



## 第二章 软件过程与方法

乔立民

2011年4月6日

## 第2章 软件过程与方法

- **2.1 软件过程**

  - 2.1.1 软件过程综述**

  - 2.1.2 瀑布模型与迭代模型

  - 2.1.3 统一过程模型简介

- **2.2 敏捷软件开发**

- **2.3 软件工程方法**

- **2.4 统一建模语言简介**

# 软件过程的类比

## 产品制造过程？

市场调研—产品评估—产品设计—加工与装配—产品

软件像产品生产过程？

↑  
购买材料零件

## 房屋建造过程？

项目规划—项目预算—房屋设计—建造—房屋

软件像房屋建造过程？

↑  
购买原材料

# 软件过程

- **软件过程：**用来开发软件系统所需要的一组活动
  - 软件过程构成了软件项目管理控制的基础，并且建立了一个环境以便于技术方法的采用、工作产品（模型、文档、数据、报告、表格等）的产生、里程碑的建立、质量的保证、正常变更的正确管理。

# 软件工程所关注的对象

- **产品**：各个抽象层次的产出物；
- **过程**：在各个抽象层次之间进行映射与转换；
- 软件工程具有“**产品与过程二相性**”的特点，必须把二者结合起来去考虑，而不能忽略其中任何一方。

软件需求



软件工程方法

软件系统



# 软件过程框架

## ■ 软件通用过程框架

- 沟通：项目启动、需求获取
- 策划：项目估算、资源需求、进度计划
- 建模：创建模型，进行分析和设计
- 构建：编码和测试
- 部署：软件交付，支持和反馈

## ■ 软件过程的管理

- 软件项目跟踪和控制
- 风险管理
- 软件质量保证
- 技术评审

# 软件工程=最佳实践

- 软件系统的复杂性、动态性使得：
    - 高深的软件理论在软件开发中变得无用武之地；
    - 即使应用理论方法来解决，得到的结果也往往难以与现实保持一致；
  - 因此，软件工程被看作一种实践的艺术：
    - 做过越多的软件项目，犯的错误就越少，积累的经验越多，随后作项目的成功率就越高；
    - 对新手来说，要通过多实践、多犯错来积累经验，也要多吸收他人的失败与教训与成功的经验。
- 当你把所有的错误都犯过之后，你就是正确的了。

## 第2章 软件过程与方法

- **2.1 软件过程**

  - 2.1.1 软件过程综述

  - 2.1.2 瀑布模型与迭代模型**

  - 2.1.3 统一过程模型简介

- **2.2 敏捷软件开发**

- **2.3 软件工程方法**

- **2.4 统一建模语言简介**

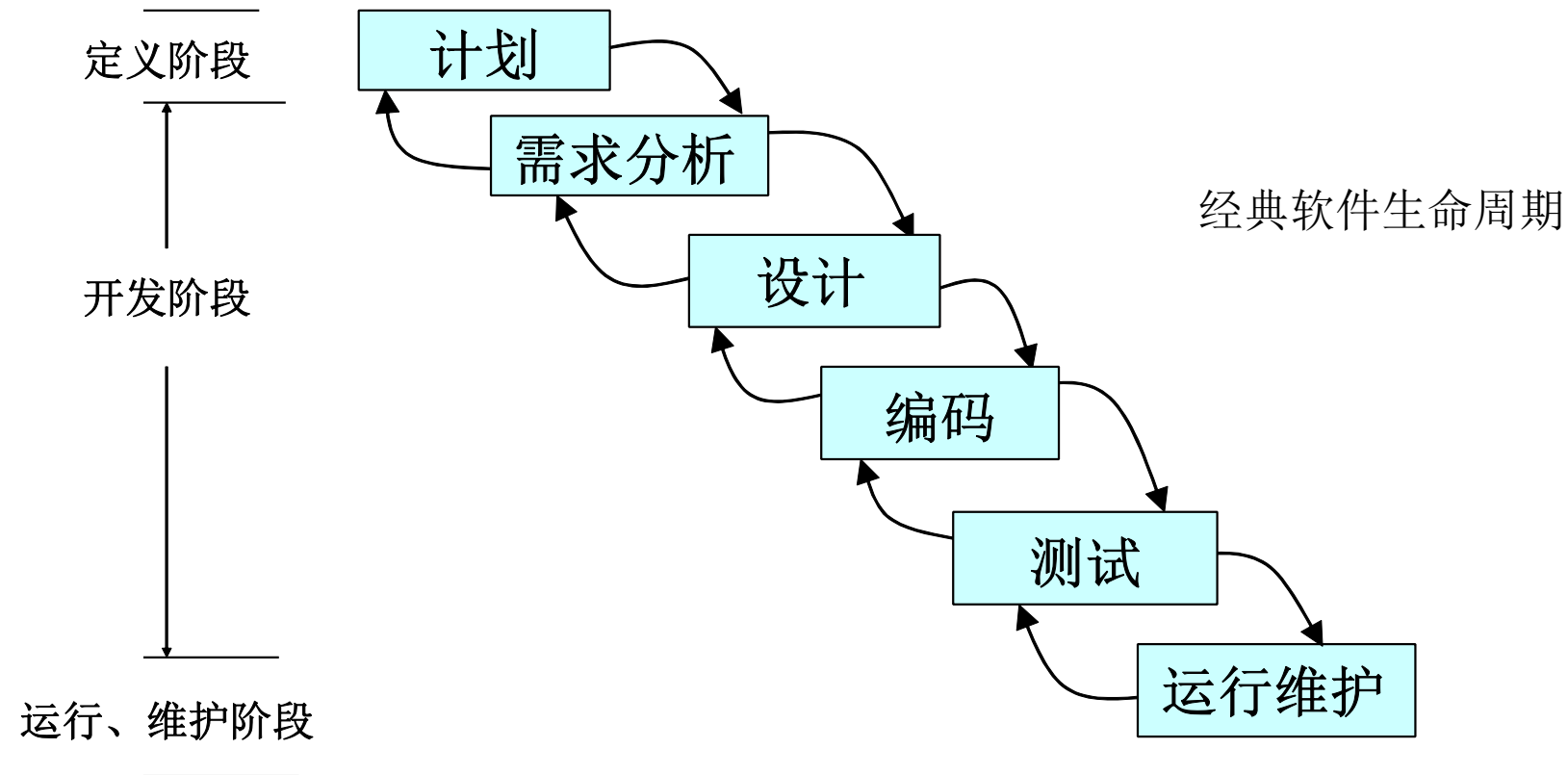


# 典型的软件过程模型

- 瀑布模型
- 迭代模型
  - 演化过程模型
    - 原型模型
    - 螺旋模型
  - 增量过程模型
    - 增量模型
    - 快速应用开发模型
  - 统一过程模型

# 瀑布模型 (Waterfall Model)

1970年, Winston Royce



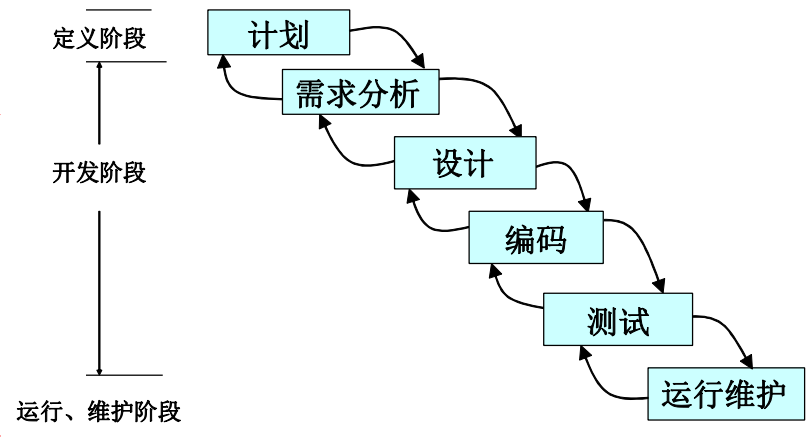
# 瀑布过程模型思想和特点

## • 基本思想

- 编码和修改代价大
- 将软件开发过程划分为分析、设计、编码、测试等阶段
- 软件开发要遵循过程规律，按次序进行
- 每个阶段均有里程碑和提交物
- 工作以线性方式进行，上一阶段的输出是下一阶段的输入；

## ■ 特点

- 需求最为重要，假设需求是稳定的
- 以文档为中心，文档是连接各阶段的关键



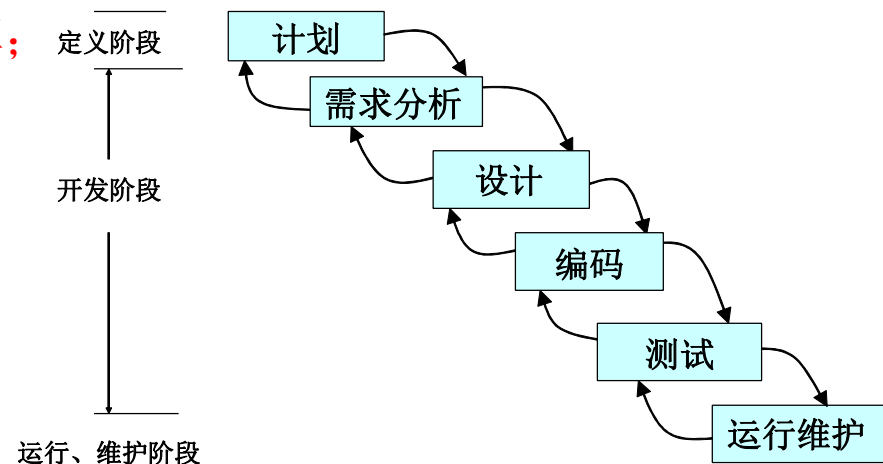
# 瀑布模型的优点和适用场合

## ■ 优点——规范，易操作

- 简单、易懂、易用；
- 为项目提供了按阶段划分的检查点，项目管理比较规范；
- 每个阶段必须提供文档，而且要求每个阶段的所有产品必须进行正式、严格的技术审查。

## ■ 适用场合

- 关键业务领域，需求控制极其严格
- 需求相当稳定，**顾客需求被全面的了解**；
- 设计直截了当，而且理解透彻
- 开发团队对于这一应用领域非常熟悉
- **外部环境的不可控因素很少**
- **小型清晰的项目或长周期的项目**



# 瀑布模型带来的问题

## ■ 稳定需求的神话

- 对于大多数软件项目，客户看到软件前，无法可靠地描述他们想要的是什么
- 开发过程能够帮助客户更好地理解自己的需求，这是需求变更的主要来源
- 开发人员对用户问题领域的看法，会在开发该领域软件的过程中逐渐熟悉
- 计划严格按照需求行事，实际上就是计划不对客户的要求作出回应
- 平均水平的项目开发过程中会有**25%**，大型项目有**40%**的需求变化
- 需求变更导致的返工占总返工量的**80%**
- 瀑布方法需求中**45%**从未被使用，时间计划与实际相差**4倍**

# 瀑布模型带来的问题

## ■ 软件工程文档

- 项目建议书
- 可行性分析报告
- 需求分析报告
- 概要设计报告
- 系统架构设计报告
- 数据库设计报告
- 详细设计报告
- 界面设计报告
- 集成测试报告
- 系统安装配置说明
- 用户使用报告

... ..

## ■ 文档的问题

- 文档过于繁杂，占用大量的时间
- 对于文档的评估需要各领域的专家，文档是否有效？
- 据统计，一个中型的瀑布过程软件项目有**68**种文档
- 当某一文档调整后的影响，不同文档间如何保持一致性？
- 产品重要还是过程重要？程序还是文档？

# 软件项目的特点

- 需求变化是软件项目中不变的因素，所以要适应需求的变化
- 软件生产中最重要因素是人，人又是最难控制因素
- 软件规模越来越大，交付期要求越来越短
- 客户更加挑剔，对软件的要求越来越高
- 其他产品的瑕疵可以容忍，而软件的问题不能容忍
- 缺少准确、直观的方法刻画需求（不同于建筑领域），与用户沟通是把握需求的最佳途径
- 缺少第三方的质量控制机构，过程管理缺少标准
- 大多数软件过程职责区分不明显，程序员往往就是设计师
- 软件的修改代价较小（相对于建筑），至少客户这样认为

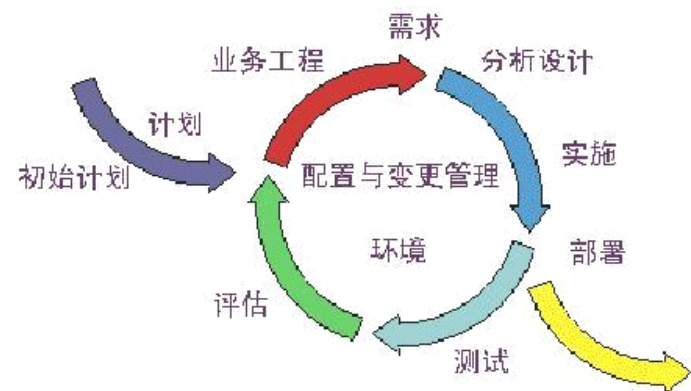
# 迭代模型

## ■ 迭代的思想

- 对一系列活动的重复应用，用以评估一系列论断，解决一系列风险，达成一系列开发目标，并逐步增量地建立并完善一个有效的解决方案

## ■ 适用场合

- 需求并没有理解透彻，或者出于其他理由你认为它是不稳定的
- 设计很复杂，或者具有挑战性，或者两者兼具
- 开发团队对于这一领域不熟悉
- 项目包含许多风险
- 具有高素质的项目管理者 and 软件研发团队





# 迭代模型

## ■ 典型的迭代模型

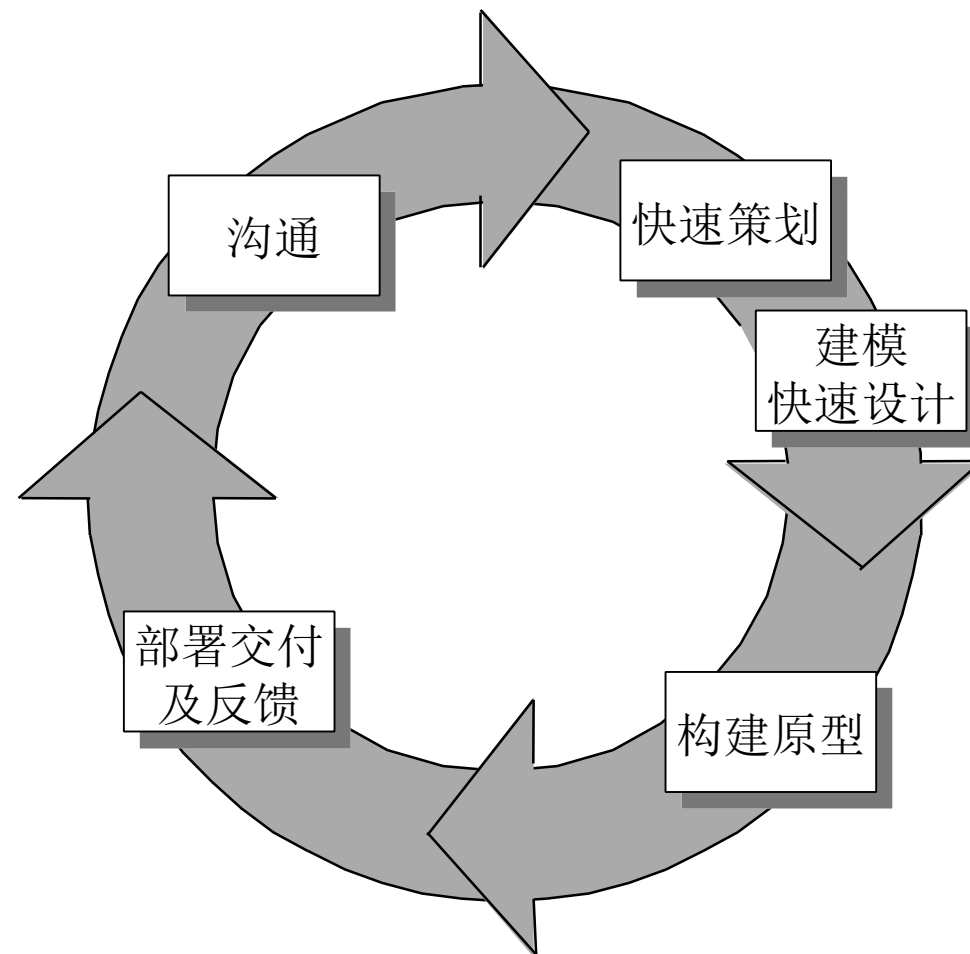
### (1) 演化过程模型

- 原型开发
- 螺旋模型

### (2) 增量过程模型

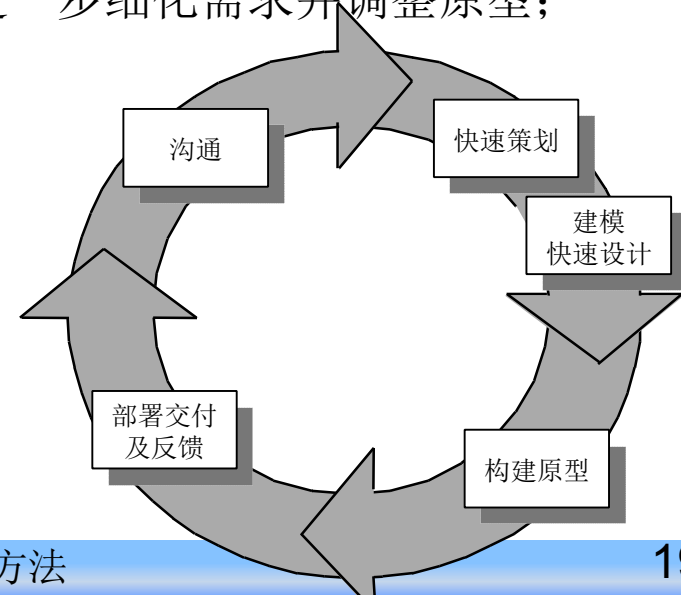
- 增量模型
- 快速应用开发模型 (RAD)

## (1) 演化过程模型——原型开发



# 原型开发模型的使用方法

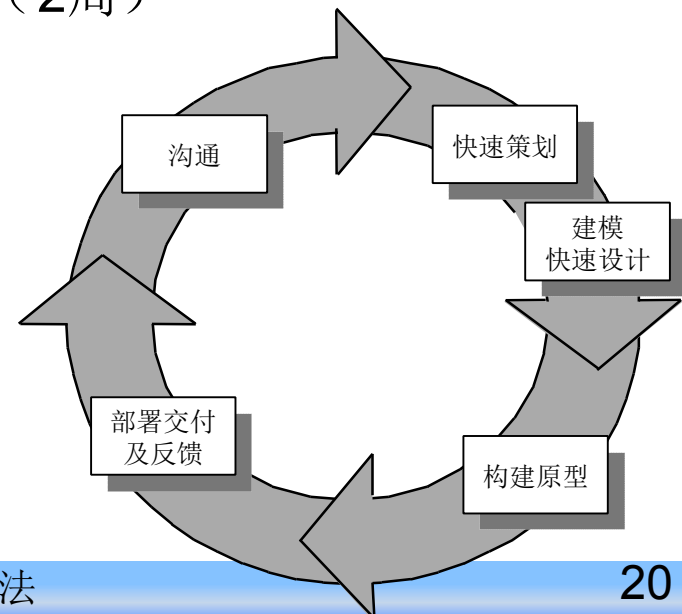
- 试验性开发，客户提出了软件的一些基本功能，但是没有详细定义其他需求；或者开发人员对与一些技术的使用不确定。
  - Step 1: 双方通过沟通，明确已知的需求，并大致勾画出以后再进一步定义的东西。
  - Step 2: 迅速策划一个原型开发迭代并进行建模，主要集中于那些最终用户所能够看到的方面，如人机接口布局或者输出显示格式等；
  - Step 3: 快速设计产生原型，对原型进行部署，由客户和用户进行评价；
  - Step 4: 根据反馈，抛弃掉不合适的部分，进一步细化需求并调整原型；
  - Step 5: 原型系统不断调整以逼近用户需求。



# 原型开发应用举例

## ■ 应用举例：开发一个教务管理系统

- 第一次迭代：完成基本的学籍管理、选课和成绩管理功能；（6周）
  - 客户反馈基本满意，但是对大数据量运行速度慢效率，不需要学生自己维护学籍的功能等
- 第二次迭代：修改细节，提高成绩统计和报表执行效率（2周）
  - 客户反馈：需要严格的权限控制，报表打印格式不符合要求
- 第三次迭代：完善打印和权限控制功能；（2周）
  - 客户反馈：可以进行正式应用验证



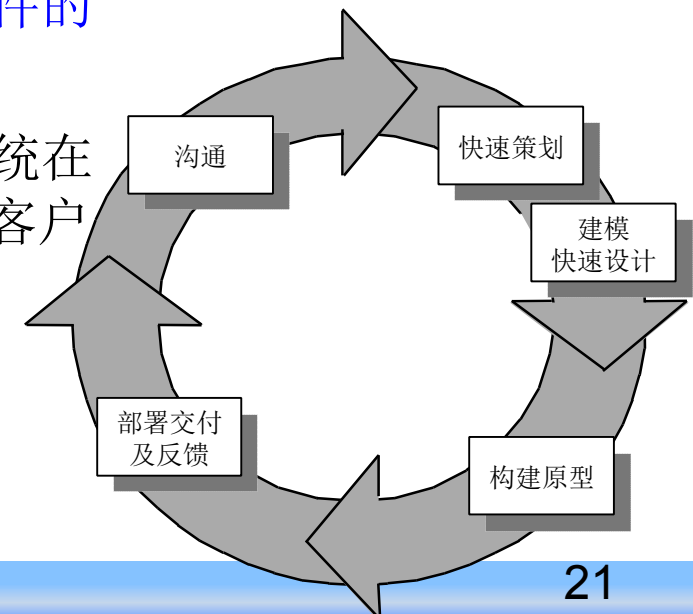
# 原型开发的优点和问题

## ■ 优点:

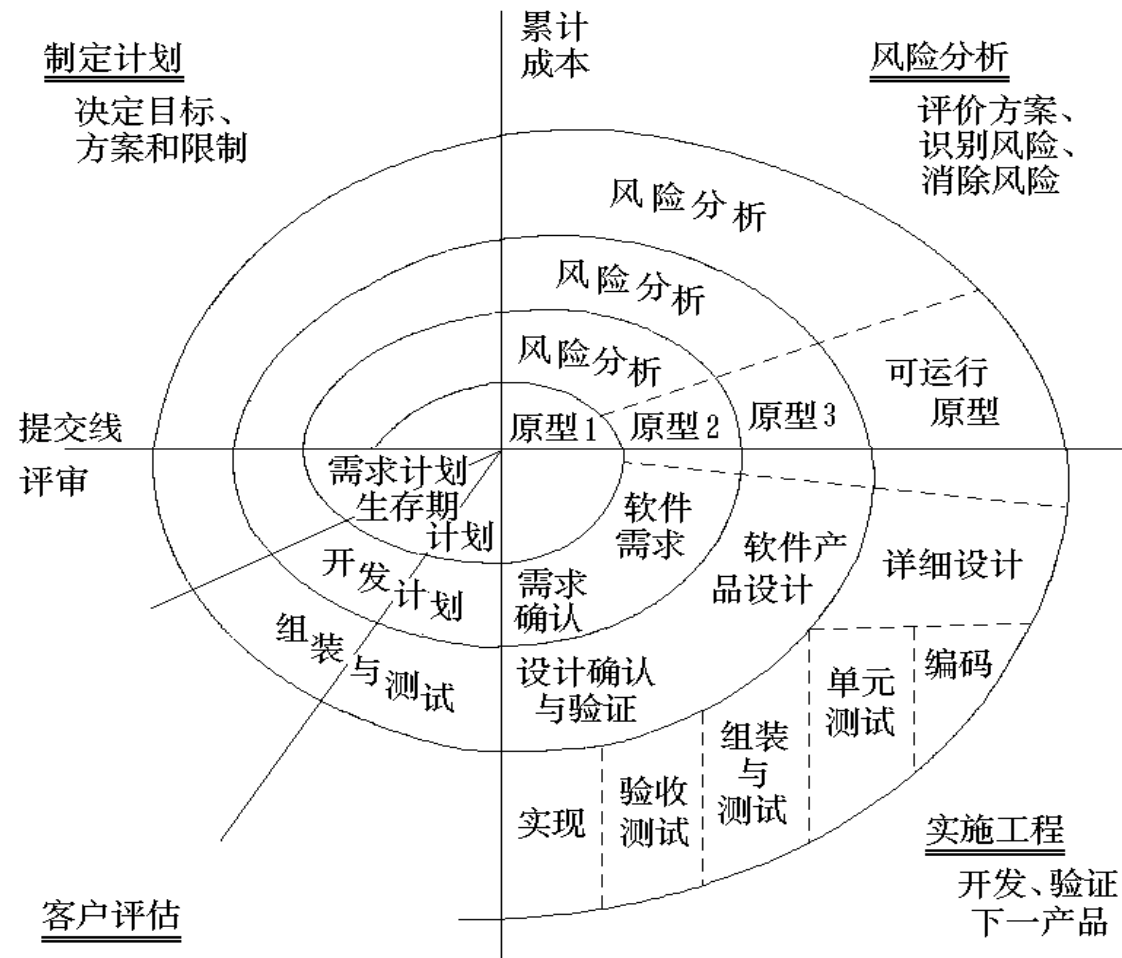
- 快速开发出可以演示的系统，方便与客户沟通；
- 采用迭代技术能够使开发者逐步弄清客户的需求；

## ■ 问题:

- 为了尽快完成原型，开发者没有考虑整体软件的质量和长期的可维护性，系统结构通常较差；
- 用户可能混淆原型系统与最终系统，原型系统在完全满足用户需求之后可能会被直接交付给客户使用；

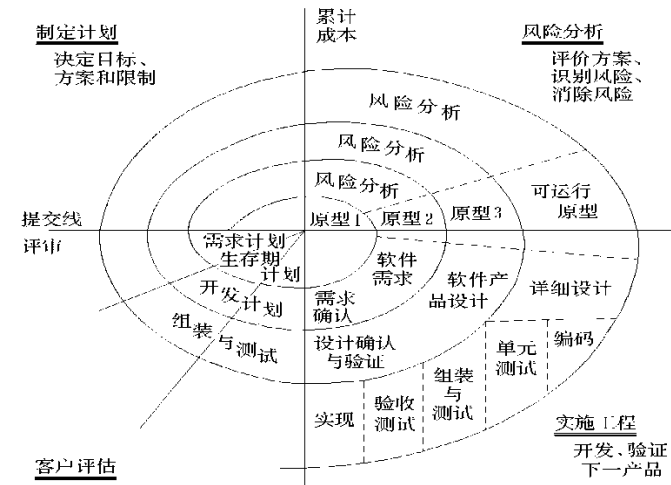


# (1)增量过程模型——螺旋模型



# 螺旋模型的特点

- 结合了原形的迭代性质和瀑布模型的系统性和可控性
- 螺旋模型沿着螺线旋转，在四个象限内表达四个方面的活动：
  - **制定计划**：确定软件目标，选定实施方案，弄清项目开发的限制；
  - **风险分析**：分析所选方案，考虑如何识别和消除风险；
  - **实施工程**：实施软件开发；
  - **客户评估**：评价开发工作，提出修正建议。
- **出发点：开发过程中及时识别和分析风险，并采取适当措施以消除或减少风险来的危害。**
- **举例：**
  - 第1圈：开发出产品的规格说明；
  - 第2圈：开发产品的原型系统；
  - 第3~n圈：不断的迭代，开发不同的软件版本；
  - 根据每圈交付后用户的反馈来调整预算、进度、需要迭代的次数；

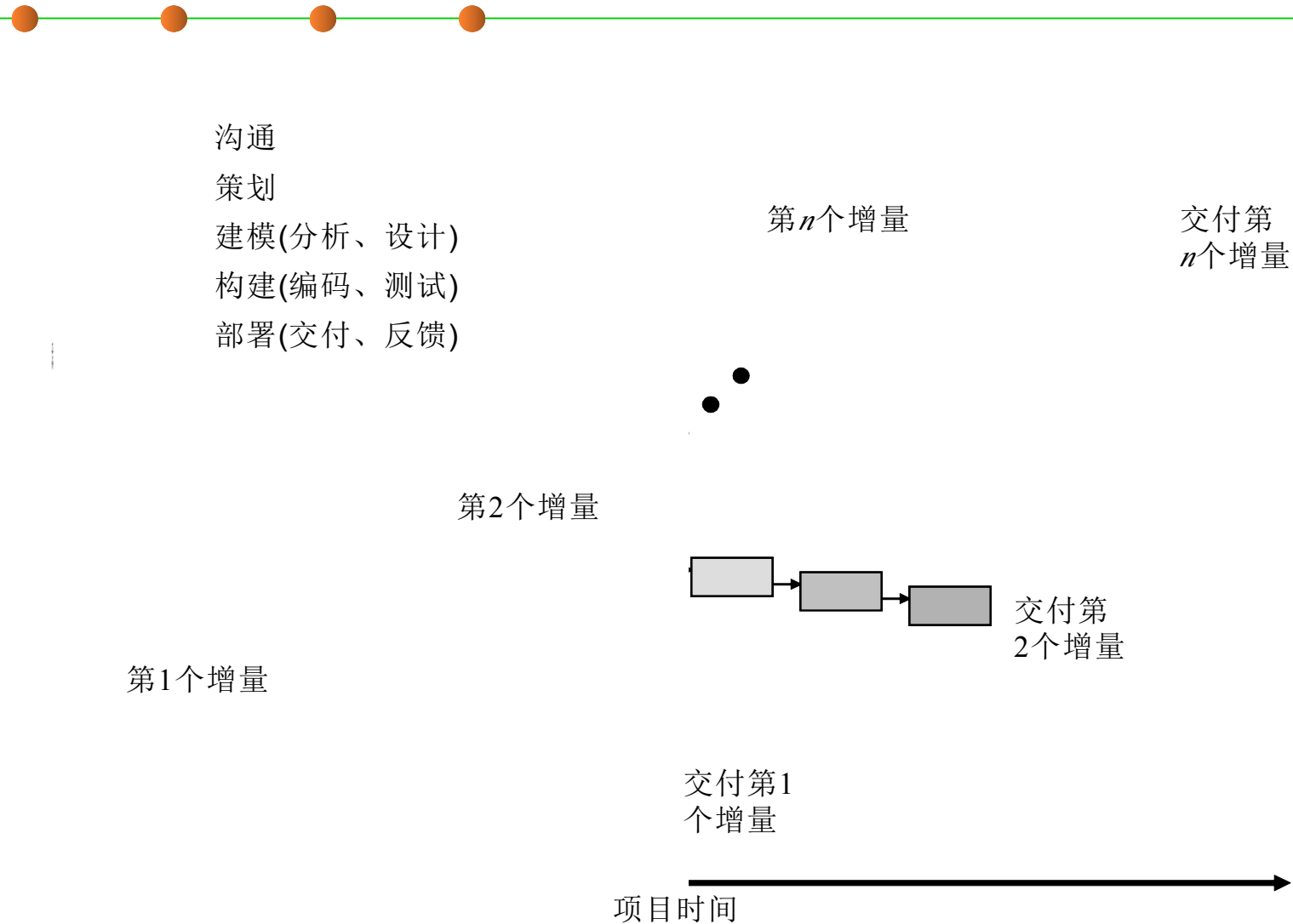


# 螺旋模型的优点和问题

- **优点：**结合了原型的迭代性质与瀑布模型的系统性和可控性，是一种**风险驱动型的过程模型**：
  - 采用循环的方式逐步加深系统定义和实现的深度，同时更好的理解、应对和降低风险；
  - 确定一系列里程碑，确保各方都得到可行的系统解决方案；
  - 始终保持可操作性，直到软件生命周期的结束；
  - 由风险驱动，支持现有软件的复用。
- **问题：**
  - 适用于大规模软件项目，特别是内部项目，周期长、成本高；
  - 软件开发人员应该擅长寻找可能的风险，准确的分析风险，否则将会带来更大的风险。

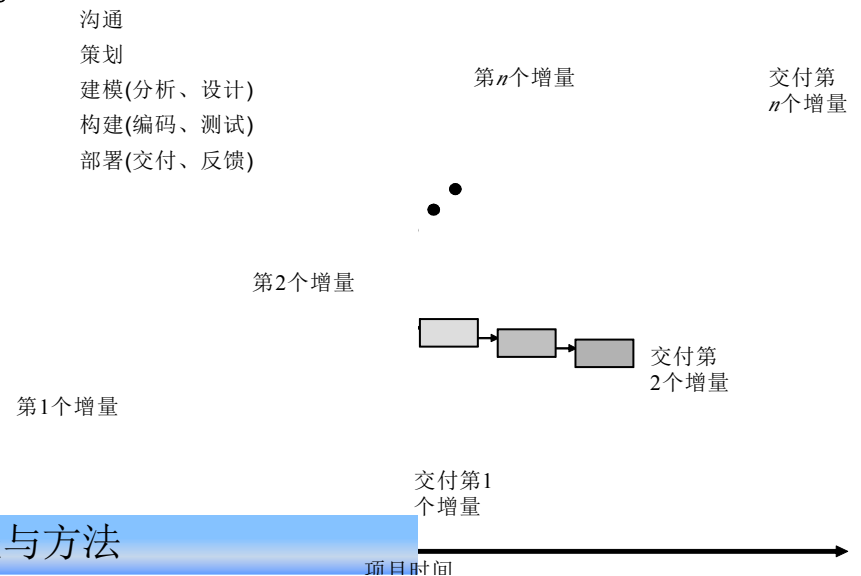


## (2) 增量过程模型——增量模型



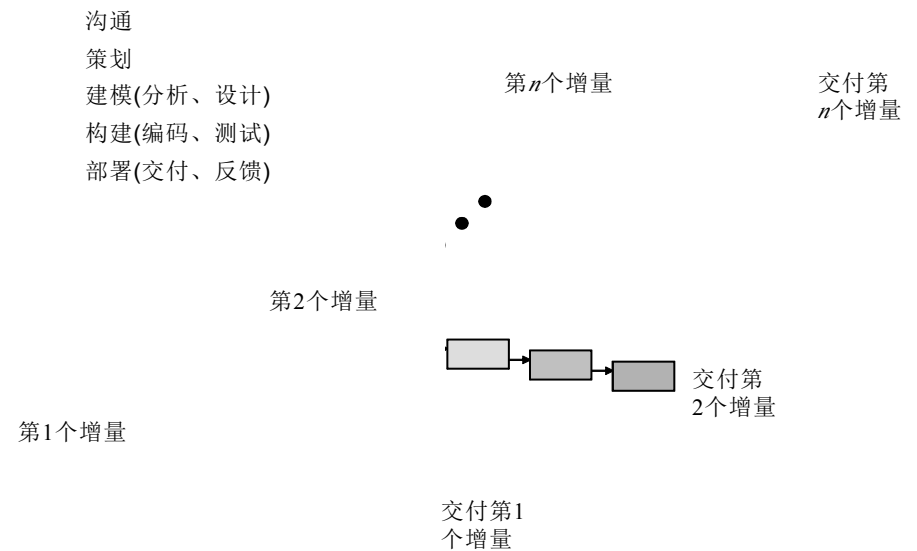
# 增量模型的使用方法和本质

- 使用方法：软件被作为一系列的增量来进行开发，每一个增量都提交一个可以操作的产品，可供用户评估。
- 第一个增量往往是核心产品：满足了基本的需求，但是缺少附加的特性；
- 客户使用上一个增量的提交物并进行自己评价，制定下一个增量计划，说明需要增加的特性和功能；
- 重复上述过程，直到最终产品产生为止。
- 本质：以迭代的方式运用瀑布模型



# 增量模型的应用举例

- 应用举例：开发一个类似于**Word**的字处理软件
  - 增量1：提供基本的文件管理、编辑和文档生成功能；
  - 增量2：提供高级的文档编辑功能；
  - 增量3：实现拼写和语法检查功能；
  - 增量4：完成高级的页面排版功能；



# 增量模型的优点和问题

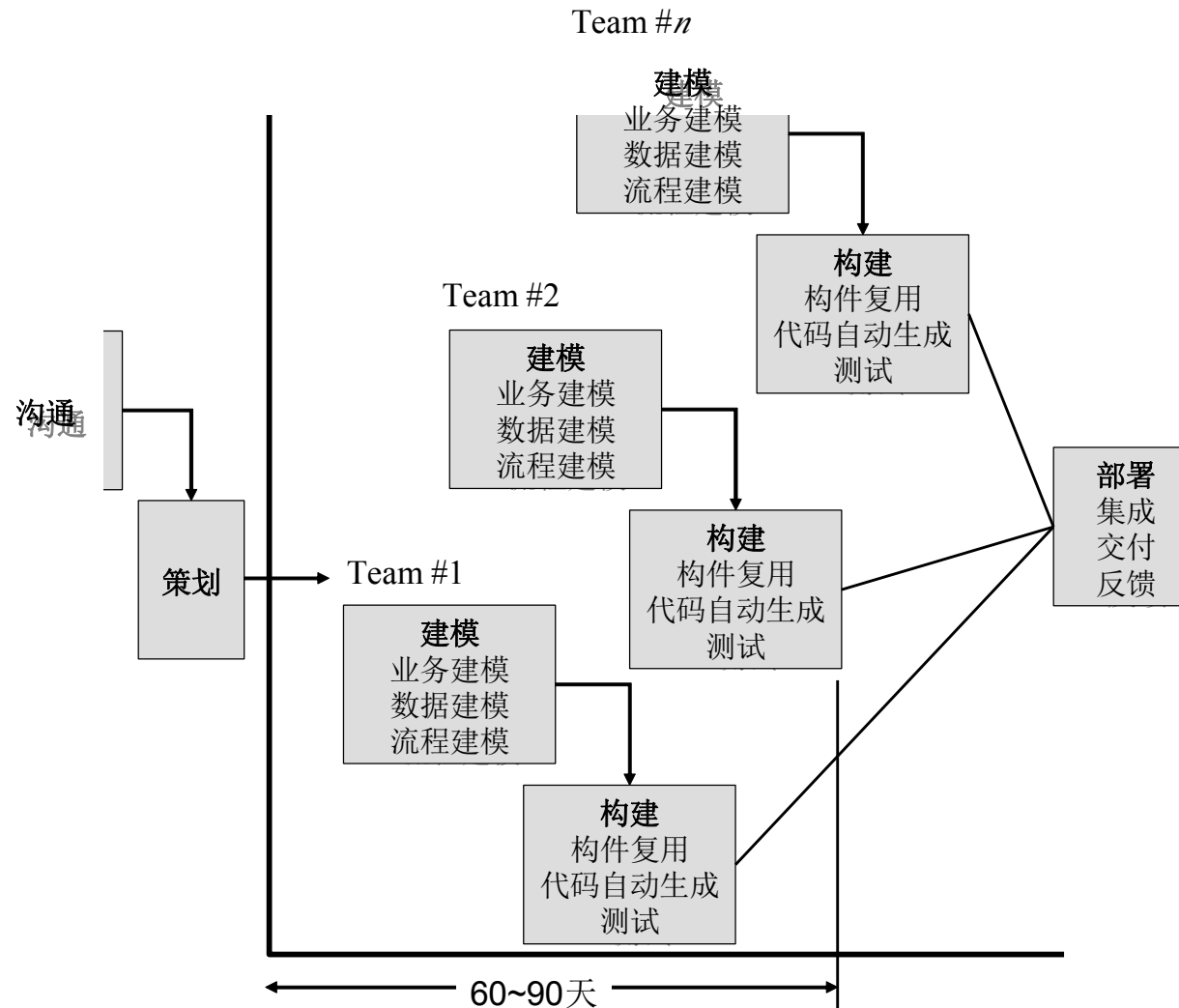
## ■ 优点：

- 提高对用户需求的响应：用户看到可操作的早期版本后会提出一些建议和需求，可以在后续增量中调整。
- 人员分配灵活：如果找不到足够的开发人员，可采用增量模型，早期的增量由少量人员实现，如果客户反响较好，则在下一个增量中投入更多的人力；
- 可规避技术风险：不确定的功能放在后面开发

## ■ 问题：

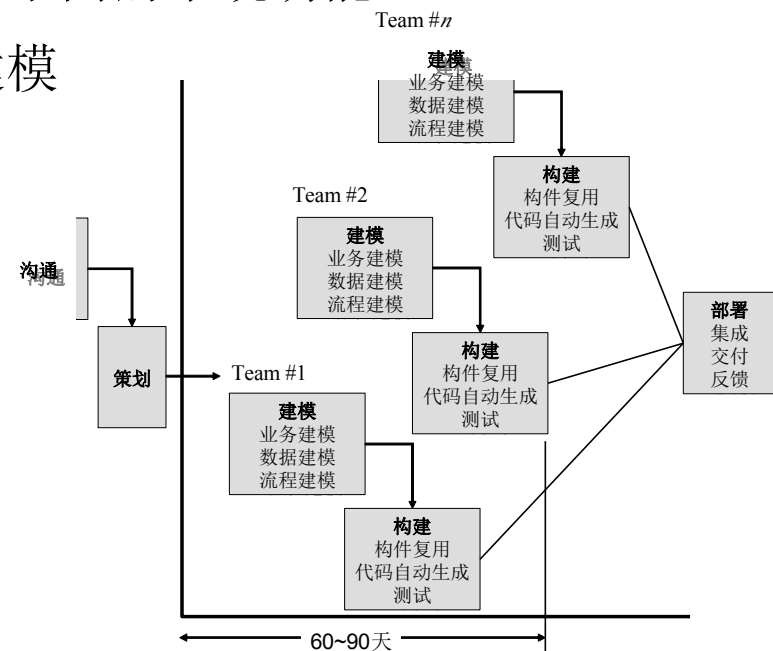
- 每个附加的增量并入现有的软件时，必须不破坏原来已构造好的东西。
- 同时，加入新增量时应简单、方便——该类软件的体系结构应当是开放的；
- 仍然无法处理需求发生变更的情况；
- 管理人员须有足够的技术能力来协调好各增量之间的关系。

## (2)增量过程模型——快速应用开发（RAD）模型



# RAD模型的使用方法

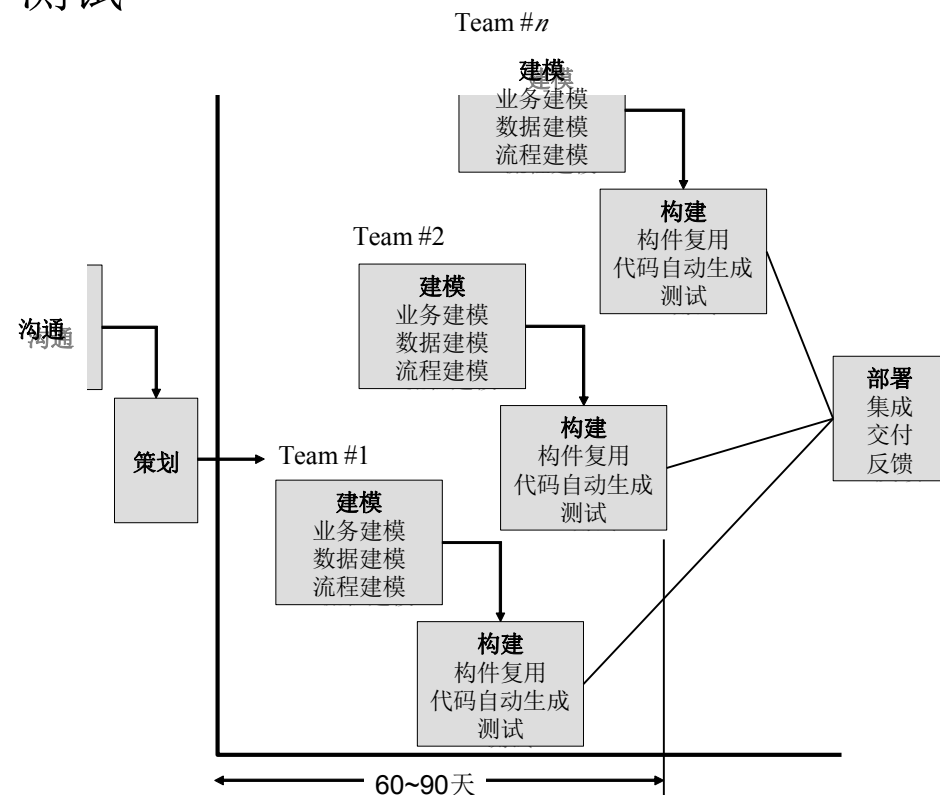
- 侧重于短开发周期(一般为**60~90天**)的增量过程模型，是瀑布模型的高速变体，通过基于构件的构建方法实现快速开发；
  - 需求被很好地理解，并且项目的边界也是固定的
  - 沟通用来理解业务问题和软件产品必须具有的特征
  - 策划确保多个软件团队能够并行工作于不同的系统功能
  - 建模包括业务建模、数据建模和过程建模
  - 构建侧重于利用已有的软件构件
  - 快速部署，为迭代建立基础



# RAD模型应用举例

## ■ 应用举例：依据已有构件开发企业**ERP**系统（1-3个月）

- 项目集中调研和规划，确定业务规范
- 划分项目组，并行建模、构建、测试
  - Team1：供应链管理体系
  - Team2：生产管理体系
  - Team3：质量、设备管理体系
  - Team4：销售管理体系
- 集成测试及交付
- 部署与反馈



# RAD模型模型的优点和问题

## ■ 优点:

- 充分利用企业已有资产进行项目开发
- 提高软件交付速度

## ■ 问题:

- 需要大量的人力资源来创建多个相对独立的RAD团队;
- 如果没有在短时间内为急速完成整个系统做好准备, RAD项目将会失败;
- 如果系统不能被合理的模块化, RAD将会带来很多问题;
- 如果系统需求是高性能, 并且需要通过调整构件接口的方式来提高性能, 不能采用RAD模型
- 技术风险很高的情况下(采用很多新技术、软件需与其他已有软件建立集成、等等), 不宜采用RAD。



## 关于迭代的错误理解

- 软件功能没有全部实现我们还不能测试，等开发迭代完成再测试吧。
- 我们的软件已经进行**2**次需求分析迭代了，正在进行设计迭代，计划迭代**2**次。
- 我们的软件才经过一次迭代，**Bug**还很多，不能使用
- 我们的软件已经测试**2**次了，再经过一次测试迭代就能发布了。

## 案例讨论

- **2003年慧通新意公司“华润酒精ERP系统”**
- **范围：**涵盖企业销售、物流、生产、质量、设备、人力、成本等业务，**10**多个部门，用户数量**300**多人。
- **项目过程模型与计划：1年**
- **参与人员：**学校老师、学生，部分刚毕业的员工
- **项目实际情况：3年**
  - 2003年底完成设计
  - 2004年底完成开发
  - 2005年底客户验收
- **总结？**

## 第2章 软件过程与方法

- **2.1 软件过程**

  - 2.1.1 软件过程综述

  - 2.1.2 瀑布模型与迭代模型

  - 2.1.3 统一过程模型简介**

- **2.2 敏捷软件开发**

- **2.3 软件工程方法**

- **2.4 统一建模语言简介**

## 最佳的软件开发实践（IBM）

在商业运作中已经证明这是一种能够解决软件开发过程中跟本问题的方法：

- 迭代开发：软件开发的特性
- 需求的管理：需求是变化的、持续的
- 应用基于构件的架构：提高重用、相互独立、适应变化
- 可视化软件建模：消除理解歧义
- 持续质量验证：迭代测试、提早发现问题
- 控制软件变更：软件过程控制与管理

# RUP(Rational Unified Process)起源

## ■ 面向对象

- 60年代 Alan Kay 发明OO语言 Smalltalk和面向对象编程（Object-Oriented programming,OOP）
- 1982年Grady Booch提出面向对象设计（Object-Oriented Design,OOD）
- 80年代末，Peter Coad创建完整的OOA/D方法

## ■ 三剑客及其建模方法

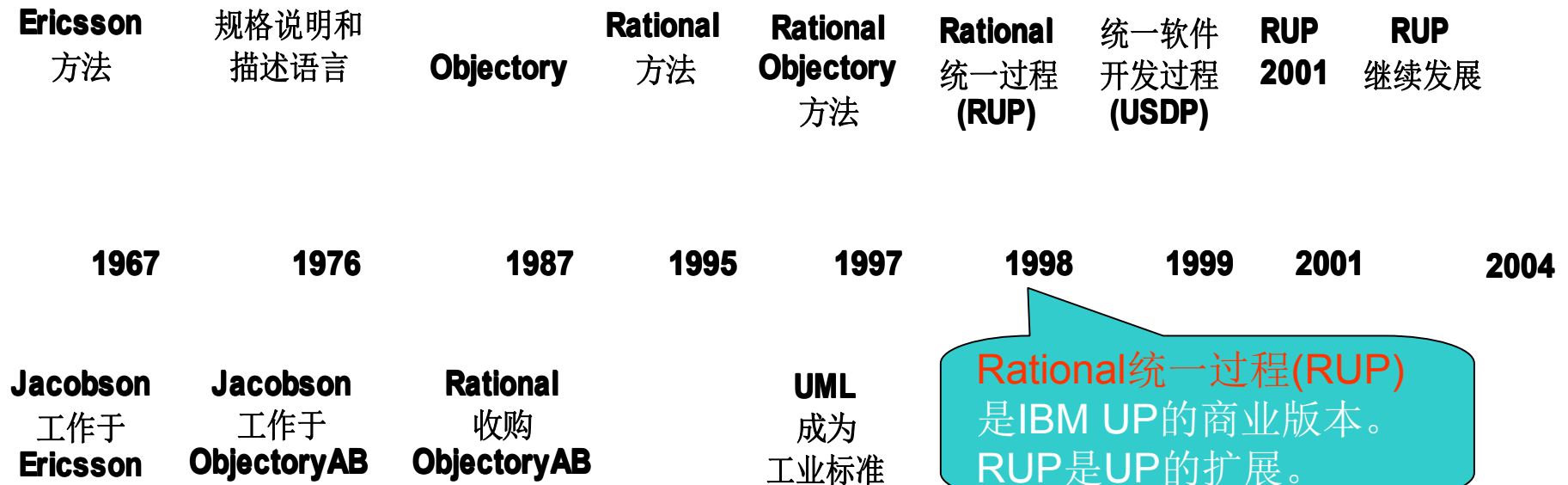
- Jim Rumbaugh提出了OMT方法
- Grady Booch提出了Booch方法
- Ivar Jacobson提出了Objectory（Object Factory）方法

## ■ UML

- 1994年Jim Rumbaugh和Grady Booch创建了统一方法（Unified Method），即UML第一个草案
- 三人加入Rational公司（后被IBM收购），领导了OMG的建模标准制定
- 1997年UML1.0发布，2004年UML2.0发布，成为OO建模标准

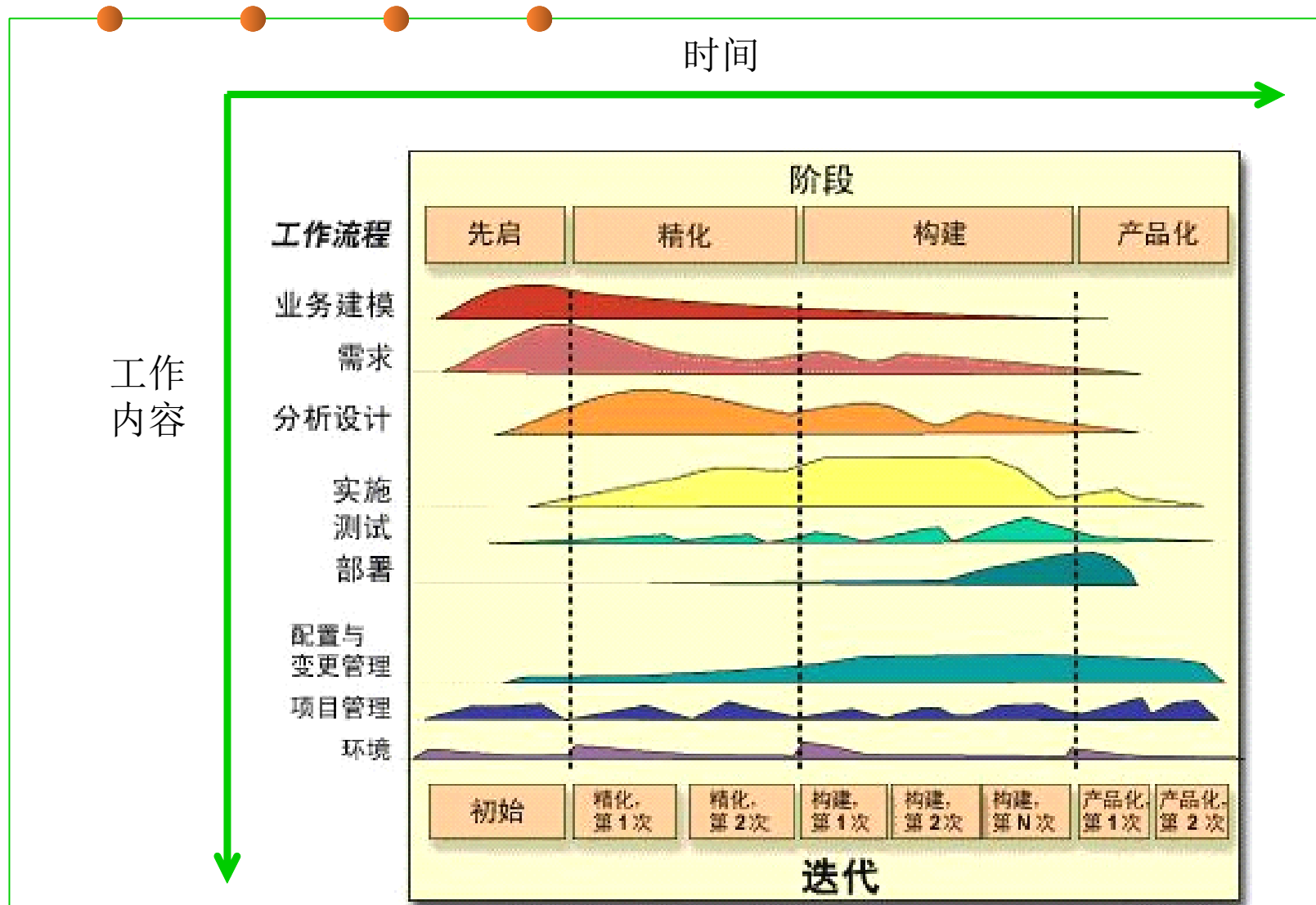
# 统一过程(Unified Process, UP)

统一软件过程(UP)是个源于UML作者的软件开发过程。



UP的历史

# 统一过程模型(Rational Unified Process)



# RUP的四个技术阶段

## ■ 初始(Inception)

- 大体上的构思、业务案例、确定范围和模糊评估

## ■ 细化(Elaboration)

- 已精化的构想、核心架构的迭代实现、高风险的解决、确定大多数需求和范围以及进行更为实际的评估

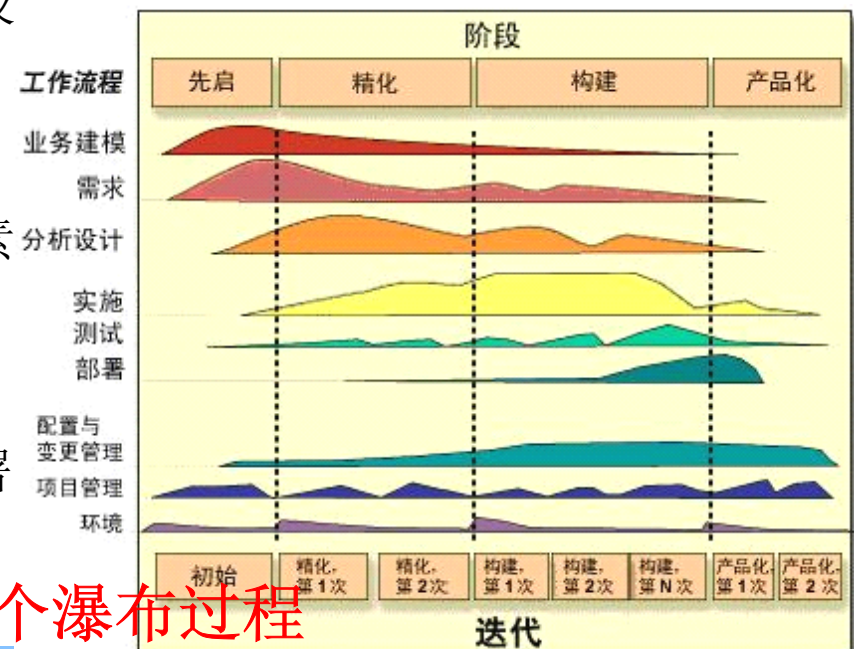
## ■ 构建(Construction)

- 对遗留下来的风险较低和比较简单的元素进行迭代实现，准备部署

## ■ 交付(Transition)

- 进行beta测试（用户测试、变更）和部署

需求  
设计  
开发  
测试



每个阶段多次迭代，一次迭代是一个瀑布过程



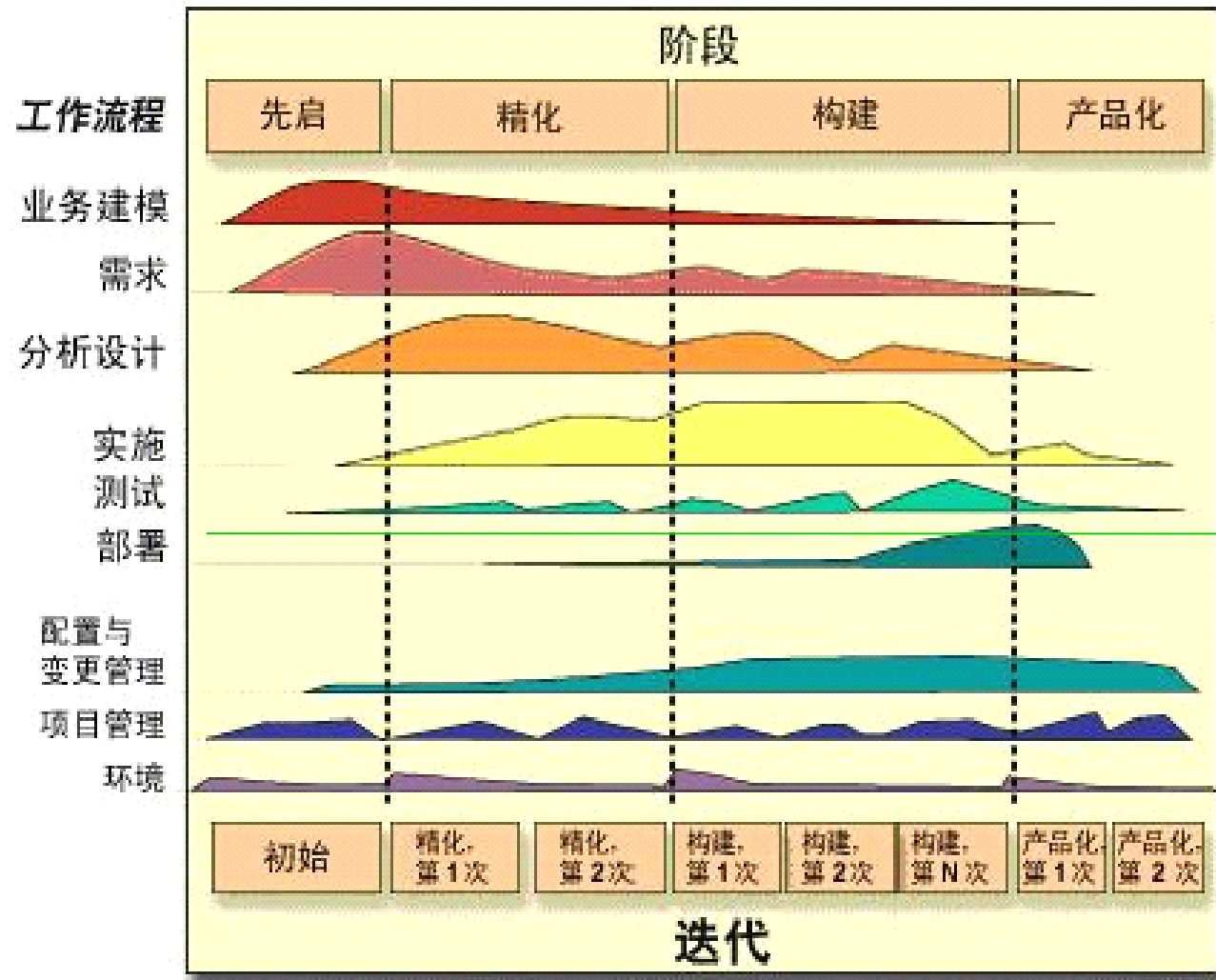
# RUP核心工作流程

## ■ 6个过程 workflows

- 业务建模
- 需求
- 分析设计
- 实现
- 测试
- 部署

## ■ 3个支持 workflows

- 配置和变更管理
- 项目管理
- 环境



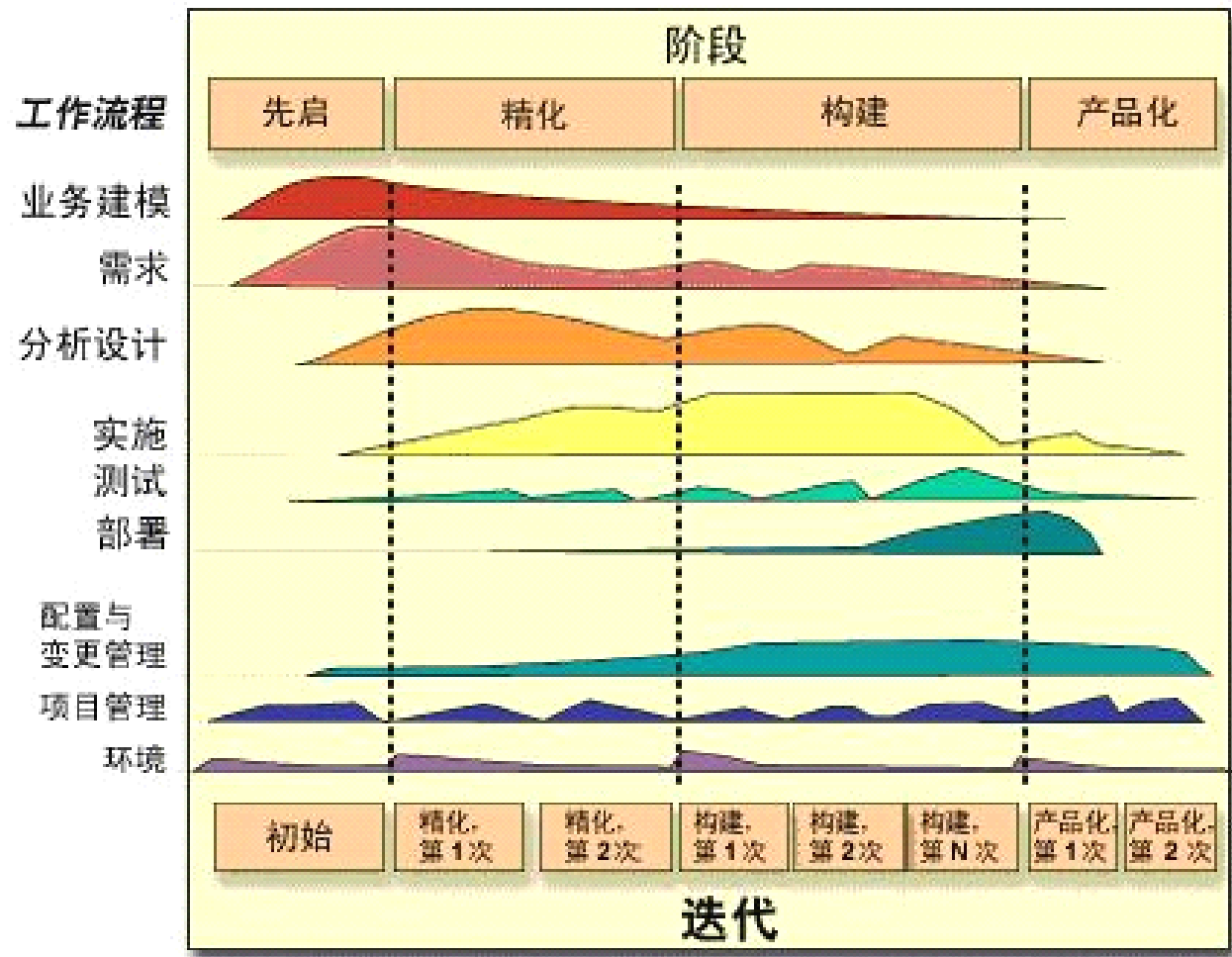
# RUP特征

## ■ 符合最佳实践

- 迭代开发
- 需求管理
- 构架和构件的使用
- 建模和UML
- 过程质量和产品质量
- 配置管理和变更管理

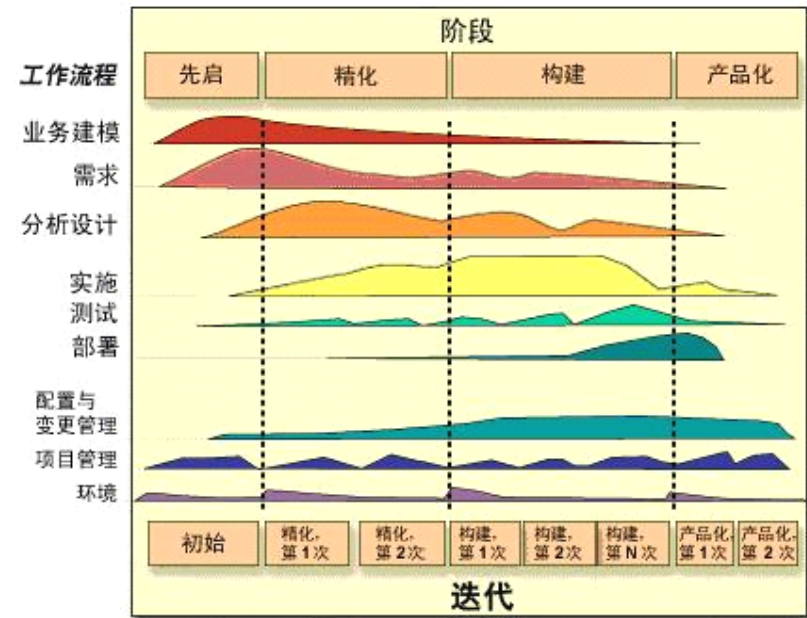
## ■ 其它特征

- 用例驱动的开发
- 过程配置
- 工具支持



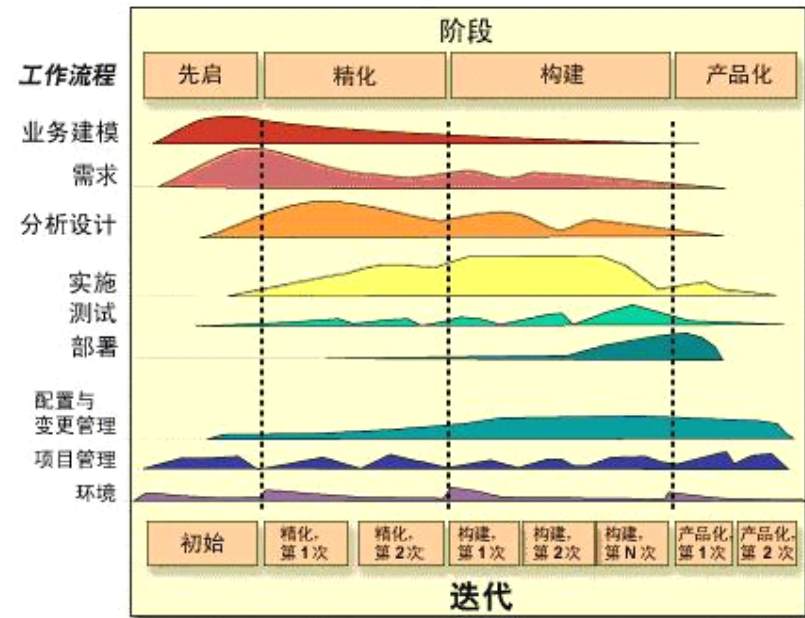
# 阶段1：初始

- 建立项目的软件规模和边界条件，包括运作前景、验收标准以及希望产品中包括和不包括的内容。
- 识别系统的关键用例（也就是系统重要设计的主要场景）。
- 对比一些主要场景，展示（也可能是演示）至少一个备选架构
- 评估整个项目的总体成本和进度（以及对即将进行的精化阶段进行更详细的评估）
- 评估潜在的风险（源于各种不可预测因素）
- 准备项目的支持环境。



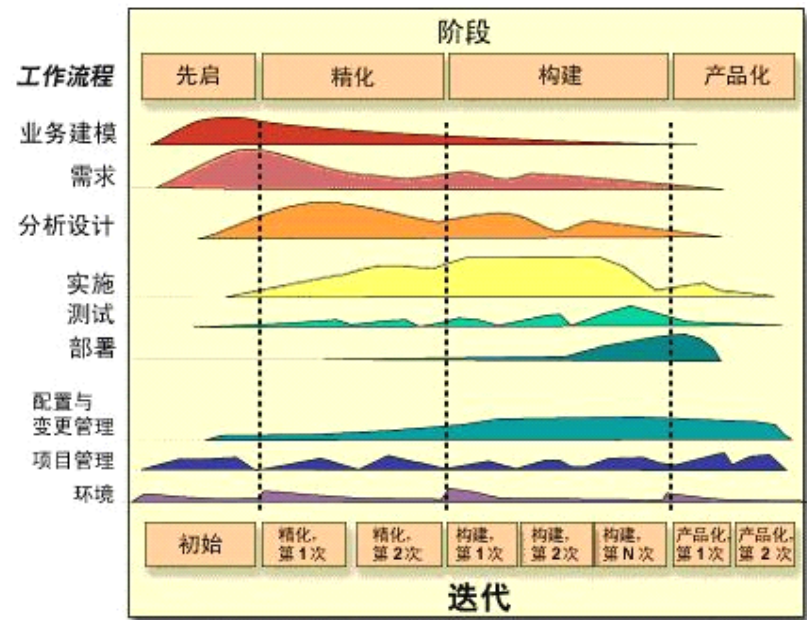
## 阶段2：细化

- 建立一个已确定基线的架构，它是通过处理架构方面重要的场景得到的，这些场景通常可以显示项目的最大技术风险。
- 确保架构、需求和计划足够稳定，充分减少风险，从而能够有预见性地确定完成开发所需的成本和进度。处理在架构方面具有重要意义的所有项目风险
- 制作产品质量构件的演进式原型，也可能同时制作一个或多个可放弃的探索性原型，以减小特定风险
- 证明已建立基线的构架将在适当时间、以合理的成本支持系统需求。
- 建立支持环境。



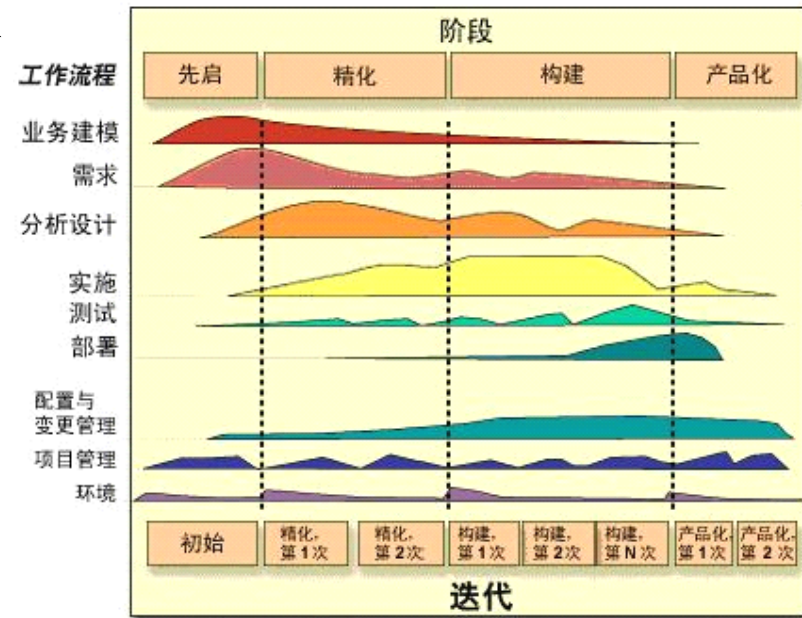
## 阶段3：构建

- 通过优化资源和避免不必要的报废和返工，使开发成本降到最低。
- 快速达到足够好的质量
- 快速完成有用的版本（**Alpha** 版、**Beta** 版和其他测试发布版）
- 完成所有所需功能的分析、开发和测试
- 迭代式、递增式地开发随时可以发布到用户群的完整产品。
- 确定软件、场地和用户是否已经为部署应用程序作好准备。
- 开发团队的工作实现某种程度的并行。



## 阶段4：交付

- 进行 **Beta** 测试，按用户的期望确认新系统
- **Beta** 测试和相对于正在替换的遗留系统的并行操作
- 转换操作数据库
- 培训用户和维护人员
- 调整活动，如进行调试、性能或可用性的增强
- 与客户一同进行软件验收





# RUP的适用范围

## ■ 大规模软件开发

### — 规模

- 大规模的软件研发（50人以上）
- 职责明确的小组划分或分布式团队
- 大项目特点：高度管理复杂性，高度技术复杂性

### — 项目持续时间

- 软件开发过程复杂，时间1年以上

### — 特点

- 规范、严格的软件开发过程
- 以文档为中心，因为各阶段的里程碑是文档提交物
- 重点在过程管理与提升

**RUP可裁剪适应小规模软件开发过程**

# RUP过程实践样例

## 开发某企业集团销售业务系统

科目	制品 迭代->	初始 I1	细化 E1...En	构建 C1...Cn	交付 T1...Tn
业务建模	领域模型		S		
需求	用例模型	S	R		
	设想	S	R		
	补充性规格说明	S	R		
	词汇表	S	R		
设计	设计模型		S	R	
	软件架构文档		S		
	数据模型		S	R	
实现	实现模型（代码）		S	R	R

S:开始时间 R:精化时间



## 第2章 软件过程与方法

- **2.1** 软件过程

- **2.2** 敏捷软件开发

  - 2.2.1 敏捷宣言

  - 2.2.2 极限编程和Scrum方法

- **2.3** 软件工程方法

- **2.4** 统一建模语言简介

## 惯例方法的问题？

- 为提高软件产品质量，不断增加过程管理控制
  - 时间↑ 成本↑ 质量？
- 庞大的重型的过程方法是否有效？
- 变化是软件的本质，软件开发必须适应变化
- 惯例模型忽视了软件开发中人的弱点，敏捷开发，重视人的作用，以人为本的软件开发方法
  - 人的弱点：不能一贯地、连续地做同一件事情

# 什么是敏捷？

- 什么是敏捷？
  - 有效地（快速、适应）响应变化
  - 参与者之间有效沟通
  - 使客户加入团队
  - 团队自我组织
- 目标：快速、增量地发布软件

## 敏捷假设：

- 提前预测哪些需求是稳定的和哪些需求会变化非常困难
- 对很多软件来说，设计和构建是交错进行的。通过构建验证之前很难估计应该设计到什么程度
- 从制定计划的角度来看，分析、设计、构建和测试并不像我们所设想的那么容易预测

# 敏捷宣言（2001）

- 我们正在通过实践和帮助其他人实践，揭示更好的开发软件的方法。我们的价值观是：

— 个体和交互	胜过	过程和工具
— 可以工作的软件	胜过	面面俱到的文档
— 客户合作	胜过	合同谈判
— 响应变化	胜过	遵循计划

- 在每对比对中，右侧很重要，但我们认为左侧更有价值。

## 敏捷12条原则

- 优先级最高的工作是通过尽早地、持续地交付有价值的软件来使客户满意
- 欢迎改变需求，即使已经到了开发后期。敏捷过程利用变化来为客户创造竞争优势
- 经常性地交付可以工作的软件，交付的间隔可以从几个星期到几个月，交付的时间间隔越短越好
- 整个项目开发期间，业务人员和开发人员必须天天都在一起工作
- 构建项目要靠积极性被激励起来的人员，要给他们提供所需的环境和支持，信任他们能够完成工作
- 在团队内部，最富有效果和效率的信息传递方法，就是面对面交谈

## 敏捷12条原则

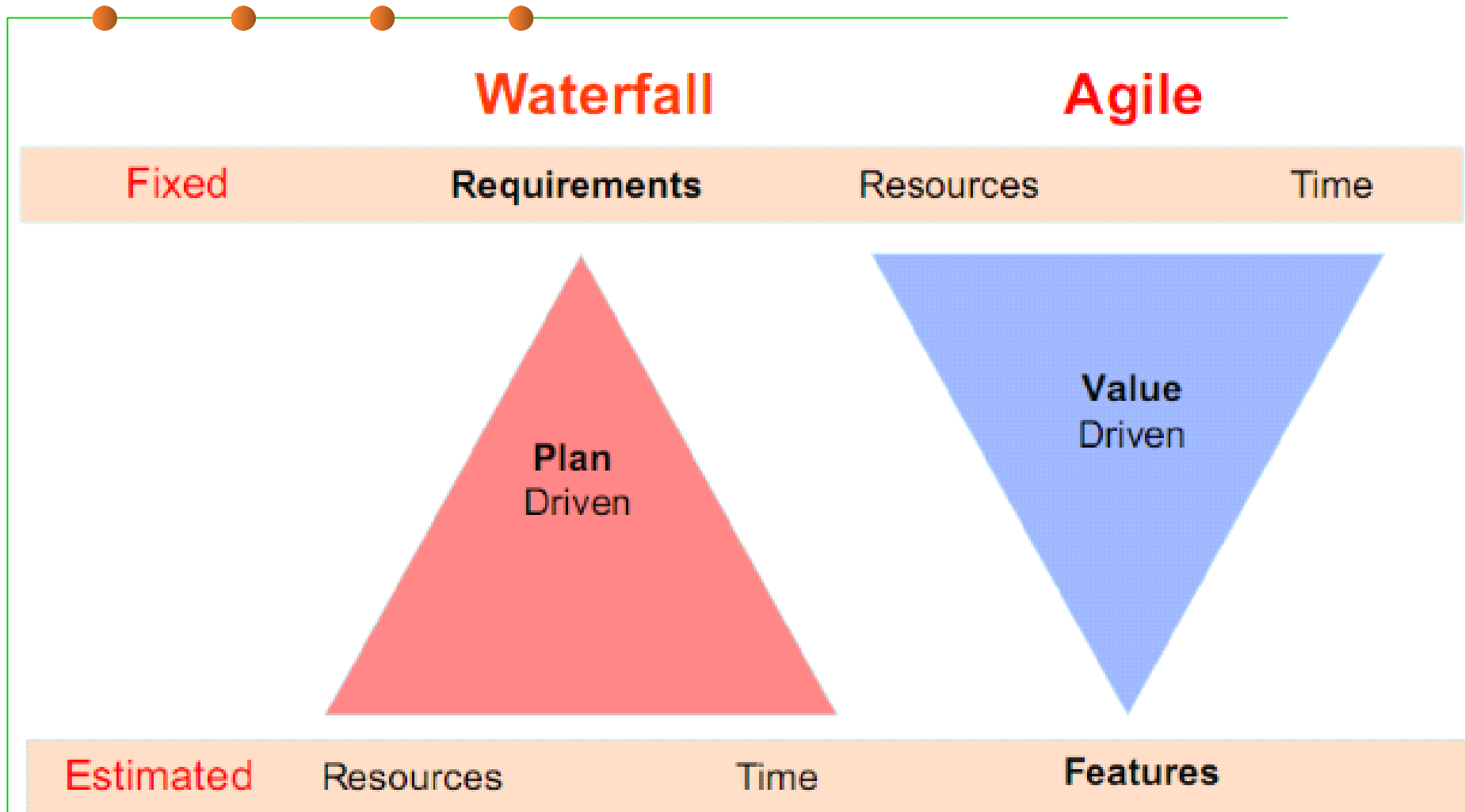
- 工作进度的主要衡量标准是能工作的软件
- 敏捷过程提倡可持续发展。出资方，开发团队和用户应能持续保持稳定的工作节拍
- 不断使技术保持领先和良好的设计会增强敏捷性
- 使要做的工作尽可能简单是根本
- 最好的架构、需求和设计出自自组织的团队
- 团队定期反思如何才能更有效地工作，然后相应地对自己的行为进行调整

# 敏捷对于人员及团队的要求

- 敏捷开发构造满足人员及团队需求的过程模型
- 对于敏捷团队及成员要求
  - 基本能力
  - 共同目标
  - 精诚合作
  - 决策能力
  - 模糊问题解决能力
  - 相互信任和尊重
  - 自我组织



# 敏捷开发与瀑布开发的区别



## 第2章 软件过程与方法

- **2.1** 软件过程

- **2.2** 敏捷软件开发

  - 2.2.1 敏捷宣言

  - 2.2.2 极限编程和Scrum方法

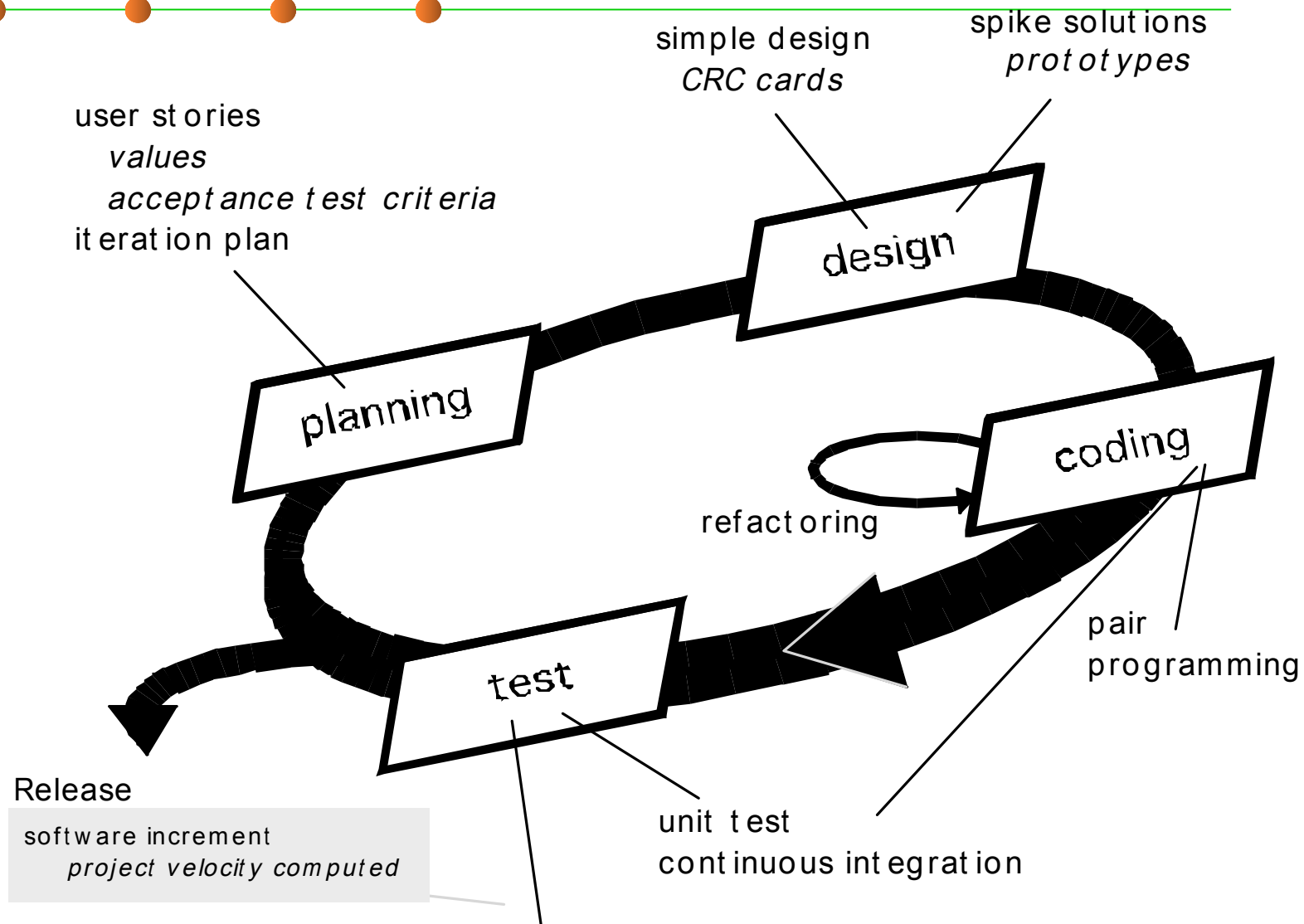
- **2.3** 软件工程方法

- **2.4** 统一建模语言简介

# 敏捷软件开发方法

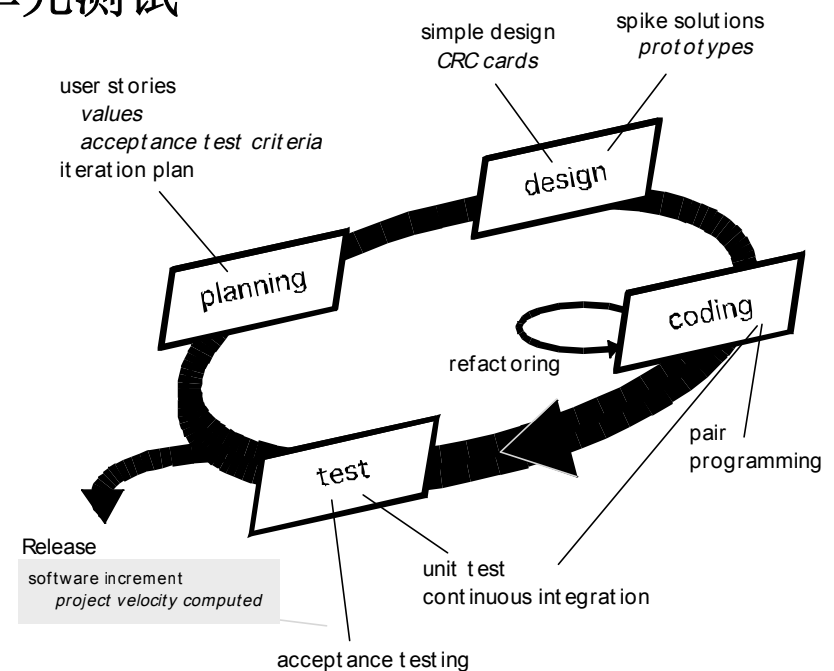
- 极限编程(**XP** , **eXtreme Programming**)
- **Scrum**

# 极限编程（XP）的过程



# 极限编程（XP）的过程

- 策划：用户故事、迭代计划
- 设计：保持简洁的原则，使用**CRC**卡设计，原型
- 编码：与设计同步进行，通过重构改进设计，结对编程方法
- 测试：测试驱动开发，先于编码建立单元测试



# 极限编程（XP, eXtreme Programming）

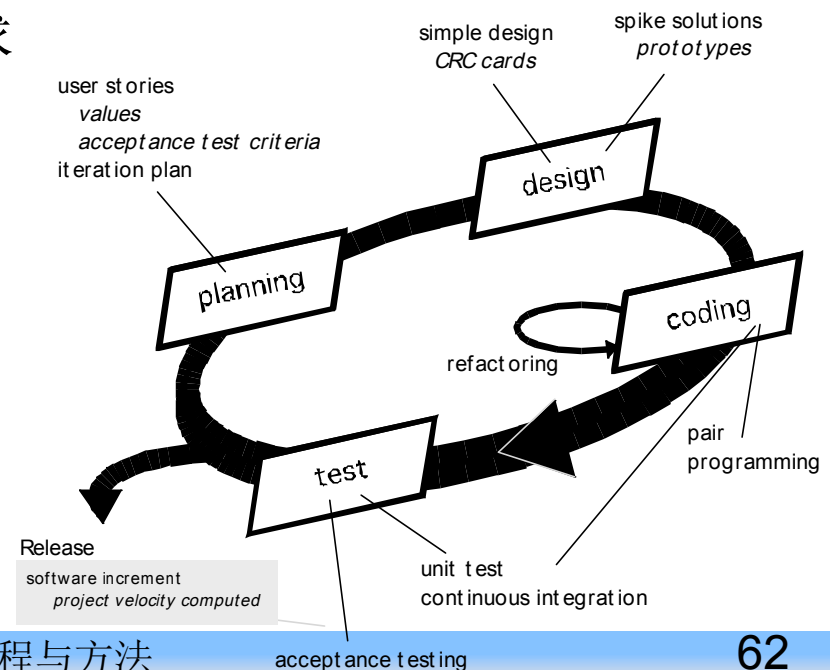
- 极限编程是一种轻量级的、灵巧的、简单的软件工程方法

- 与传统的开发过程不同，极限编程的核心活动体现在

需求→测试→编码→设计 过程中

- 适用于规模小、进度紧、需求变化大、质量要求严的项目

- 它希望以最高的效率和质量来解决用户目前的问题，以最大的灵活性和最小的代价来满足用户未来的需求



# 极限编程特点

## ■ 优点

- 重视客户的参与;
- 重视团队合作和沟通;
- 制定计划前做出合理预测;
- 让编程人员参与软件功能的管理;
- 重视质量;
- 简单设计;
- 高频率的重新设计和重构;
- 高频率及全面的测试;
- 递增开发;
- 连续的过程评估;
- 对过去的工作持续不断的检查

## ■ 缺点

- 以代码为中心, 忽略了设计;
- 缺乏设计文档, 局限于小规模项目;
- 对已完成工作的检查步骤缺乏清晰的结构;
- 质量保证依赖于测试;
- 缺乏质量规划;
- 没有提供数据的收集和使用的指导;
- 开发过程不详细;
- 全新的管理手法带来的认同度问题;
- 缺乏过渡时的必要支持。

# Scrum

- **Scrum**是一种迭代式增量软件开发过程框架，预先假定混乱的存在
  - 待定项：项目需求列表，具有优先级
  - 迭代冲刺开发：选择工作任务开发（1-4周）
    - 冲刺与Scrum例会：每天15分钟，交流
    - 演示：向客户交付软件增量
    - 回顾：总结本次迭代

产品订单  
Product Backlog

冲刺订单  
Sprint Backlog

高优先级

工作项  
分解

迭代  
每30天

新的功能  
增量

可运行的软件

冲刺规划会议  
Sprint Plan

一般不超过8小时。  
前4个小时：产品负责人向团队展示最高优先级的产品，团队则向他询问产品Backlog的内容、目的、含义及意义。  
后4小时：团队计划本次Sprint的安排。

冲刺复审会议  
Sprint Review

一般4个小时，由团队成员向产品负责人或其他利益相关人展示Sprint周期内的产品开发情况。

冲刺回顾会议  
Sprint Retrospective

一般3个小时，ScrumMaster将鼓励团队在SCRUM过程框架和实践范围内，对开发过程做出修改，使它在下一个Sprint周期中更加有效和令人愉快。



## 各种过程模型的比较

- 针对本次课程所讲授的软件过程模型，通过对比分析，找出这些过程模型之间的异同，并分析各自的优缺点；
- 分别用若干个简短的词来描述每一种软件过程模型的核心特征；
- 考虑以下维度：
  - 时间效率、成本、人力资源、开发质量、顾客满意度、需求扩展、需求变化、风险、与顾客交互程度、适用项目规模、适用 **deadline** 紧急程度、项目管理的方便程度、等等。

	商业系统	使命攸关的系统	性命攸关的嵌入式系统
典型应用	Internet站点 Intranet站点 游戏 管理信息系统（MIS） 企业资源计划（ERP）	嵌入式软件 游戏 Internet站点 盒装软件 软件工具	航空软件 嵌入式软件 医疗设备 操作系统 盒装软件
生命周期模型	敏捷开发、渐进原型	分阶段交付 渐进交付 螺旋型开发	分阶段交付 螺旋形开发 渐进交付
计划与管理	增量式项目计划 按需测试与QA计划 非正式的变更控制	基本的预先计划 基本的测试计划 按需QA计划 正式的变更控制	充分的预先计划 充分的测试计划 充分的QA计划 严格的变更控制
需求	非形式化的需求规格	半形式化的需求规格 按需的需求评审	形式化的需求规格 形式化的需求检查
设计	设计与编码是结合的	架构设计 非形式化的详细设计 按需的设计评审	架构设计 形式化的架构检查 形式化的详细设计 形式化的详细设计检查
构建	结对编程或独立编码 非正式的check-in手续或没有check-in手续	结对编程或独立编码 非正式的check-in手续 按需代码评审	结对编程或独立编码 正式的check-in手续 正式的代码检查
测试与QA	开发者测试自己的代码 测试先行开发 很少或没有测试（由单独的测试小组来做）	开发者测试自己的代码 测试先行开发 单独的测试小组	开发者测试自己的代码 测试先行开发 单独的测试小组 单独的QA小组
部署	非正式的部署过程	正式的部署过程	正式的部署过程

## 第2章 软件过程与方法

- **2.1** 软件过程
- **2.2** 敏捷软件开发
- **2.3** 软件工程方法
  - 2.3.1 结构化方法
  - 2.3.2 面向对象方法
- **2.4** 统一建模语言简介

# 软件工程方法

## ■ 结构化

- 复杂世界—>复杂处理过程（事情的发生发展）
- 设计一系列功能（或算法）以解决某一问题
- 寻找适当的方法存储数据

## ■ 面向对象

- 任何系统都是由能够完成一组相关任务的对象构成
- 如果对象依赖于一个不属于它负责的任务，那么就需要访问负责此任务的另一个对象（调用其他对象的方法）
- 一个对象不能直接操作另一个对象内部的数据，它也不能使其它对象直接访问自己的数据
- 所有的交流都必须通过方法调用

## 举例：五子棋游戏

- 面向过程（事件）的设计思路就是首先分析问题的步骤：

1. 开始游戏，初始化画面
2. 黑子走，绘制画面，
3. 判断输赢，如分出输赢，跳至步骤6
4. 白子走，绘制画面，
5. 判断输赢，如未分出输赢，返回步骤2，
6. 输出最后结果。

- 面向对象的设计思路是分析与问题有关的实体：

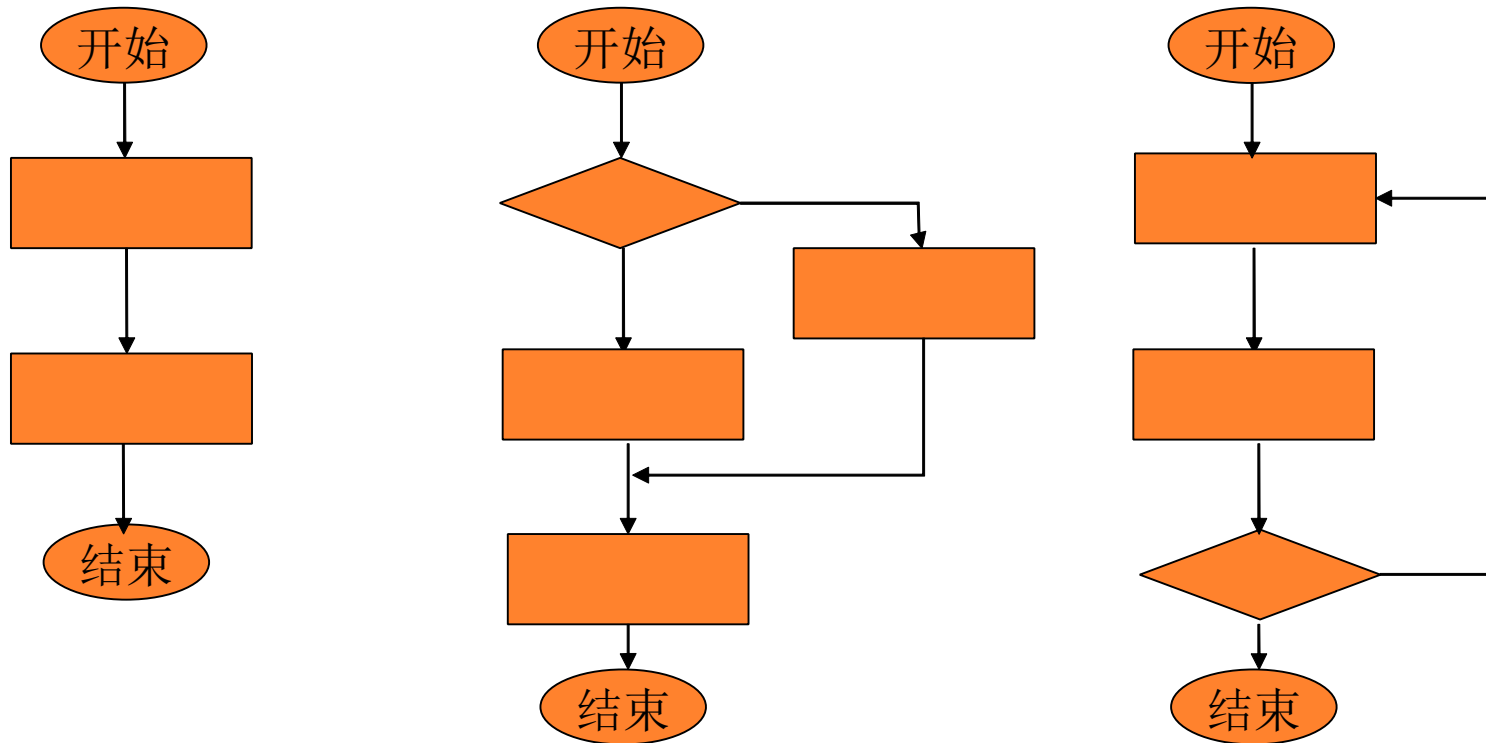
1. 玩家：黑白双方，这两方的行为是一模一样的，
2. 棋盘：负责绘制画面
3. 规则：负责判定诸如犯规、输赢等。

# 结构化方法

- 使用结构化编程、结构化和结构化设计技术的系统开发方法
  - 结构化编程
  - 结构化分析
  - 结构化设计
- 程序=数据结构+算法
- 以过程（事件）为中心的设计方法

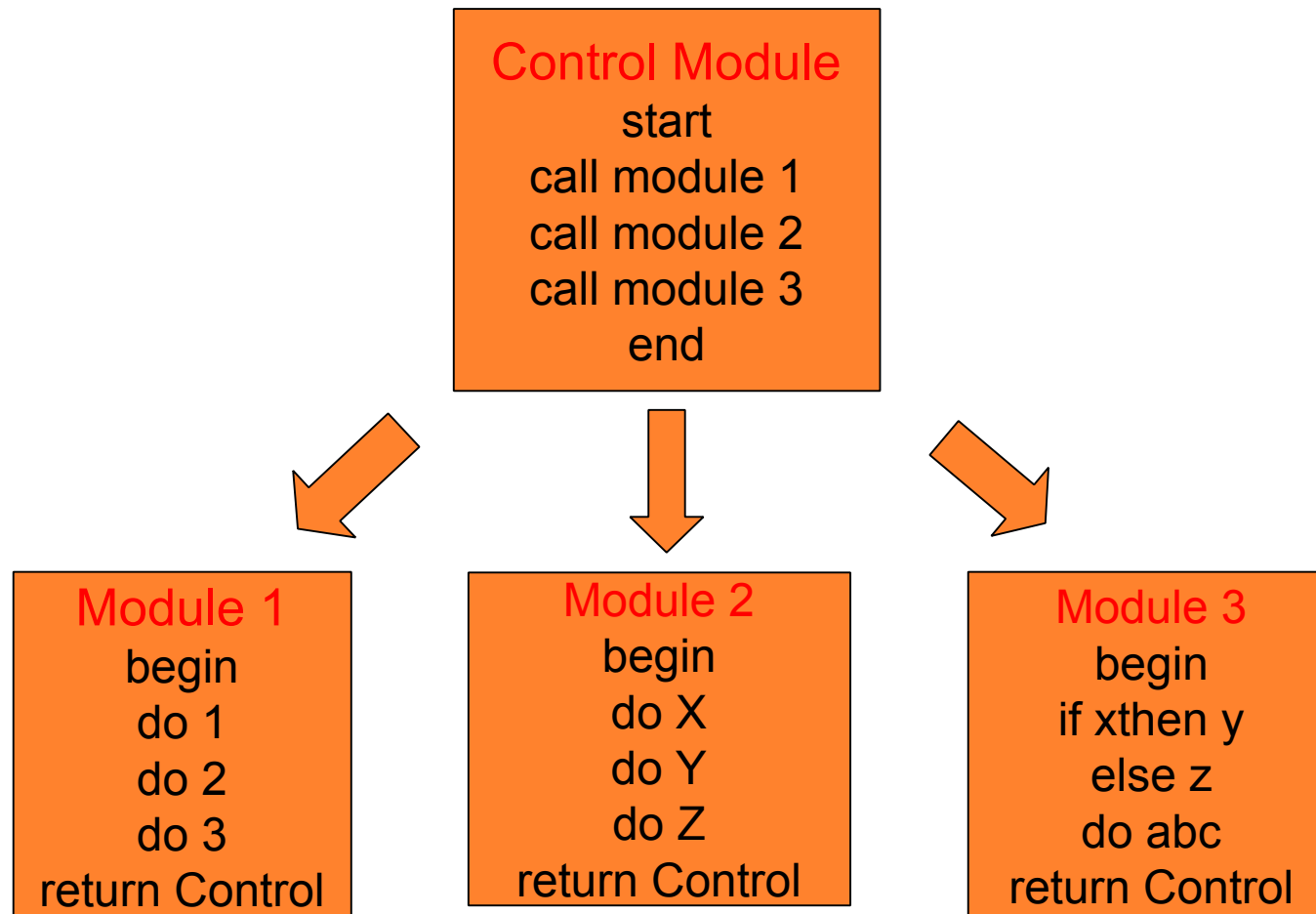
# 结构化编程

- 具有一个开始和一个结束的程序或程序模块，并且在程序执行中的每一步都由三个部分之一组成：顺序、选择或循环结构



# 结构化编程

- 自顶向下程序设计（逐步求精）：把更复杂的程序分解为程序模块的层次图





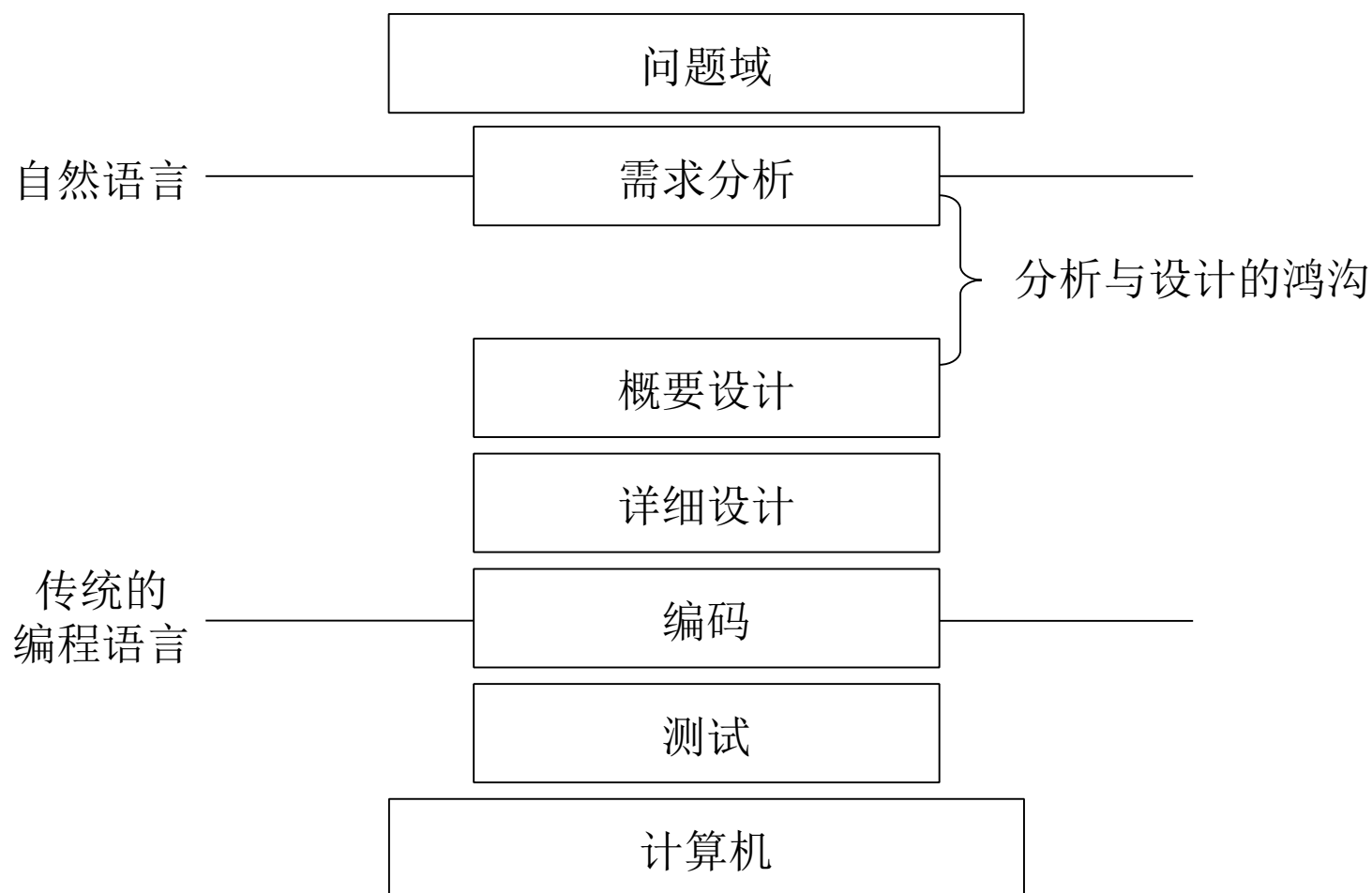
# 结构化分析

- 帮助开发人员定义系统需要做什么（处理需求），系统需要存储和使用哪些数据（数据需求），系统需要什么样的输入和输出以及如何把这些功能结合在一起来完成任任务。
  - 数据流图
  - 实体-联系图

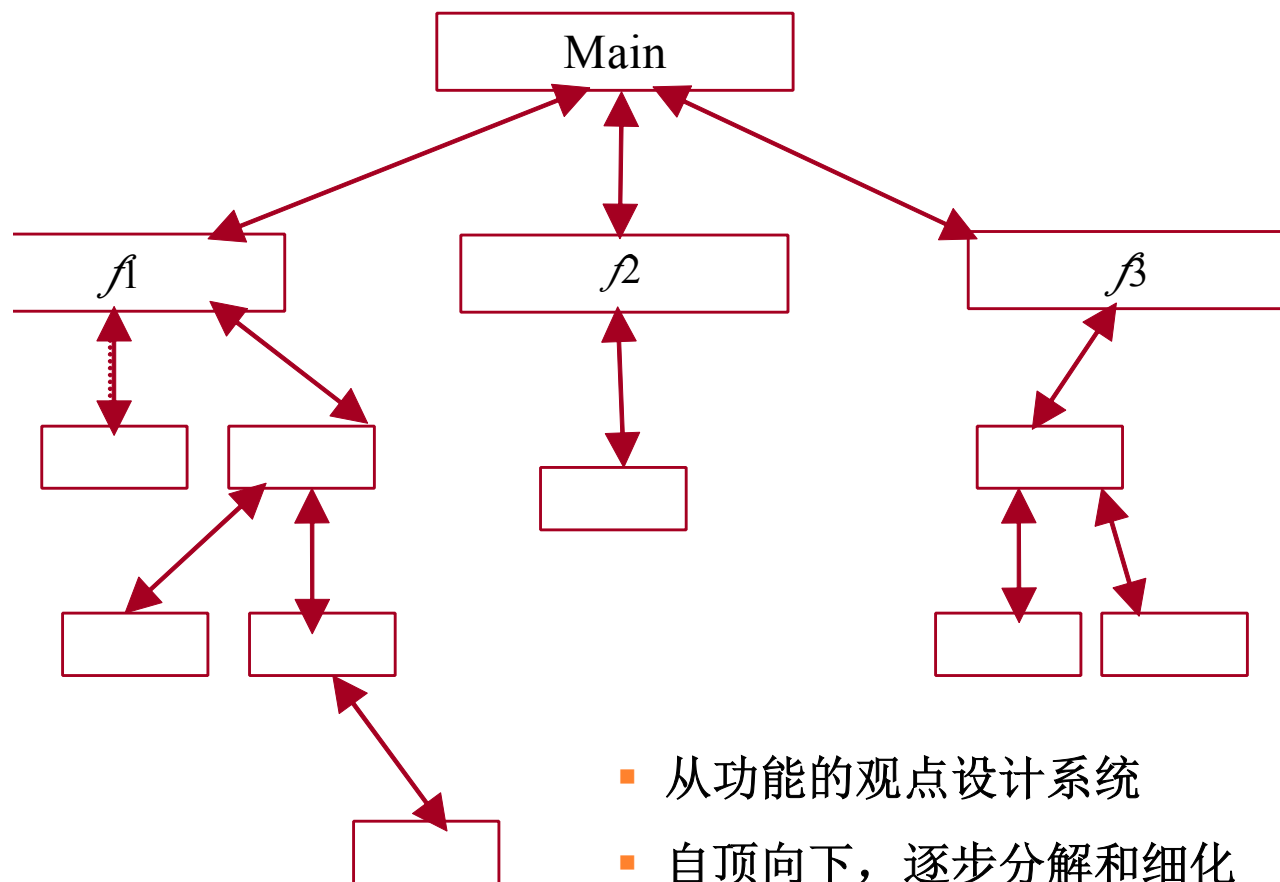
# 结构化设计

- 用来为确定下列事物提供指导，包括：程序是什么，每一个程序应该实现哪些功能以及如何把这些程序组织成一张层次图。
  - 结构图
  - 原则：低耦合、高内聚

# 传统软件工程方法：结构化方法



先看



- 从功能的观点设计系统
- 自顶向下，逐步分解和细化
- 将大系统分解为若干模块，主程序调用这些模块实现完整的系统功能

## 结构化程序开发的特点

- 把软件视为处理数据的流，并定义成由一系列步骤构成的算法
- 每一步骤都是带有预定输入和特定输出的一个过程；
- 把这些步骤串联在一起可产生合理的稳定的贯通于整个程序的控制流，最终产生一个简单的具有静态结构的体系结构。
- 数据抽象、数据结构根据算法步骤的要求开发，它贯穿于过程，提供过程所要求操作的信息；
- 系统的状态是一组全局变量，这组全局变量保存状态的值，把它们从一个过程传送到另一个过程。
- 结构化软件=算法+数据结构
- 结构化需求分析 = 结构化语言+DD+数据流图（DFD）

# 结构化方法的常见问题

## ■ 需求的错误

- 不完整、不一致、不明确
- 开发人员和用户无法以同样的方式说明需求
- 需求分析方法与设计方法不一致，分析的结果不能平滑过渡到设计

## ■ 需求的变化

- 需求在整个项目过程中始终发生变化
- 设计后期发生改变

## ■ 持续的变化

- 系统功能不断变化
- 许多变化出现在项目后期
- 维护过程中发生许多变化

## ■ 系统结构的崩溃

- 系统在不断的变化中最终变得不可用

## 造成上述问题的根本原因...

- 结构化方法以功能分解和数据流为核心，但是...  
系统功能和数据表示极有可能发生变化；
  - 以ATM银行系统为例：帐户的可选项、利率的不同计算方式、**ATM**的不同界面；
- 而软件设计应尽可能去描述那些极少发生变化的  
稳定要素：对象
  - 银行客户、帐户、**ATM**

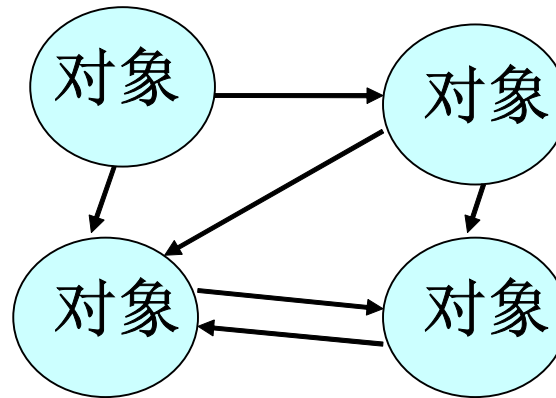
## 第2章 软件过程与方法

- **2.1** 软件过程
- **2.2** 敏捷软件开发
- **2.3** 软件工程方法
  - 2.3.1 结构化方法
  - 2.3.2 面向对象方法
- **2.4** 统一建模语言简介



## 再看面向对象的程序开发...

- 系统被看作对象的集合；
- 每个对象包含一组描述自身特性的数据以及作用在数据上的操作(功能集合)。



# 面向对象的程序开发

- 在结构化程序开发模式中优先考虑的是过程抽象，在面向对象开发模式中优先考虑的是实体(问题论域的对象)；
- 在面向对象开发模式中，把标识和模型化问题论域中的主要实体做为系统开发的起点，主要考虑对象的行为而不是必须执行的一系列动作；
  - 对象是数据抽象与过程抽象的综合；
  - 系统的状态保存在各个数据抽象所定义的数据存储中；
  - 控制流包含在各个数据抽象中的操作内；
  - 消息从一个对象传送到另一个对象；
  - 算法被分布到各种实体中。

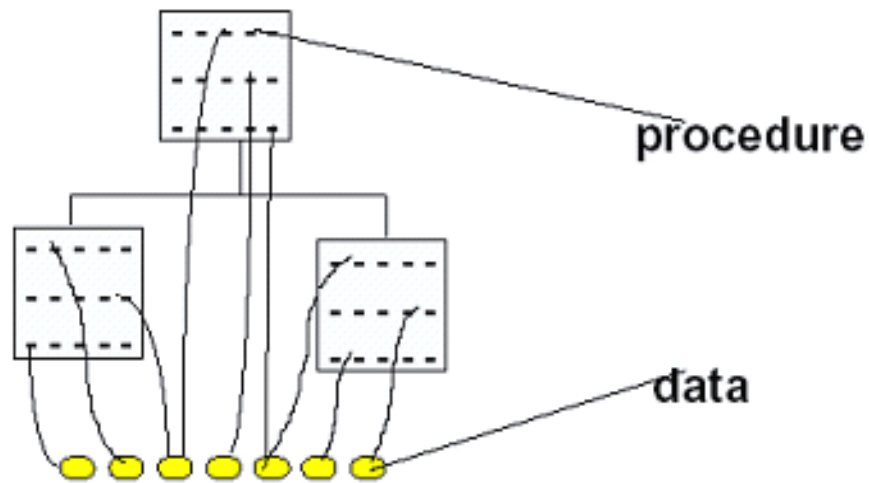
# 面向对象方法的优势

- 面向对象模型更接近于问题域(尽可能模拟人类习惯的思维方式)
  - 以问题域中的对象为基础建模
  - 以对象、属性和操作对问题进行建模
- 反复细化高层模型直到可以实现的程度
  - 努力避免在开发过程中出现大的概念跳变
- 将模型组织成对象的集合
  - 真实世界中的具体事物
    - 售货员、商品、仓库、顾客
    - 飞机、机场等
  - 逻辑概念
    - 商品目录、生产计划、销售
    - 操作系统中的分时策略、军事训练中的冲突解决规则等

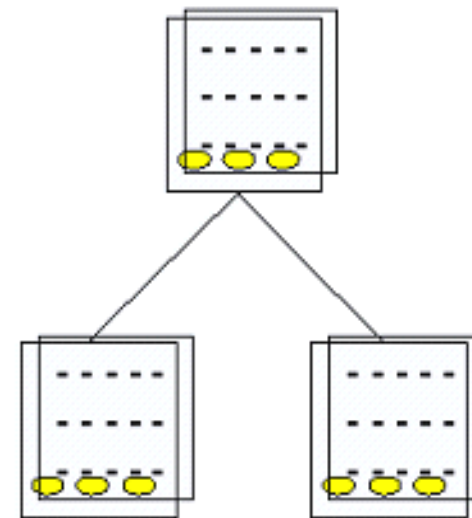
## 二者的本质区别

- 面向过程的结构化系统 = 功能 + 数据 (软件=算法+数据结构)
- 面向对象的系统 = 对象 + 消息 (对象=数据属性+操作)

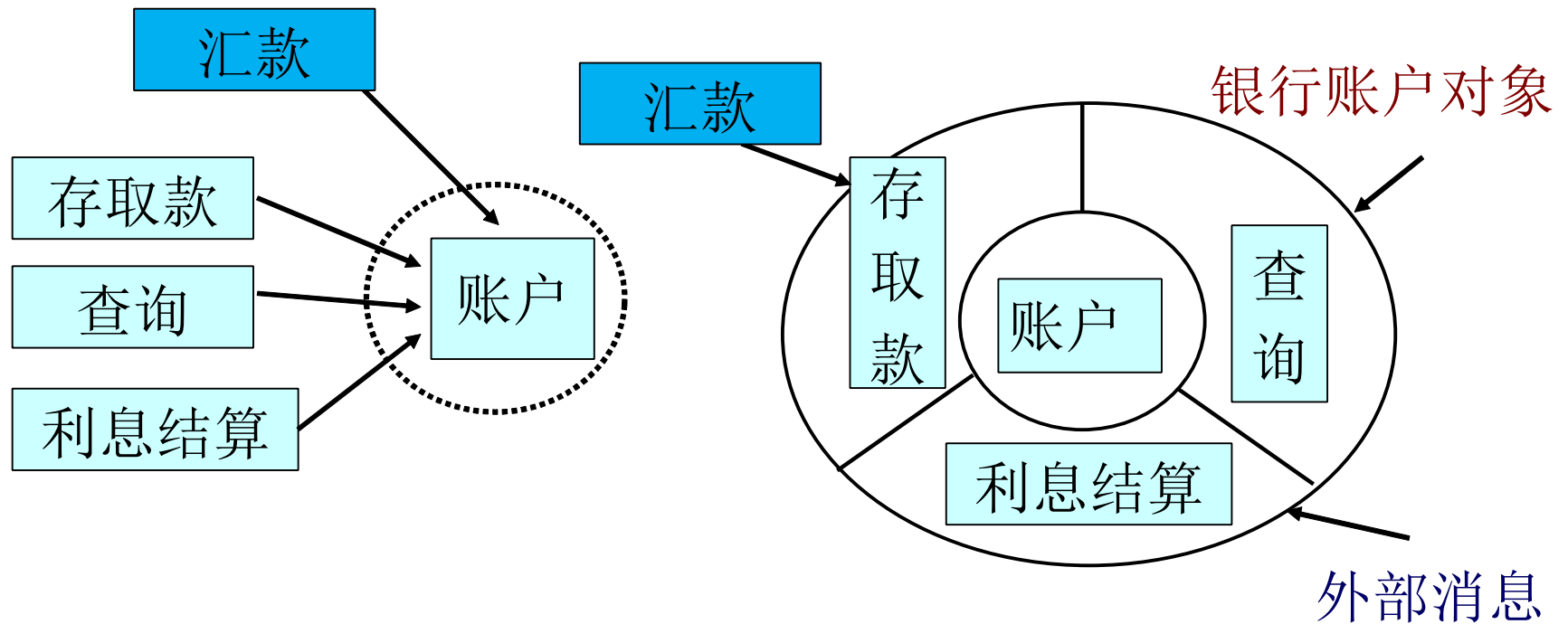
Traditional approach



Object Oriented approach



## 二者的本质区别



# 面向对象方法

- 把系统看做是一起工作来完成某项任务的相互作用的对象的集合
  - 面向对象分析
  - 面向对象设计
  - 面向对象编程
- 程序 = 对象 + 消息

# 面向对象方法

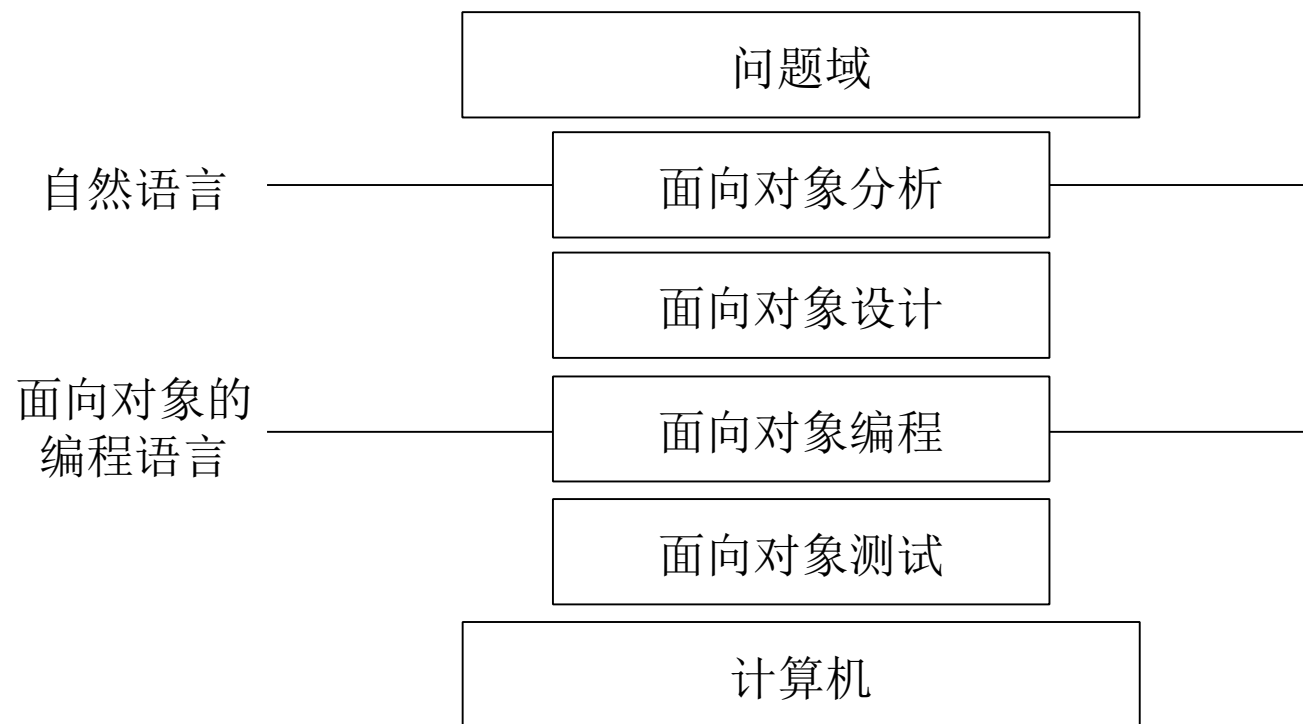
- **面向对象分析(Object Oriented Analysis, OOA)**
  - 分析和理解问题域，找出描述问题域和系统责任所需的类及对象，分析它们的内部构成和外部关系，建立OOA模型。
- **面向对象设计(Object Oriented Design, OOD)**
  - 将OOA模型细化，描述对象间交互，变成OOD模型，并且补充与一些实现有关的部分，如人机界面、数据存储、操作细节等。
- **面向对象编程(Object Oriented Programming, OOP)**
  - 用一种面向对象的编程语言将OOD模型中的各个成分编写成程序，由于从OOA→OOD→OOP实现了无缝连接和平滑过渡，因此提高了开发工作的效率和质量。

# OOA/D

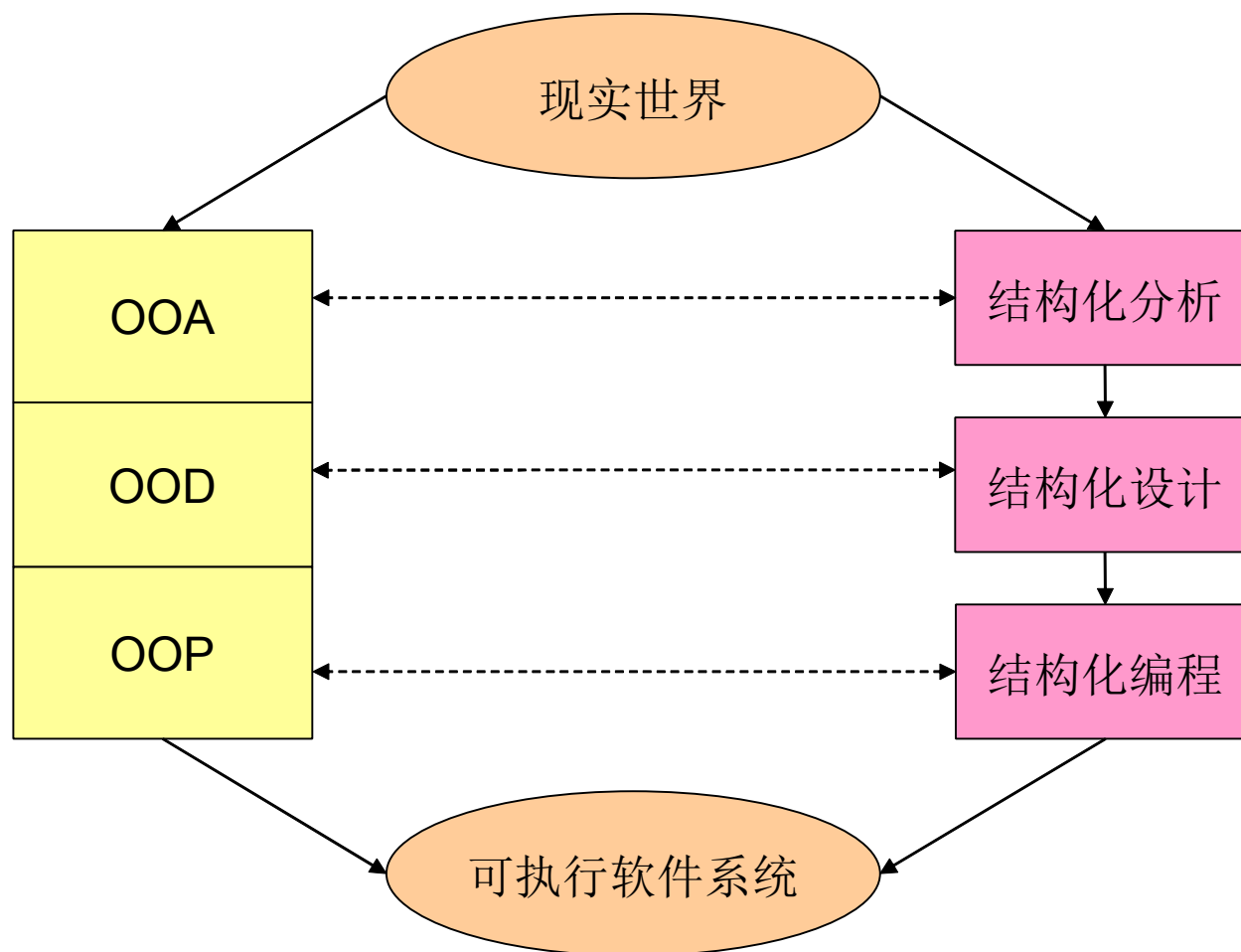
- 分析：强调的是对问题和需求的**调查研究**，而不是解决方案
  - 面向对象分析过程中，强调的是在问题领域内发现和描述对象（或概念）
- 设计：强调的是满足需求的概念上的**解决方案**（在软件方面和硬件方面），而不是其实现。
  - 面向对象设计过程中，强调的是定义软件对象以及它们如何协作以实现需求。
- 有价值的分析和设计可以概括为：**做正确的事**（分析）和**正确地做事**（设计）



# 面向对象方法



# 面向对象的软件工程



# 面向对象的基本概念

- 对象(**Object**)
- 类(**Class**)
- 消息(**Message**)
- 封装(**Encapsulation**)
- 继承(**Inheritance**)
- 多态(**Polymorphism**)

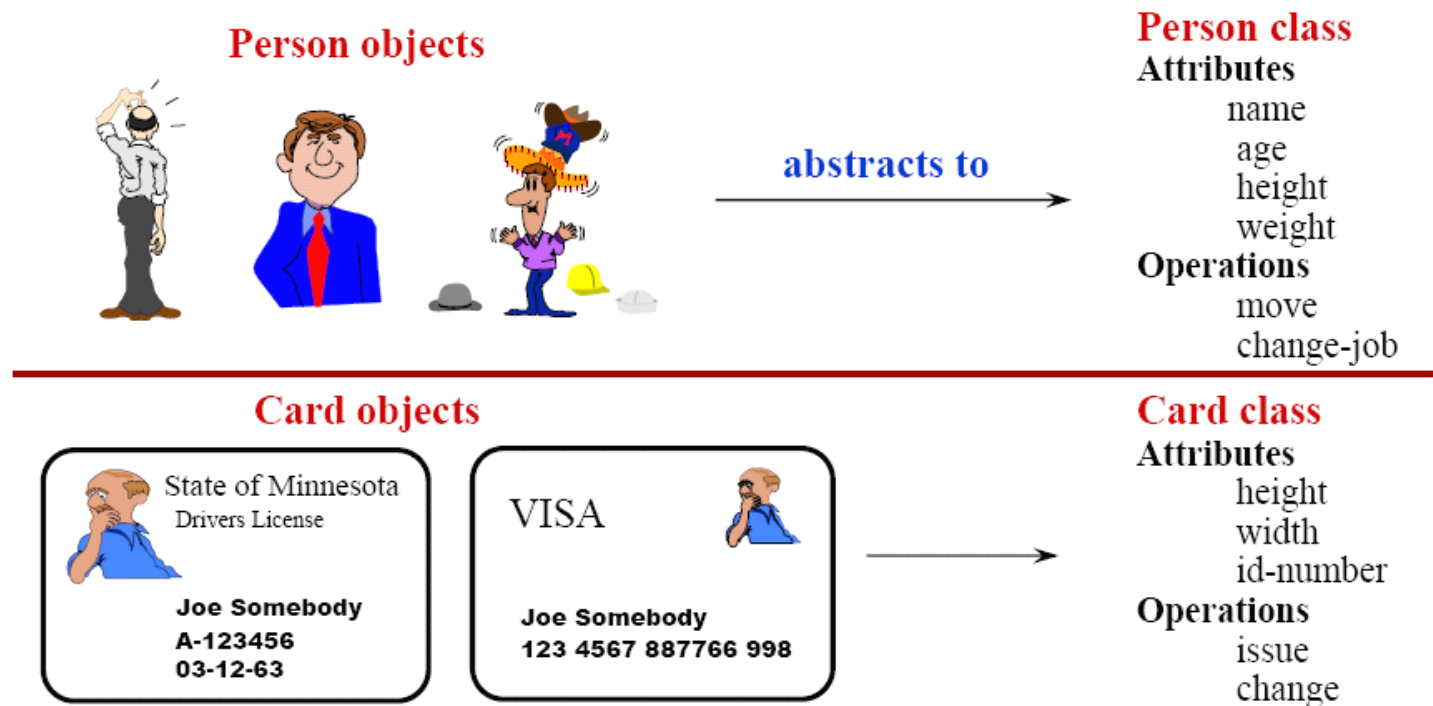
# 对象(Object)

- 对象(**Object**): 具有**责任**的实体。一个特殊的, 自成一体的容器, 对象的数据对于外部对象是受保护的。
- 特性: **标识符** (区别其他对象)、**属性** (状态) 和**操作** (行为)。



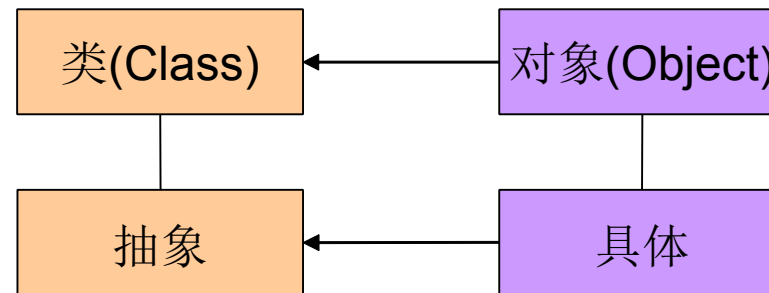
# 属性和操作

- **属性(Attribute):** 与对象关联的数据, 描述对象**静态**特性;
- **操作(Operation):** 与对象关联的程序, 描述对象**动态**特性;



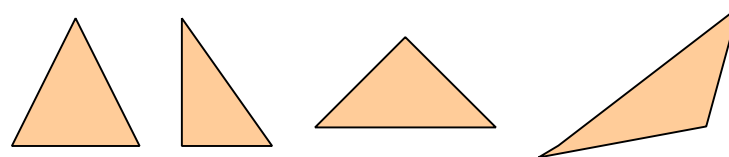
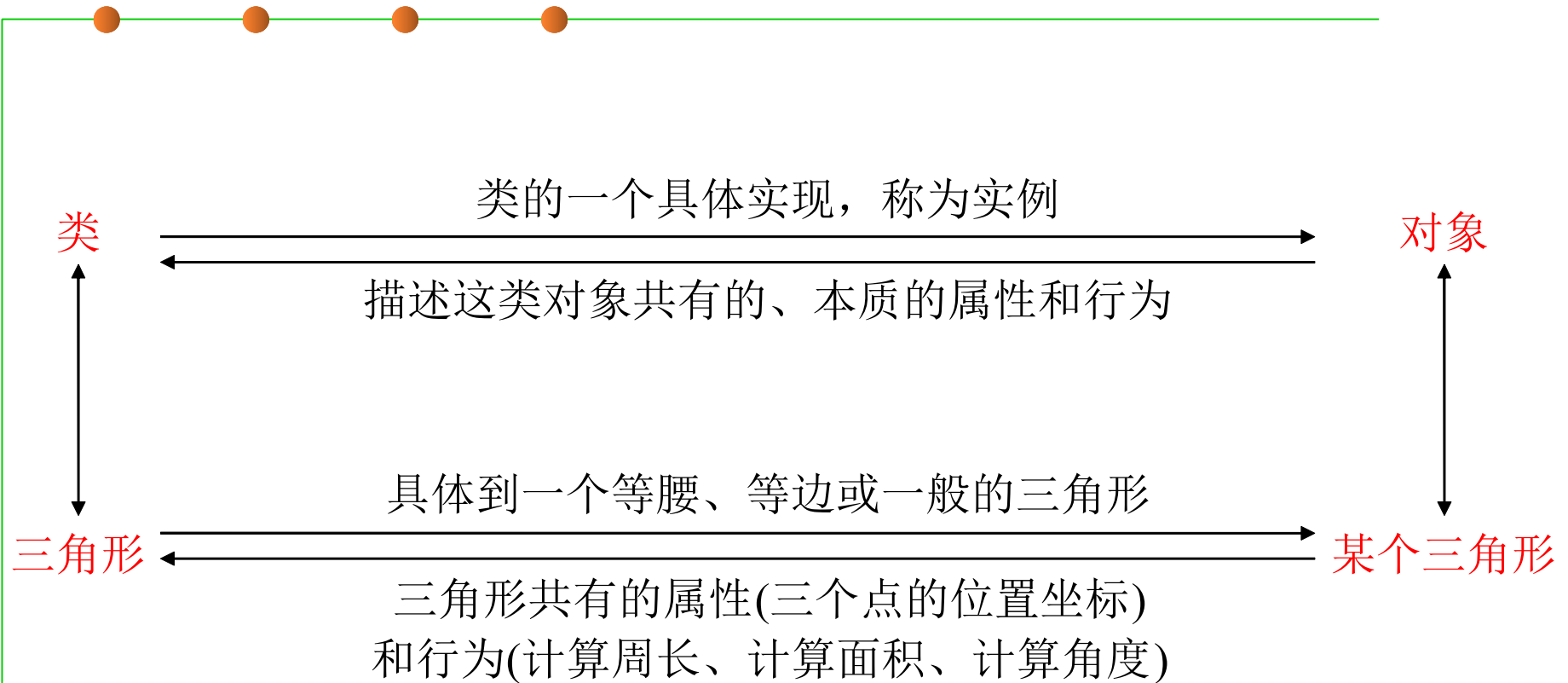
# 类(Class)

- **类(Class):** 具有相同属性和操作的一组**对象的抽象**，它为属于该类的全部对象提供了统一的抽象描述。



- “类”所代表的是一个抽象的概念或事物，现实世界中并不存在，而“对象”是客观存在的。
- [例] “学生”是一个类，计算机学院**1080310501**号学生则是“学生”类的一个实例，是一个具体的“对象”。

# 类与对象的对比



# 类与对象的对比

## ■ 类与对象的比较

- “同类对象具有相同的属性和操作”是指它们的定义形式相同，而不是说每个对象的属性值都相同。
  - 类是静态的，类的存在、语义和关系在程序执行前就已经定义好了。
  - 对象是动态的，对象在程序执行时可以被创建和删除。
- 
- 在面向对象的系统分析和设计中，并不需要逐个对对象进行说明，而是着重描述代表一批对象共性的类。



# 类与对象的对比

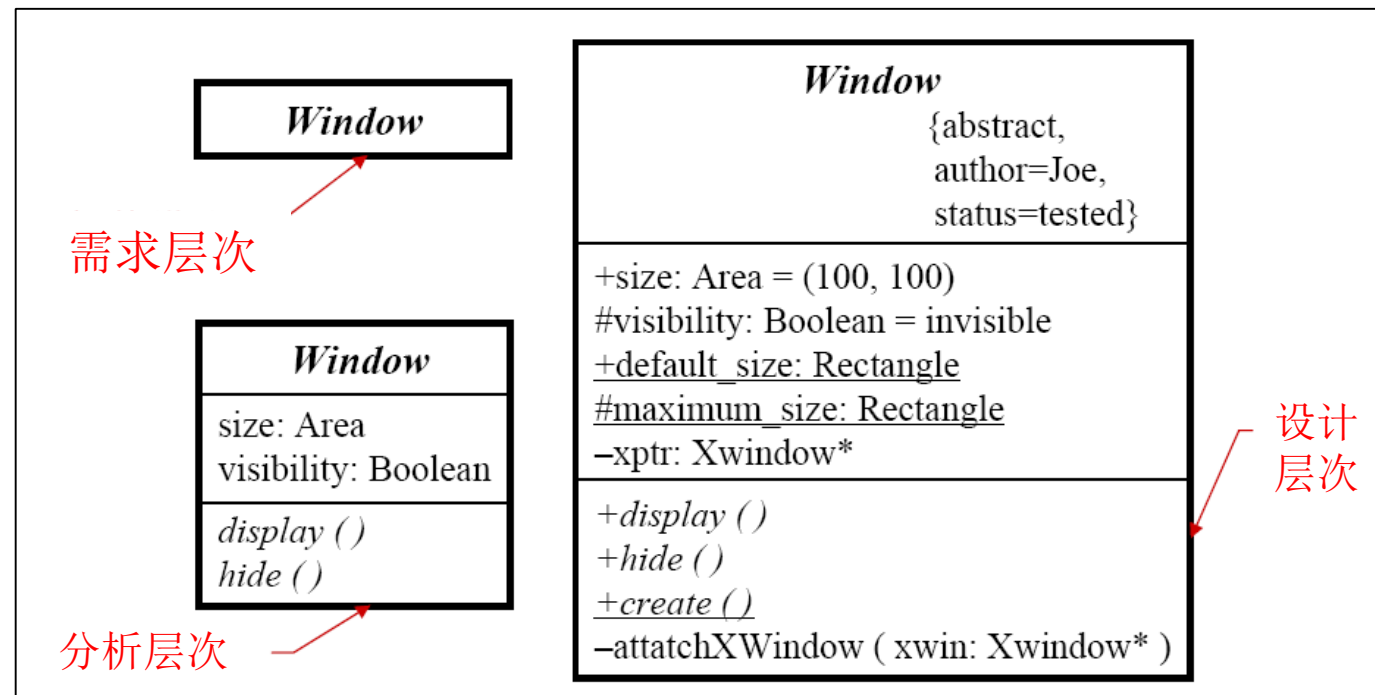
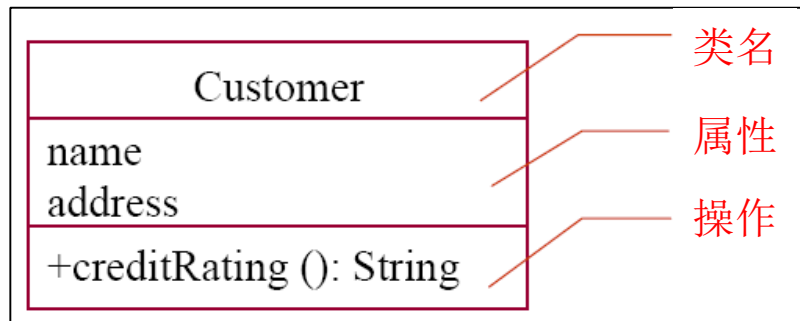
定义一个学生类:

```
Class Student {  
    String sno;  
    String sname;  
    String dept;  
  
    public Student (String sno, string sname,  
                    string dept) { };  
  
    public boolean RegisterMyself() { };  
    public boolean SelectCourses() { };  
    private float QueryScore(int courseID) { };  
}
```

使用这个类:

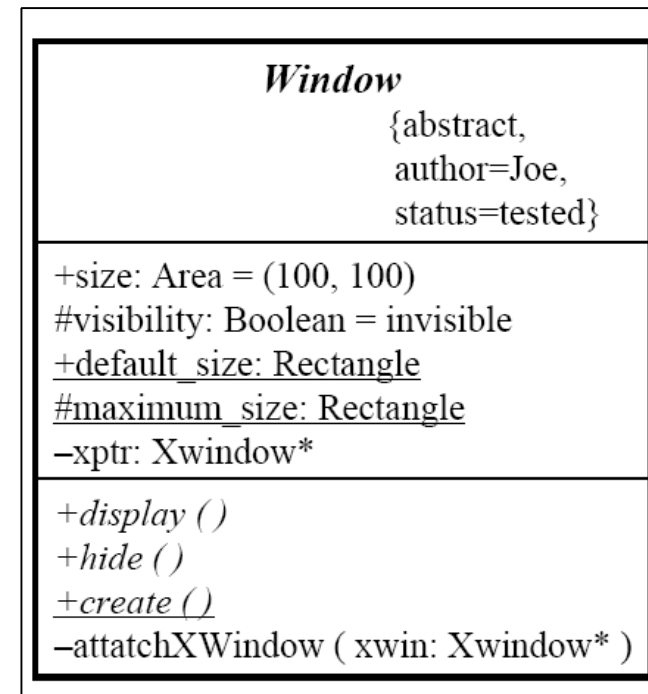
```
Student 张三 = new Student  
    ("1080310501", "张三", "CS" );  
  
Student 李四 = new Student  
    ("1080310502", "李四", "CS" );  
  
Student 王五 = new Student  
    ("1080310503", "王五", "CS" );  
  
If ( 张三.RegisterMyself() == true) {  
    张三.SelectCourses();  
}  
  
float score = 张三.QueryScore (32);
```

# 类(Class)的三种抽象层次



# 类的属性(Attribute)

- 类的属性：描述对象“静态”(结构)特征的一个数据项；
- 属性的“可见性”(Visibility)分类：
  - 公有属性(public) +
  - 私有属性(private) -
  - 保护属性(protected) #
- 属性的表达方式：
  - 可见性 属性名：数据类型=初始值

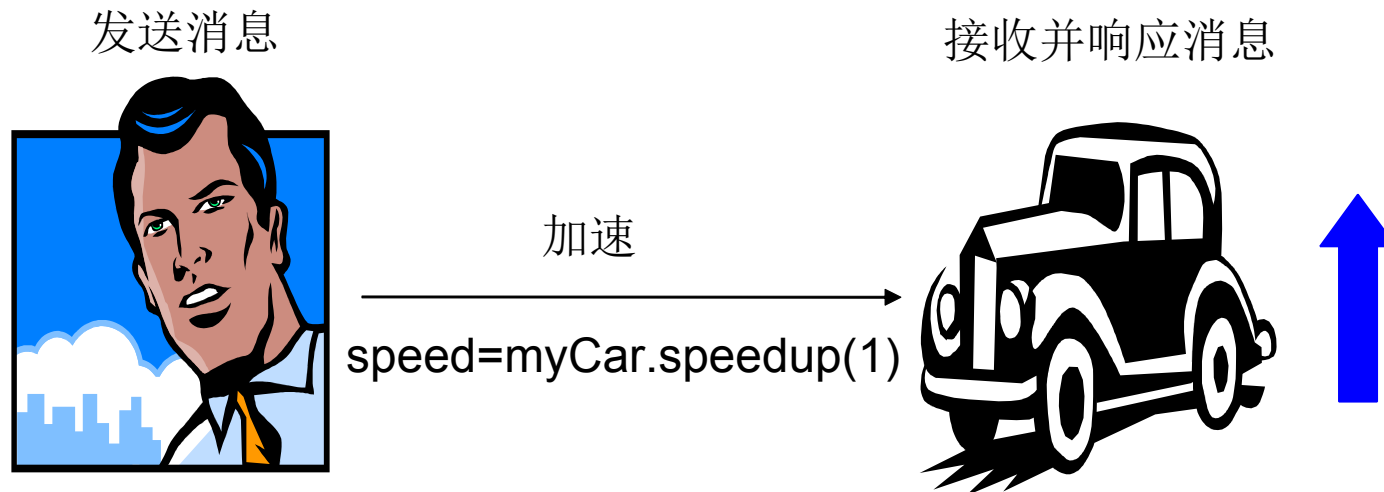


# 类的操作/方法(Operation/Method)

- 类的操作/方法：描述对象“动态”(行为)的特征的一个函数或过程；
- 方法的“可见性”(Visibility)分类：
  - 公有属性(public) +
  - 私有属性(private) -
  - 保护属性(protected) #
- 方法的表达方式：
  - 可见性 方法名(参数列表)：返回值数据类型
  - 例如：+ getNextSentence (int i) : string

# 消息(Message)

- **消息(Message):** 一个对象向其他对象发出的请求，一般包含消息接收对象、接收对象所采用的方法、方法需要的参数、返回信息等；
  - 一个对象向另一个对象发出消息请求某项服务；
  - 另一个对象接收该消息，触发某些操作，并将结果返回给发出消息的对象；
  - 对象之间通过消息通信彼此关联在一起，形成完整的系统。



# 封装(Encapsulation)

- **封装(Encapsulation):** 把对象的**属性和操作**结合成一个独立的单元，并尽可能对外界**隐藏数据**的实现过程；
- 一个对象不能直接操作另一个对象内部数据，它也不能使其他对象直接访问自己的数据，所有的交流必须通过方法调用。
  - getXXX()
  - setXXX()
- 例如：对“汽车”对象来说，“司机”对象只能通过方向盘和仪表来操作“汽车”，而“汽车”的内部实现机制则被隐藏起来。

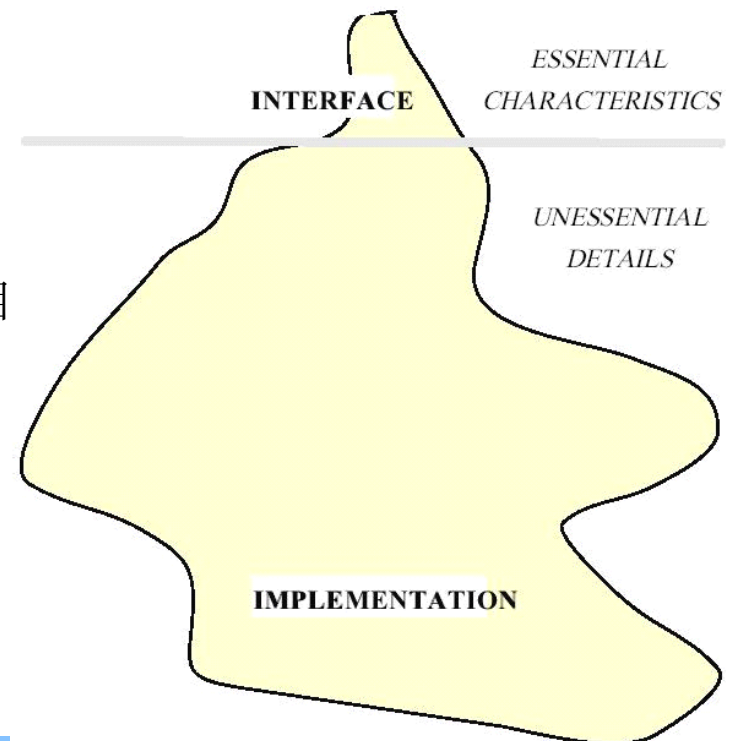


# “封装”的作用

- 使对象形成两个部分：接口(可见)和实现(不可见)，将对象所声明的功能(行为)与内部实现(细节)分离

——信息隐藏(Information Hiding)

- “封装”的作用是什么？
  - 保护对象，避免用户误用；
  - 保护客户端，其实现过程的变化不会影响到相应客户端的改变。



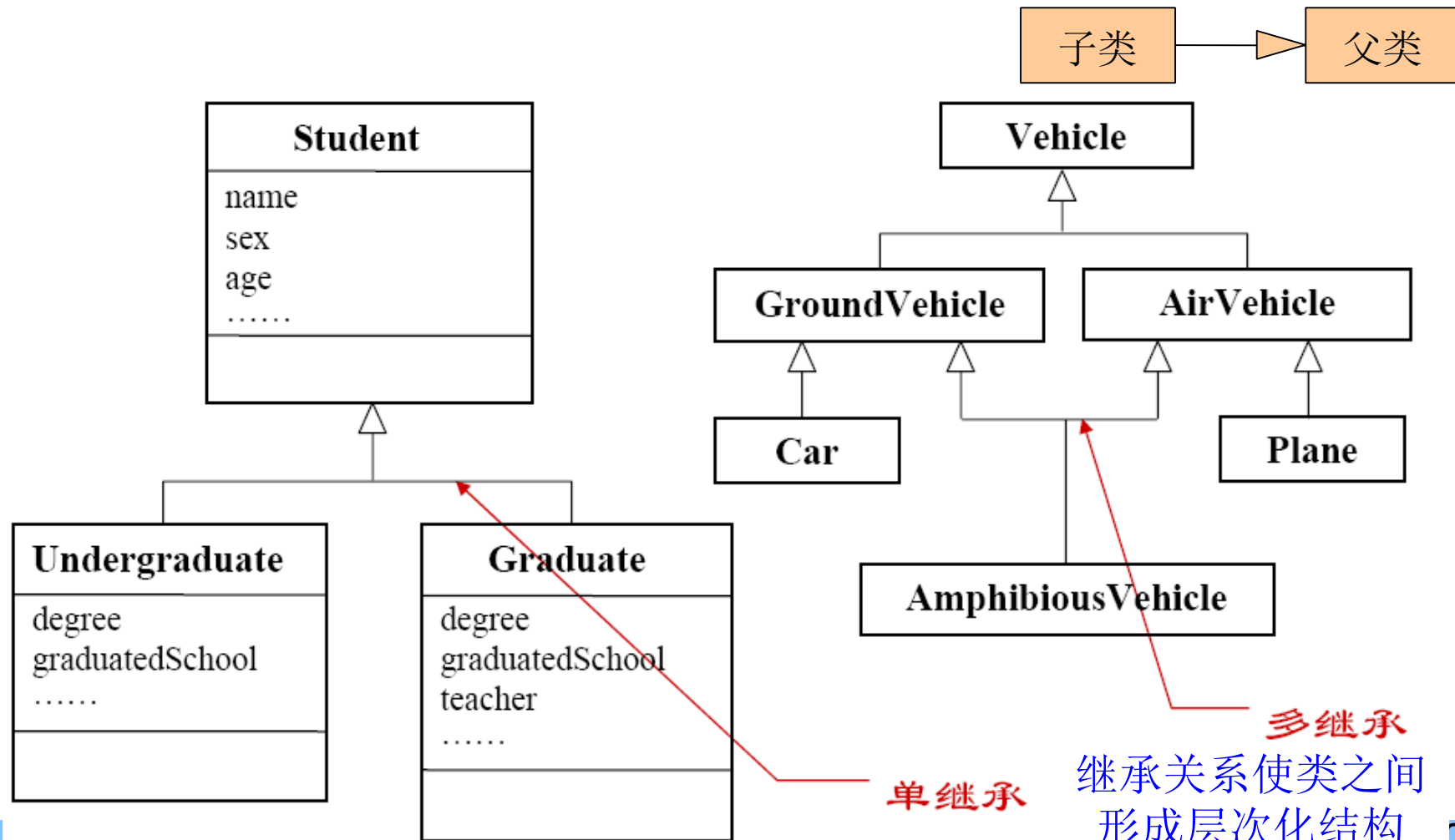
# 泛化/继承

- **泛化**关系是类元的一般描述和具体描述之间的关系，具体描述建立在一般描述的基础之上，并对其进行了扩展。
- 泛化的用途：
  - 在共享祖先所定义的成分的前提下允许它自身定义增加的描述，这被称作**继承**。当一个变量（如参数或过程变量）被声明承载某个给定类的值时，可使用类（或其他元素）的实例作为值，这被称作**可替代性原则**。
  - 泛化使得**多态**操作成为可能，即操作的实现是由它们所使用的对象的类，而不是由调用者确定的。



# 继承(Inheritance)

- **继承(Inheritance):** 子类可自动拥有父类的全部属性和操作。



## 定义的父亲类型可用子类型替换

```
Vehicle[] vehicles=new Vehicle[3];  
vehicles[0]=new Car();  
vehicles[1]=new Plane();  
vehicles[2]=new Train();  
  
for(int i=0;i<vehicles.length;i++){  
    vehicles[i].start();  
    vehicles[i].ring();  
}
```

# 继承(Inheritance)

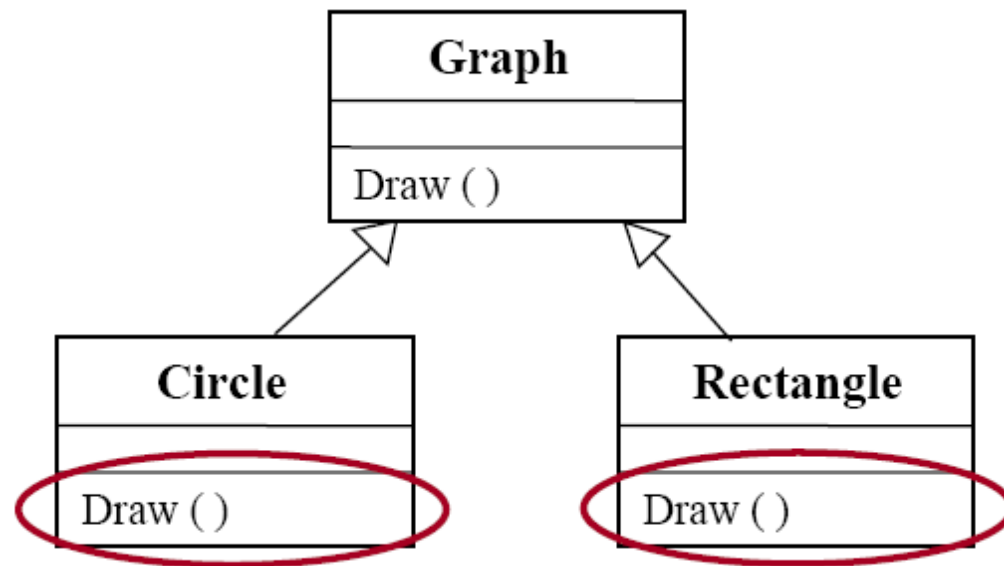
- **单一继承**：一个子类只有唯一的一个父类
- **多重继承**：一个子类有一个以上的父类

```
class Graduate extends Student{  
  
    //methods and fields  
  
}
```

- **抽象类**：把一些类组织起来，提供一些公共的行为，但不能使用这个类的实例(即从该类中派生出具体的对象)，而仅仅使用其子类的实例。称不能建立实例的类为抽象类。
  - 抽象类中至少有一个方法被定义为“abstract”类型的。

# 多态(Polymorphism)

- **多态性(Polymorphism):** 在父类中定义的属性或服务被子类继承后，可以具有不同的数据类型或表现出不同的行为。



# 多态(Polymorphism)

- **多态性**：同一个操作作用于不同的对象上可以有不同的解释，并产生不同的执行结果。
- 多态性是面向对象程序设计语言的基本机制，是将一个操作与不同方法关联的能力
- **静态绑定**：传统程序设计语言的过程调用与目标代码的连接(即调用哪个过程)放在程序运行前进行
- **动态绑定**：把这种连接推迟到运行时才进行

## 接口（Interface）

- 接口与类类似，但是只为其成员提供规约而不提供实现。它与只含有抽象方法的抽象类很相似。
- 以接口为中心的设计方法

```
public interface Comparable{  
    int compareTo(Object other);  
}
```

```
class Student implements Comparable{  
    public int compareTo(Object otherobject){  
        Student other=(Student)otherobject;  
        if (age<>other.age) return -1;  
        if (age==other.age) return 1;  
        return 0;  
    }  
}
```

## 小结：面向对象技术

### ■ 面向对象技术(Object Oriented Technology)

- 客观世界是由对象组成的，任何客观事物或实体都是对象；复杂对象可以由简单对象构成；
- 具有相同数据和相同操作的对象可以归并为一个统一的“类”，对象是类的实例；
- 类可以派生出子类，子类继承父类的全部特性(数据和操作)，同时加入了自己的新特性；子类和父类形成层次结构；
- 对象之间通过消息传递相互关联；
- 类具有封装性，其数据和操作对外是不可见的，外界只能通过消息请求某些操作。
- 具体的计算则是通过新对象的建立和对象之间的通信来执行的。

## 第2章 软件过程与方法

- **2.1** 软件过程
- **2.2** 系统工程
- **2.3** 软件工程方法
- **2.4** 统一建模语言简介
  - 2.4.1 建模与UML
  - 2.3.2 UML的图
  - 2.3.3 用UML建模



# 建模是一种设计技术

- 模型是某个事物的抽象，其目的是在构建这个事物之前先理解它
  - 在构建物理实体之前先验证
  - 通过抽象降低复杂度
  - 可视化，便于与客户和其他小组成员交流
  - 为维护 and 升级提供文档
- 建模的主要目的是为理解，而非文档
- 建模的过程
  - 分析建模
  - 设计建模
  - 实现建模
  - 部署建模

# 建模的三个不同视角

- **类模型：** 表示系统静态的、结构化的“数据”层面
  - 描述系统中对象的结构：它们的标识、属性和操作
  - 描述与其他对象的关系
- **状态模型：** 表示对象时序的、行为的“控制”层面
  - 标记变化的事件
  - 界定事件上下文的状态
  - 一个类有一个状态图
- **交互模型：** 表示独立对象的协作“交互”层面
  - 系统行为如何完成
  - 对象间如何协作

# 什么是UML?

- 统一建模语言（**UML**）是描述、构造和文档化系统制品的可视化语言（**OMG03a**）。
  - 由Booch、Rumbaugh和Jacobson合作创建
  - 统一了主流的面向对象的分析设计的表示方法
- 面向对象建模的图形化表示法的标准
  - UML不是可视化程序设计语言，而是一个可视化的建模语言
  - UML不是OOA/D，也不是方法，它只是图形表示工具

## 应用UML的三种方式

- **UML作为草图**：非正式、不完整的，用于探讨问题和交流
- **UML作为蓝图**：相对详细的设计图，用于代码生成和逆向工程
- **UML作为编程语言**：用**UML**完成系统可执行规格说明，模型驱动体系结构（**MDA**）的应用方式，尚在发展阶段

## 第2章 软件过程与方法

- **2.1** 软件过程
- **2.2** 系统工程
- **2.3** 软件工程方法
- **2.4** 统一建模语言简介
  - 2.4.1 建模与UML
  - 2.4.2 UML的图**
  - 2.4.3 用UML建模

# UML视图和图

	视图	图	主要概念
结构分类	静态视图	类图	类、关联、泛化、依赖关系、实现、接口
	用例视图	用例图	用例、参与者、关联、扩展、包括、用例泛化
	物理视图	构件图	构件、接口、依赖关系、实现
		部署图	节点、构件、依赖关系、位置
动态行为	状态机视图	状态图	状态、事件、转换、动作、
	活动视图	活动图	状态、活动、完成转换、分叉、结合
	交互视图	顺序图	交互、对象、消息、激活
		协作图	协作、交互、协作角色、消息

# UML视图

- **结构分类：描述了系统中的结构成员及其相互关系**
  - **静态视图**对应用领域中的概念以及与系统实现有关的内部概念建模
  - **用例视图**是外部用户所能观察到的系统功能的模型图
  - **物理视图**对应用自身的实现结构建模。物理视图有两种：实现视图和部署视图
    - 实现视图为系统的构件模型及构件之间的依赖关系。
    - 部署视图描述位于节点实例上的运行构件实例的安排。
- **动态行为：描述了系统随时间变化的行为**
  - **状态机视图**是一个类对象所可能经历的所有历程的模型图。
  - **活动视图**是状态机的一个变体，用来描述执行算法或工作流程中涉及的活动
  - **交互视图**描述了执行系统功能的各个角色之间相互传递消息的顺序关系。

# UML图

- **类图**：描述系统的各个对象类型以及存在的各种静态关系
- **用例图**：描述用户与系统如何交互
- 构件图：构件间的组织结构及链接关系
- 部署图：制品在节点上的部署
- 状态图：事件在对象的周期内如何改变状态
- **活动图**：过程及并行行为
- **顺序图**：对象间的交互；强调顺序
- 协作图：对象间交互，重点在对象间链接关系



# UML图的适用场合

## ■ 需求分析

- **用例图**：建立应用场景，如何使用系统
- 从概念视角绘制的**类图**：用于领域分析
- **活动图**：明确组织机构的工作流程，软件如何与人交互
- **状态图**：描述概念的状态改变及事件

## ■ 设计

- 常用用例的**顺序图**：重要用例的对象交互顺序
- 软件视角的**类图**：设计软件中的类及相互联系
- **包图**：设计软件的组织结构
- 具有复杂生命周期的类的**状态图**
- **部署图**：设计软件的物理布局

## 核心 工作流程



## 第2章 软件过程与方法

- **2.1** 软件过程
- **2.2** 系统工程
- **2.3** 软件工程方法
- **2.4** 统一建模语言简介
  - 2.4.1 建模与UML
  - 2.4.2 UML的图
  - 2.4.3 用UML建模（类图）**

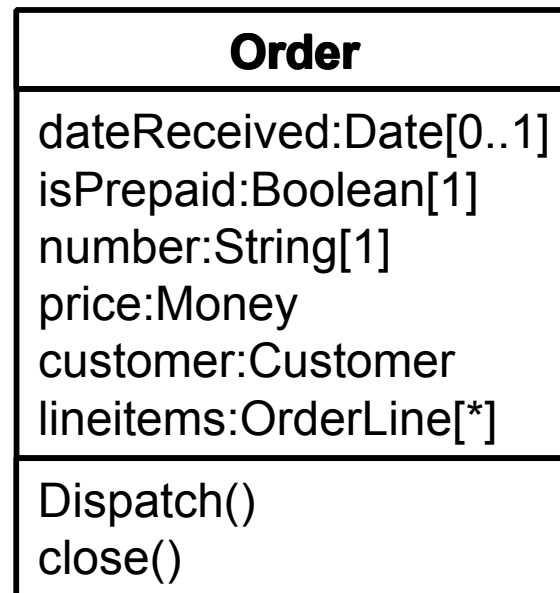
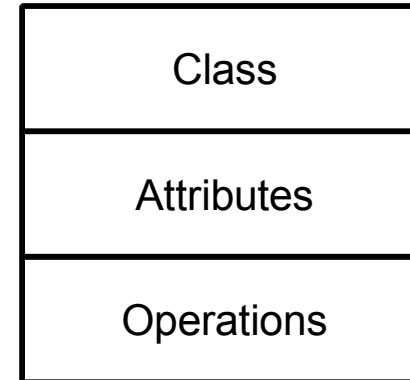
# 标准建模语言UML ——类图

- 在面向对象的建模技术中，类、对象和它们之间的关系是最基本的建模元素。对于一个想要描述的系统，其类模型、对象模型以及它们之间的关系揭示了系统的结构。
- 静态视图中的关键元素是类元及它们之间的关系。类元是描述事物的建模元素。有几种类元，包括类、接口和数据类型。
- 类图描述了系统中的类元及其相互之间的各种关系，类元之间的关系有关联、泛化及各种不同的依赖关系，包括实现和使用关系。

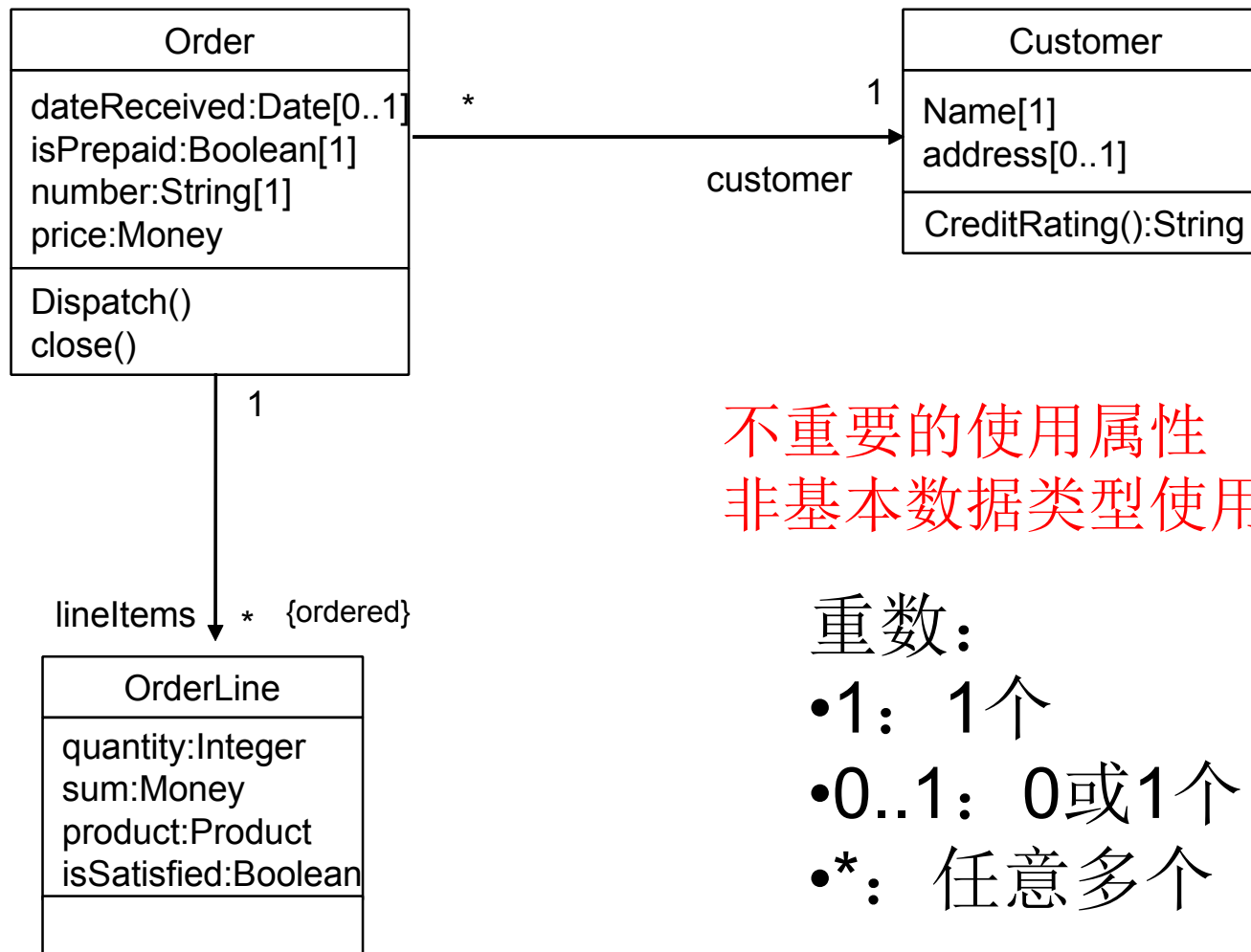
# 标准建模语言UML ——类图

## ■ 类模型

- 类名
- 属性
- 操作



# 属性及关联



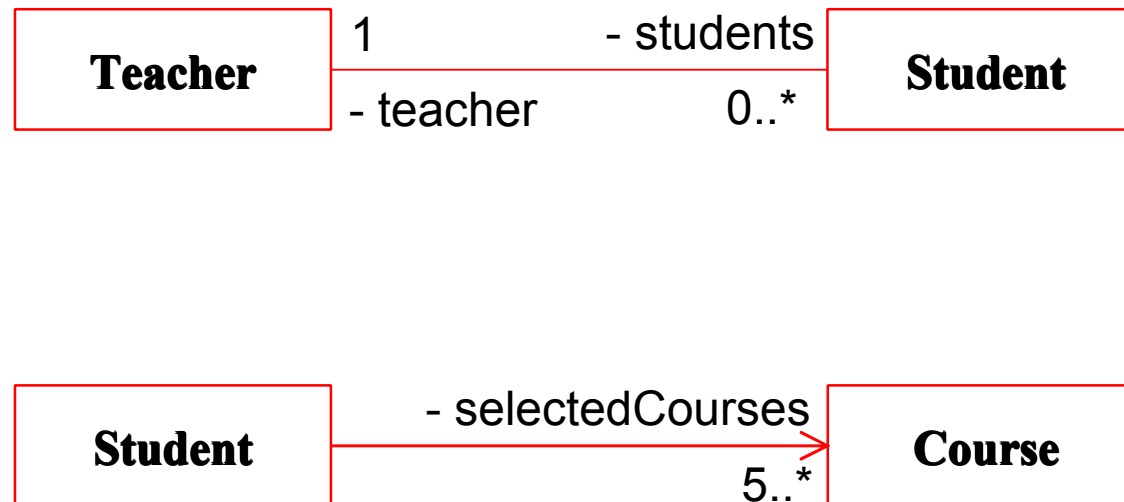
不重要的使用属性  
非基本数据类型使用关联

重数:

- 1: 1个
- 0..1: 0或1个
- \*: 任意多个

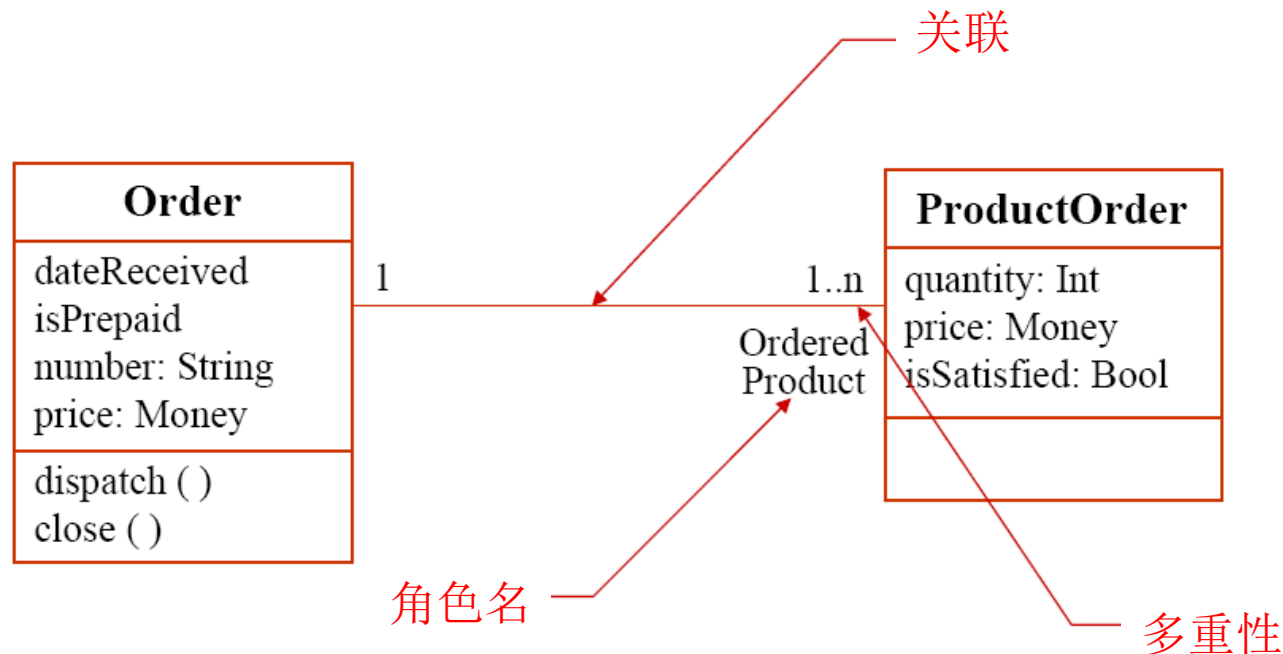
# 关联关系

- 例如：
  - “教师”与“学生”是两个类，它们之间存在“教-学”关系。
  - “学生”与“课程”是两个类，它们之间存在“学习-被学习”的关系。



# 关联关系

- 关联具有多重性(重数): 表示可以有多少个对象参与该关联
- 关联具有方向性:
  - 单向关联: 两个类是相关的, 但是只有一个类知道这种联系的存在
  - 双向关联: 两个类彼此知道它们间的联系





# 关联关系

```
class Course {
```

```
class Student {
```

```
    private Course [ ] selectedCourses;
```

```
}
```

**Student**

- selectedCourses

5..\*

**Course**

```
class Teacher {
```

```
    private Student [ ] students;
```

```
}
```

```
class Student {
```

```
    private Teacher teacher;
```

```
}
```

**Teacher**

1

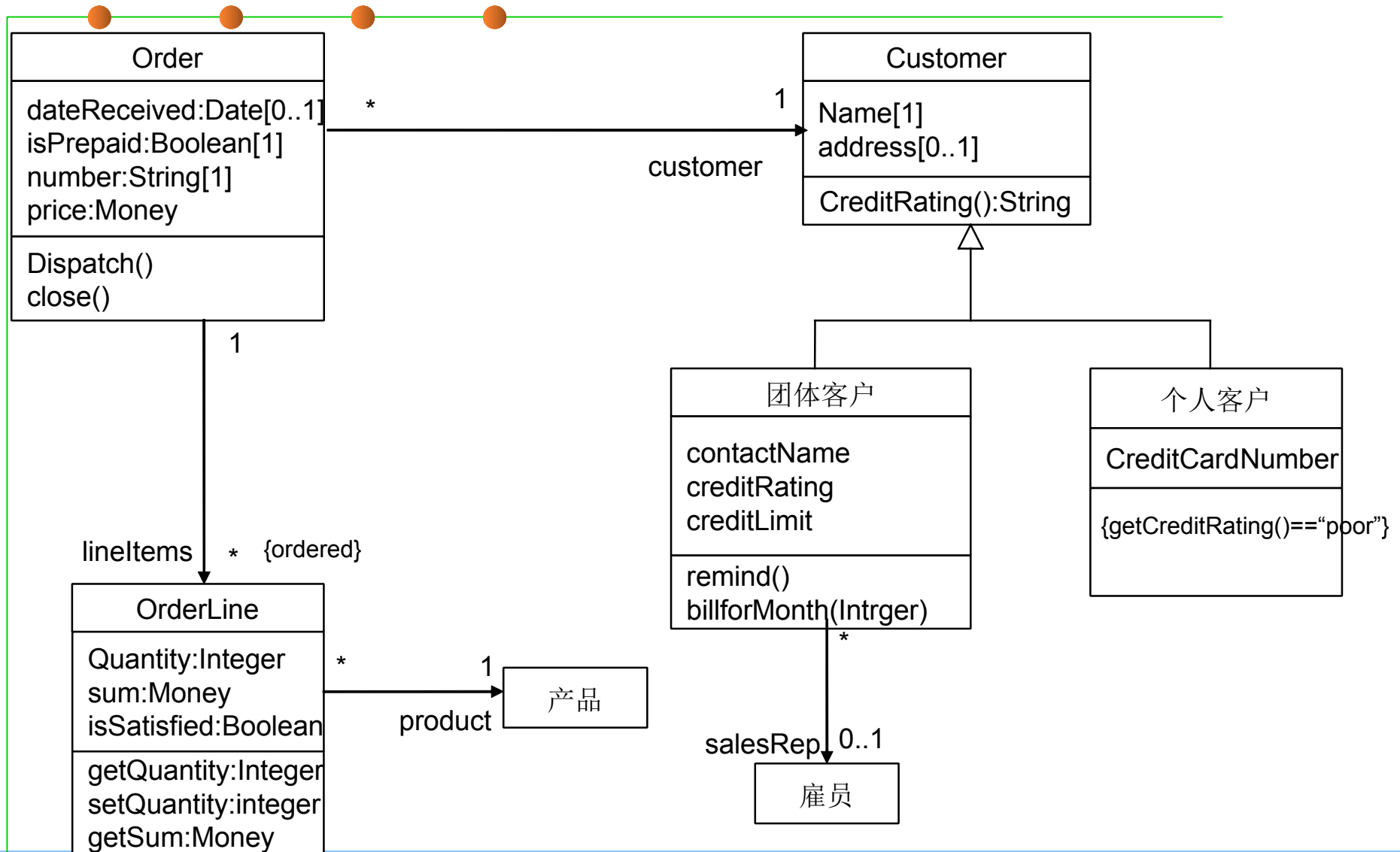
- students

- teacher

0..\*

**Student**

# 类图示例1

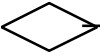



## 类的程序实现

```
public class OrderLine ...  
    private int quantity;  
    private Product product;  
    private Money sum;  
    public int getQuantity(){  
        return quantity;  
    }  
    public void setQuantity(int quantity){  
        this.quantity=quantity;  
    }  
    public Money getSum(){  
        return product.getPrice().multiply(quantity);  
    }
```

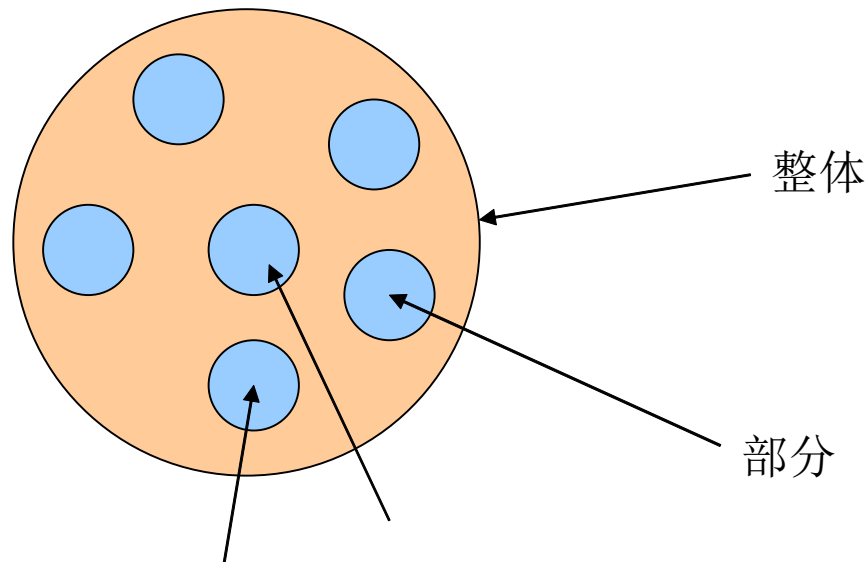
# 聚合与组合

类图中的图符：

-  — 聚合：整体与部分的关系
-  — 组合：整体拥有各部分，部分与整体共存，如整体不存在了，部分也会随之消失。

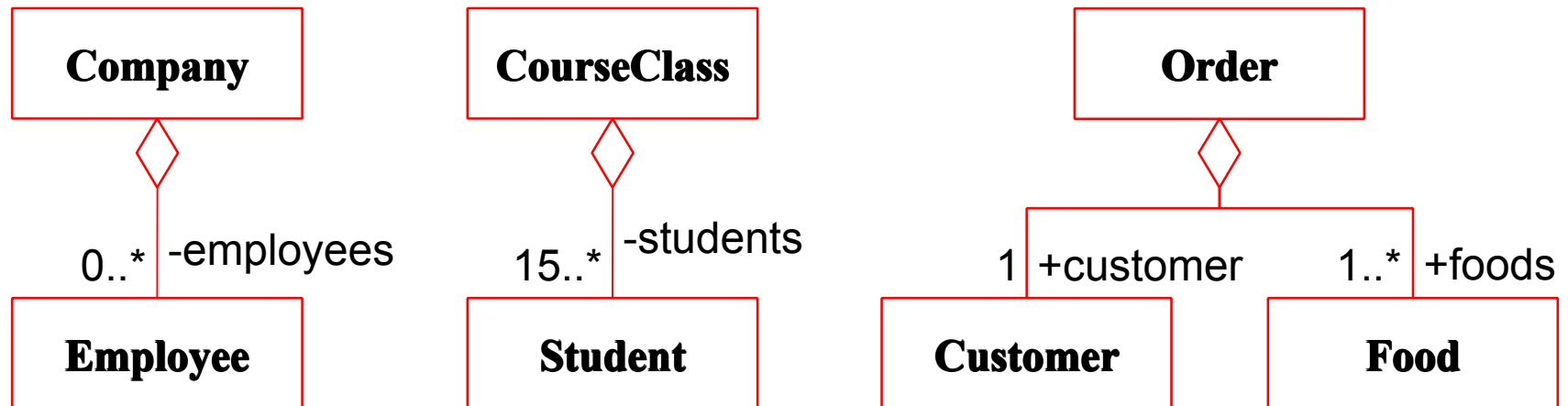
## 关联的特例：聚合与组合关系

- 组成结构：表示对象类之间的组成关系，一个对象是另一个对象的组成部分，即“部分-整体”关系。
- 分为两个子类：
  - 聚合(Aggregation): 整体与部分在生命周期上是独立的 (...owns a...);
  - 组合(Composition): 整体与部分具有同样的生命周期(...is part of...);



## 关联的特例：聚合关系

- 聚合(**Aggregation**): 整体与部分在生命周期上是独立的



**A company owns zero or multiple employees;  
A course's class owns above 15 students;  
An order owns a customer and a set of foods;**

## 关联的特例：聚合关系

定义两个类：

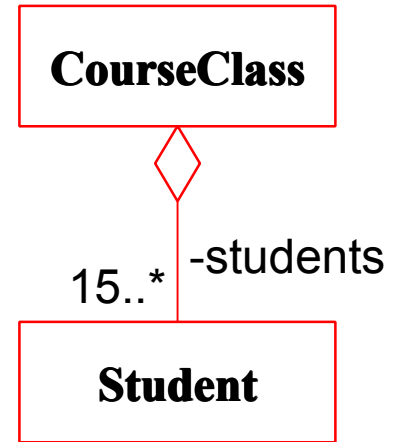
```
class Student {}
```

```
class CourseClass {  
    ...  
    private Student[] students;  
    public addStudent (Student s) {  
        students.append(s);  
    }  
    ...  
}
```

使用时的代码：

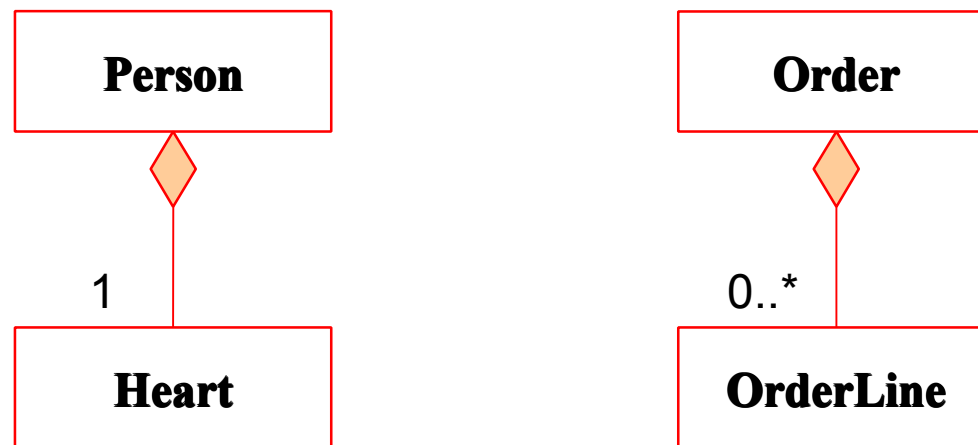
```
Student a = new Student ();  
Student b = new Student ();  
Student n = new Student ();
```

```
CourseClass SE = new CourseClass();  
SE.addStudent (a);  
SE.addStudent (b);  
SE.addStudent (n);
```



## 关联的特例：组合关系

- 组合(**Composition**): 是聚合的特例, 强调整体与部分具有同样的生命周期;





## 关联的特例：组合关系

```
class Heart {}
```

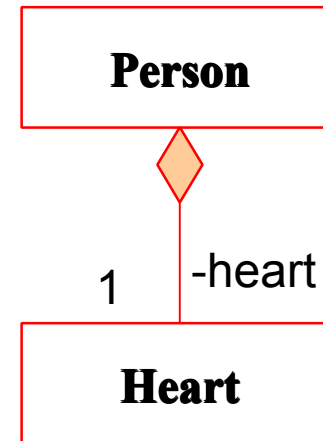
```
class Person {
```

```
...
```

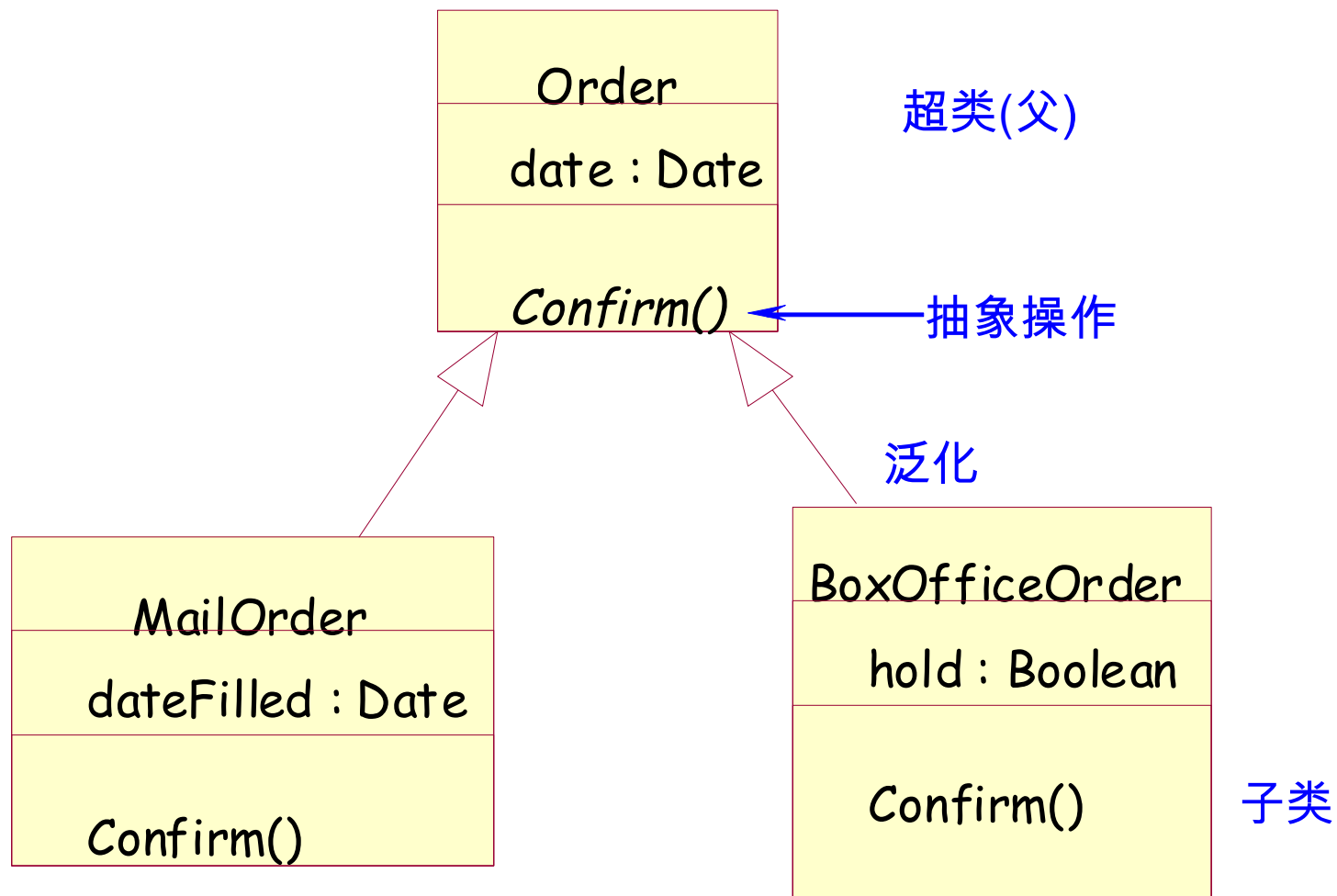
```
private Heart heart = new Heart();
```

```
...
```

```
}
```

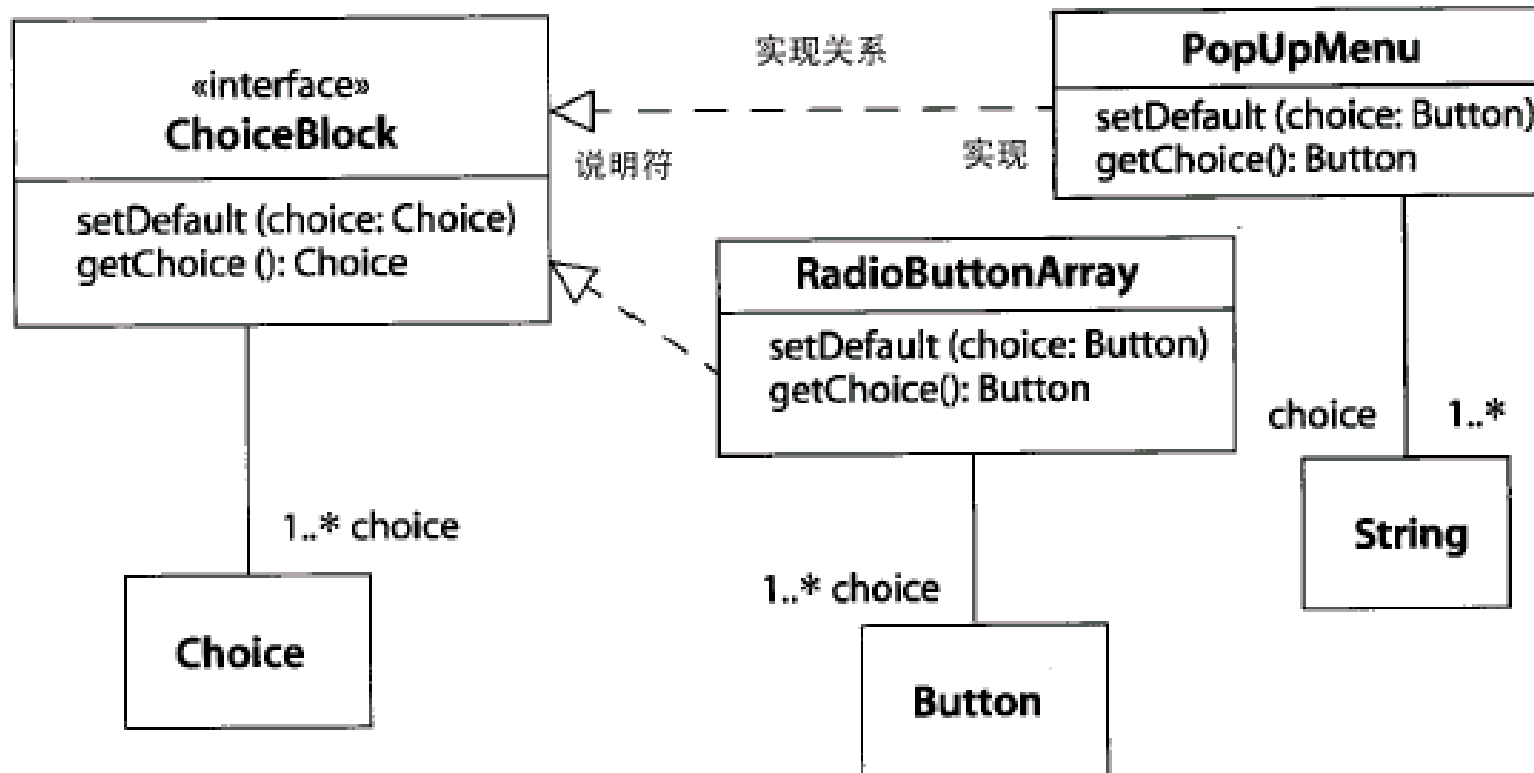


# 泛化/继承

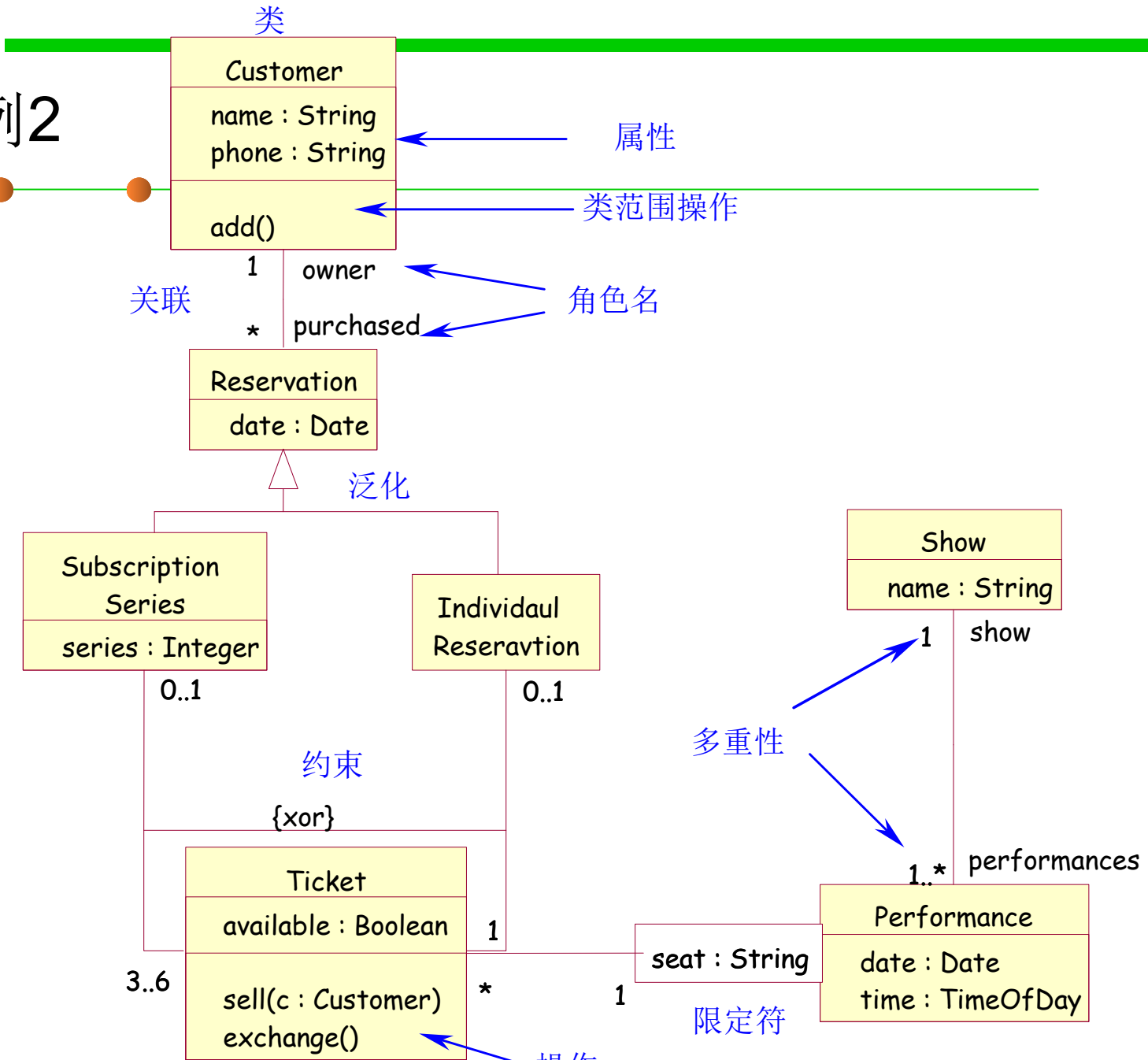


# 实现

- 实现关系将一种模型元素（如类）与另一种模型元素（如接口）连接起来，其中接口只是行为的说明而不是结构或者实现。

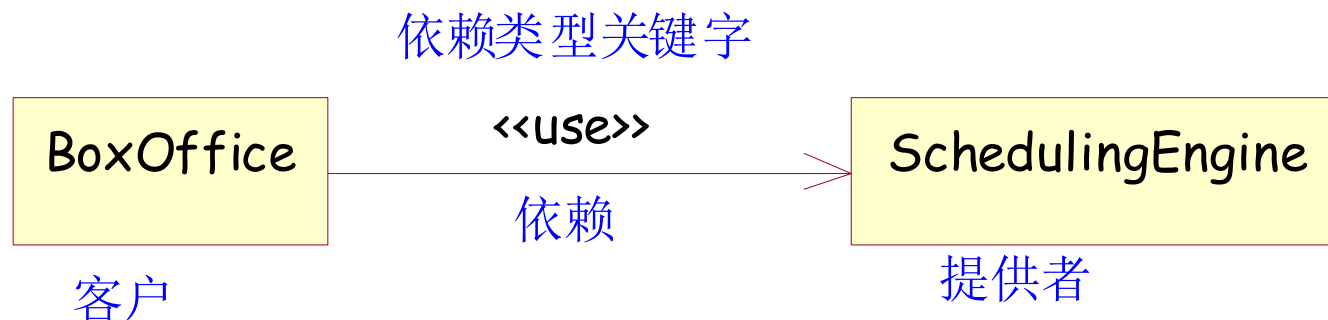


## 类图示例2



# 依赖

- 如果改动一方的定义可引起另一方的改动，则这两方之间存在**依赖**
- **原因**
  - 一个类调用另一个类的方法（消息）
  - 一个类以另一个类作为其数据部分（创建新对象实例）
  - 一个类用到另一个类作为操作参数



# 依赖

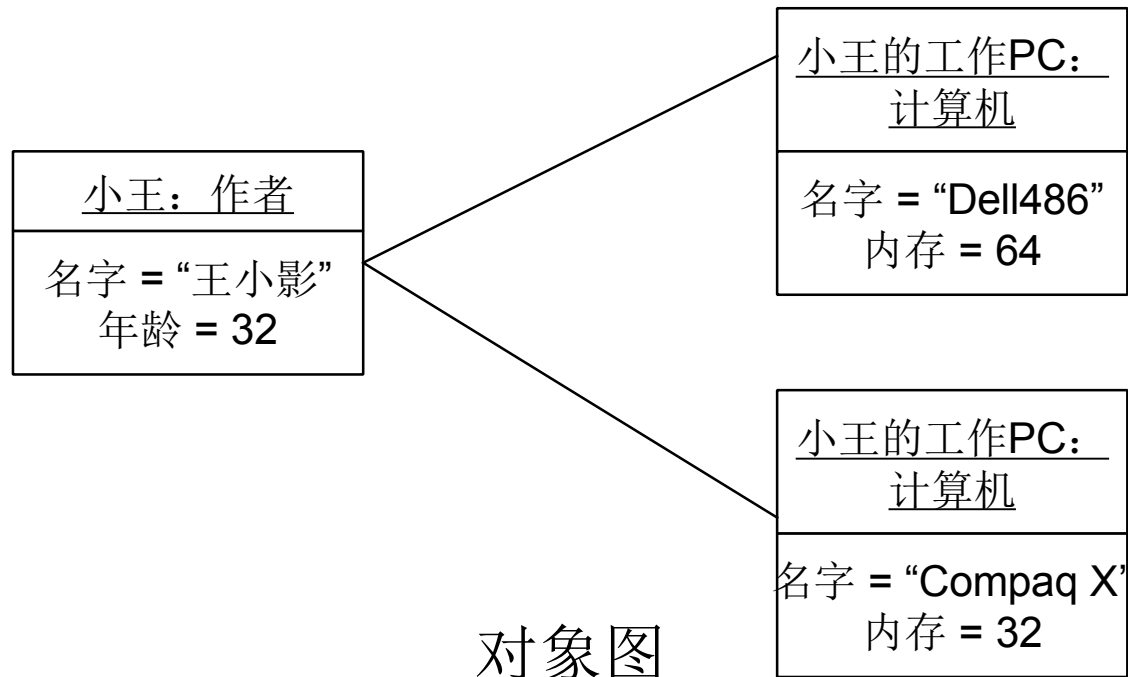
- 依赖表示两个或多个模型元素之间语义上的关系。

依赖关系	功能	关键字
访问	允许一个包访问另一个包的内容	access
绑定	为模板参数指定值，以生成一个新的模型元素	bind
调用	声明一个类调用其他类的操作的方法	call
派生	声明一个实例可以从另一个实例导出	derive
友员	允许一个元素访问另一个元素，不管被访问的元素是否具有可见性	friend
输入	允许一个包访问另一个包的内容并为被访问包的组成部分增加别名	import
实例化	关于一个类的方法创建了另一个类的实例的声明	instantiate
参数	一个操作和它的参数之间的关系	parameter
实现	说明和对这个说明的具体实现之间的映射关系	realize
精化	声明具有两个不同语义层次上的元素之间的映射	refine
发送	信号发送者和信号接收者之间的关系	send
跟踪	声明不同模型中的元素之间存在一些连接，但不如映射精确	trace
使用	声明使用一个模型元素需要用到已存在的另一个模型元素，这样才能正确实现使用者的功能（包括了调用、实例化、参数、发送）	use

## 标准建模语言UML——对象图

- 对象图是类图的一种变形。除了在对象名下面要加下划线以外，对象图中所使用的符号与类图基本相同。
- 对象图是类图的一种实例化。一张对象图表示的是与其对应的类图的一个具体实例，即系统在某一时期或者某一特定时刻可能存在的对象实例以及它们相互之间的具体关系。



# 对象图示例





# 学习材料

- 参考论文（已上传）
  - Process Models in Software Engineering
  - Characterizing the Software Process A Maturity Framework
- 掌握一门面向对象程序语言
  - JAVA: Java核心技术(卷1)、Head First Java
  - C#, C++
- 学习**UML**（已上传）
  - UML Distilled A Brief Guide to the Standard Object Modeling Language
  - UML参考手册



结束

**2011年4月6日**