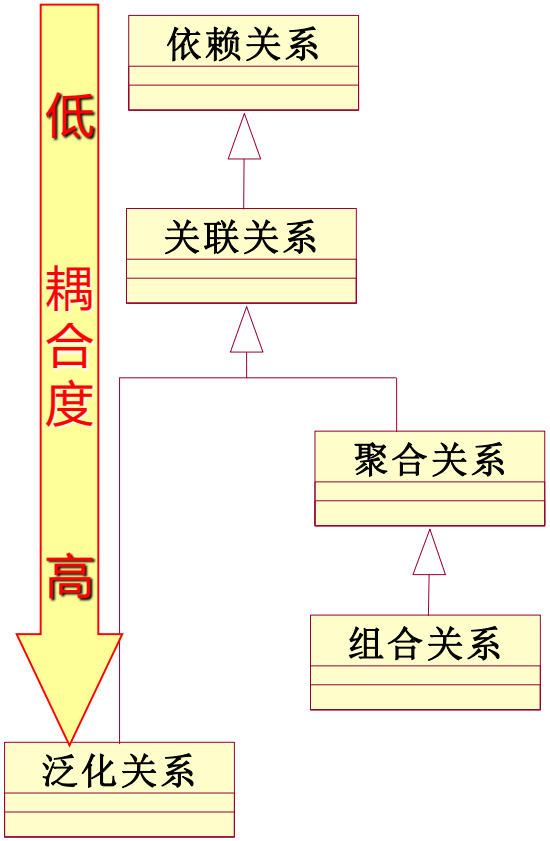
示例：定义属性

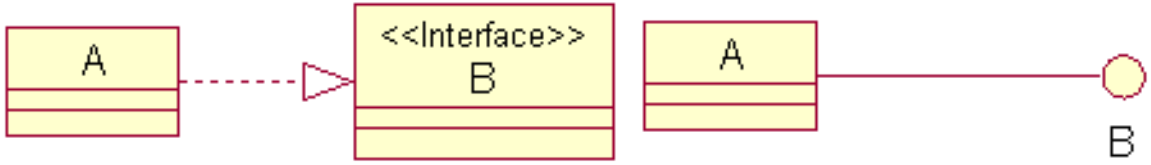


5.定义关系

- 依赖关系
- 关联关系
- 聚合关系
- 组合关系
- 泛化关系
- 实现关系(类与

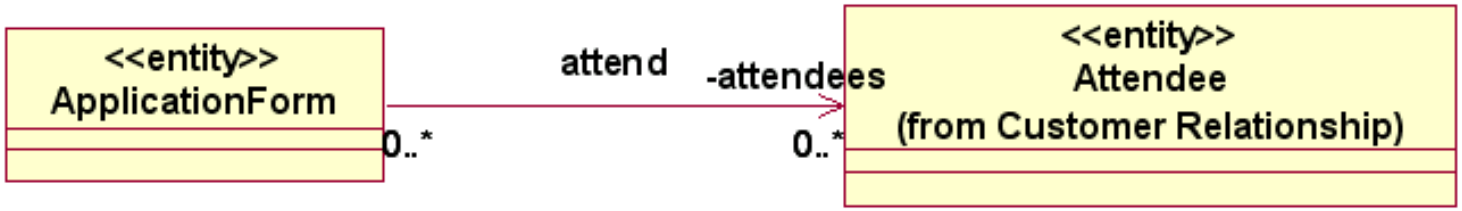






关联关系的表示方法

- 在分析阶段许多关系被建模为关联，但在设计时需要对这些关系的细节进行描述
  - 这些细节在分析时并没有被完整地描述
- 关联具有：名称、多重性表达式、导航符号、角色名称
  - 名称：动词短语
  - 多重性表达式：*，1..*，1-40，5，3,5,8，…
  - 导航性、端点名称



关联的导航性

- 关联的方向性(导航)是指对象间链接的方向(也可理解为消息发送的方向)
  - 分析阶段，没有考虑方向性；则此时默认为双方向的关联(互相知道对方对象，可以互相发送消息)
  - 设计阶段，在有可能的情况下将关联关系改为单方向的关联
    - 好的设计目标是最小化类间耦合，在没有导航性的方向上就没有类间的耦合
    - 双方向关联难以实现，需要消耗成本、内存等

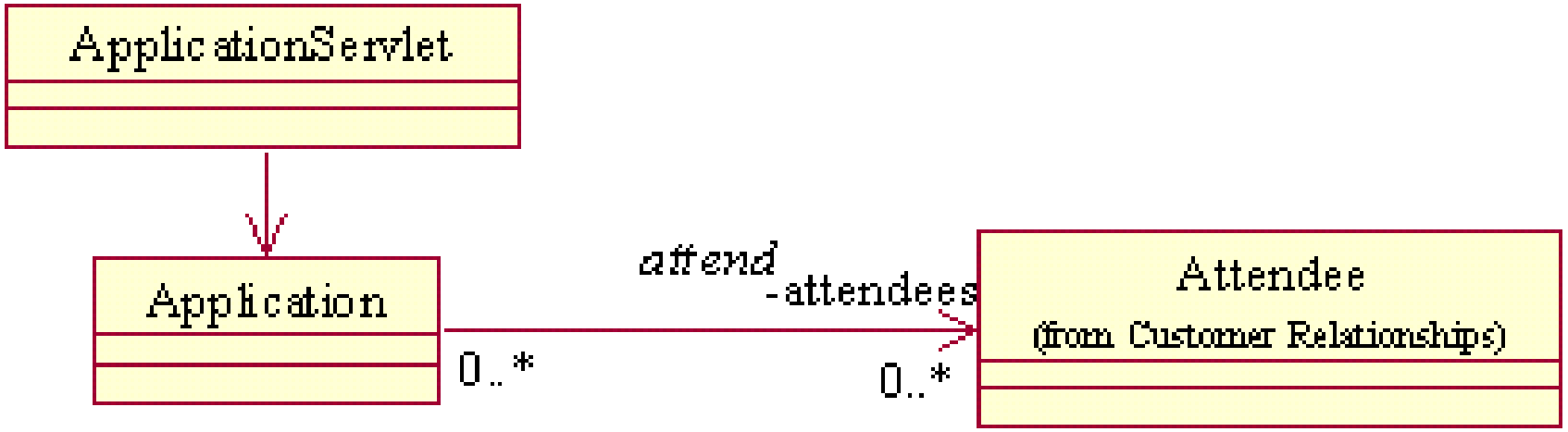
单方向关联的设计

- 设计考虑：类A与类B关联时
  - 类A的对象是否需要知道类B的对象(即类A的对象是否向类B的对象发送消息)
  - 类B的对象是否需要知道类A的对象(即类B的对象是否向类A的对象发送消息)
- 设计规则：
  - 通过分析通信图(或顺序图)，如果只向一个方向发送消息，则定义为单方向的关联(方向与消息的发送方向一致)
  - 如果双向发送消息时，则需要进一步的考虑

双方向关联的设计

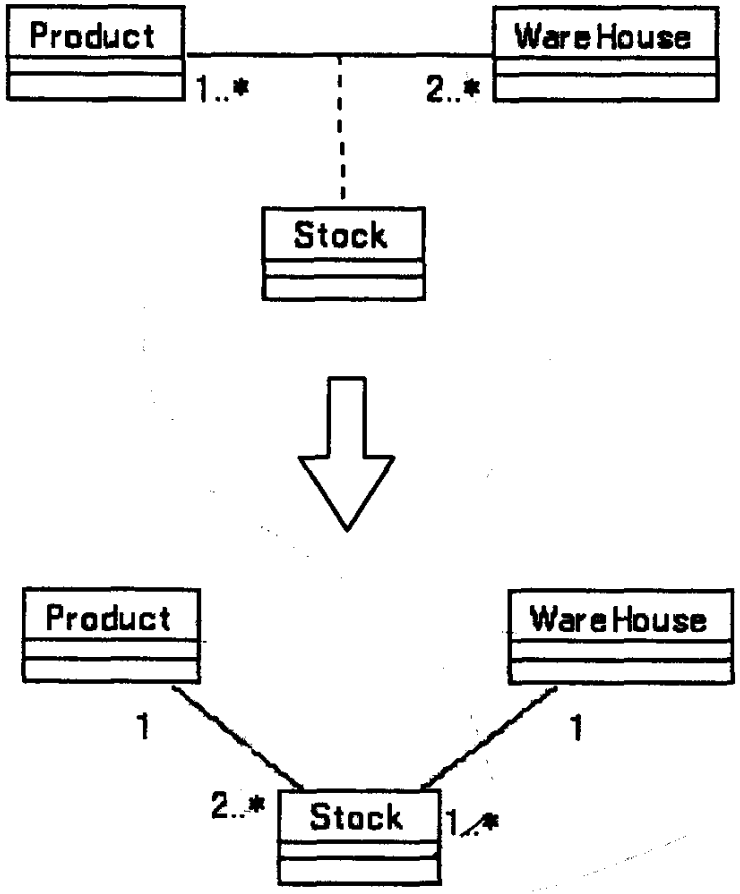
- 如果双向发送消息时，可能的方案：
  - 方案1：采用双方向的关联
  - 方案2：改变原有的消息发送顺序，从而将消息改成单方向的发送，从而采用单方向关联
- 选择依据
  - 双方向：考虑维护双方向关联的成本
  - 单方向：所发生的顺序变更及速度降低的影响

实例：导航性


ApplicationServlet
Application
attend
-attendees
0..*
0..*
Attendee
(from Customer Relationships)

关联类的设计

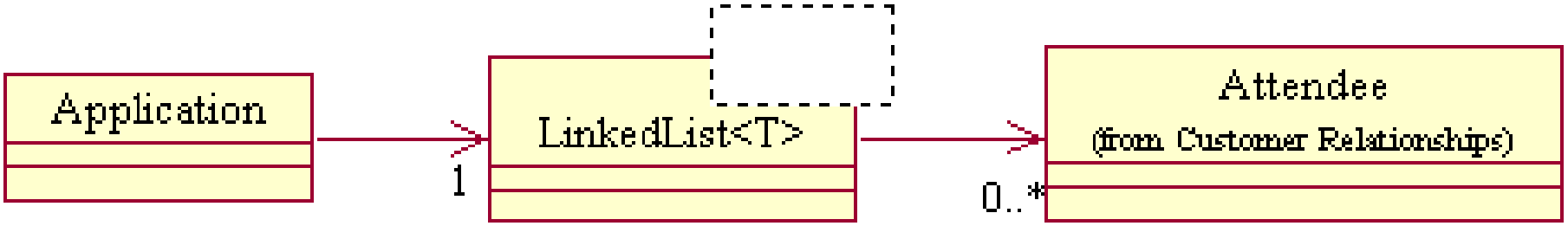
- 分析阶段使用关联类来描述关系本身的属性
  - 面向对象的编程语言不支持关联类的实现
  - 设计时需要将关联类直接定义为普通的类，从而将一个多对多的关系转变为两个一对多的关系



多重性设计

- 分析阶段，只设定了具体的重数
- 设计阶段，要考虑重数对实现的影响
  - 多重性“1”
    - 所链接的对象一定存在
  - 多重性“0..1”
    - 所链接的对象也有不存在的情况
    - 需要添加判断链接的对象是否存在的操作
  - 多重性“*”
    - 实现时准备容器类
    - 如，在java中使用Array类和List类等

多重性>1的设计方案


Application
1
LinkedList<T>
0..*
Attendee
(from Customer Relationships)

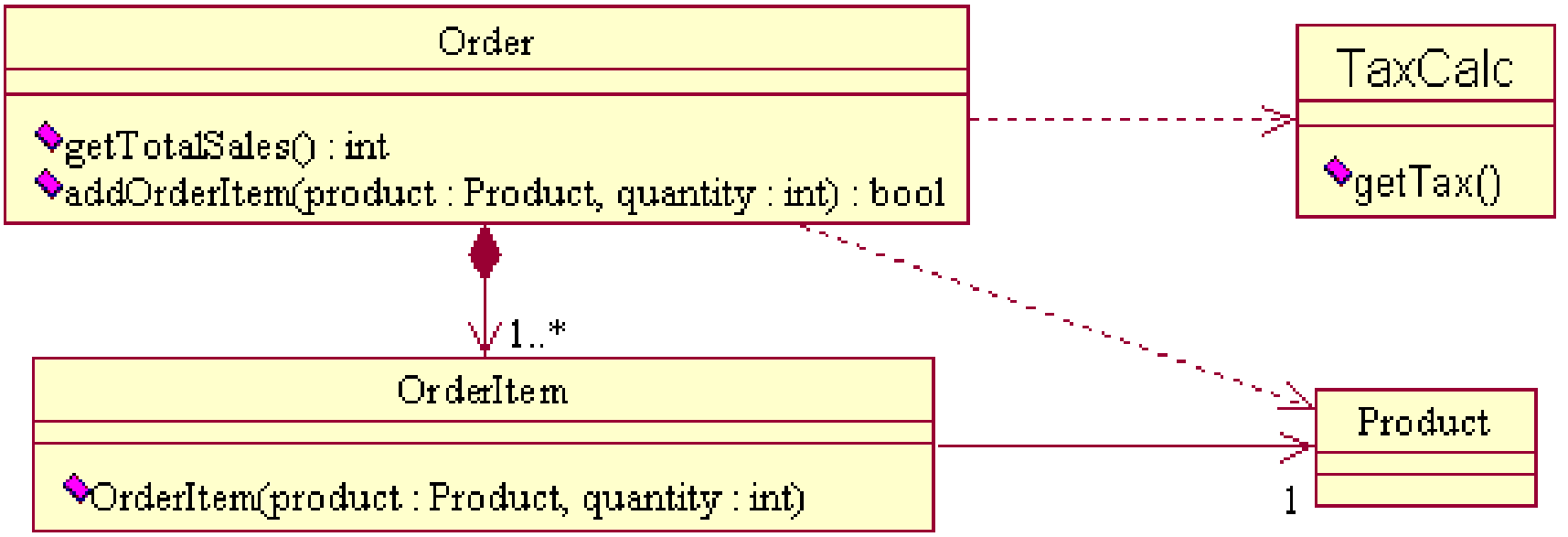
将关联关系精化为聚合/组合关系

- 聚合(Aggregation)关系和组合(Composition)关系是一种特殊的关联关系
  - 由关联关系精化而来
  - 表示整体和部分的含义，整体拥有部分
- 组合是聚合的一种形式，具有很强的归属关系和一致的生 存期
  - 部分不能脱离整体而存在

将关联关系退化为依赖关系

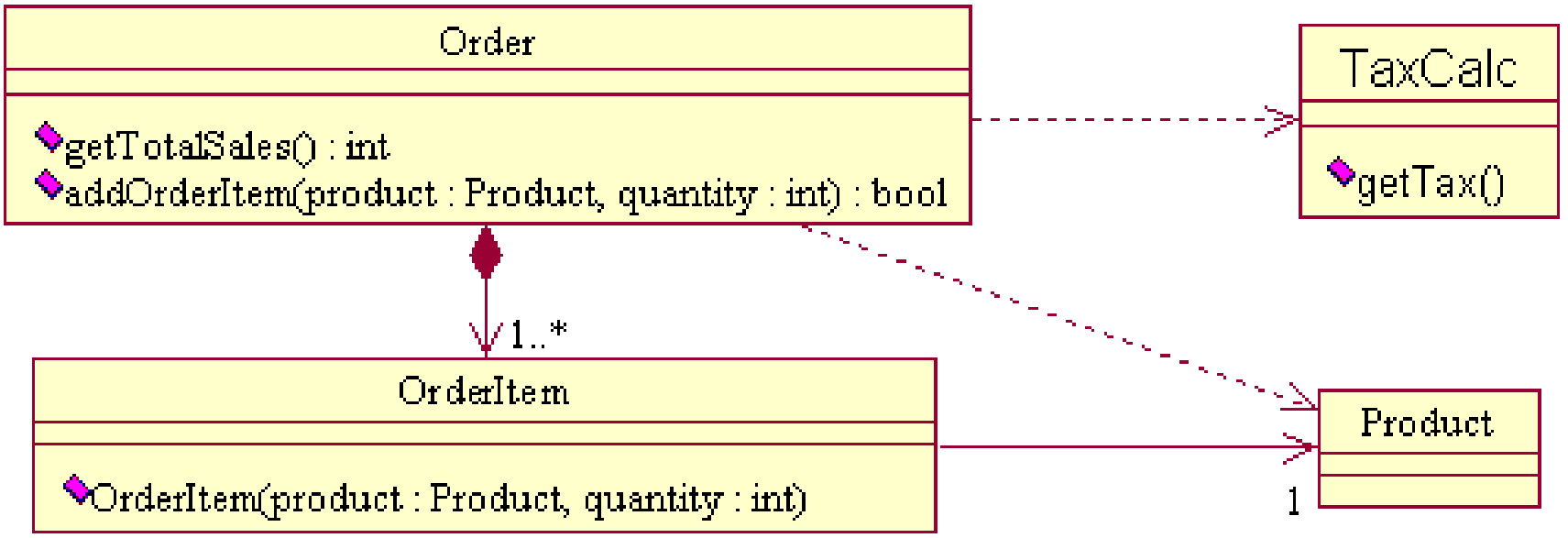
- 关联关系是“结构化”的关系
- 依赖关系是“非结构化”的、短暂的关系
- 利用对象间的引用(即对象相互了解的程度，与Demeter准则对应)来区分
  - 属性引用：B对象作为A对象的某个属性(关联关系)
  - 参数引用：B对象作为A对象某个操作的参数(依赖关系)
  - 局部声明引用：B对象作为A对象某个操作内部临时构造的对象 (依赖关系)
  - 全局引用：B对象是一个全局对象 (依赖关系)

示例：参数可见性




```
public class Order{
  public void addOrderDetail(Product product, int quantity)
  {
    OrderDetail od= new OrderDetail(product, quantity);
    …
  }
}
```

示例：局部声明可见性




```
public class Order{
  public int getTotalSales(){    …
    TaxCalc tx = new TaxCalc();
    long includeTax = tx.getTax(sales);
    …
  }
}
```

泛化关系的设计

- 只有在两个设计类之间存在清晰明确的“is a”关系或为了复用代码才使用继承（但是注意不要因此引入耦合）
- 缺点
  - 类间可能耦合的最强形式：派生类会继承基类的属性、方法、关系
  - 类层次中的封装是脆弱的，它会导致“脆弱基类”问题—基类的改动会直接波及底下的层次
  - 在大多数语言中，继承非常不易改变—关系是在编译时确定的，关系在运行时是固定的
- 注意LSP原则的应用