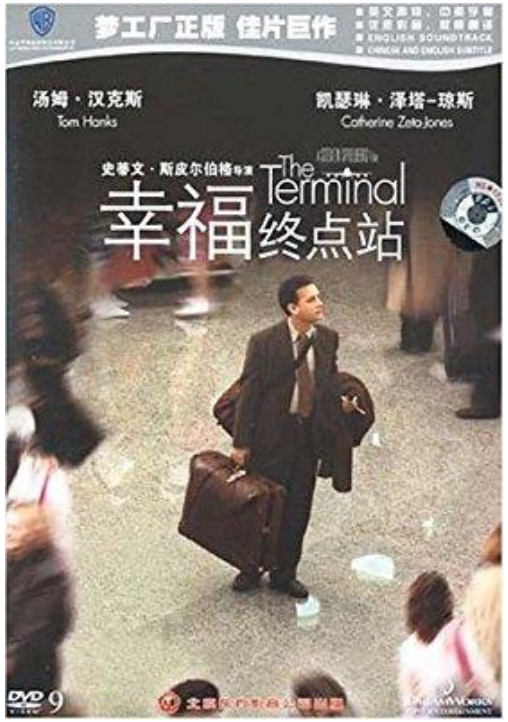
第十五讲
如何更好的进行设计



OO2018课程组

计算机学院

Sometimes you land on a small fish. You unhook him very carefully. You place him back in the water. You set him free so that somebody else can have the pleasure of catching him.

摘要

- 面向对象程序设计
- 面向对象程序的结构特点
- 面向对象程序的行为特点
- 设计过程与设计表示
- 结构的UML表示
- 行为的UML表示
- 作业

面向对象程序设计

- 我们前面讨论过
  - 继承/抽象设计
  - SOLID设计原则
  - 线程安全设计
  - 规格化设计
- 设计是一个过程
  - 分析和细化问题，重用领域知识和经验，获得解决方案的过程
  - 如何提高过程的效率？
- 设计是一个结果
  - 对系统概念、结构和行为的抽象结果，既能满足系统需求，又具有一系列良好的特性
  - 如何准确又简洁的表示？

面向对象程序的结构特点

- 层次性
  - 继承层次
  - 接口实现层次
- 交互性
  - 内部交互
  - 外部交互
- 内聚性
  - 类方法的聚合
  - 类属性的聚合

层次结构

- 层次化是一种典型的程序结构
  - 结构化程序中：模块依赖，函数调用
  - 面向对象程序：继承、接口实现
- 层次化是对程序进行设计抽象的结果
  - 结构化程序
    - 高层：对问题解决方案的高层定义
    - 低层：对问题解决方案的某个细节定义
  - 面向对象程序
    - 高层：对问题解决方案的抽象定义
    - 低层：对问题解决方案的细化定义

层次结构

- 高层是对低层的抽象
- 低层是对高层的细化
- 抽象与细化的连接是什么？
  - FileReader类与FileWriter是否具有层次关系？
  - FileReader与WebReader是否具有层次关系？

✓ 高层类一定存在一些属性或操作能够概况低层类的相关属性和操作

✓ 低层类一定存在一些属性或操作来对高层类的相关属性或行为进行补充/细化

交互结构

- 交互是OO程序的一个基本结构模式：达到均衡目标的主要手段
  - 功能分解
  - 数据分散
- 类之间形成交互关系
  - 数据交互
  - 控制交互
  - 共享
- 交互结构和层次结构常常相互交织
  - 一个类交互的另一个类，在运行时的交互对端不见得是那个类产生的对象

聚合结构

- 内聚是面向对象的另一个重要特征
  - 控制数据的可见范围，确保其状态变化受控
  - 线程安全的基础
- 内聚与层次和交互结构互相支撑
  - 低层次的类内聚具体的属性数据
  - 高层次类聚合具体数据和类之间的交互，形成代理结构(delegation)
- 内聚是控制故障传播范围的有效措施
  - 故障是bug在运行时的显现，导致程序状态出现错误
  - 故障随着数据共享进行传播

面向对象程序的行为特点

- 协同性
  - 对象之间协同完成功能
  - Delegation行为
  - Control center行为
- 并发性
  - 每个对象都是一个独立的行为体，本质上并发执行各自的行为
  - 但是每个对象只能看到局部程序空间
- 静态不确定性
- 动态不确定性

面向对象程序的行为特点

- 静态不确定性
  - 在程序静态分析层面，无法确定一个变量引用的对象在运行时的实际类型，无法确定调用的是哪个具体方法
  - 静态不确定性提升了设计的抽象表示能力，但易引入运行时错误
  - 遵循设计原则来加以控制
- 动态不确定性
  - 多线程/并行是现代软件的一个基本特征
  - 线程是并发执行控制单位，执行控制由下层计算模型决定
  - 程序层次可通过计算模型提供的控制原语来控制线程间的同步

设计过程与设计表示

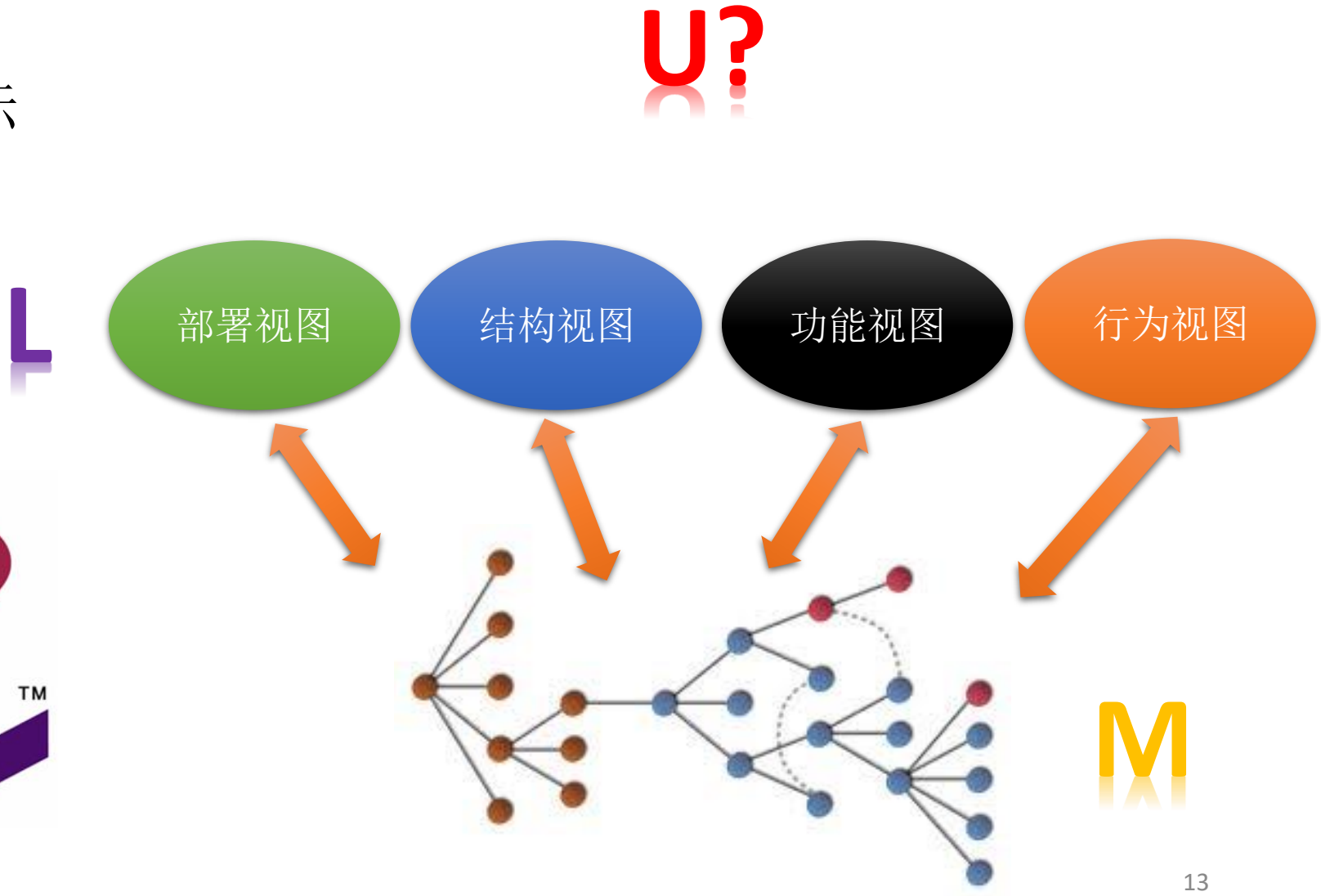
- 设计是一个综合性很强的智力活动，需要设计者对系统功能、环境、边界条件等有清晰的把握，还要求设计者在软件结构对未来行为的影响方面有很强的预见性
- 抽象思维能力是关键
- 设计过程一般需要几轮迭代和修改
  - 确定核心的类、接口和线程
  - 设计类层次结构
  - 设计类的交互关系
  - 设计类的规格

设计过程与表示

- 规模小的程序，画画草图也许就能完成设计
- 我们希望使用一种清晰、易理解且表达直接的语言来开展设计，并记录设计结果
  - UML
- 整个设计可以从三个维度开展
  - 数据维：定义抽象数据及其层次抽象关系
  - 控制维：结合功能和用户特点，定义线程及其协同方式
  - 处理维：针对类和方法，设计其功能和交互

UML语言简介

- UML的语言设计目标
- UML的建模理念
- UML模型组成



UML语言设计目标

- 提供一种面向对象式的抽象又直观的描述逻辑
  - 抽象：把系统抽象表示为类和类之间的协同
  - 直观：通过可视化的模型图来描述和展示系统功能、结构和行为
- UML经过了近二十年的发展(UML 2.x)
  - 绘画式语言→仅用于人之间的交流
  - 描述性建模语言→机器能够理解模型的部分含义
  - 可执行建模语言→机器能够理解和执行模型的准确语义

UML建模理念

- 语法明确、语义清晰的可视化语言
- 多种描述视角
  - 功能视角：系统或子系统要提供哪些功能(use case)？
  - 结构视角：系统有哪些组件(component)/类(class)/接口(interface)，相互间有什么关系(relation)？
  - 行为视角：组件/类能够做什么？组件/类之间如何协同？
  - 部署视角：组件/类如何分配到不同的可安装软件模块？
- 每个视角可以通过若干UML图来描述
  - 每个图有明确的主题
  - 控制每个图的规模

UML建模理念

- 用例模型定义系统的需求
  - 使用可视化图来形象展示系统功能整体
  - 基于模板来描述每个用例(需求)的规格
    - 输入/输出，处理流程，异常情况，前置条件和后置条件
- 类模型定义系统的解决方案：使用“这些类”来实现相应的需求
  - 使用可视化图来形象展示系统解决方案的整体（类、类之间的关系）
  - 基于模板(属性、操作、约束条件)定义类的结构规格
- 状态模型定义类的行为机制：“这个类”将按照这样的行为逻辑运行
  - 使用可视化图来形象展示一个类受到关注的状态空间
  - 基于模板(状态行为、迁移行为)来定义类的行为规格→方法规格
- 交互模型定义类之间的协作机制：“这些类”在一起完成“这个业务”
  - 使用可视化图来形象展示类之间的交互序列
  - 基于模板(消息、消息时序控制)来定义类之间的交互规格→方法规格

UML模型组成

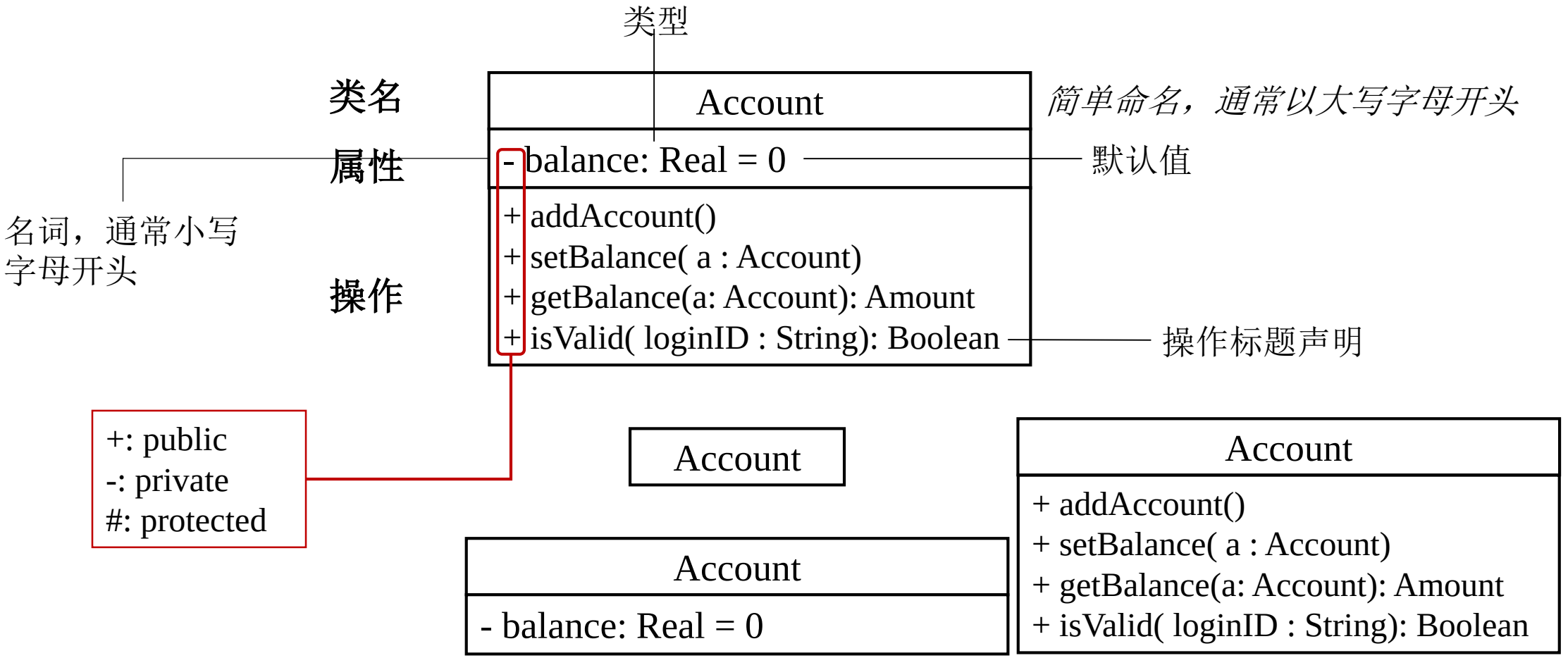
- 在UML建模工具中建立的各种图都是对模型的一种观察
- UML模型在哪儿？
  - 内存中、文件中
  - 是一组基于图的数据结构
- 在UML建模工具看来，各种图中的每个要素都是一个对象
  - 用例、类、属性、操作、关联、继承、消息、迁移…
  - 通过一系列数据结构来管理这些对象
- 一组数据结构：UML元模型
- 所建立的UML模型实际上就是UML元模型的实例化结果
  - 通过复杂的图数据结构来管理

UML类图---对象建模的根本

- 最常使用的UML建模图
- 围绕一个具体主题，展示相关的类、接口，它们之间的关系（依赖dependency、继承generalization、关联association、实现realization），以及必要的注释说明
- 三个层次的描述抽象
  - 概念层描述：用来分析**问题域描述**中可看到的类（分析模型）
  - 规格层描述：关注类的规格和接口
  - 实现层描述：可直接映射到代码细节的类

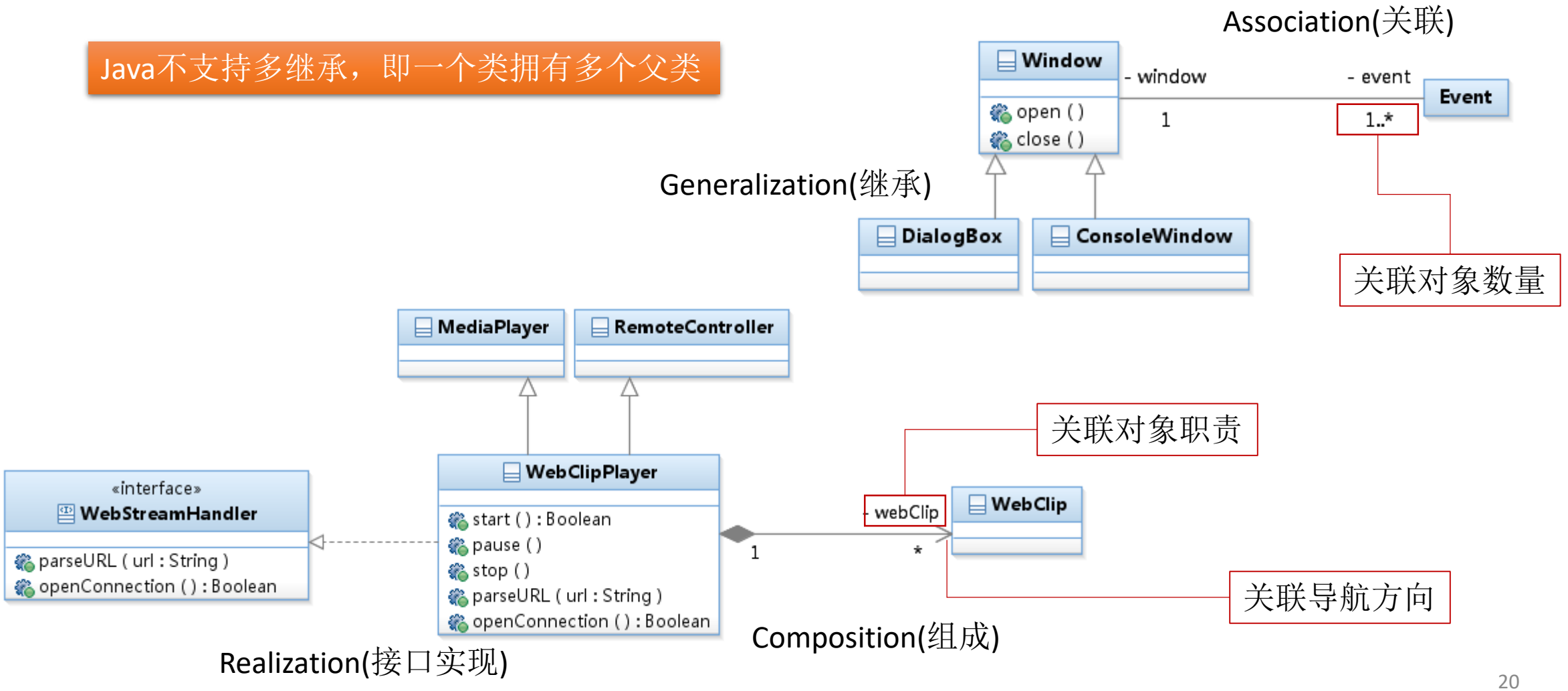
Most users of OO methods take an implementation perspective, which is a shame because the other perspectives are often more useful.

类的表示语法


类型
类名
属性
操作
Account
- balance: Real = 0
+ addAccount()
+ setBalance(a : Account)
+ getBalance(a: Account): Amount
+ isValid(loginID : String): Boolean
简单命名，通常以大写字母开头
默认值
操作标题声明
名词，通常小写字母开头
+: public
-: private
#: protected
Account
Account
- balance: Real = 0
Account
+ addAccount()
+ setBalance(a : Account)
+ getBalance(a: Account): Amount
+ isValid(loginID : String): Boolean

类之间的关系

Java不支持多继承，即一个类拥有多个父类



类之间的关联关系

- 一个类需要另一个类的协助才能完成自己的工作
  - 需要获得一些信息
  - 需要协助做一些处理
  - 需要通知对方自己的状态变化
- 从对象的角度来理解关联
  - 从当前对象顺着关联方向可以找到相关联的对象
  - 注意关联对象的数目
    - *：表示为0到多个对象
    - 1..*：表示为1到多个对象
    - m..n：表示为m到n个对象
    - n：表示n个对象

```
public class Course{
  …
  private Vector<Student> student;
  …
}
```

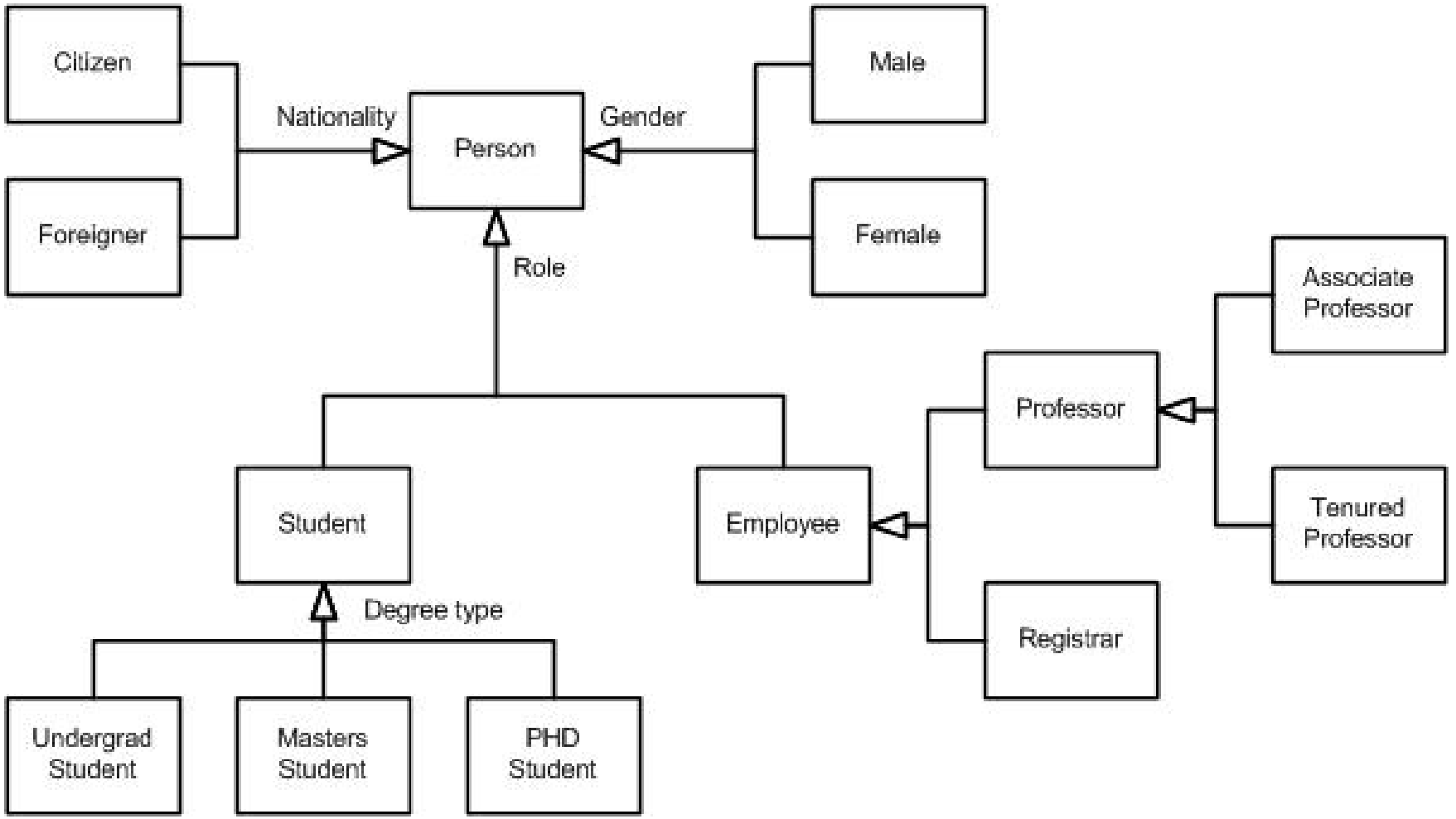

```
public class Student{
  …
  private Vector<Course> course;
  …
}
```

类之间的继承关系

- 父类与子类
  - 父类概括子类
  - 子类扩展父类
- UML支持灵活的多继承
  - Java不支持
  - 建议在使用UML时不用多继承
- 一旦建立继承关系，子类将自动拥有父类的所有属性和操作
  - 不要重复定义

```
public class OOCourse extends Course{
  …
  …
}
```

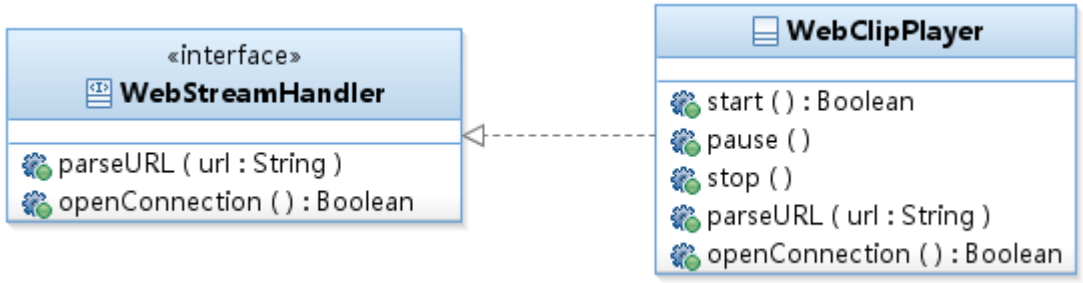
通过继承来进行分类


Citizen
Foreigner
Nationality
Person
Gender
Male
Female
Role
Student
Degree type
Undergrad Student
Masters Student
PHD Student
Employee
Professor
Associate Professor
Tenured Professor
Registrar

类对接口的实现关系

- UML与Java具有一致性
  - 一个非抽象类必须实现接口中定义但未实现的所有操作
- 接口是UML语言预定义的一种特殊的类
- 一个类可以实现多个接口
  - 与此同时还可以继承一个父类
- 实现类需要显示列出要实现的操作
  - 和继承机制不同！

```
public class A extends B implements C,D,E{
  …
  …
}
```



面向对象程序行为的UML表示

- 基础行为
  - 不依赖于其他类的行为
  - 通过方法来规格化和实现
- 交互行为
  - 内部交互
  - 对象交互
  - 线程交互
- 控制行为
  - 协调行为
  - 共享访问控制行为

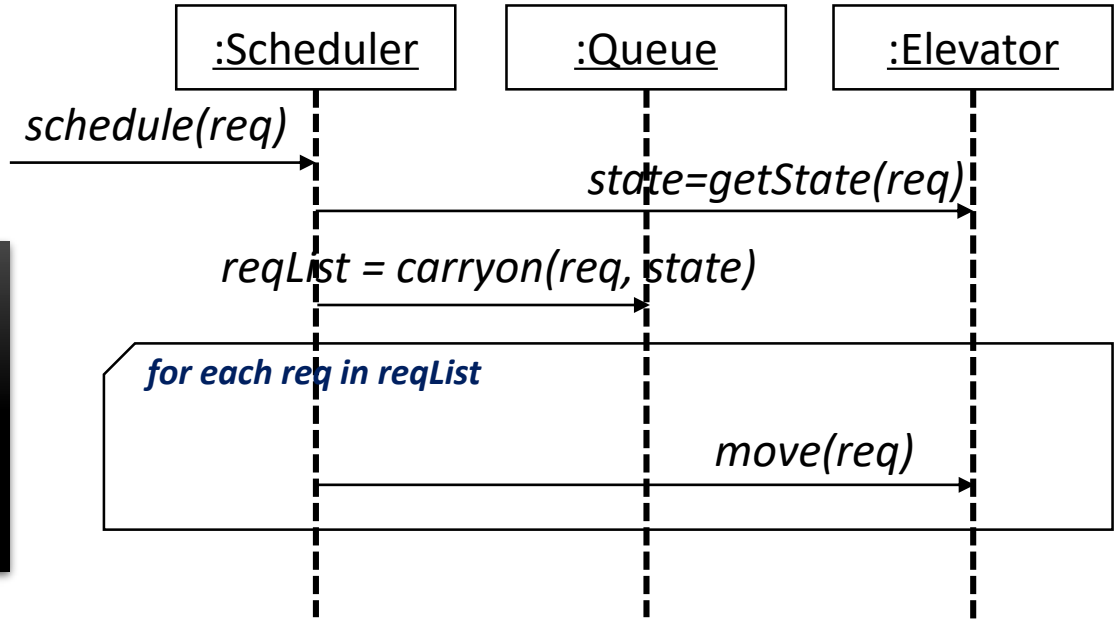
行为的UML表示

- 顺序图
  - 交互行为
  - 控制行为
- 状态图
  - 对象行为

UML顺序图模型

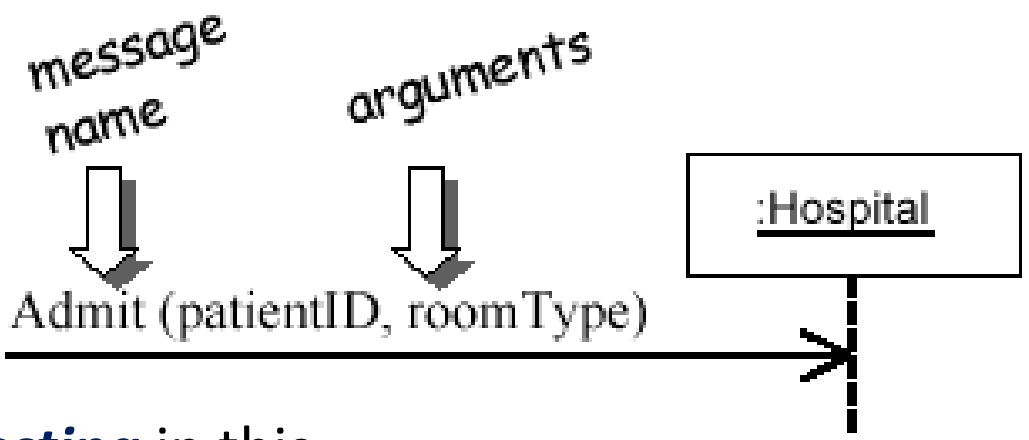
- 顺序图的组成
  - 参与对象(participant)：参与交互的对象
  - 消息：对象间的交互
  - 对象生命线：描述对象的存活生命期

顺序图具有典型的笛卡尔坐标图性质:
水平坐标：排列参与交互的对象
垂直坐标：消息时序和时间信息(时间从上往下增长)



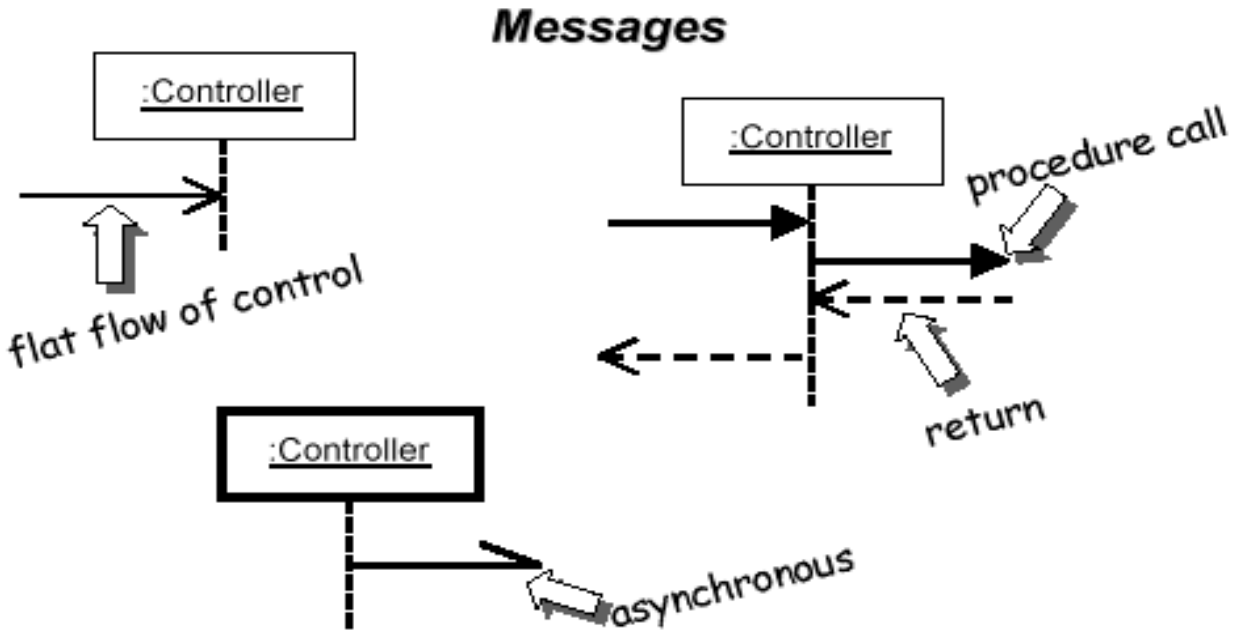
UML顺序图模型

- 对象格式
  - 矩形框，对象名：类型名
  - 对象名有时可省略
  - 每个对象都有一条垂直的生命线
- 消息
  - 对象之间的交互手段
  - [var=]消息名([消息参数])
- 消息形成合作
  - Sender: I need ***help /*** Someone is ***interesting*** in this.
  - Receiver: Someone is asking for ***help*** / Someone is ***telling*** me what I need/interest



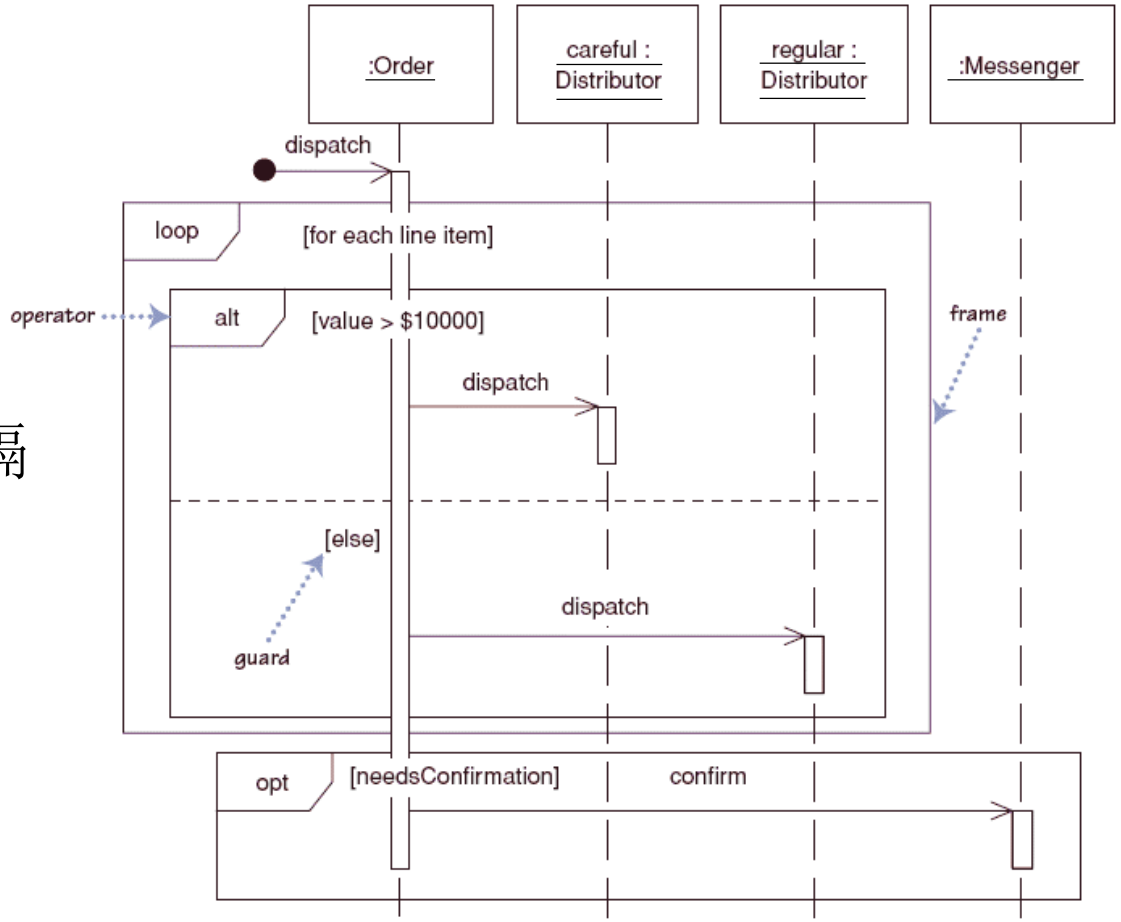
UML顺序图模型

- 多种消息格式
  - 同步消息，等同于方法调用
  - 异步消息，如线程调度方法
  - 显式返回消息



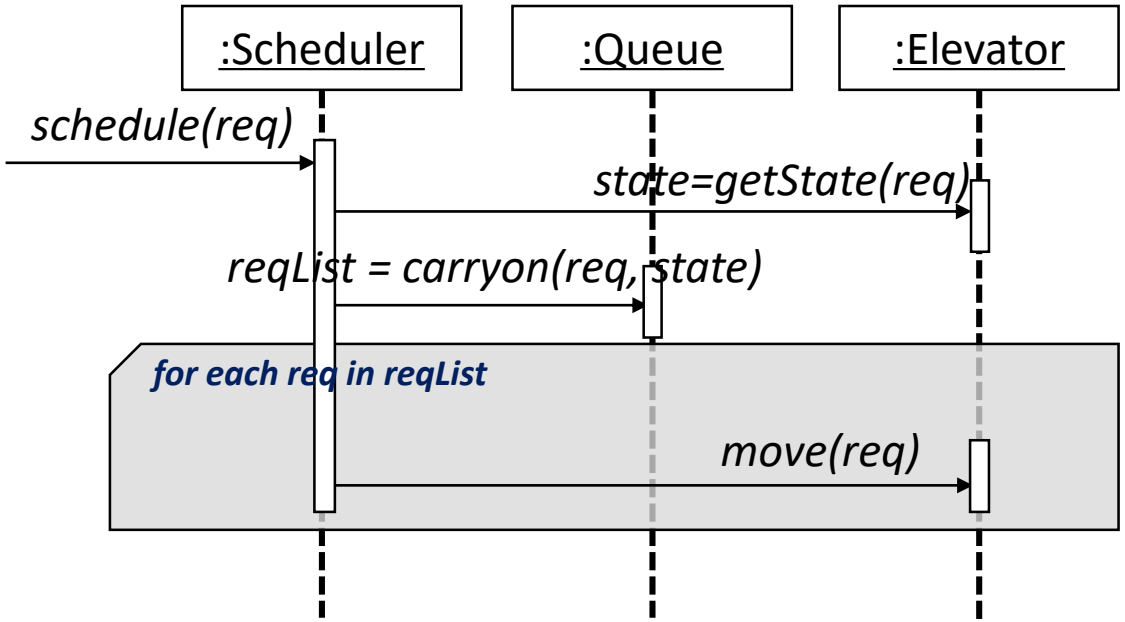
UML顺序图模型

- 基于消息块的交互流程控制
- if控制-> 可选消息快
  - (opt) [控制条件]
- if/else-> 多分支消息块
  - (alt) [控制条件], 通过水平虚线来分隔多个分支控制
- loop -> 循环消息快
  - (loop) [循环控制条件或循环事项]

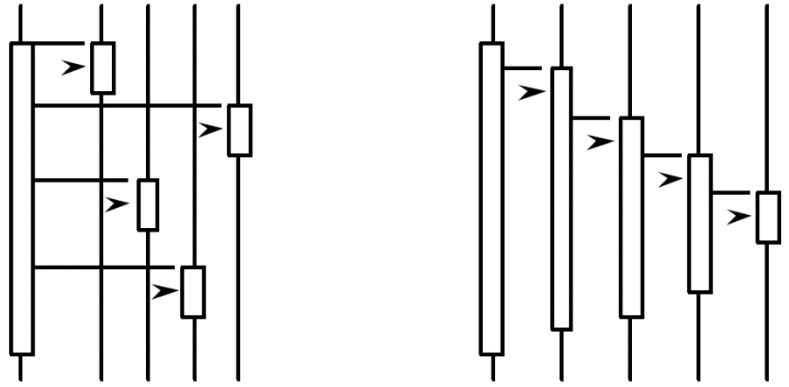


UML顺序图模型

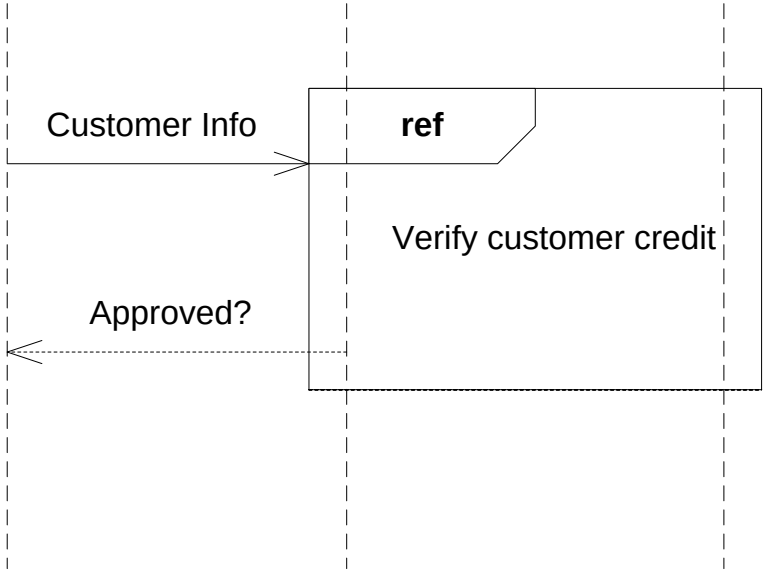
- 每个顺序图描述一个交互主题
  - 如图所示，调度器根据调度要求，从请求队列中取出捎带的请求进行调度
  - 有一个触发消息
  - 一系列响应消息
- 一致性规则
  - 每个对象都提供相应的消息处理能力
  - 对象必须提供相应的操作，且与消息名称、参数和返回值类型一致
  - 如果返回值有异常或者特殊值，Sender对象必须要进行处理



UML顺序图模型

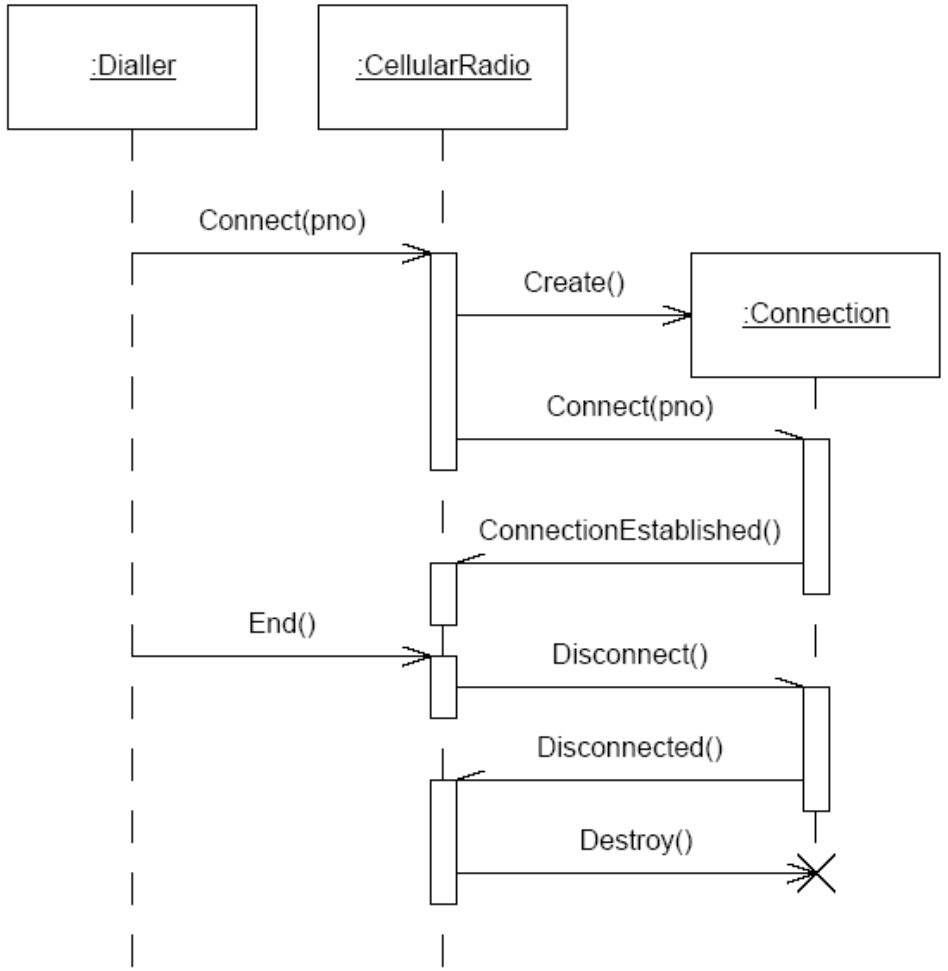


- 两类典型的控制模式
  - 集中式控制
  - 分布式控制
- 适用于表示层次化设计原则，一个主题对应的交互尽量简单，多个顺序图之间可以引用(类似于调用)



UML顺序图模型

- 这个顺序图有什么问题？
- 为什么不直接写代码？
  - 规格抽象关注于类的方法和类整体
  - 顺序图关注与对象之间的协作
  - 是一种重要的设计手段
    - 清楚展示哪些对象参与一个处理
    - 清楚展示每个对象的职责
    - 容易识别“松垮”的交互场景

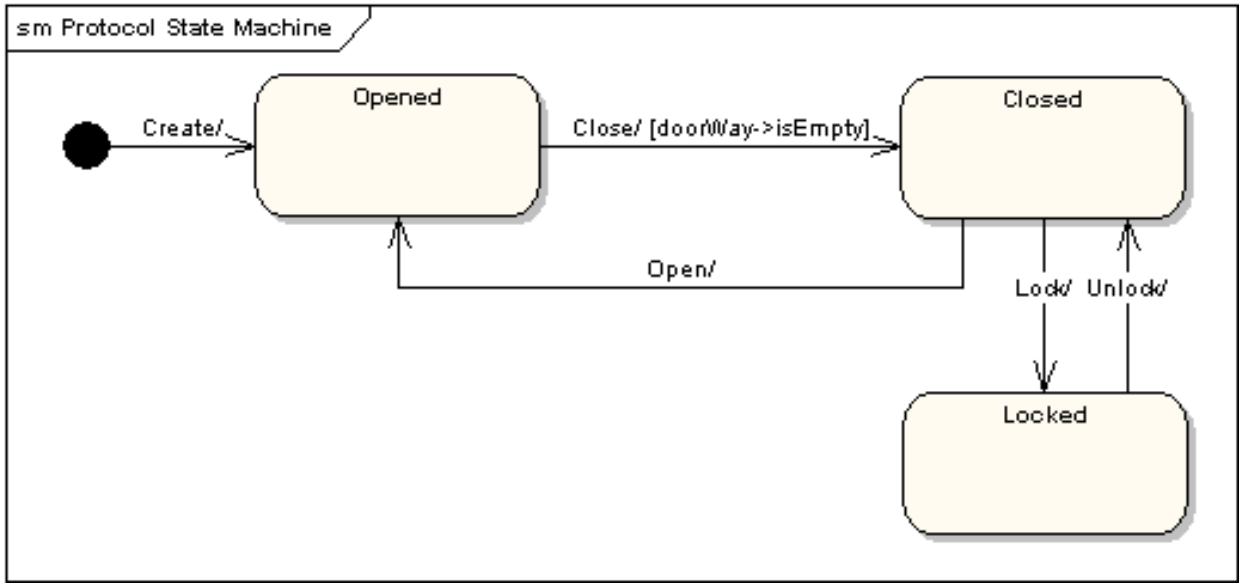


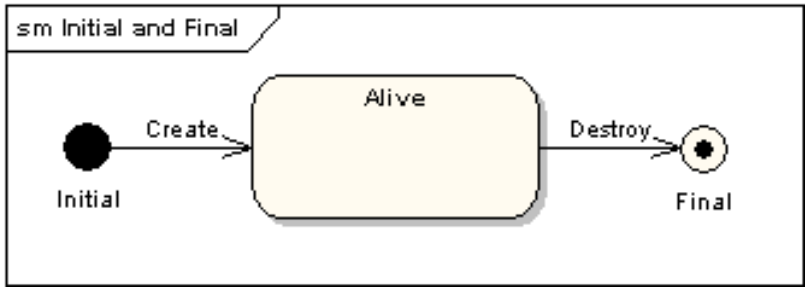
UML状态图模型

- 对象是一种状态化的存在
  - 状态由数据定义
  - 外部状态、内部状态
- 对象行为引发状态变化
  - 状态迁移
- 使用UML状态图来描述外部可见的状态
  - 类的行为规格设计

UML状态图模型

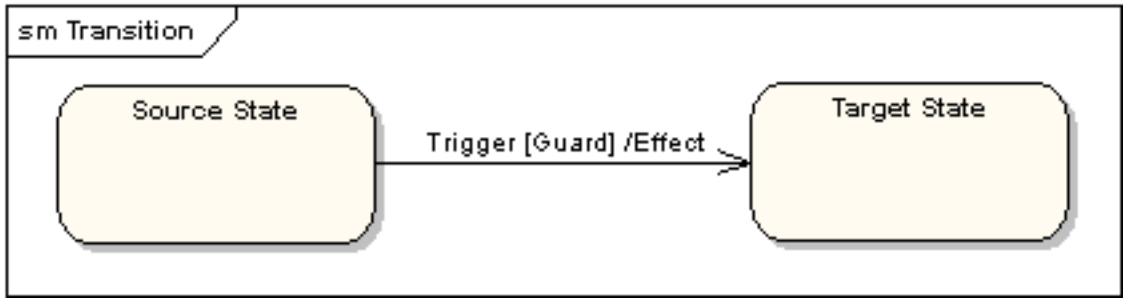
- 只用来描述一个对象的行为
  - 不能跨越“边界”
- 状态使用圆角矩形框表示
  - 初始状态
  - 终止状态(可能没有)
- 迁移使用带箭头的线表示
  - 一个迁移只能连接一个源状态、一个目标状态
  - 任何一个状态都必须从初始状态可达
  - 任何一个状态都能够迁移到终止状态(如果有)

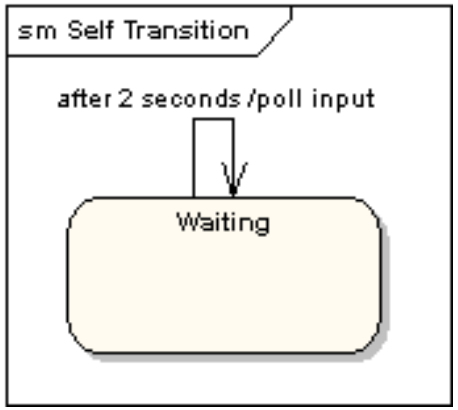




UML状态图模型

- 迁移的定义
  - trigger[guard]/effect
  - 触发事件trigger
  - 守护条件guard
  - 迁移动作effect
- Trigger是引起迁移的原因
- Guard是迁移能够发生的前置条件
- Effect是迁移发生的后置条件之一
  - Effect
  - 对象状态改变为迁移的目标状态





UML状态图模型

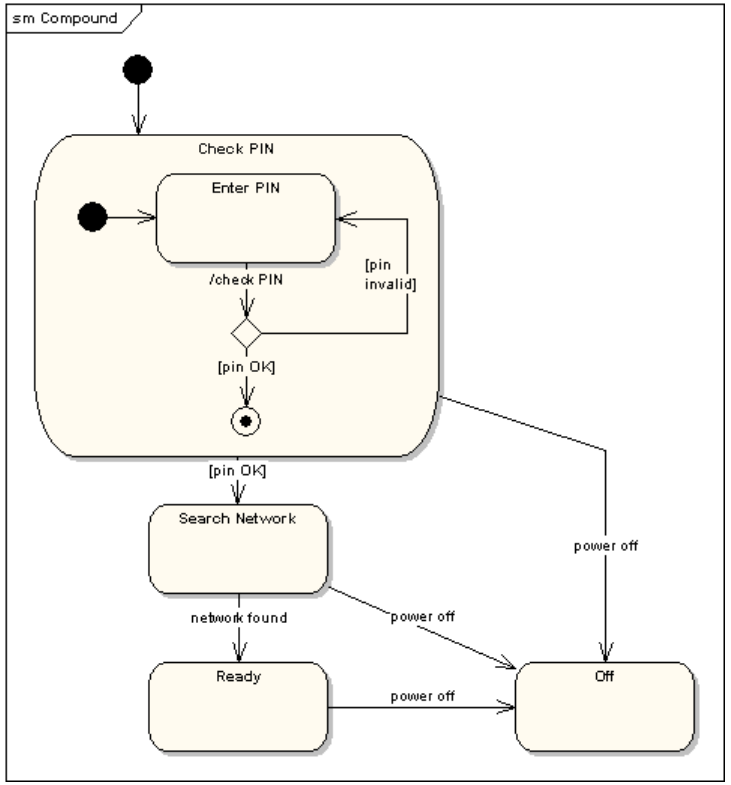
- 对象在某些状态下可完成一定的动作(较少用到这种表达)
  - 进入状态动作entry action
  - 退出状态动作exit action
  - 处于状态中的动作do action
- 可以为一个状态构造任意数目的这三种类型动作
  - 一定要从问题域出发

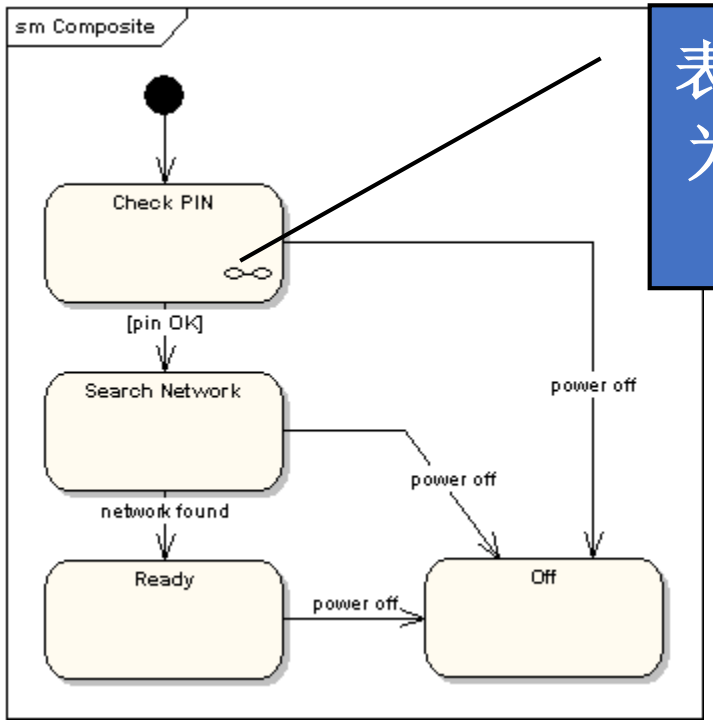
UML状态图模型

- 状态与程序代码的对应
  - 对象运行时属性取值的划分
  - 不能简单对应到静态的代码
- 状态迁移与程序代码的对应
  - 从源状态来看
    - Trigger：方法调用或者事件通知
    - Guard：调用时的相关检查
  - 从目标状态来看
    - Trigger：方法体
    - Guard：方法入口处的检查
    - Effect：方法执行后的效果，即迁移到目标状态
- 状态动作与程序代码的对应
  - 对应到方法，通常外部用户不会调用，对象为了满足相关规格而实施的行为
  - 要求不改变对象状态

UML状态图模型

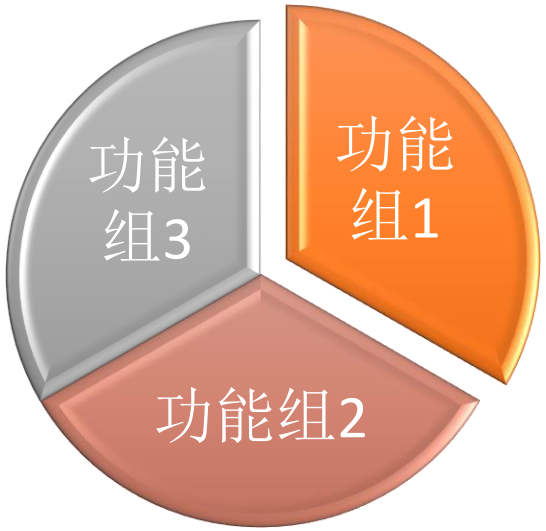
- 组合状态：(***Compound State or Composite state***)
  - 用来按照层次化方式定义一个复杂状态的行为





表示Check PIN 的行为细节在另一个子状态图中定义

使用UML开展设计的典型过程



设计类图，确定类属型、操作和类之间的关系

设计顺序图，描述类（对象）之间如何协作完成每一个功能

针对具有复杂逻辑的类，设计状态图来描述其行为

作业

- 博客作业
- 1. 论述测试与正确性论证的效果差异，比较其优缺点
- 2. 调研OCL语言，并比较其与课程所介绍的JSF规格之间的相似和不同之处
- 3. 根据第十四次作业的单电梯系统，针对调度器、电梯、请求队列和请求，至少整理出 一幅UML类图、一幅顺序图和一幅状态图，并使用图(graph)来表示出模型
- 4. 整理总结一个学期所学所练
  - 4.1 阐述四个单元模块知识点之间的关系
  - 4.2 梳理自己所设计实现的程序，分析自己在设计、测试和质量上的进步
  - 4.3 阐述自己对工程化开发的理解
  - 4.4 对课程的任何期望或建议