

类的协作

刘 钦

南京大学软件学院

Outline

- 回顾类的职责
- 类的协作
- 类之间的关系
- 类的开发指南

回顾类的职责

Outline

- 回顾类的职责
 - 概念与原则
 - 对象的发现
 - 职责的分配
- 类的协作
- 类之间的关系

面向对象方法的原则与要素

- 基本问题求解的原则
 - 分解与抽象
- 面向对象方法的原则
 - 职责与协作
- 面向对象方法三要素
 - 封装
 - 继承
 - 多态

面向对象方法中的一些概念

- 软件 = 一组相互作用的对象
- 对象 = 一个或多个角色的实现
 - 状态
 - 行为
- 责任 = 执行一项任务或掌握某种信息的义务
- 角色 = 一组相关的责任
- 协作 = 对象或角色（或两者）之间的互动

常见对象

- 和系统存在交互的外部实体，
 - 例如人、设备、其他的软件系统等；
- 问题域中存在的事物，
 - 例如报表、信息展示、信号等；
- 在系统的上下文环境中发生的事件，
 - 例如一次外部控制行为、一次资源变化等；
- 人们在与系统的交互之中所扮演的角色，
 - 例如系统管理人员、用户管理人员、普通用户等；
- 和应用相关的组织单位，
 - 例如分公司、部门、团队、小组等；
- 问题域中问题发生的地点，
 - 例如车间、办公室等；
- 事物组合的结构关系，
 - 例如部分与整体的关系等。

发现策略 - I

- 写一个简要的设计提纲，其中列出应用系统的重要部分
- 根据这个提纲，确定几个与应用系统核心问题相关的主题
- 查找哪些围绕和支持每个主题的对象
- 检查哪些描述关键概念及软件外部表征的对象
- 查找哪些描述了附加机制和结构的对象

发现策略 - 2

- 命名、描绘、刻画每个对象
- 组织对象，寻找自然的方法把应用系统划分成一些解决共同问题而相互关联的对象族
- 查看对象是否合适于系统。确定它们时候描述某种合理的抽象实体
- 讨论每个对象的存在原因
- 当工作进行缓慢的时候，要不断地利用责任和协作对系统进行建模

设计的提纲

- 用自己的语言
- 开始判断系统的要点
- 明确系统中你有把握的和不知道的东西
- 粗略点
- 快速
- 设计草图

寻找策略

- 系统完成的工作
- 直接受应用程序影响或与其有关联的东西
- 软件中的信息流
- 决策、控制与协调等行为
- 结构和对象群
- 对应用程序有意义，代表现实事物的对象

名字

- 修饰通用名称
- 名字里只能包含将最有启迪性和最突出的因素
- 给服务提供者以worker式命名
- 为那些名字暗示了广泛责任的对象寻找辅助对象
- 选择不限制行为的名称
- 选择一个适合当前背景的名称
- 不要重载名称
- 通过添加形容词来消除命名冲突
- 通过选择相似意义的名称来消除冲突
- 选择容易理解的名字

候选对象

- 特殊化候选对象
- 连接候选对象
- 审核对象

责任

- 对象执行的动作。
- 对象持有的信息。
- 能够影响到其他对象的决定。

获得责任 - I

- 确定系统责任声明或将其隐含在用例中
- 通过添加低级别的责任来弥补用例和系统描述之间的沟壑
- 从设计主题和设计提供中提炼出额外的系统行为
- 遵循what if...then... and how的推理链

获得责任 - 2

- 认识匹配对象角色的构造型责任
- 搜寻每个候选对象的更深处本质特征
- 识别出支持对象之间的关联和从属依赖的责任
- 识别出与对象的主要动作相关联的责任
- 识别出对象适合特定软件环境所具备的技术责任

水壶的责任

- 可以倒水，不会将水溅出。
- 可以盛水，并且加热直至沸腾。
- 当水煮沸的时候，发出通知。
- 提供一种安全，简便的提携方法。

对象与系统

- “不要试着把对象在现实世界中可以想象到的行为都实现到设计中去。相反，只需要让对象能够合适于应用系统即可。对象能做的，所知的最好是一点不多一点不少。”

---Jon Kern

职责与角色

- 一个对象维护其自身的状态需要对外公开一些方法，行使其职能也要对外公开一些方法，这些方法组合起来定义了该对象允许外界访问的方法，或者说限定了外界可以期望的表现，它们是对对象需要对外界履行的协议（Protocol）。
- 一个对象的整体协议可能会分为多个内聚的逻辑行为组
 - 例如，一个学生对象的有些行为是在学习时发生的，而另外一些可能是在购物时发生的，这样，学生对象的行为就可以分为两组。
- 划分后的每一个逻辑行为组就描述了对对象的一个独立职责，体现了对象的一个独立角色。
- 如果一个对象拥有多个行为组，就意味着该对象拥有多个不同的职责，需要扮演多个不同的角色。
 - 例如，上例的学生对象就需要同时扮演学生和顾客两个角色。每一个角色都是对象一个职责的体现，所有的角色是对象所有职责的体现。所以，理想的单一职责对象应该仅仅扮演一个角色。

协作

Outline

- 回顾类的职责
- 类的协作
 - 消息传递
 - 用例图
- 类之间的关系

协作

- 一组对象共同协作履行整个应用软件的责任。
- 设计的焦点是从发现对象及其责任转移到对象之间如何通过互相协作来履行责任。

“如何做”，“何时做”和“与谁工作”

- 在此之前，对于“这个对象做这件事”和“那个对象做那件事”的讨论都是以一个概念为依据的，即对象随时可以获得他们所需要的信息或服务。
- 随着执行请求在对象之间流传，对象之间的协作关系也不断地发生变化。
- 直到我们描述了对对象之间的联系及相互作用时，模型才是完整的。
- 协作模型描述的是一些关于“如何做”，“何时做”和“与谁工作”的动态行为。

职责分配

- 当将具有共同责任的对象组织成“邻域”时，需要细致地安排邻里之间的协作关系以履行其更大的责任。同样也要指定邻域之外的对象如何与邻域对象提供的服务进行交互。这样做的好处之一是，当我们对系统部分作修改时不会涉及整个系统。一个设计良好的面向对象应用软件应包含一定量的变化，而不用做大幅度的改动。
- 软件实现了一个责任系统。不同的角色通过协作履行不同的责任，良好的软件结构可以有效地执行这些责任。我们的设计工作从创建对象开始，将特定的责任分配给对象，使其了解某些信息或完成某些工作，而这些对象的集体行为将履行更大的责任。
- 一个对象所提供的服务和所持有的信息定义了该对象的行为方式，并由此和其他对象区分开来。在早期设计中，只需要将具体责任分配给对象，这就已经足够了。最重要的是，保证责任为他人服务。设计模型负责在对象之间分配责任。

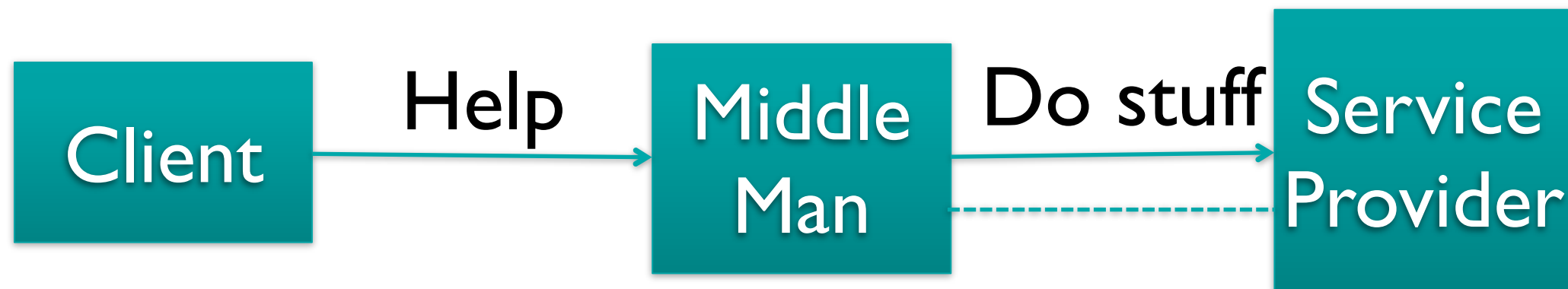
对象的角色

- 从消息传递的角度
 - 客户
 - 服务器
 - 代理

客户服务器模式



客户代理服务器模式



Case Study: 智能热水器

- 智能控制水温
 - 周末水温高
 - 夜晚水温低
 - 生病等特殊情况水温高
 - 度假水温低

概念模型

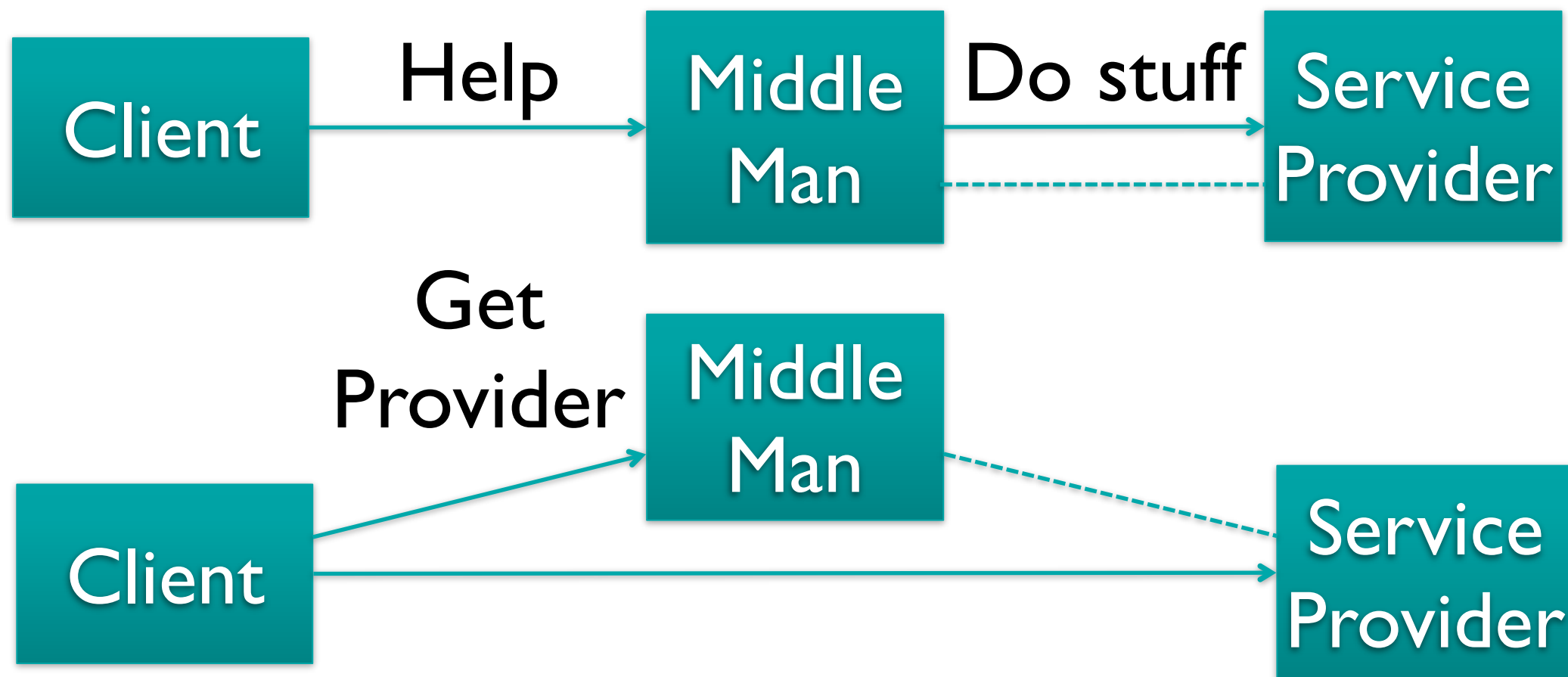
- Class:
 - WaterHeaterController
 - mode
 - lowTemp
 - highTemp
 - weekendDays
 - Special Time
- Interface:
 - ThermostatDevice

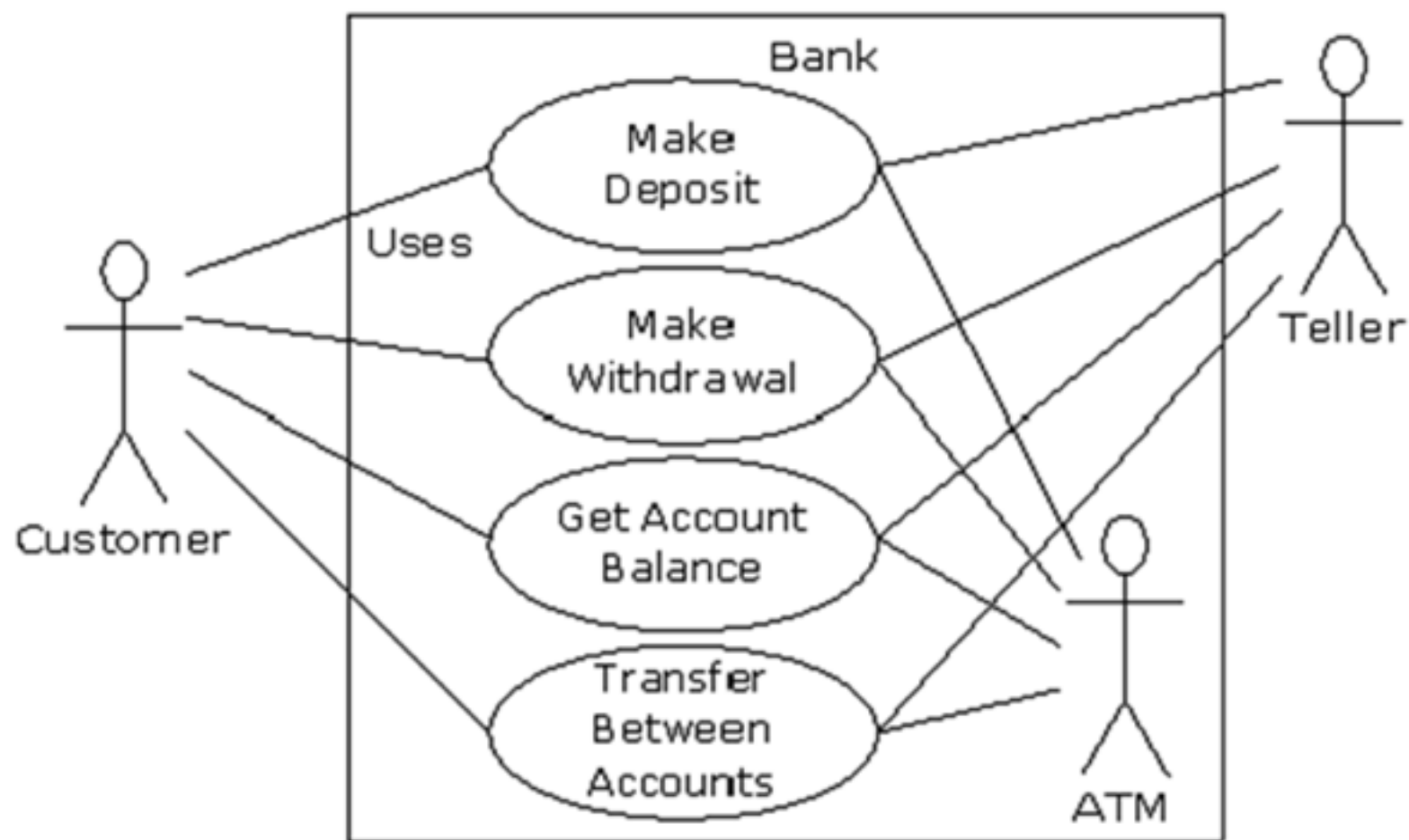
怎么知道当前时间是不是特殊时期

- Controller自己保存特殊时间并计算（比较当前时间和特殊时间）
 - Bad：多个职责。
- 由SpecialTime类保存特殊时间；Controller调用getSpecialTime（）得到特殊时间，再计算
 - Bad：数据职责与行为职责的分离
- 由SpecialTime类保存特殊时间，并提供isSpecialTime（）；Controller调用方法
 - Good：单一职责

协作对象

- 该对象自身
- 任何以参数形式传入的对象
- 被该对象直接创建的对象
- 其所持有的对象引用





用例图

寻找参与者

- 谁对系统有着明确的目标和要求并且主动发出动作?
- 系统是为谁服务的?

用例的特征

- 用例是相对独立的
 - 取钱？ 填写取款单？
- 用例的执行结果对参与者来说是可观测的和有意义的
 - 登陆系统？ 后台进程监控？
- 这件事必须有一个参与者发起。
 - **ATM吐钞票？**
- 用例必须是以动宾短语形式出现的
 - 统计？ 报表？
- 一个用例就是一个需求单元、分析单元、设计单元、开发单元、测试单元，甚至部署单元

目标和步骤

- 参与者： 寄信人
- 用例：
- 买信封
- 买邮票
- 付钱
- 投递

ID:	用例的标识，通常会结合用例的层次结构使用 X.Y.Z 的方式
名称:	对用例内容的精确描述，体现了用例所描述的任务，通常是“动词+名词”
用例属性	包括创建者、创建日期、更新历史等
参与者:	描述系统的主参与者、辅助参与者和每个参与者的目标
描述:	简要描述用例产生的原因，大概过程和输出结果
优先级:	用例所描述的需求的优先级
触发条件:	标识启动用例的事件，可能是系统外部的事件，也可能是系统内部的事件，还可能是正常流程的第一个步骤
前置条件:	用例能够正常启动和工作的系统状态条件
后置条件:	用例执行完成后的系统状态条件
正常流程:	在常见和符合预期的条件下，系统与外界的行为交互序列
分支流程:	用例中可能发生的非常见的其他合理场景
异常流程:	在非预期的错误条件发生时，系统对外界进行响应的交互行为序列
相关用例:	记录和该用例存在关系的其他用例，关于用例之间的关系见 10.4.4
业务规则:	可能会影响用例执行的业务规则
特殊需求:	和用例相关的其他特殊需求，尤其是非功能性需求
假设:	在建立用例时所做的假设
待确定问题:	一些当前的用例描述还没有解决的问题

模版 10-1、常见的用例描述格式

用例文本描述

ID
1
名称
处理销售
创建者
最后一次更新者
创建日期
最后更新日期

参与者
收银员，目标是快速、正确地完成商品销售，尤其不要出现支付错误。

触发条件
顾客携带商品到达销售点

前置条件
收银员必须已经被识别和授权。

后置条件
存储销售记录，包括购买记录、商品清单、赠送清单和付款信息；更新库存和会员积分；打印收据。

优先级
高

正常流程

1如果是会员，收银员输入客户编号

2系统显示会员信息，包括姓名和积分

3收银员输入商品标识

4系统记录商品，并显示商品信息，商品信息包括商品标识、描述、数量、价格、特价（如果有商品特价策略的话）和本项商品总价

5系统显示已购入的商品清单，商品清单包括商品标识、描述、数量、价格、特价、各项商品总价和所有商品总价
收银员重复3-5步，直到完成所有商品的输入

6收银员结束输入，系统计算并显示总价，计算根据总额特价策略进行

7系统根据商品赠送策略和总额赠送策略计算并显示赠品清单，赠品清单包括各项赠品的标识、描述与数量

8收银员请顾客支付账单

9顾客支付，收银员输入收取的现金数额

10系统给出应找的余额，收银员找零

11收银员结束销售，系统记录销售信息、商品清单、赠送清单和账单信息，并更新库存
系统打印收据

扩展流程

- 1a、非法客户编号：
- 1、系统提示错误并拒绝输入
- 3a、非法标识：
- 1、系统提示错误并拒绝输入
- 3b、有多个具有相同商品类别的商品（如5把相同的雨伞）
- 1、收银员可以手工输入商品标识和数量
- 5-8a、顾客要求收银员从已输入的商品中去掉一个商品：
- 1、收银员输入商品标识并将其删除
 - 1a、非法标识
 - 1、系统显示错误并拒绝输入
 - 2、返回正常流程第5步
- 5-8b、顾客要求收银员取消交易
- 1、收银员在系统中取消交易
- 9a、会员使用积分
- 1. 系统显示可用的积分余额
 - 2. 营业员输入使用的积分数额，每50个积分等价于1元RMB
 - 3. 系统显示剩余的积分余额和余下的现金数额
 - 4. 收银员输入收取的现金数额
- 11a、会员
- 1、系统记录销售信息、商品清单、赠送清单和账单信息，并更新库存
 - 2、计算并更新会员积分，将积分总额和积分余额都增加现金数额
- 特殊需求**
- 1、系统显示的信息要在1米之外能看清
 - 2、因为在将来的一段时间内，超市都不打算使用扫描仪设备，所以为输入方便，要使用5位0～9数字的商品标识格式。
- 将来如果超市采购了扫描仪，商品标识格式要修改为标准要求：13位0～9的数字

类与类之间的关系

Outline

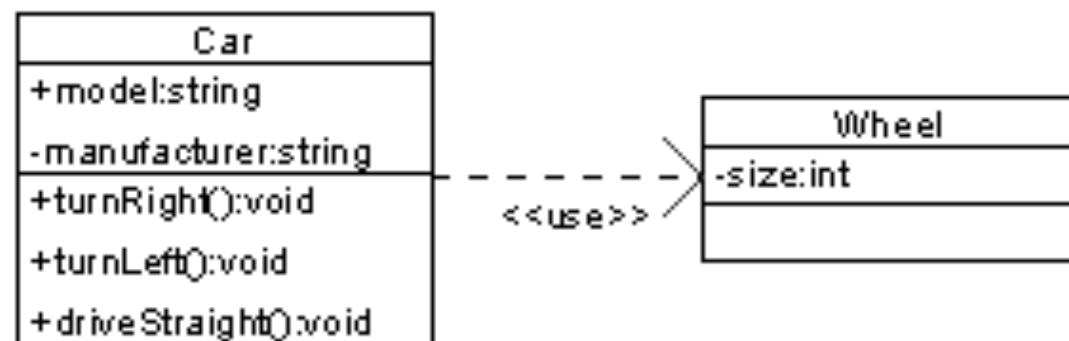
- 面向对象编程的概念与原则
- 类的协作
- 类之间的关系
 - General Relationship
 - Instance Level Relationship
 - Class Level Relationship

类之间的关系

- General Relationship
 - 依赖
- Instance Level Relationship
 - 连接
 - 关联
- Class Level Relationship
 - 继承
 - 实现

类之间的关系 - General Relationship

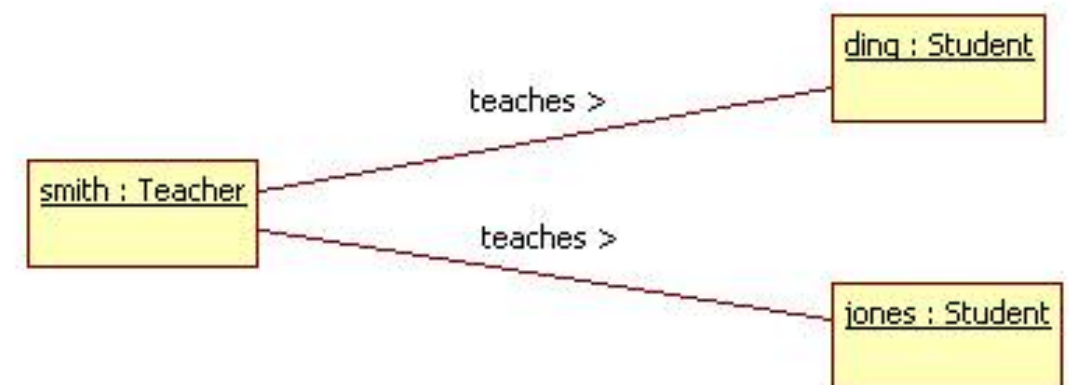
- 依赖 (Dependency)
- 物理关系(model-time relationship between definitions)



- Dependencies can exist between other elements than classes. (Requirements, use cases, objects, packages, etc.)

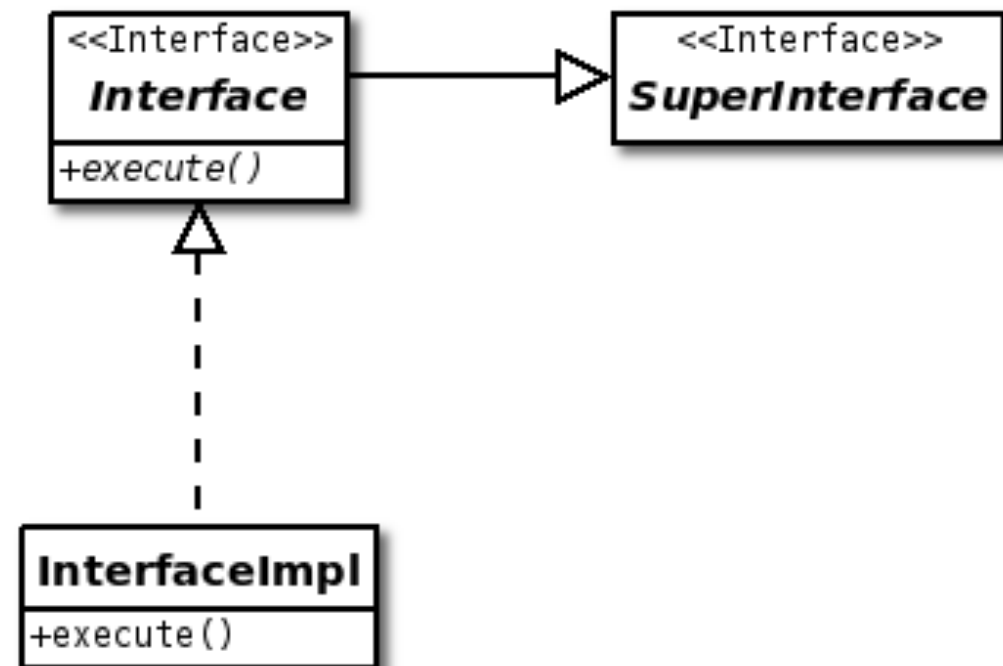
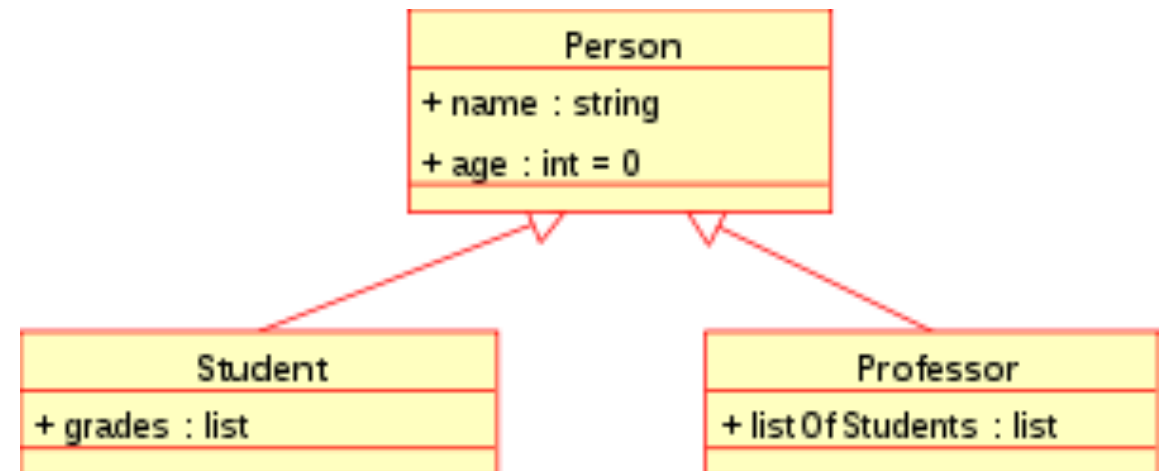
类之间的关系 – Instance Level Relationship

- 连接 (External Links)
 - Relationship among objects
 - A link is an instance of an association
- 关联 (Association)
 - 逻辑关系(run-time relationship between instances or classes)
- 关联的分类
 - 普通关联
 - 可导航关联
 - 聚合 (Aggregation)
 - 组合(Composition)
- 它们的强弱关系是没有异议的：依赖 < 普通关联 < 聚合 < 组合



类之间的关系 – Class Level Relationship

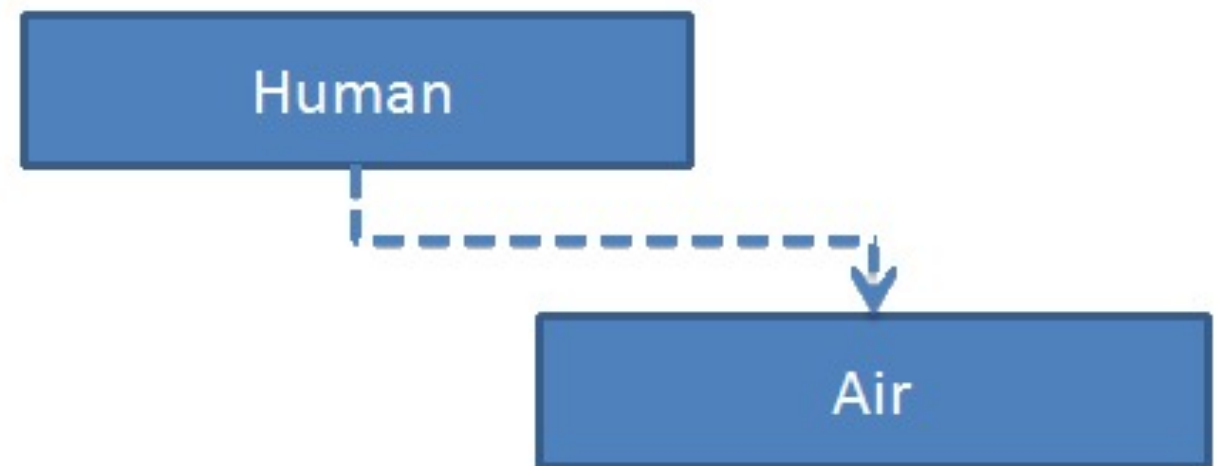
- Generalization
- 继承 (extends)
- Realization
- 实现 (implements)



依赖




- 关系: "... uses a ..."
- 所谓依赖就是某个对象的功能依赖于另外的某个对象，而被依赖的对象只是作为一种工具在**使用**，而并不持有对它的引用。



```

1. class Human
2. {
3.     public void breath()
4.     {
5.         Air freshAir = new Air();
6.         freshAir.releasePower();
7.     }
8.     public static void main()
9.     {
10.        Human me = new Human();
11.        while(true)
12.        {
13.            me.breath();
14.        }
15.    }
16. }
17.
18. class Air
19. {
20.     public void releasePower()
21.     {
22.         //do sth.
23.     }
24. }

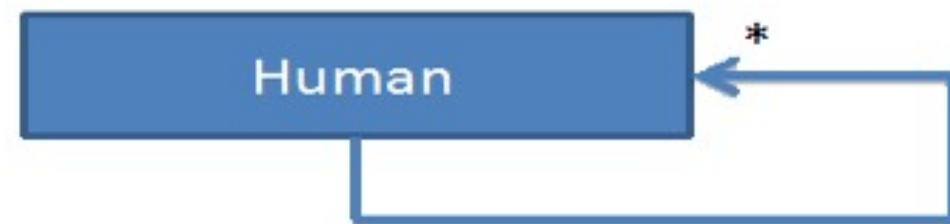
```



- 释义：一个人自创生就需要不停的呼吸，而人的呼吸功能之所以能维持生命就在于吸进来的气体发挥了作用，所以说空气只不过是人类的一个工具，而人并不持有对它的引用。

关联

- 关系: "... has a ..."
- 所谓关联就是某个对象会**长期的持有**另一个对象的引用, 而二者的关联往往也是相互的。关联的两个对象彼此间没有任何强制性的约束, 只要二者同意, 可以随时解除关系或是进行关联, 它们的生命期问题上没有任何约定。被关联的对象还可以再被别的对象关联, 所以关联是可以共享的。

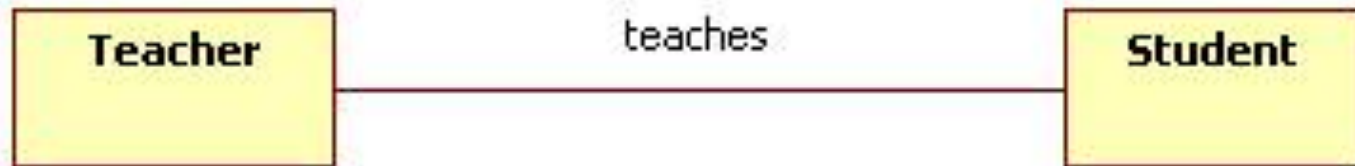


```
1. class Human
2. {
3.     ArrayList friends = new ArrayList();
4.     public void makeFriend(Human human)
5.     {
6.         friends.add(human);
7.     }
8.     public static void main()
9.     {
10.         Human me = new Human();
11.         while(true)
12.         {
13.             me.makeFriend(mySchool.getStudent());
14.         }
15.     }
16. }
17.
```

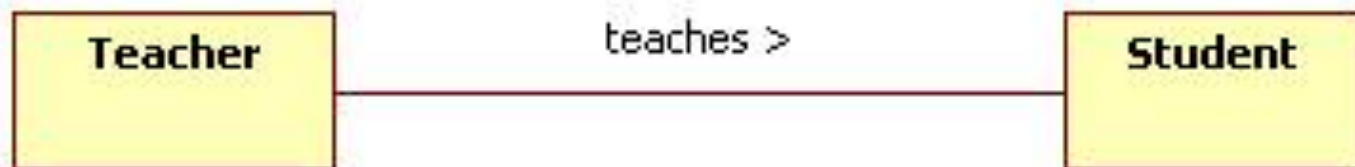
人从生至死都在不断的交朋友，然而没有理由认为朋友的生死与我的生死有必然的联系，故他们的生命期没有关联，我的朋友又可以是别人的朋友，所以朋友可以共享。

普通关联和可导航关联

- 普通关联



- 可导航关联



- 单向

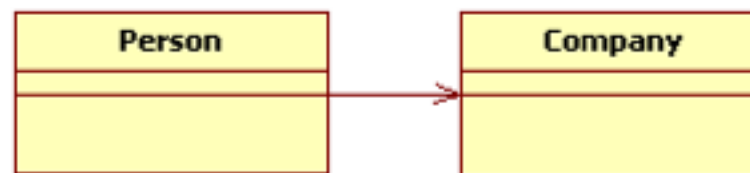
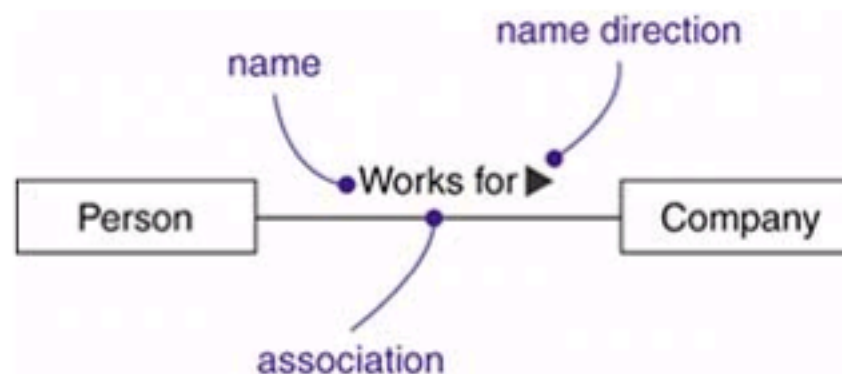


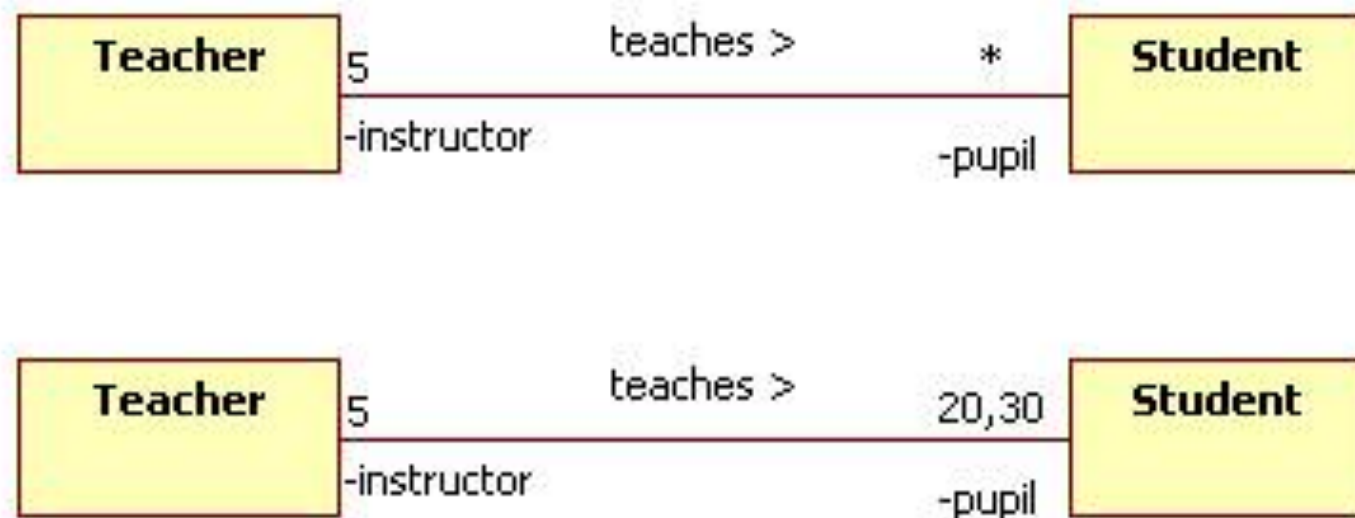
Figure 5-4. Association Names

- 双向



多重性

- 多重性 (Multiplicity)
 - the number of objects that participate in the association

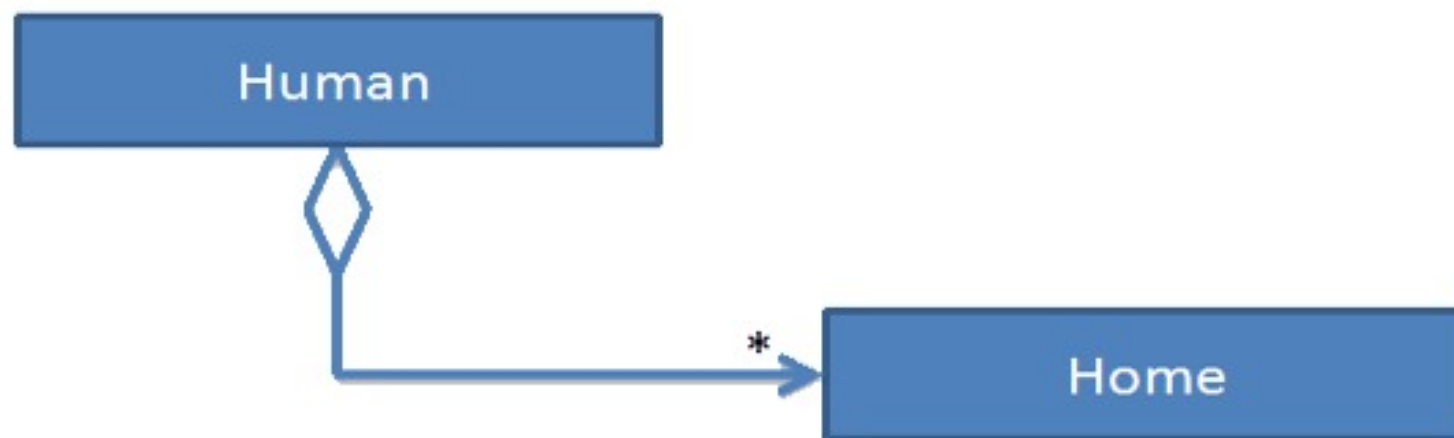


- 0..1 No instances, or one instance (optional, may)
- 1 Exactly one instance
- 0..* or * Zero or more instances
- 1..* One or more instances (at least one)

聚合



- 关系： "... owns a ..."
- 聚合是强版本的关联。它暗含着一种**所属关系以及生命期关系**。被聚合的对象还可以再被别的对象关联，所以被聚合对象是可以共享的。虽然是**共享**的，聚合代表的是一种更亲密的关系。



```
1. class Human
2. {
3.     Home myHome;
4.     public void goHome()
5.     {
6.         //在回家的路上
7.         myHome.openDoor();
8.         //看电视
9.     }
10.    public static void main()
11.    {
12.        Human me = new Human();
13.        while(true)
14.        {
15.            //上学
16.            //吃饭
17.            me.goHome();
18.        }
19.    }
```

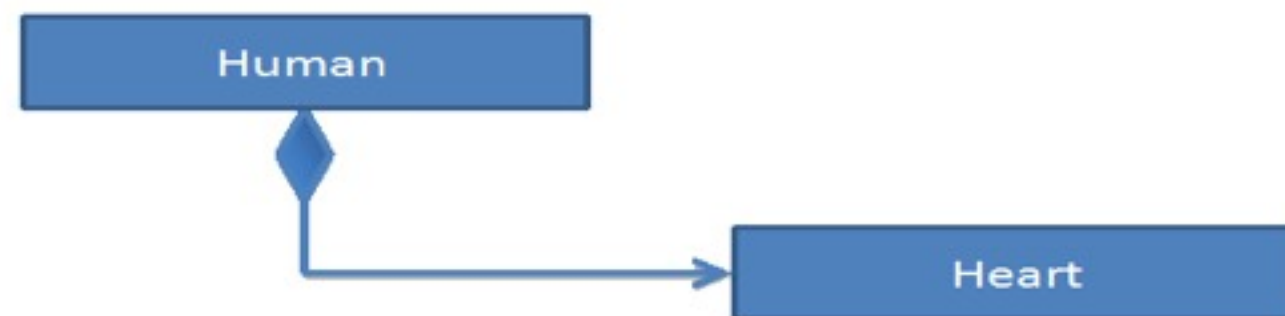


我的家和我之间具有着一种强烈的所属关系，我的家是可以分享的，而这里的分享又可以有两种。其一是聚合间的分享，这正如你和你媳妇儿都对这个家有着同样的强烈关联；其二是聚合与关联的分享，如果你的朋友来家里吃个便饭，估计你不会给他配一把钥匙。

组合



- 关系: "... is a part of ..."
- 组合是关系当中的最强版本，它直接要求包含对象对被包含对象的拥有以及包含对象与被包含对象生命期的关系。被包含的对象还可以再被别的对象关联，所以被包含对象是可以共享的，然而绝不存在两个包含对象对同一个被包含对象的共享。



```
1. class Human
2. {
3.     Heart myHeart = new Heart();
4.     public static void main()
5.     {
6.         Human me = new Human();
7.         while(true)
8.         {
9.             myHeart.beat();
10.        }
11.    }
```



组合关系就是整体与部分的关系，部分属于整体，整体不存在，部分一定不存在，然而部分不存在整体是可以存在的，说的更明确一些就是部分必须创生于整体创生之后，而销毁于整体销毁之前。

部分在这个生命期内可以被其它对象关联甚至聚合，但有一点必须注意，一旦部分所属于的整体销毁了，那么与之关联的对象中的引用就会成为空引用，这一点可以利用程序来保障。

心脏的生命期与人的生命期是一致的，如果换个部分就不那么一定，比如阑尾，很多人在创生后的某个时间对其厌倦便提前销毁了它，可它和人类的关系不可辩驳的属于组合。

类的开发指南

类的开发指南 - I

- 类需要有一个目的
 - 每个类都需要有职责。如果没有清楚的类的职责和所需的操作，那么它可能是其他类的一个部分。如果类没有目的，那就不应该存在。
- 类与属性
 - 如果类拥有一组定义良好的属性，这些属性有些相关的操作，这些操作实际上并不是类的一部分，而且有可能被其他的类独立地使用，那么这些属性和方法是一个候选的独立的类。另一方面，如果一个类没有操作，那么它的属性可以作为其他类的简单属性。由于Java没有与简单的C结构等价的东西，所以也有可能存在一些Java类实际上起到了结构的作用。它们不需要任何操作，但可以作为简单数据结构来使用。
- 类不能什么事情都做
 - 别把一个类弄得太大。类的职责应该适合在该类中实现并且与其他类无关。如果一个类试图做实际与它的主要职责无关的事，那么哪些操作可能应该属于别的类。

类的开发指南 - 2

- 名字重要
 - 为类，属性，方法和变量选择好的名字对写出好的软件是至关重要的。名字应该具有意义，有助于解释该事物的角色。避免缩写。好的，描述性的名字减少了对解释性注释的需要。键入长的名字可能要多花一些力气，但是减少注释和增强可读性带来的好处对补偿这种一次性的输入功夫还是绰绰有余的。但是请记住，占据了大半行的名字对可读性是有影响的。
- 一次做一件事
 - 类的操作应该完成单件定义良好的任务。不要让一个取值方法同时完成改变对象状态的操作。避免副作用
- 不要重新发明轮子
 - 避免去解决那些已经被解决了的问题。任何时候只要可能，请重用已经存在的代码。在任何可能的时候使用已有的库和框架。学习设计模式，再恰当的地方使用它们。

类的开发指南 - 3

- 不会一次就搞定
 - 不管是多么优秀的设计师，都做不到一次就做出正确的设计。要认识到设计会有问题，然后尽可能快地进行修正。修正问题最终产生更好的软件系统，更易于修改和维护。请学习使用重构技术。
- 简洁性
 - 让设计尽可能地简单。从一个层面上看，这意味着不应该仅仅因为您认为它会表现更好，就尝试采用那些很花哨的解决方案。有时简单的线性搜索与更花哨的，较难编码的二分搜索同样都能够很好的完成工作。在另一个层面上看，简洁性并不是指使用您首先想到的东西。发现一个简洁的，优雅的设计可能需要很大的努力，但从长远看是有回报的。。
- 您的软件不会消失
 - 软件系统常常在超过它预期的生命周期后仍在使用。所有的软件系统在设计时都应该假设它们可能永远使用下去。这意味着要为长期做设计。比起那些走捷径或假设有限生命周期的系统来，设计良好的系统更易于维护。