

第一章

软件工程导论

本章要点

- 工程的概念
- 软件工程的发展
- 软件工程分析
- 三种过程模型
- 工程化思考

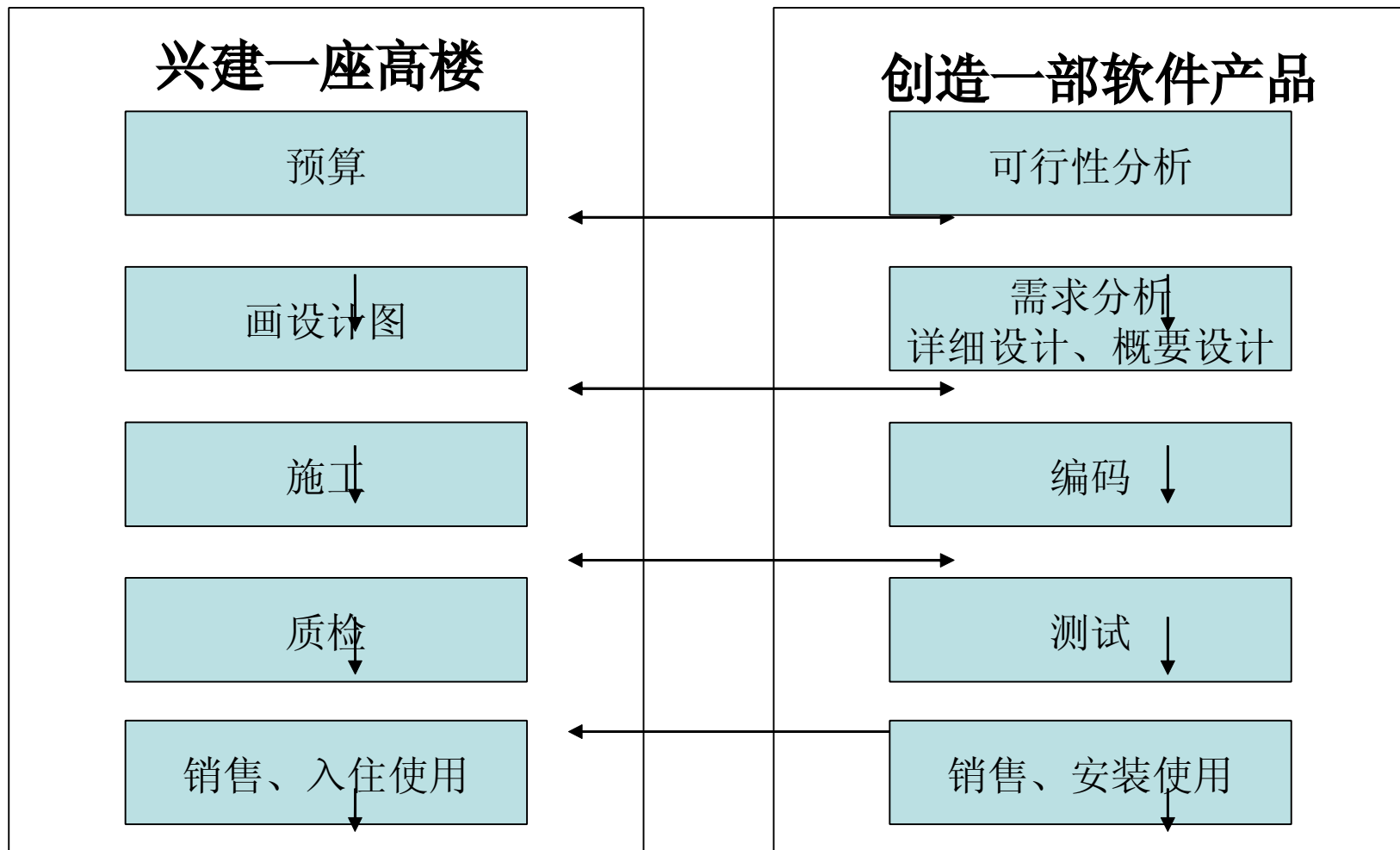
工程是什么？

- 工程简而言之就是多人参与并有计划、有步骤地完成一项任务的活动
- 工程强调
 - 目的
 - 计划
 - 步骤

软件发展与软件工程起源

- 软件的发展四个阶段：
 - 1950年前后到1960年前后，程序设计阶段；
 - 1960年前后到1970年前后，软件系统阶段；
 - 1970年前后到1980年前后互联网络兴起，软件工程阶段；
 - 1980年前后到现在，分布式软件工程阶段；
- 1968年，北大西洋公约组织的计算机科学家召开国际会议，第一次提出软件危机的概念，产生了应对软件危机的对策---软件工程。

软件工程与建筑工程的对比

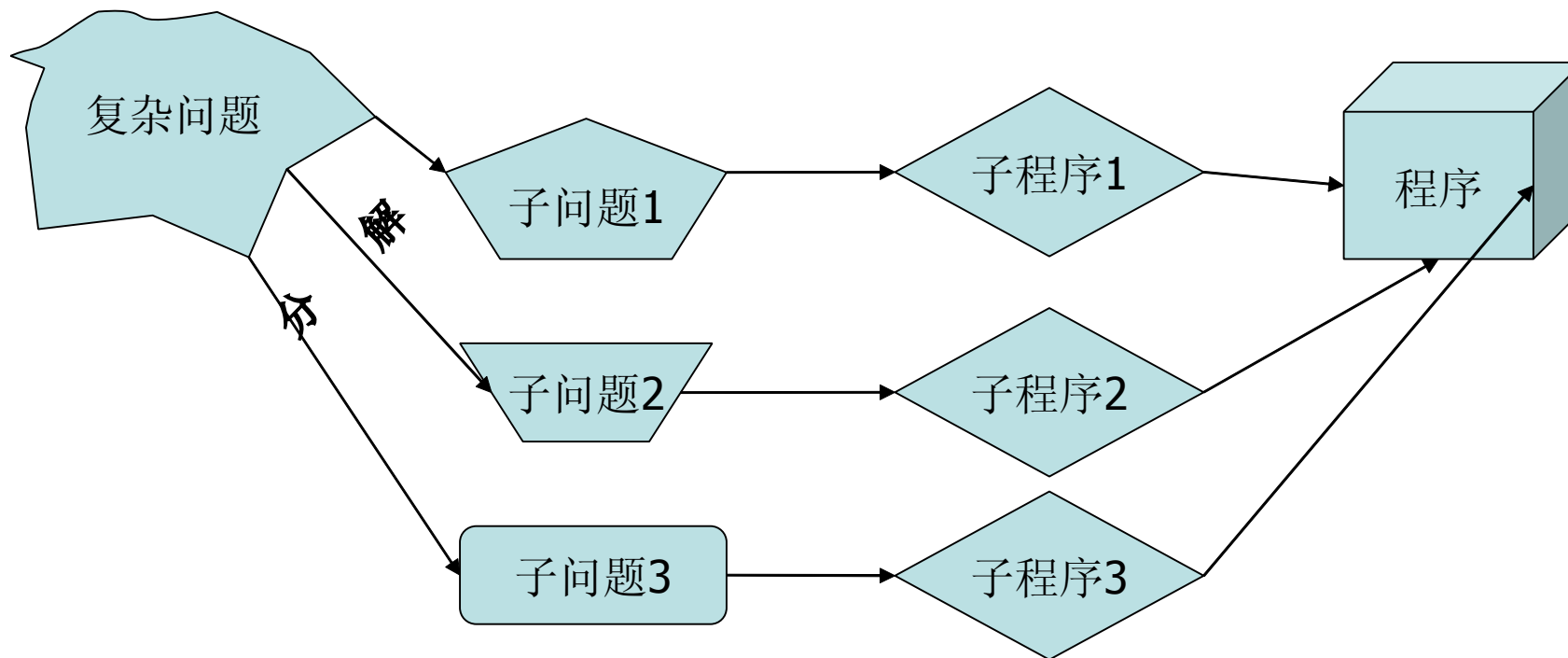


工程策略

- 任何工程都有如下的策略：
 - 分而治之
 - 复用
 - 折衷优化
 - 检验并保证质量
- 软件工程也会充分利用这些策略

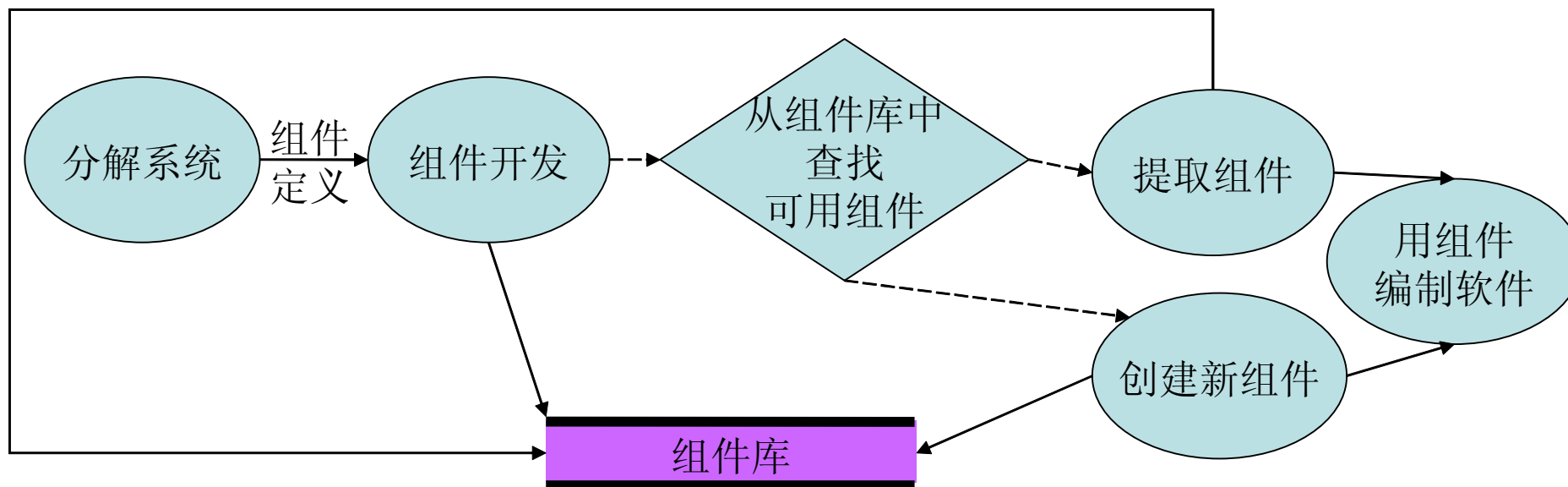
分而治之

- 把复杂的问题分解为小的问题并一一解决
- 分而治之图示



复用

- 利用现有的组件来构筑软件的一部分功能
- 组件技术有：**CORBA**、**EJB**、**COM**
- 软件复用图示：



软件开发的发展与变化

软件技术的发展带来了一些变化：

- 1 用户对软件要求的变化：软件规模在扩大；对软件的质量要求在提高；
- 2 软件技术本身的变化：新的理念、新的方法和新的工具
- 3 软件开发队伍的变化：从单人开发、小组开发，到大规模团队开发；从稳定、相对稳定到全员流动

软件开发的发展与变化

- 应对这些变化的是：
- 1 市场化：软件开发由个人爱好行为转变为企业行为，需要大量的投资、大量的人力，并且要按照市场规律来运作
- 2 知本化：要求技术的积累、模块的积累和成果的积累；
- 3 开发过程的规范化：来应对需求多变，人员流动
- 4 标准化：能力成熟度，质量控制

软件工程的目标

- 软件工程的目标是提高软件的质量与生产率，最终实现合格的软件。
 - 质量是软件需求方最关心的问题。
 - 生产率是软件供应方最关心的问题。

软件工程的四项基本原则

- 选取适宜开发范型
- 采用合适的设计方法
- 提供高质量的工程支持
- 重视开发过程的管理

软件工程准则

- 七条基本准则
 - 1) 生命周期计划；
 - 2) 阶段评审；
 - 3) 变更控制；
 - 4) 改进程序设计技术；
 - 5) 控制人员规模；
 - 6) 定义评审；
 - 7) 不断改进软件工程；

软件工程的要素

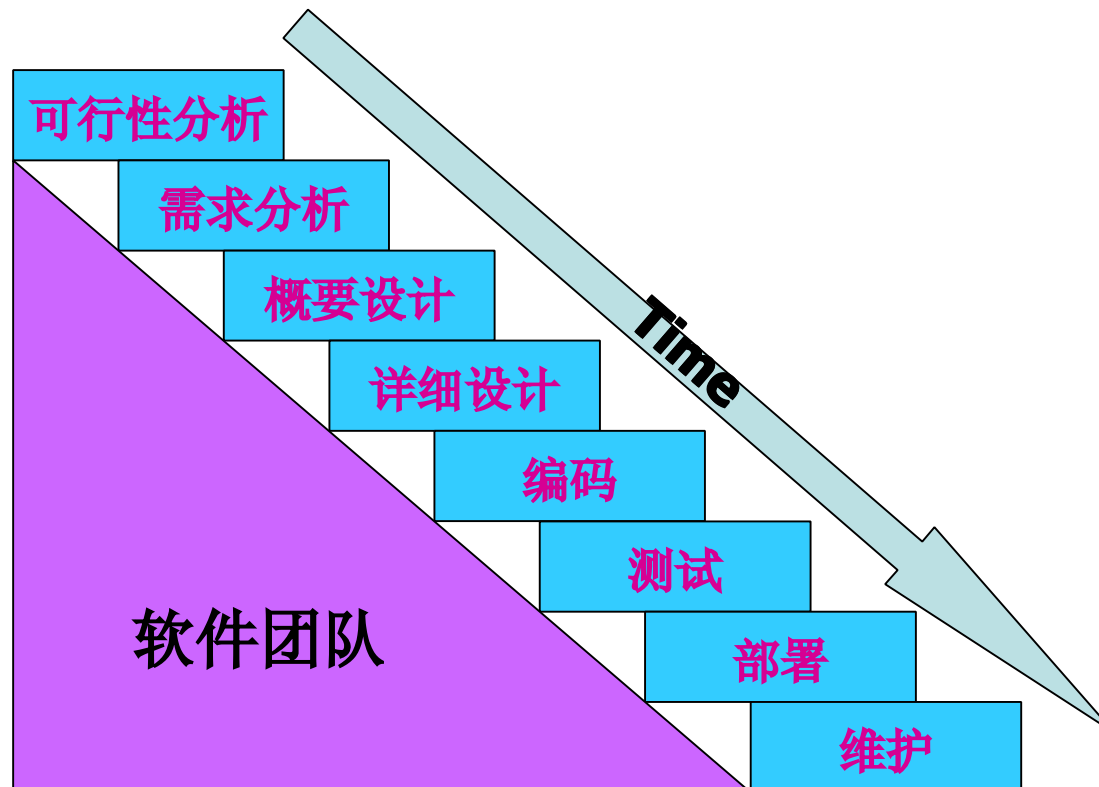
1. **方法：**软件工程方法为软件开发提供了“如何做”的技术，是完成软件工程项目的手段；
2. **工具：**软件工具是在开发软件的活动中智力和体力的扩展和延伸，为软件工程方法提供了自动的或半自动的软件支撑环境；
3. **过程：**软件工程的过程则是将软件工程的方法和工具综合起来以达到合理、及时地进行计算机软件开发的目的。

软件工程的组成

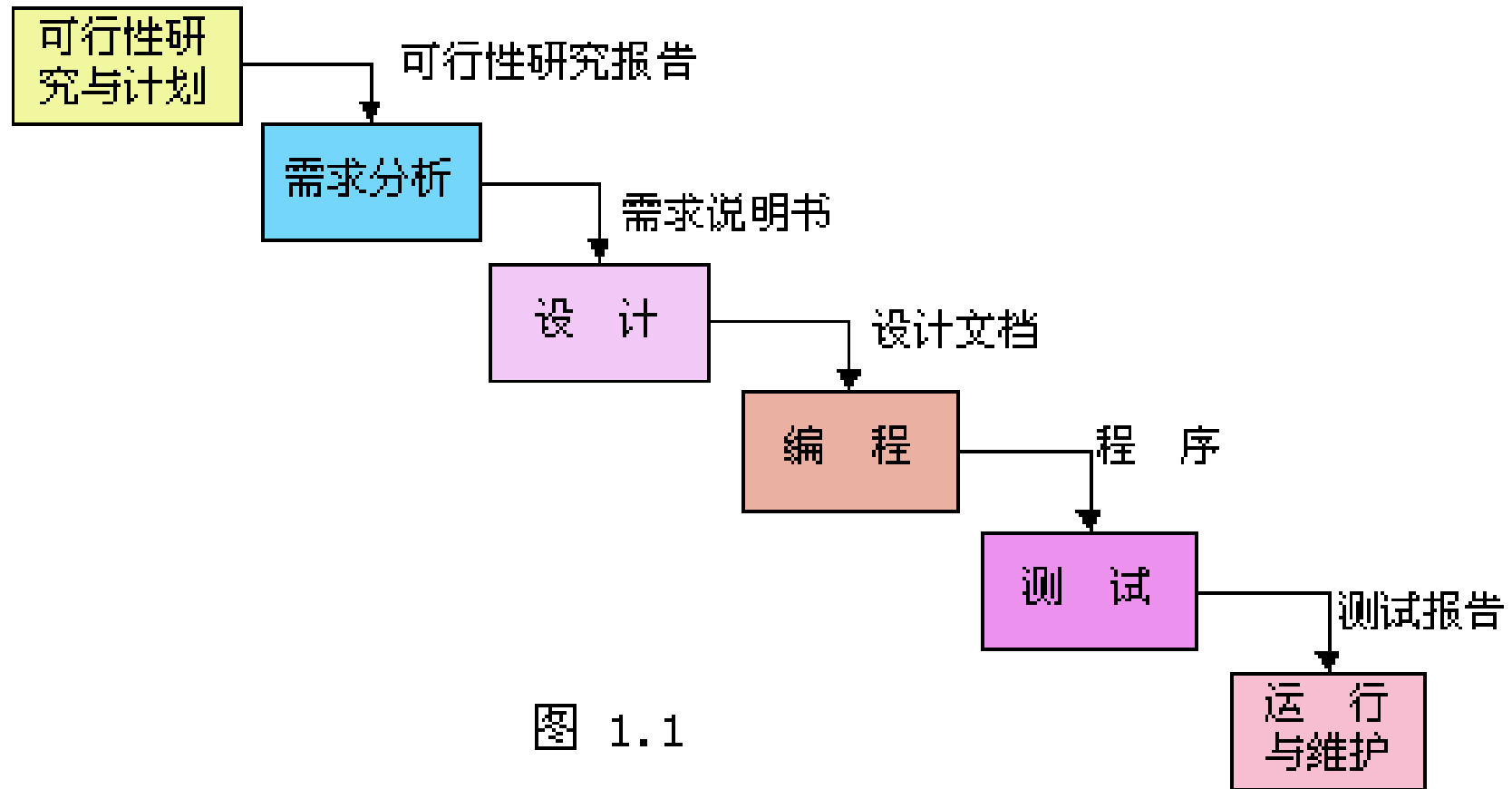
- 人员管理
- 项目管理
- 过程管理

瀑布模型

- 瀑布模型将软件生命周期的各项活动顺序进行，形如瀑布流水，最终得到软件产品
- 是最早的软件工程模型，是所有现代模型的基础



瀑布模型 continue



阶段任务、结果及人员

●

| | 阶段 | 基本任务 | 工作结果 | 参加者 |
|-----|----------|-------------|----------|-------------|
| 计划期 | 可行性研究与计划 | 研究开发该项目的可行性 | 可行性研究报告 | 用户、高级程序员 |
| | 需求分析 | 理解和表达用户的要求 | 需求说明书 | 用户、高级程序员 |
| 开发期 | 设计 | 建立系统的结构 | 模块、数据说明书 | 用户、高级程序员 |
| | 编程 | 编写程序 | 程序 | 高级程序员、初级程序员 |
| | 测试 | 发现错误和排除错误 | 测试报告 | 另一独立的部门 |
| 运行期 | 运行与维护 | 维护 | 改进的系统 | 用户、高级程序员 |

瀑布模型特征

- 从上一项活动接收该项活动的工作对象，作为输入；
- 利用这一输入实施该项活动应完成的内容；
- 给出该项活动的工作成果，作为输出传给下一项活动；
- 对该项活动实施的工作进行评审，若其工作得到确认，则继续下一项活动，否则返回前项，甚至更前项的活动进行返工。

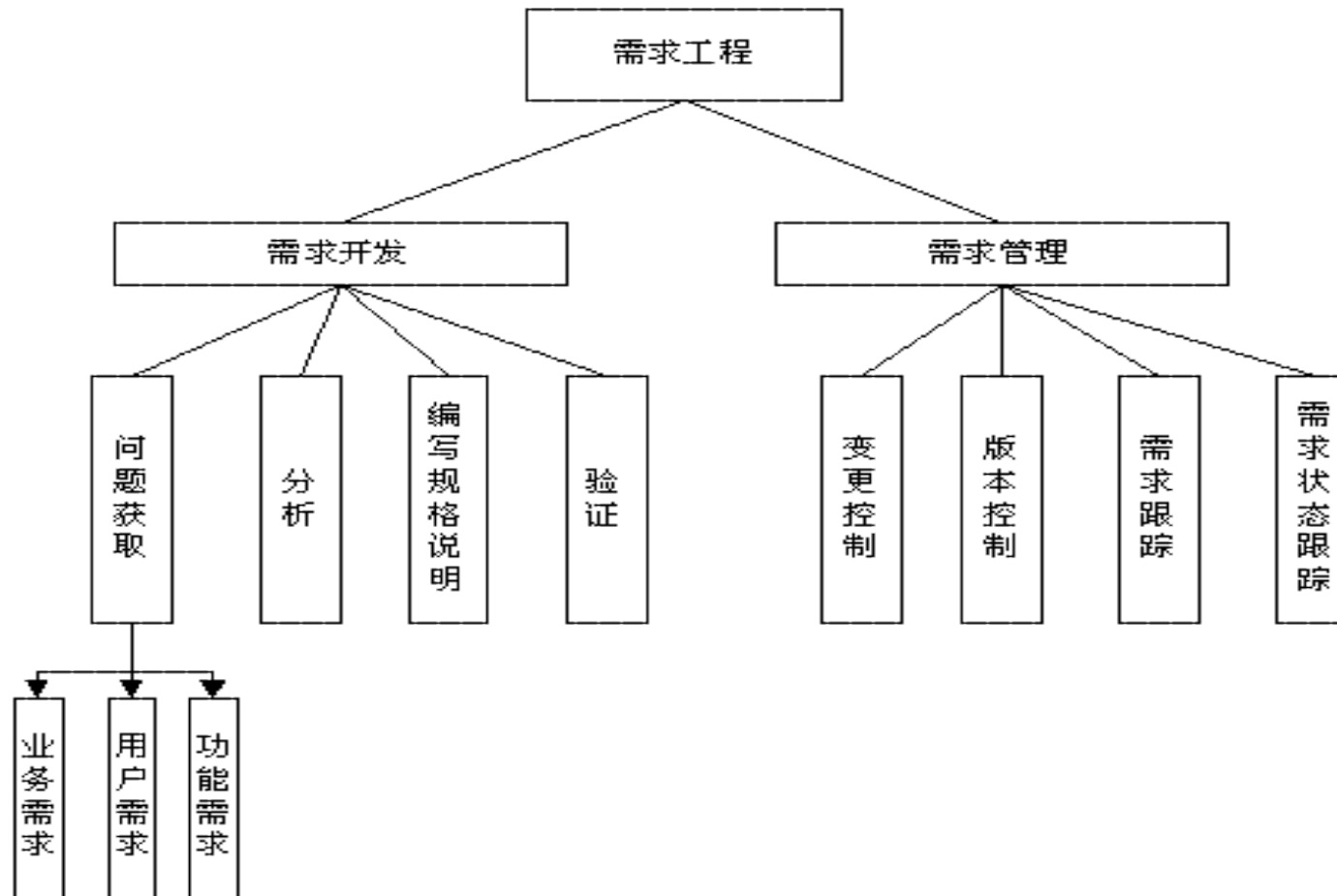
瀑布模型各个阶段概述

- 可行性分析：做还是不做
- 需求分析： 都有什么功能
- 概要设计： 供有多少子功能
- 详细设计： 子功能怎么实现
- 编码： 子功能实现了吗
- 测试： 功能完备吗
- 部署： 需要多少设备和软件的支持
- 维护： 软件运行的正常吗

可行性分析

- 可行性分析因素
 - 经济
 - 技术
 - 社会环境
 - 人才

需求分析



概要设计

- 提供多少子功能
- 面向对象分析（OOA）

详细设计

- 子功能如何实现
- 面向对象设计（OOD）

编码

- 子功能是否实现？
- 程序员严格按照规范编码；

测试

- 单元测试
- 系统测试
- 用户测试

部署

- 部署要求
 - 增强自动化程度，用**ant**等工具
 - 培训最终用户
 - 要有详细计划
 - 记录详细的过程数据
 - 及时反馈软件兼容性缺陷

维护

- 一般维护分三类：
 - 纠错性维护
 - 改正软件漏洞、发布补丁程序
 - 适应性维护
 - 使得软件在新的硬件、操作系统、编译器和解释器下运行
 - 完善性维护
 - 增加新功能、更改原有的设计等

影响维护成本的因素

- 非技术因素
 - 需求的复杂性
 - 开发人员的岗位稳定性
 - 软件的生命期
 - 外部环境的变化，例如财政政策影响财务软件
- 技术因素
 - 软件运行环境
 - 编程语言
 - 编程风格
 - 测试工作的有效性
 - 文档的质量

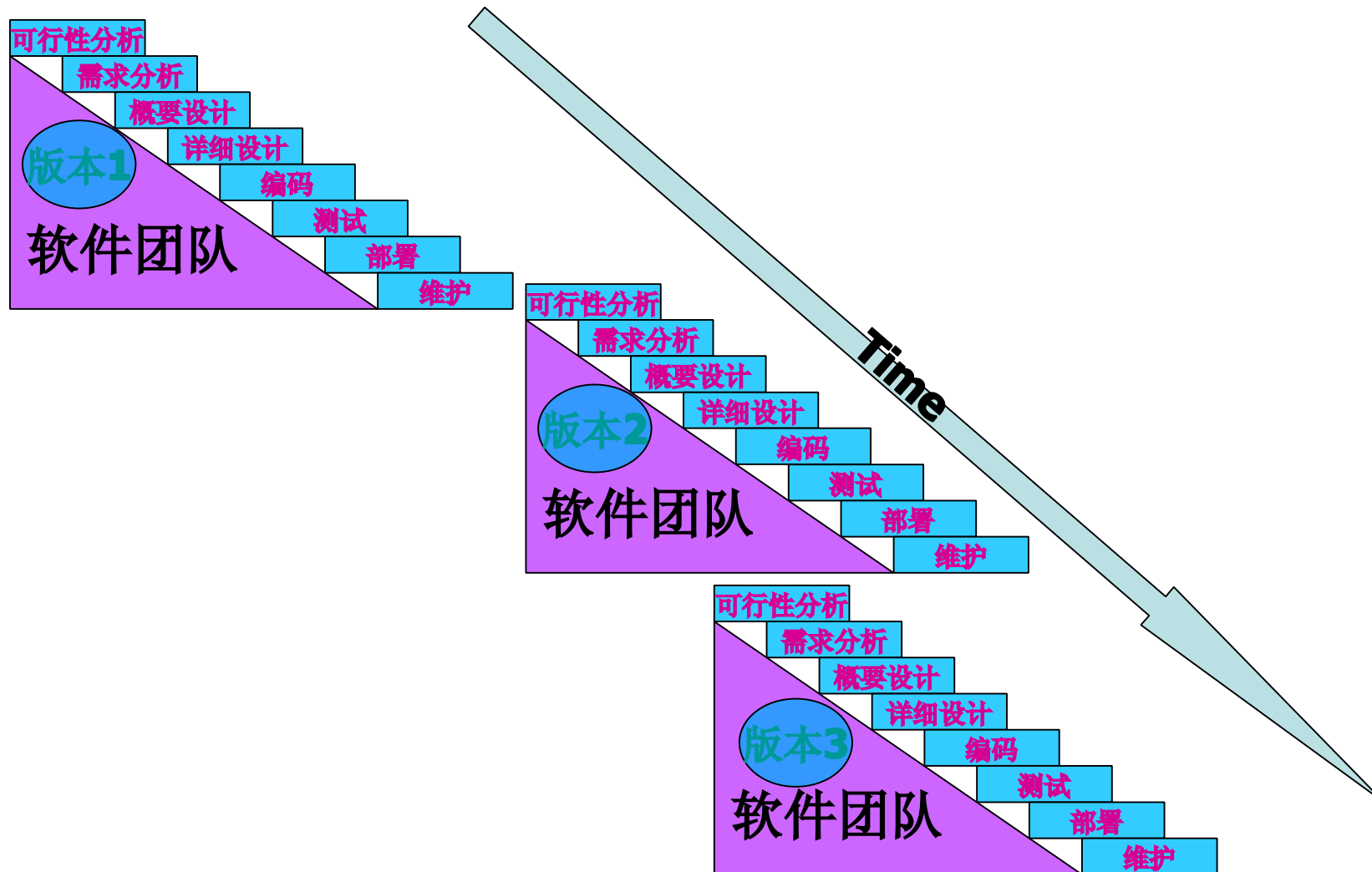
瀑布模型的优点

- 通过设置里程碑，明确每阶段的任务与目标
- 可为每阶段制定开发计划，进行成本预算，组织开发力量
- 通过阶段评审，将开发过程纳入正确轨道
- 严格的计划性保证软件产品的按时交付

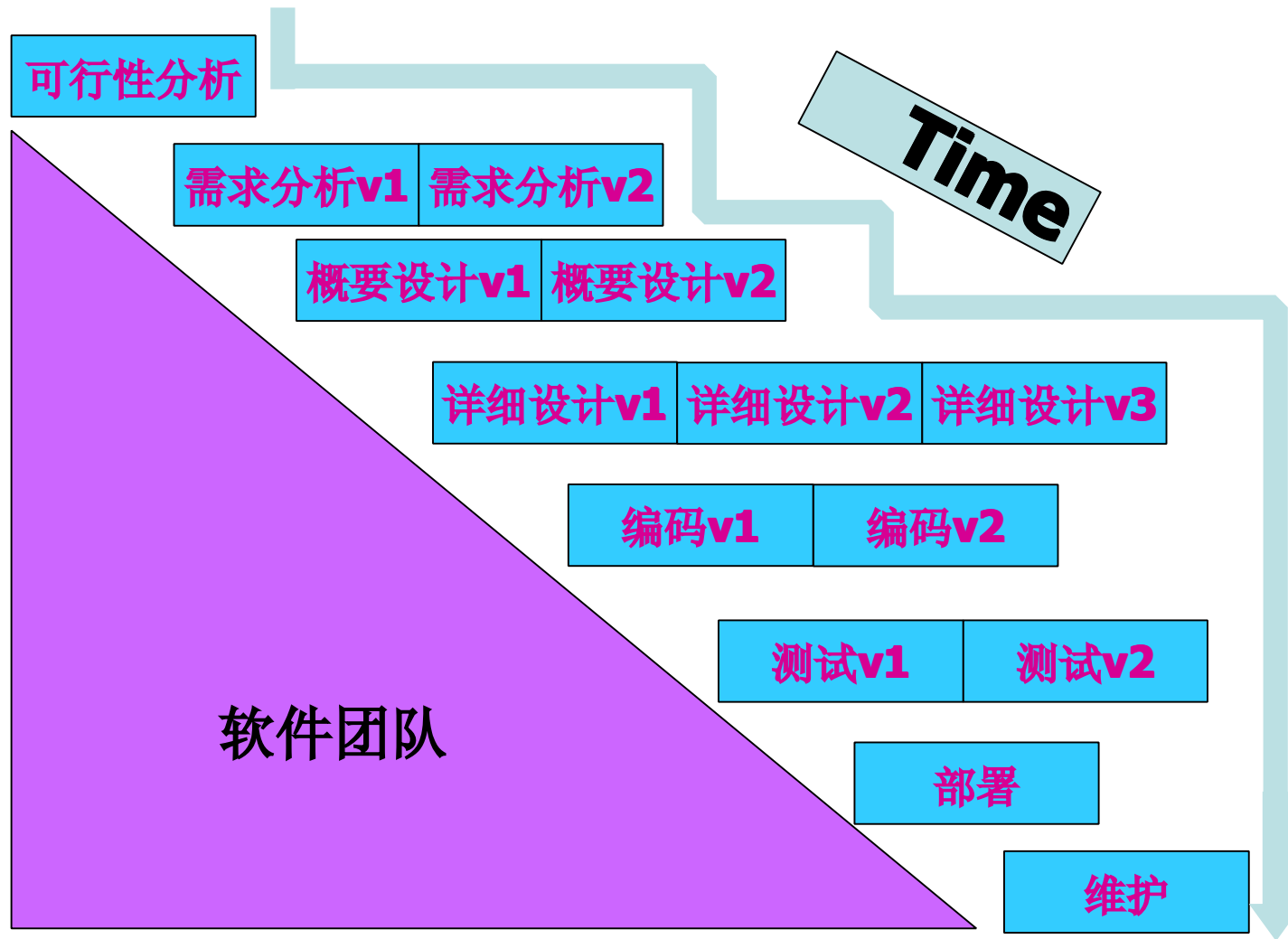
瀑布模型的缺点

- 缺乏灵活性，不能适应用户需求的变化
- 开始阶段的小错误被逐级放大，可能导致软件产品报废
- 返回上一级的开发需要十分高昂的代价
- 随着软件规模和复杂性的增加，软件产品成功的机率大幅下降

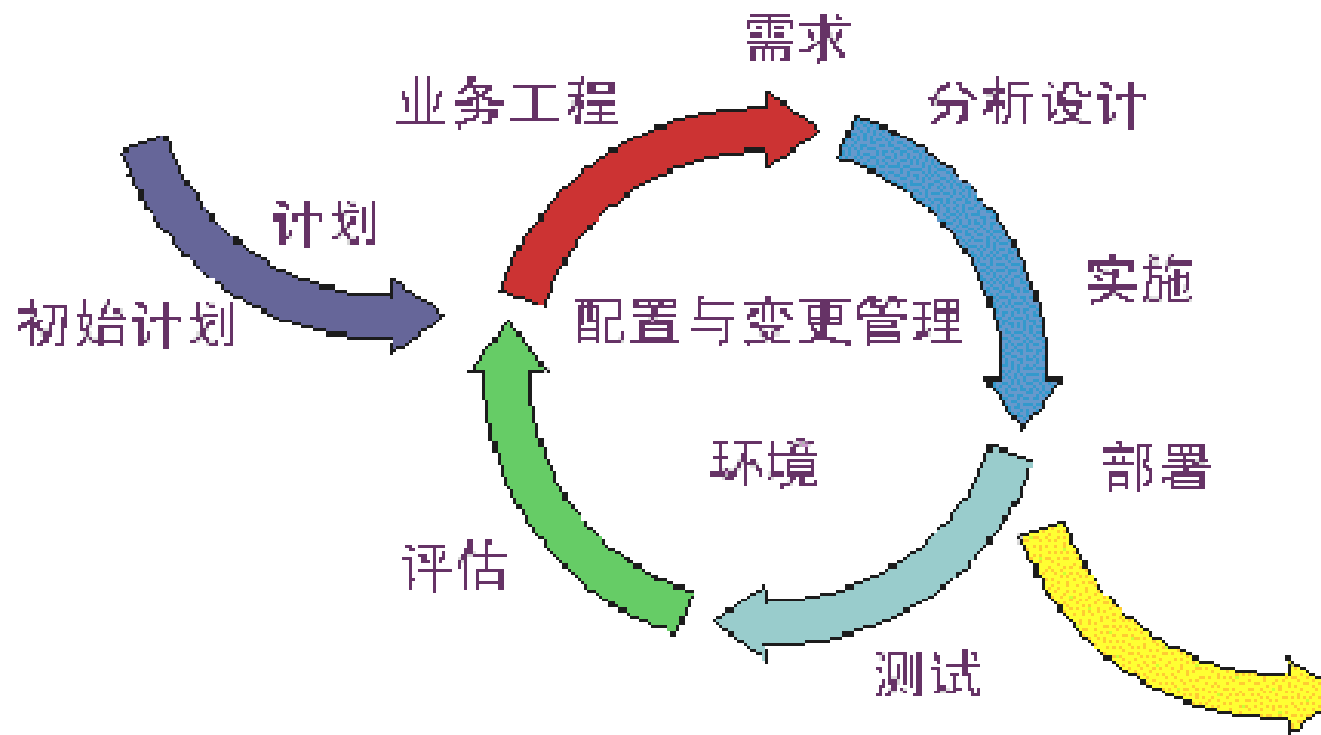
演化模型



迭代模型



迭代模型 continue



迭代模型 continue

- 迭代的定义：迭代包括产生产品发布（稳定、可执行的产品版本）的全部开发活动和要使用该发布必需的所有其他外围元素。
- 生命周期是基于对一个系统进行连续的扩充和精化，需要经历若干个开发周期，每个周期都需要经历分析、设计、实现和测试阶段。每个开发周期只针对比较小的一部分需求
- 在某种程度上，开发迭代是一次完整地经过所有工作流程的过程：（至少包括）需求工作流程、分析设计工作流程、实施工作流程和测试工作流程；
- 类似小型的瀑布式项目。**RUP**认为，所有的阶段（需求及其它）都可以细分为迭代。每一次的迭代都会产生一个可以发布的产品，这个产品是最终产品的一个子集。

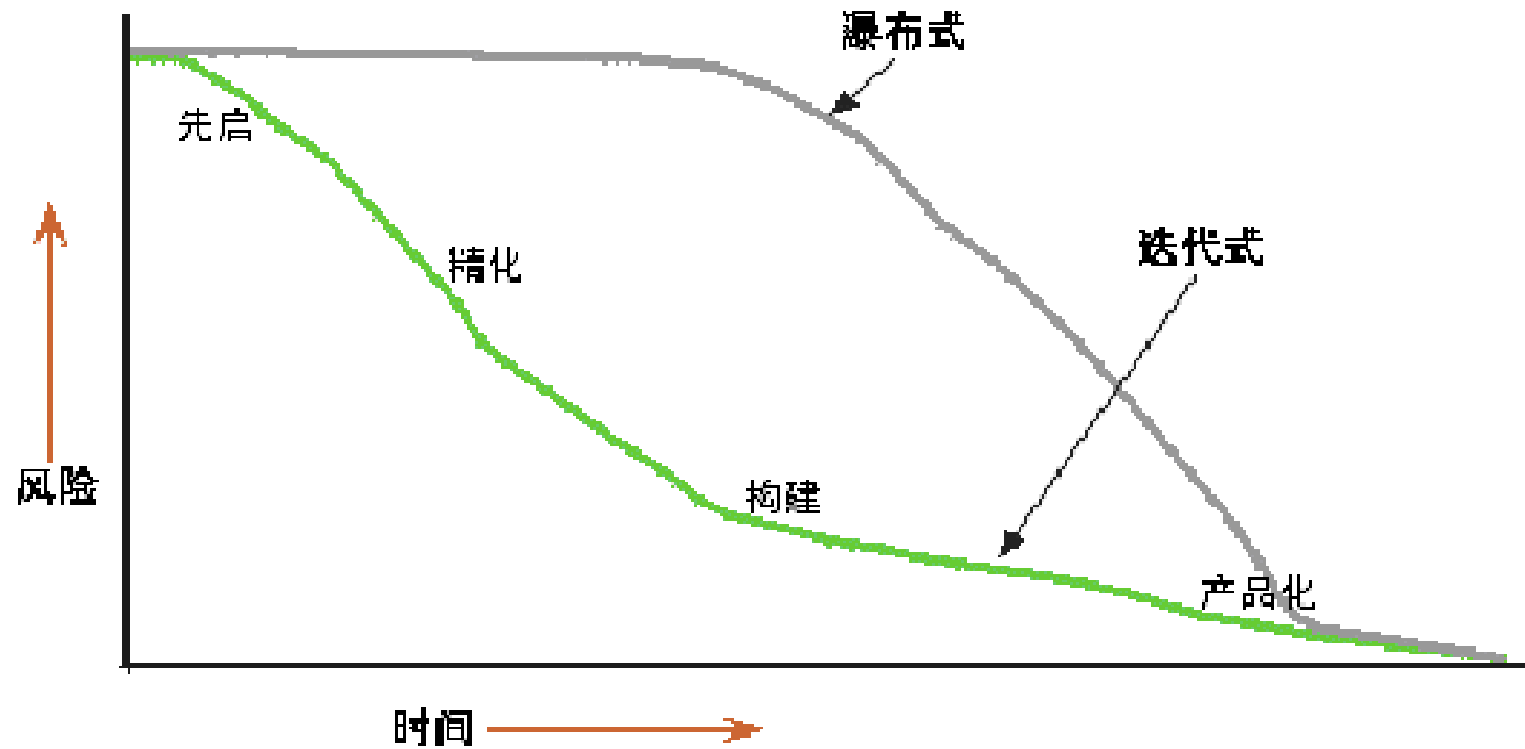
迭代的优点

- 降低了在一个增量上的开支风险。如果开发人员重复某个迭代，那么损失只是这一个开发有误的迭代的花费。
- 降低了产品无法按照既定进度进入市场的风险。通过在开发早期就确定风险，可以尽早来解决而不至于在开发后期匆匆忙忙。
- 加快了整个开发工作的进度。因为开发人员清楚问题的焦点所在，他们的工作会更有效率。
- 由于用户的需求并不能在一开始就作出完全的界定，它们通常是在后续阶段中不断细化的。因此，迭代过程这种模式使适应需求的变化会更容易些。

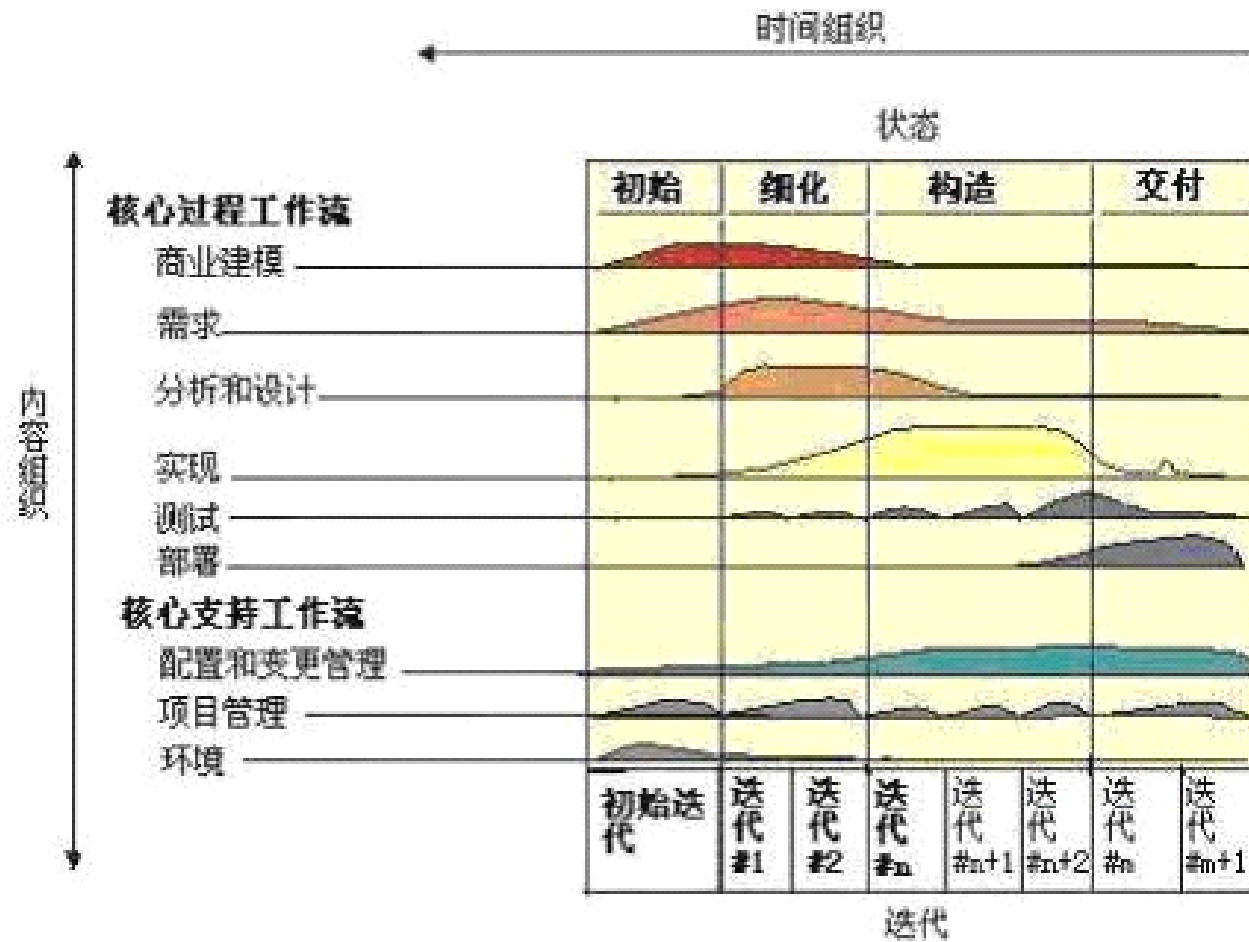
迭代模型和瀑布模型的差别

- 最大的差别在于风险的暴露时间上。
- 任何项目都会涉及到一定的风险。如果能在生命周期中尽早确保避免了风险，那么计划自然会更趋精确。
- 有许多风险直到已准备集成系统时才被发现。不管开发团队经验如何，都绝不可能预知所有的风险。

迭代模型和瀑布模型的差别



统一软件过程RUP模型



RUP中的软件生命周期

- 初始阶段(Inception): 目标是为系统建立商业案例并确定项目的边界。
- 细化阶段(Elaboration): 目标是分析问题领域，建立健全的体系结构基础，编制项目计划，淘汰项目中最高风险的元素。
- 构造阶段(Construction): 所有剩余的构件和应用程序功能被开发并集成为产品，所有的功能被详细测试。
- 交付阶段(Transition): 重点是确保软件对最终用户是可用的。
每个阶段结束于一个主要的里程碑(*Major Milestones*)；每个阶段本质上是两个里程碑之间的时间跨度。在每个阶段的结尾执行一次评估以确定这个阶段的目标是否已经满足。如果评估结果令人满意的话，可以允许项目进入下一个阶段。

RUP模型的优缺点

优点

- 提高了团队生产力，在迭代的开发过程、需求管理、基于组件的体系结构、可视化软件建模、验证软件质量及控制软件变更等方面，针对所有关键的开发活动为每个开发成员提供了必要的准则、模板和工具指导，并确保全体成员共享相同的知识基础。
- 建立了简洁和清晰的过程结构，为开发过程提供较大的通用性。

缺点

- **RUP**只是一个开发过程，并没有涵盖软件过程的全部内容，例如它缺少关于软件运行和支持等方面的内容；
- 没有支持多项目的开发结构，这在一定程度上降低了在开发组织内大范围实现重用的可能性。

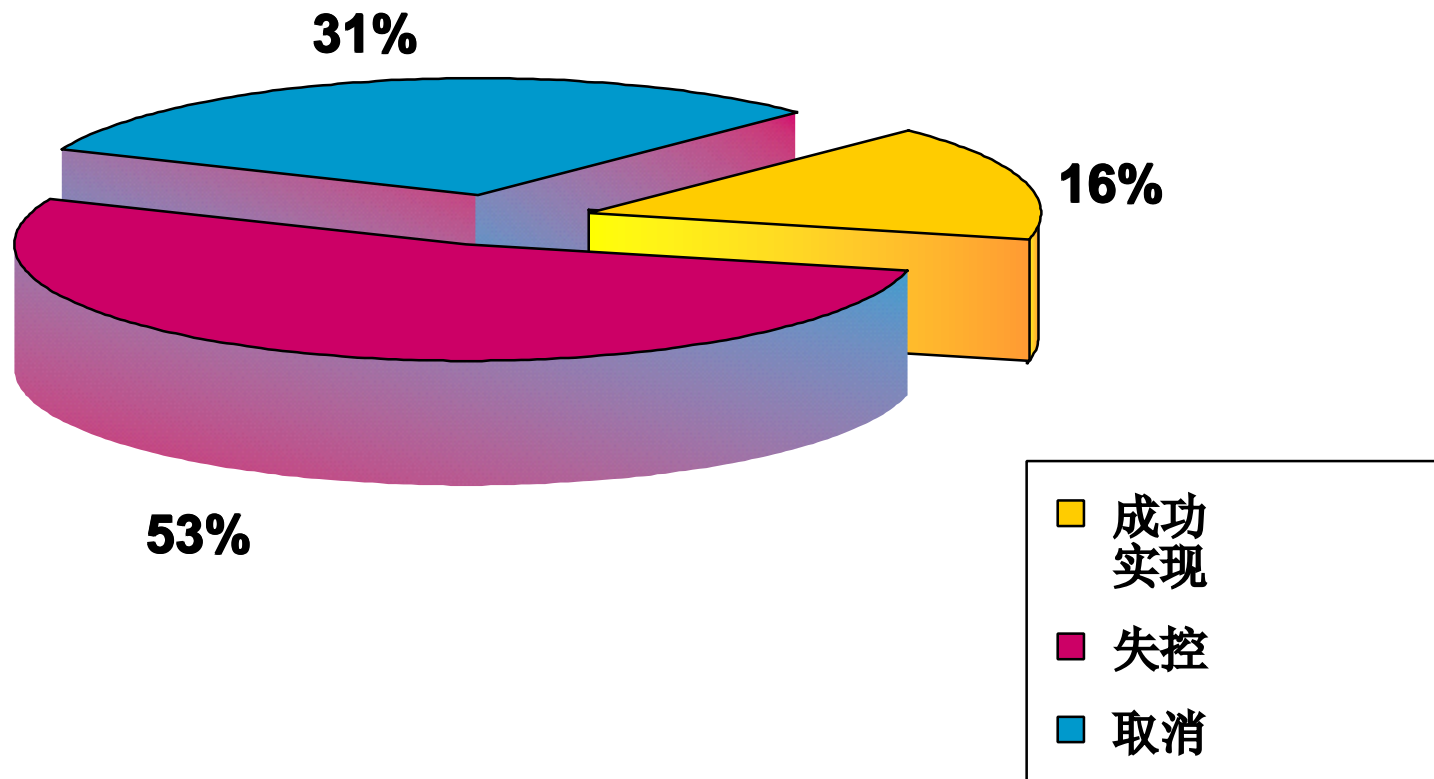
第二章

软件项目管理

本章要点

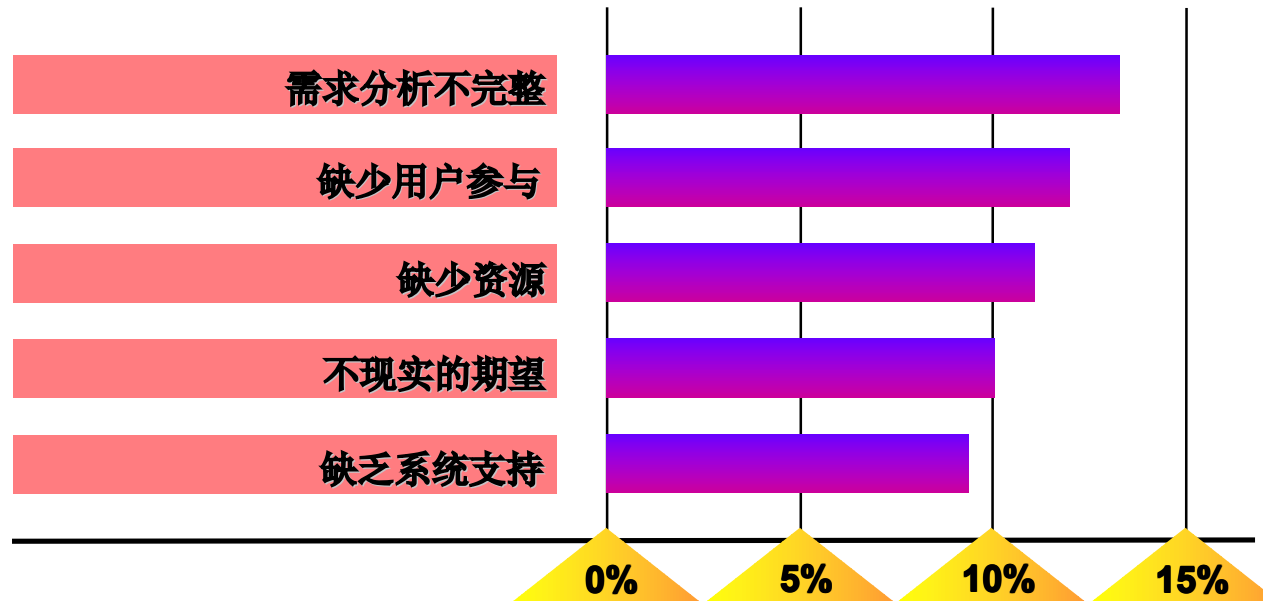
- 项目管理一般原理
- **Project 2002**中的项目管理概念
- 用**Project2002**做项目计划
- 关键路径、关键任务计算法则

成功的软件项目有多少？



为什么失败？

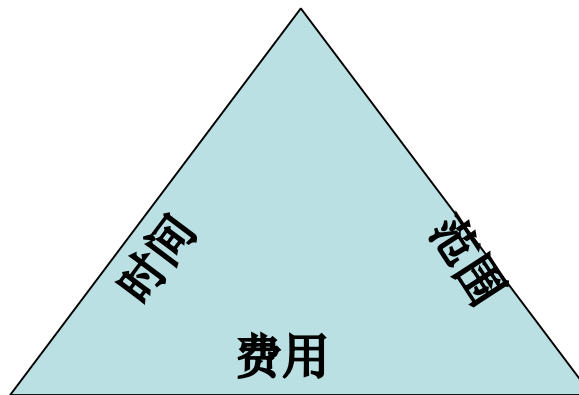
缺少项目管理



项目管理

- 项目管理的定义
- 项目管理分三个阶段：
 - 制定项目计划
 - 管理和跟踪项目
 - 结束项目

项目管理三角形



项目轮廓定义

- 目标
- 前提
- 限制
- 范围

项目计划要素

- 任务
- 任务相关性
- 工期
- 成本
- 资源

任务相关性

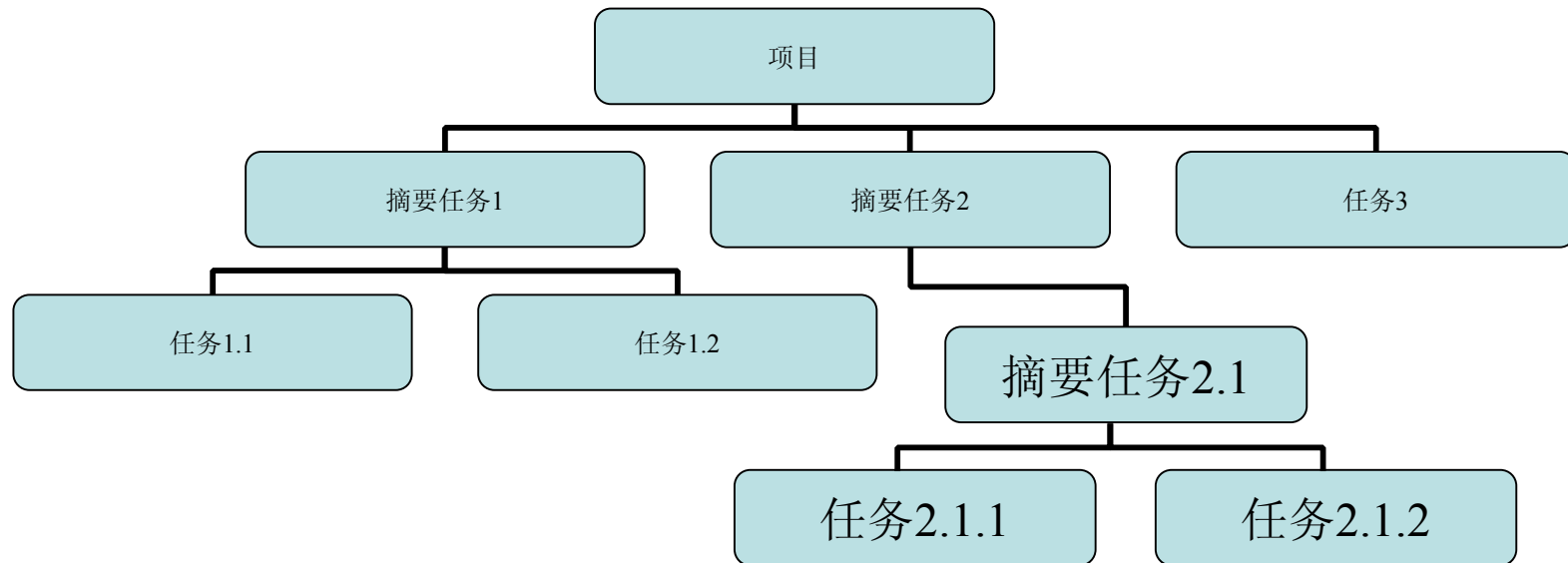
- 任务相关性：两个链接任务之间的关系；通过完成日期和开始日期之间的相关性进行链接。有四种任务相关性类型：“完成-开始”(FS)、“开始-开始”(SS)、“完成-完成”(FF)、“开始-完成”(SF)。
- 当关键任务完成或另一系列中的任务发生延迟时，关键路径就会更改。

任务相关性

| 任务相关性 | 描述 | 例子 |
|-----------|-----------------------|--------------------------|
| 完成-开始（FS） | 只有在任务 A 完成后任务 B 才能开始。 | 地基要先建好才能盖房子 |
| 开始-开始（SS） | 只有在任务 A 开始后任务 B 才能开始。 | 所有的人员都到齐后会议才能开始 |
| 完成-完成（FF） | 只有在任务 A 完成后任务 B 才能完成。 | 所有的资料全部准备齐全后才能结案 |
| 开始-完成（SF） | 只有在任务 A 开始后任务 B 才能完成。 | 站岗时，下一个站岗的人来了，原本站岗的人才能回去 |

工作分解结构

任务关系树型图



WBS代码列

| WBS | 任务名称 | 工期 | 开始时间 | 完成时间 | 前置任务 | 资源名称 |
|-----|-----------------------|-----------|------------|------------|------|------------|
| 0 | ☐ 软件开发 | 95.75 工作日 | 2004年1月1日 | 2004年5月13日 | | |
| 1 | ☐ 项目范围规划 | 3.5 工作日 | 2004年1月1日 | 2004年1月6日 | | |
| 1.1 | 确定项目范围 | 4 工时 | 2004年1月1日 | 2004年1月1日 | | 管理人员 |
| 1.2 | 获得项目所需资金 | 1 工作日 | 2004年1月1日 | 2004年1月2日 | 2 | 管理人员 |
| 1.3 | 定义预备资源 | 1 工作日 | 2004年1月2日 | 2004年1月5日 | 3 | 项目经理 |
| 1.4 | 获得核心资源 | 1 工作日 | 2004年1月5日 | 2004年1月6日 | 4 | 项目经理 |
| 1.5 | 项目范围规划完成 | 0 工作日 | 2004年1月6日 | 2004年1月6日 | 5 | |
| 2 | ☐ 分析/软件需求 | 14 工作日 | 2004年1月6日 | 2004年1月26日 | | |
| 2.1 | 行为需求分析 | 5 工作日 | 2004年1月6日 | 2004年1月13日 | 6 | 分析人员 |
| 2.2 | 起草初步的软件规范 | 3 工作日 | 2004年1月13日 | 2004年1月16日 | 8 | 分析人员 |
| 2.3 | 制定初步预算 | 2 工作日 | 2004年1月16日 | 2004年1月20日 | 9 | 项目经理 |
| 2.4 | 工作组共同审阅软件规范/预算 | 4 工时 | 2004年1月20日 | 2004年1月20日 | 10 | 项目经理, 分析人员 |
| 2.5 | 根据反馈修改软件规范 | 1 工作日 | 2004年1月21日 | 2004年1月21日 | 11 | 分析人员 |
| 2.6 | 确定交付期限 | 1 工作日 | 2004年1月22日 | 2004年1月22日 | 12 | 项目经理 |
| 2.7 | 获得开展后续工作的批准(概念、期限和预算) | 4 工时 | 2004年1月23日 | 2004年1月23日 | 13 | 管理人员, 项目经理 |
| 2.8 | 获得所需资源 | 1 工作日 | 2004年1月23日 | 2004年1月26日 | 14 | 项目经理 |
| 2.9 | 分析工作完成 | 0 工作日 | 2004年1月26日 | 2004年1月26日 | 15 | |

Project 中创建项目计划文档

- 新建项目文档
- 定义常规工作时间
- 添加分层任务
- 读取来自**Excel**的资料
- 添加资源
- 给任务配备资源

任务相关操作

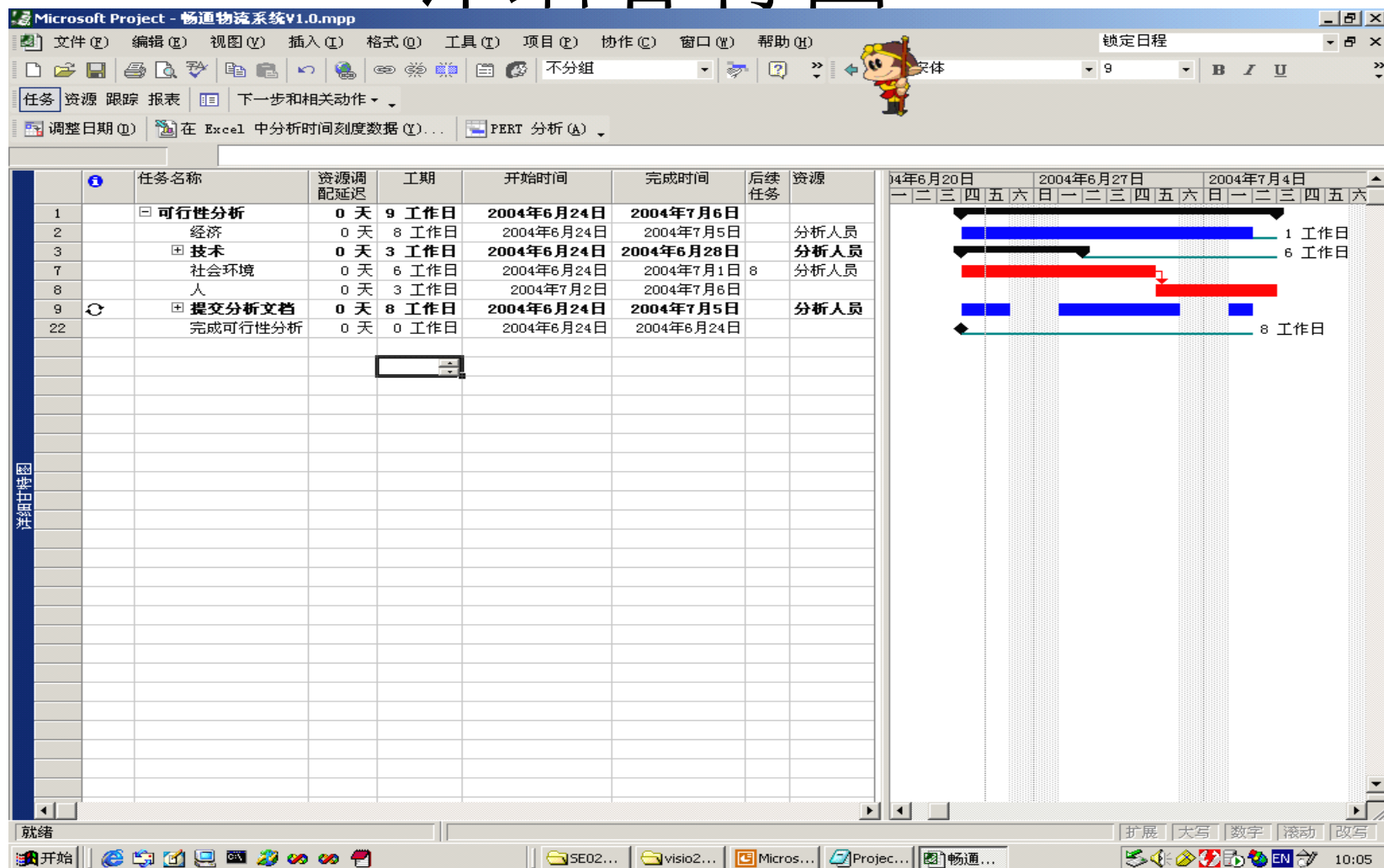
- 创建里程碑(0天)
- 创建周期性任务
- 创建和删除任务链接
- 创建任务相关性
- 设置任务限制

工时计算公式

工时=工期×单位（资源工作分配单位）
工期是完成任务所经历的实际时间

- 工时是资源执行任务的工作时间
- 单位是资源的分配量

- 全职工作人员的单位一般是**100%**
兼职工作人员的单位一般是**50%**



关键路径

- 是贯穿整个项目的一条路径，表明在限定的时间成功完成项目涉及的各任务间的依赖关系。
- 调整关键路径上任务的时间进度将会影响整个项目的交付时间。
- 关键路径方法（**CRM**）图是一种网络图，用于项目的进度控制和协调项目的活动和事件。

最早/晚完成日期

- 最早完成日期：根据前置任务和后续任务的最早完成日期、其他限制以及任何调配延迟，任务可能完成的最早日期。
即任务在开始日期和预计工期的基础上能够最早完成的日期。
- 最晚完成日期：在不延迟项目完成时间的情况下，任务可以完成的最晚日期。该日期基于任务最晚开始日期、前置任务和后续任务的最晚开始和完成日期及其他限制。
即任务在不延迟项目完成时间的情况下能够最晚完成的日期。

关键任务/时差

- 关键任务是指一旦延迟就会影响项目完成日期的任务。
- 时差：在不影响其他任务或项目完成日期的情况下，任务可以落后的时间量。
 - 可用时差是在不延迟其他任务的情况下，任务可以落后的时间量。
 - 总时差是在不延迟项目的情况下，任务可以落后的总时差。
- 在典型的项目中，很多任务都有时差，因此即使延迟一小段时间也不会影响其他任务或项目完成日期。

关键任务/时差 continue

当一项任务满足以下任何一个条件时，该任务即成为关键任务：

- 该任务没有时差。
- 该任务有“必须开始于”(MSO) 或“必须完成于”(MFO) 的日期限制（限制：对任务的开始日期或完成日期设置的限制。可以指定任务在特定日期开始，或者不晚于特定日期完成。限制可以是弹性的（未指定特定日期），也可以是非弹性的（指定了特定日期））。
- 在一个从开始日期排定的项目中，该任务有“越晚越好”(ALAP) 的限制。
- 在一个从完成日期排定的项目中，该任务有“越早越好”(ASAP) 的限制。
- 该任务的完成日期等同于或超出期限（期限：表明希望完成任务的目标日期。如果期限日期已过而任务仍未完成，*Project* 就会显示一个标记。）日期。
- 当一项任务标记为已完成时，就不再是关键任务，因为它不再影响后续任务的完成或项目完成日期。

关键路径显示项目的内容

- 通过了解和跟踪项目的关键路径和分配给关键任务的资源，就可以确定哪些任务会影响项目的完成日期以及项目能否按时完成。

任务节点表示

- 任务节点

| | | |
|--------|------|--------|
| 最早开始时间 | 持续时间 | 最早结束时间 |
| 任务标识号 | | 时差 |
| 最晚开始时间 | | 最晚结束时间 |

=最早结束时间-最早开始时间
=最晚结束时间-最晚开始时间

=最晚结束时间-最早结束时间
=最晚开始时间-最早开始时间

- 任务链接

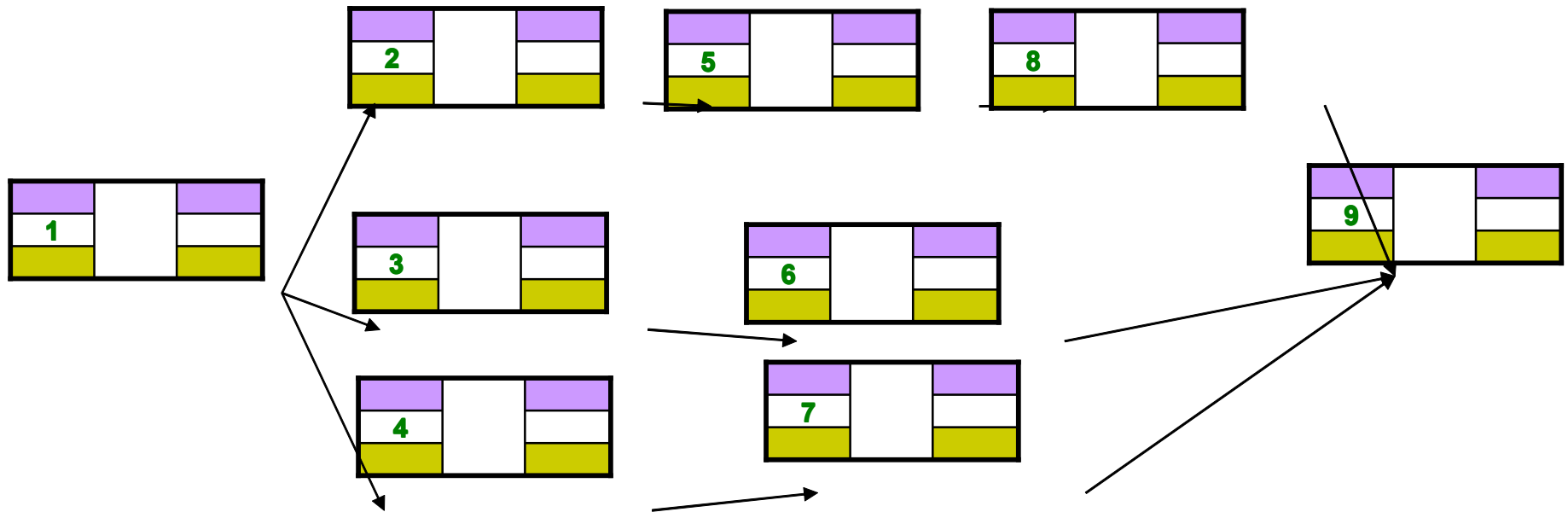
工期

| | | |
|----|---|----|
| 5 | 4 | 9 |
| 4 | | 21 |
| 26 | | 30 |

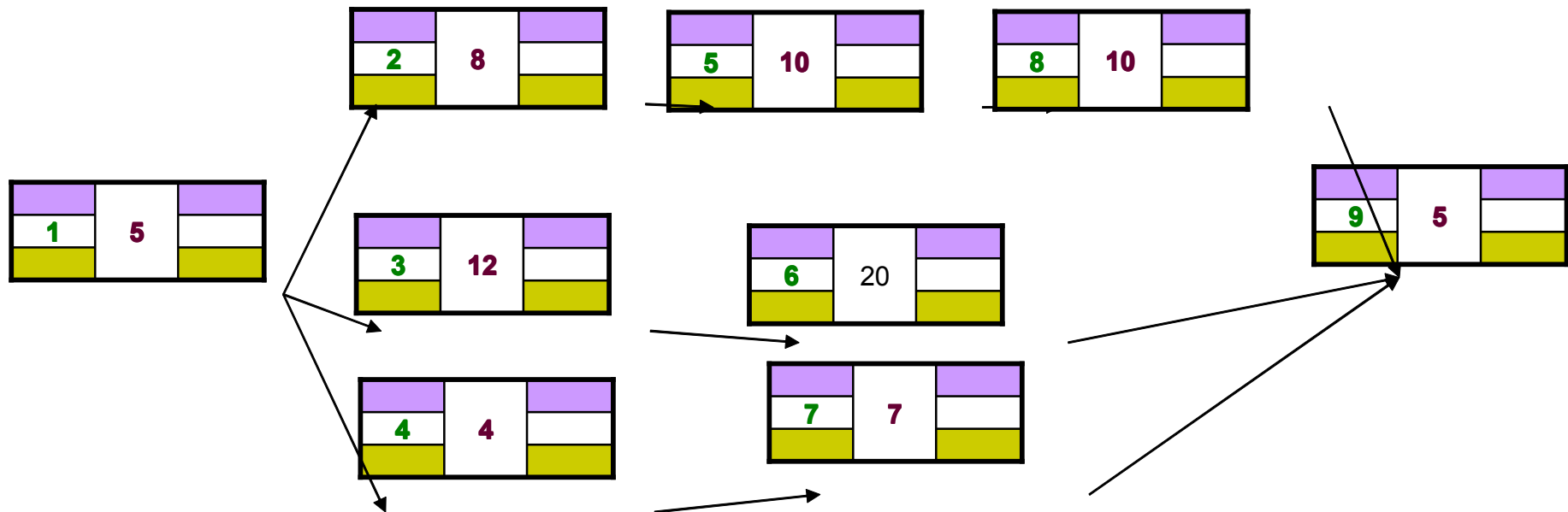
| | | |
|----|---|----|
| 9 | 7 | 16 |
| 7 | | 11 |
| 30 | | 37 |



网络初始状态

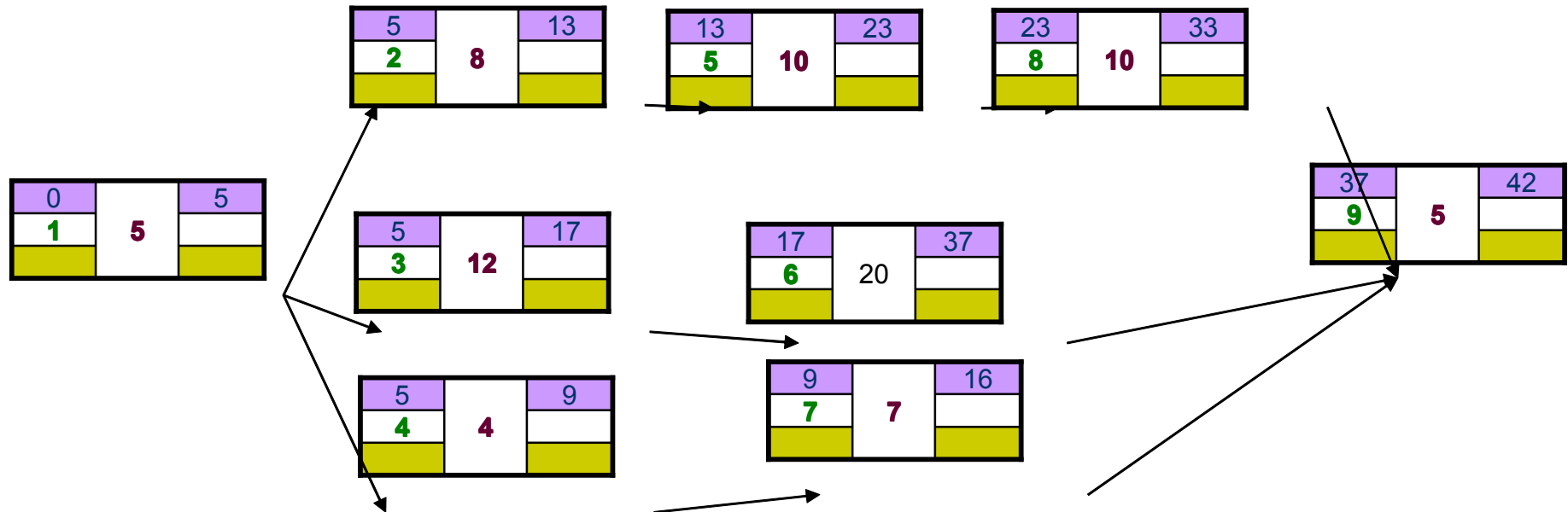


计算工期

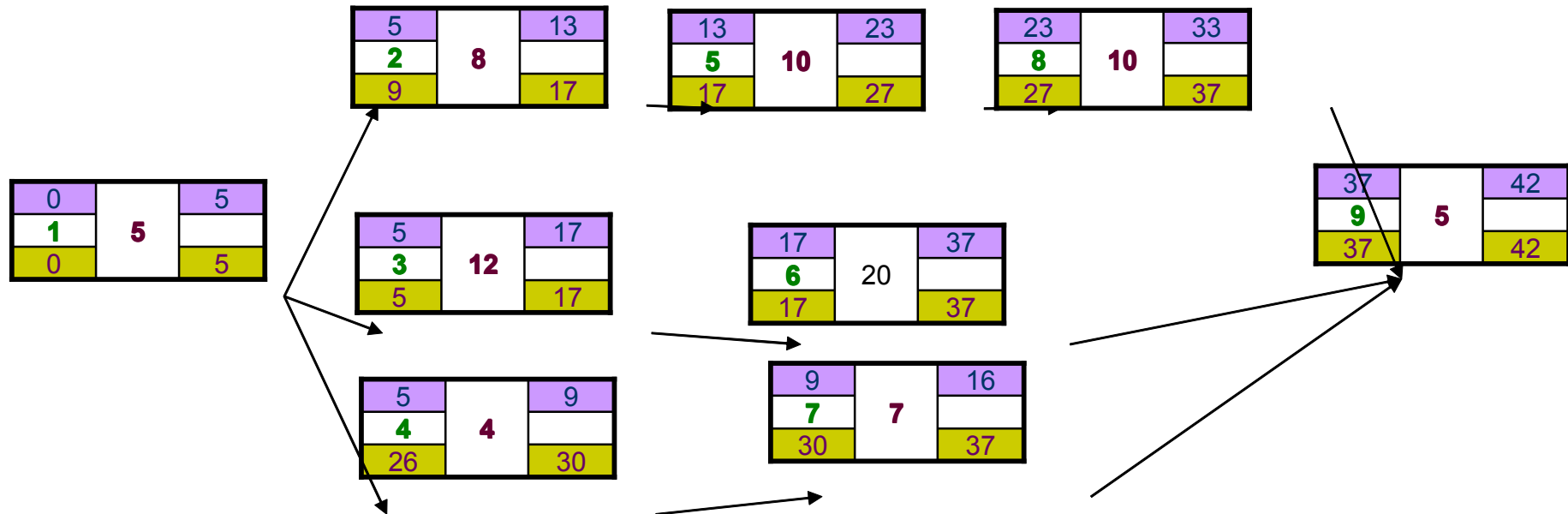


因为举例子，所以直接填写工期

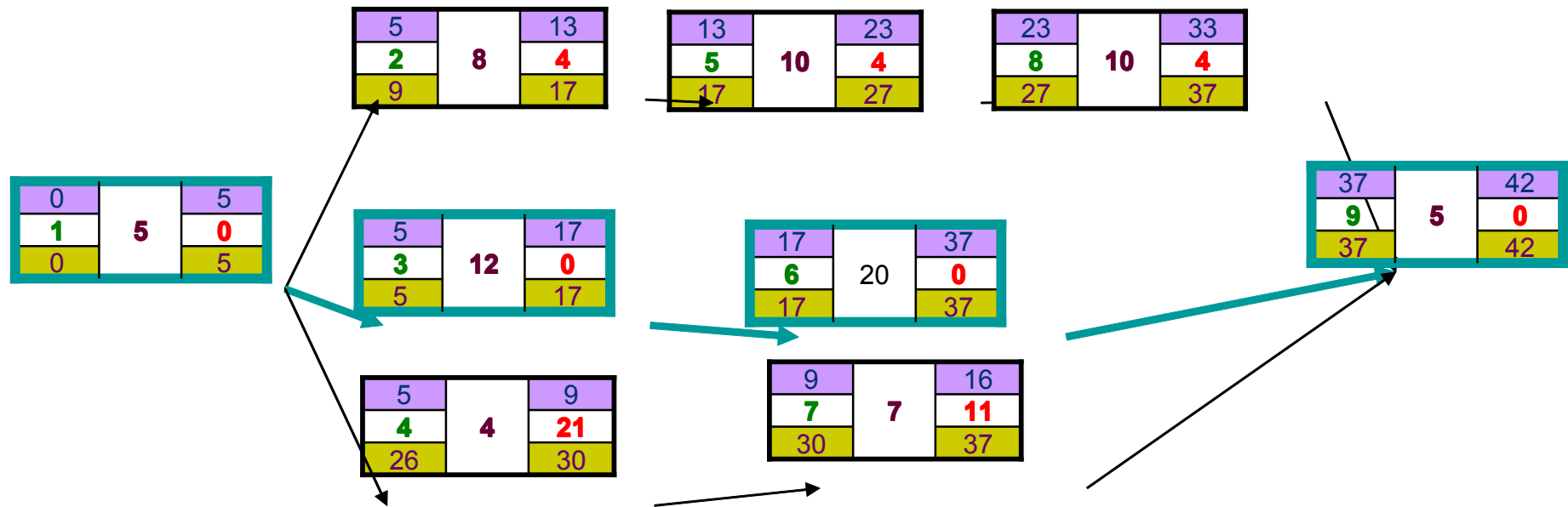
计算最早开始时间和最早结束时间



计算最晚开始时间和最晚结束时间

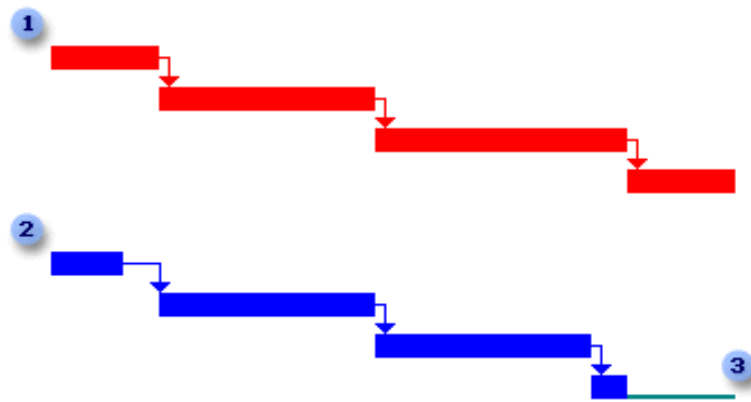


计算时差



完成这些任务最多用42单位工期
关键路径为1→3→6→9

关键任务实例



- (1)此任务序列没有时差，因而会推动项目的完成日期尽早到来。此序列中的所有任务都在关键路径上，称为关键任务。在“详细甘特图”视图中，关键任务显示为红色。
- (2)此任务序列不会推动项目的完成日期尽早到来，因此不是关键任务。在“详细甘特图”视图中，非关键任务显示为蓝色。
- (3)总时差是指此任务序列在不影响项目的完成日期的条件下可跳过的时间量。在“详细甘特图”视图中，总时差显示为浅蓝绿色线。

如果项目必须如期完成，应该密切关注关键路径上的任务以及为其分配的资源。这些要素将决定项目能否按时完成。

如何缩短关键路径

- 冲击：在不更改任务关系的情况下缩短项目的总体工期。对项目进行冲击通常需要为任务分配额外的资源。
- 缩短关键路径上任务的工期或工时。
- 更改任务限制，以增加日程排定的灵活性。
- 将关键任务分解成多个由不同资源同时进行的小任务。
- 更改任务相关性，以增加日程排定的灵活性。
- 在适用的相关任务间设置前置重叠时间。
- 排定加班工时。
- 为关键路径上的任务分配额外资源。

第三章

MSF介绍

本章要点

- MSF内容
- 组队模型
- 过程管理模型
- 应用程序模型

什么是MSF

- MSF(Microsoft Solution Framework)是指微软解决方案框架。
- MSF描述了微软公司从众多大小软件产品研发实践中总结的**管理软件开发过程**的经验

程序员眼中的MSF三个核心

- 组队模型
- 过程管理模型
- 应用程序模型

三种模型的关系



组队原则

- 清晰的责任，共同的职责
- 赋予小组成员权力
- 聚焦业务价值
- 共同的项目设想
- 保持灵活，预测变化
- 推动开放式沟通

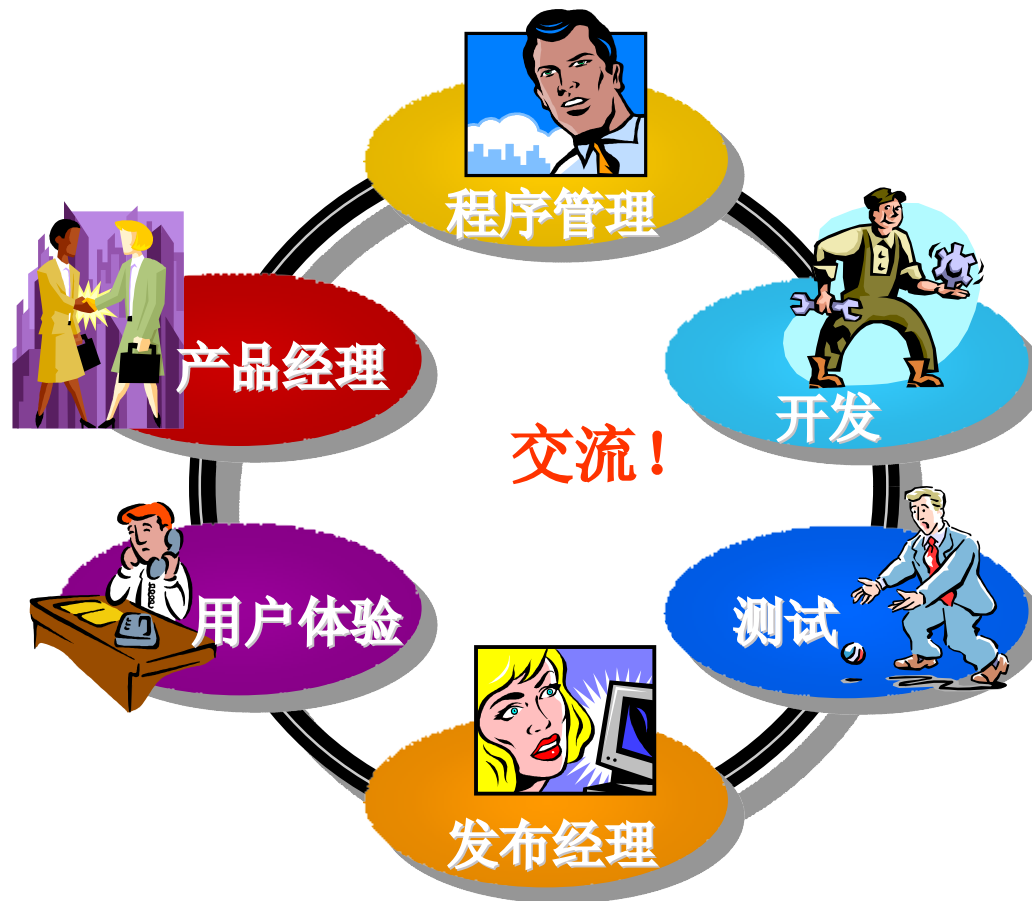
关键概念

- 同级小组
- 以客户为中心
- 产品理念体系
- 零缺陷
- 乐意学习
- 有动力的小组有效率

成功的做法

- 小型的多学科小组
- 一起工作
- 全员参与设计

组队模型



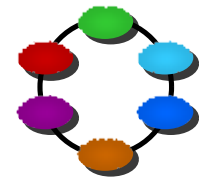
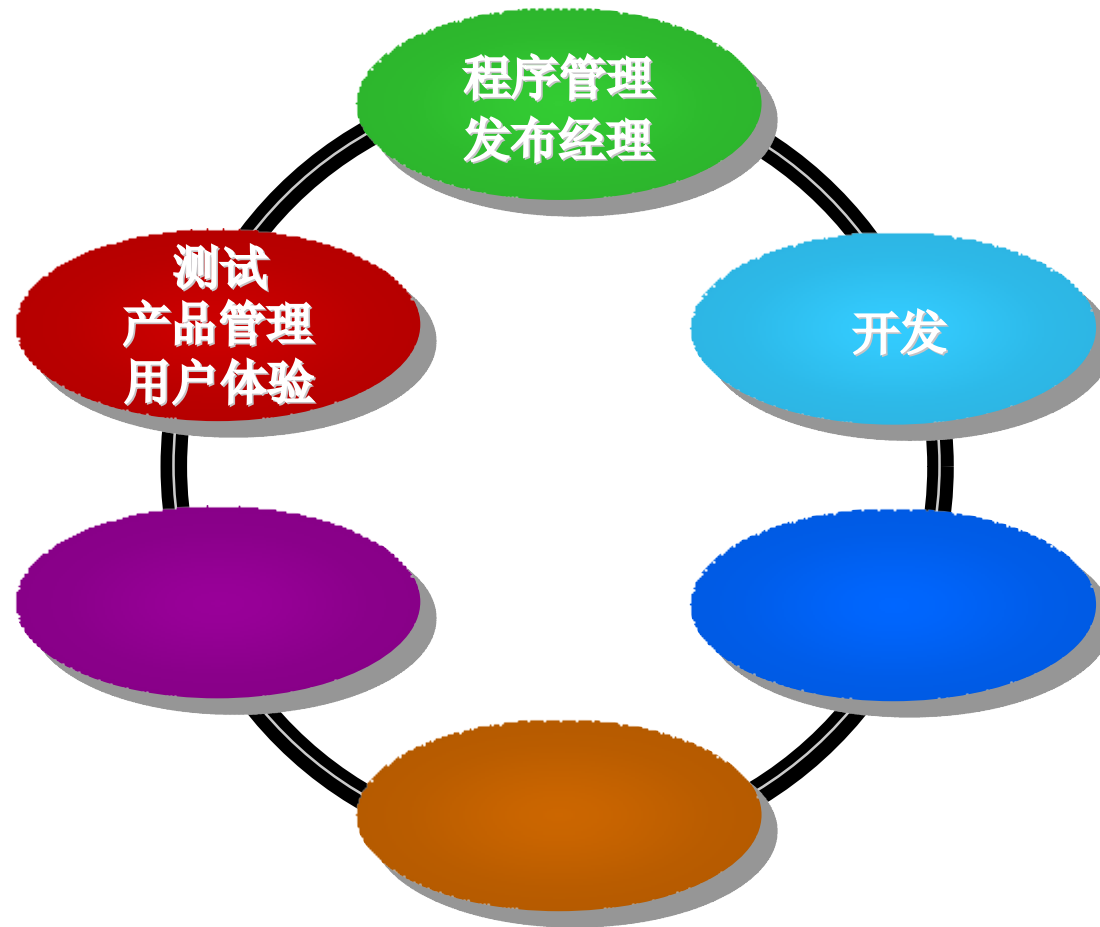
角色与目标

| 目标 | 组队角色 |
|----------------|------|
| 满足客户 | 产品经理 |
| 提高用户效率 | 用户体验 |
| 根据规格说明创建解决方案 | 开发 |
| 确保产品质量事宜被识别并处理 | 测试 |
| 交付满足项目约束的解决方案 | 程序管理 |
| 进行平滑的部署及日常运行 | 发布经理 |

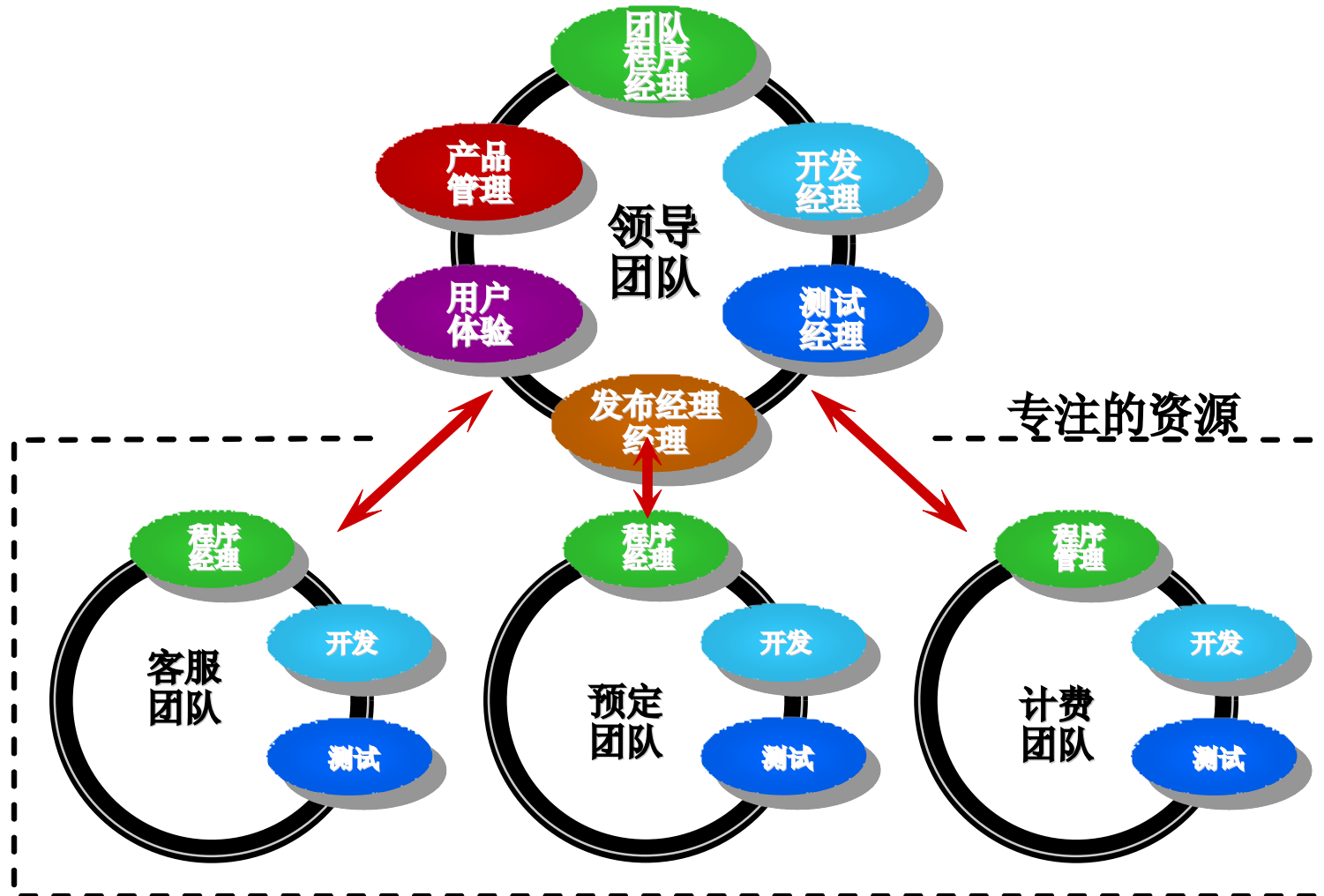
可合并角色表

| | 产品管理 | 程序管理 | 开发 | 测试 | 用户体验 | 发布经理 |
|------|------|------|----|----|------|------|
| 产品管理 | | N | N | P | P | U |
| 程序管理 | N | | N | U | U | P |
| 开发 | N | N | | N | N | N |
| 测试 | P | U | N | | P | P |
| 用户体验 | P | U | N | P | | U |
| 发布经理 | U | P | N | P | U | |

案例1：角色合并了的小团队



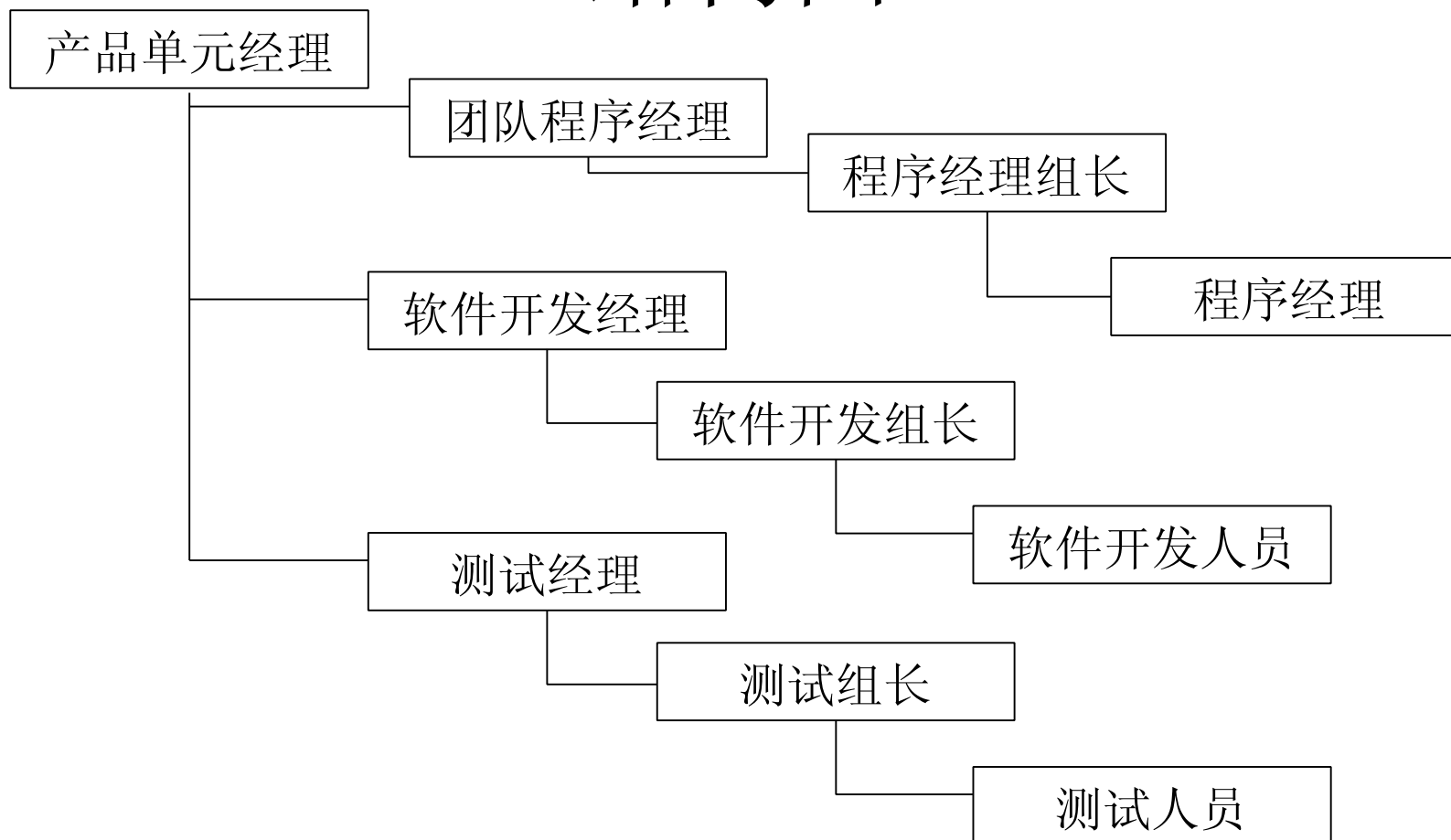
案例2：大项目的功能团队



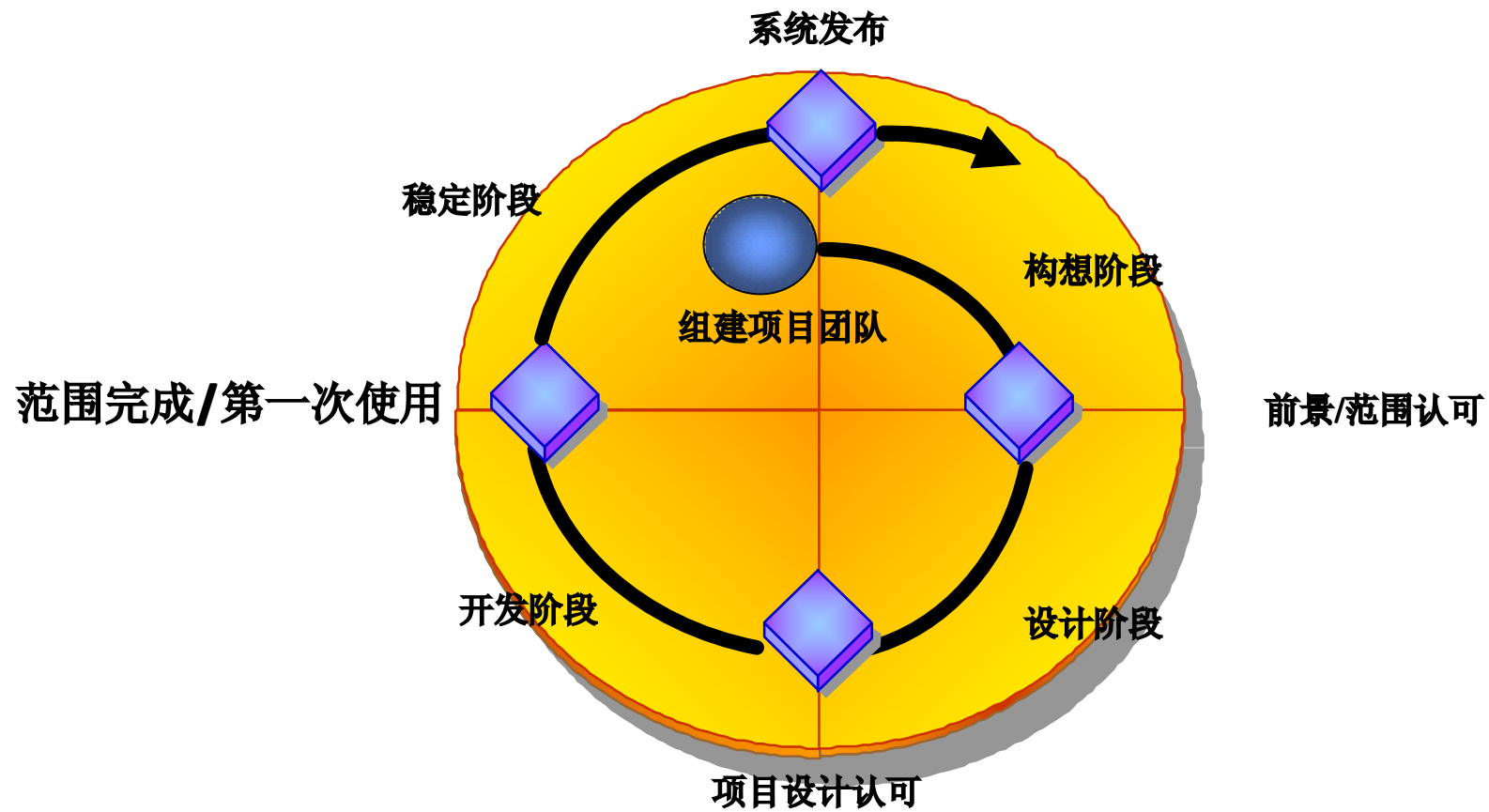
案例3：一个微软的开发小组示

- 一个项目经理
- 两个软件工程师
- 两个软件测试工程师

案例4：微软的项目产品组行政结构图



过程模型



构想阶段

- 定义初步的商业需求（持续性工作）
- 风险管理
- 定义项目结构
- 研究和收集设想
 - 进行初步的用户访问
 - 定义使用场合
 - 确立设计目标
 - 制定初步的解决方案概念
- 制定初步的项目范围
 - 定义关键的成功因素
 - 定义衡量成功标准
 - 定义主要的可交付结果（初步）
 - 确定构想/范围

设计阶段

- 创建功能描述
- 开发计划
- 测试计划
- 用户培训计划
- 后勤计划
- 产品管理计划
- 程序管理计划
- 合并项目计划

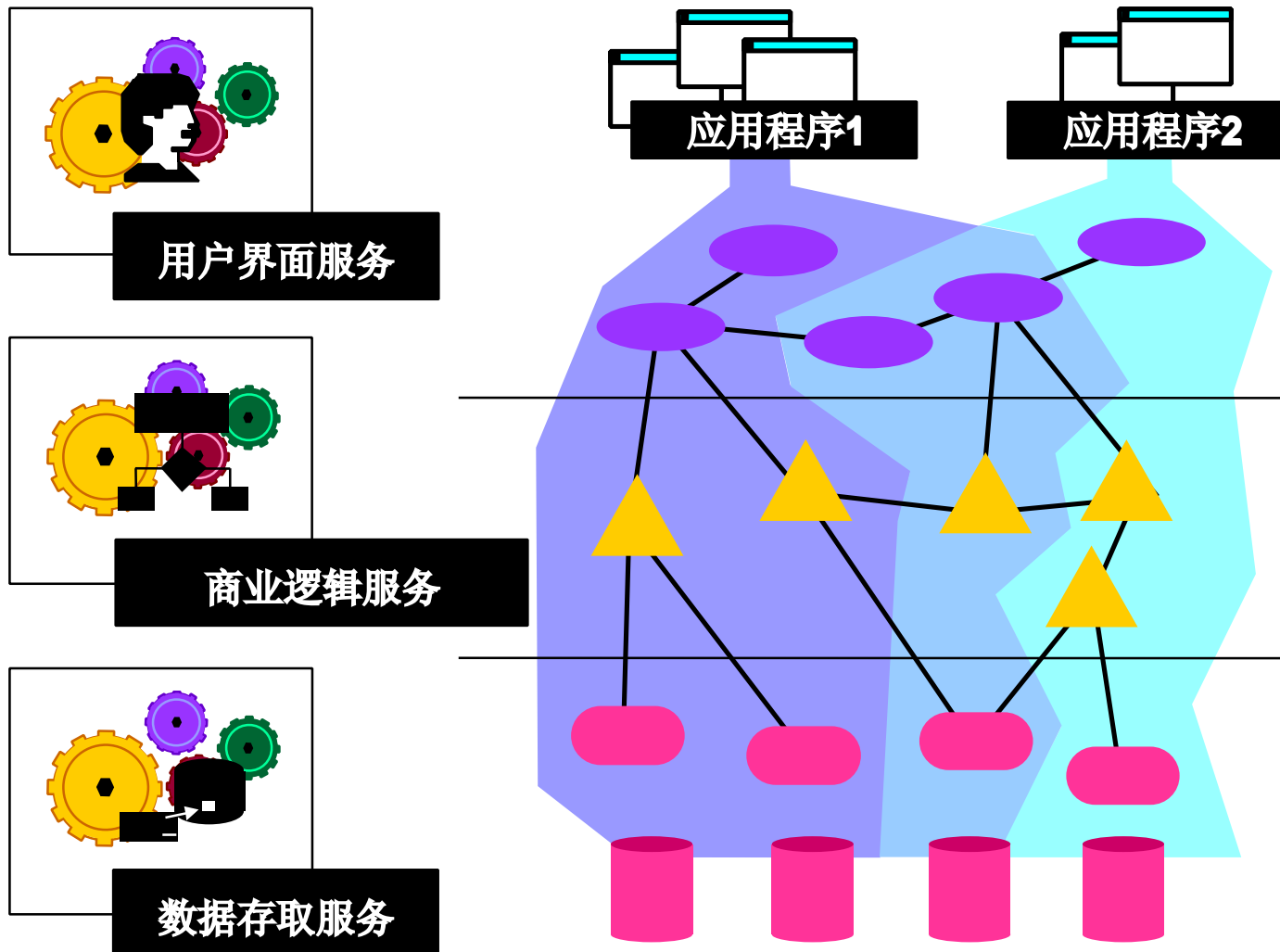
开发阶段

- 迭代开发一到多次的内部发布版
 - 开发目标组件
 - 测试单个组件
 - 测试组装为整体的应用程序
- 功能说明冻结
- 最后的特性开发
- 最后的后勤开发
- 最后的性能支持开发

稳定阶段

- 发布一到多个测试版，包括 α 测试版和 β 测试版
- 收集错误
- 改正高优先级的错误，发布无错误版
- 进行最后的错误分类
- 黄金发布版

应用程序模型



第四章

UML：系统结构

本章要点

- OOAD与UML
- 用例图
- 类图与类关系
- 组件图、包和部署图

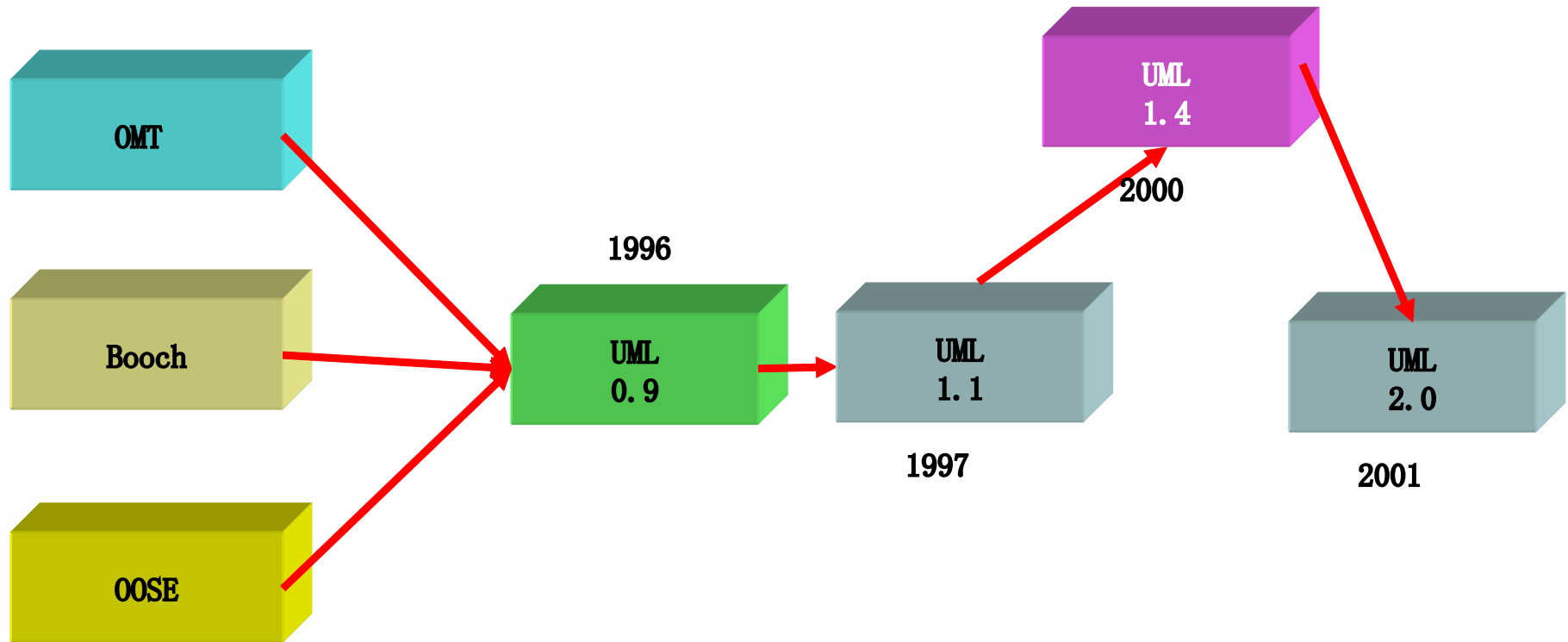
分析、设计、编程

- 面向对象分析(OOA)
- 面向对象设计(OOD)
- 面向对象编程(OOP)

UML诞生

- 面向对象分析和设计的结果需要统一符号来描述并记录。
- 统一建模语言——
UML（Unified Modeling Language）

UML发展

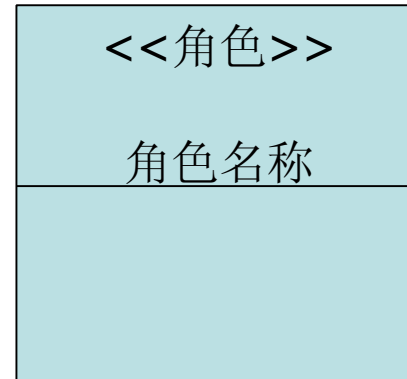
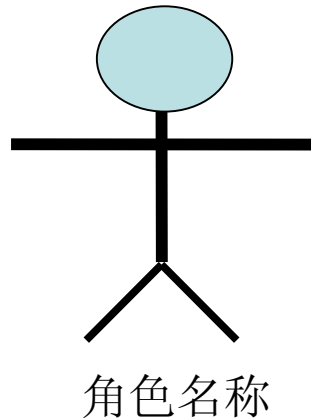


系统需求（学生注册）

- 管理员按照学校计划预先设置课程
- 学生可以通过该系统选择课程。
- 学生选好课程以后财务系统可以发费用清单给学生。
- 学生可以增删选择的课程。
- 教授可以获得自己课程的名单。
- 所有角色都必须经过登陆。

用例图图元（角色）

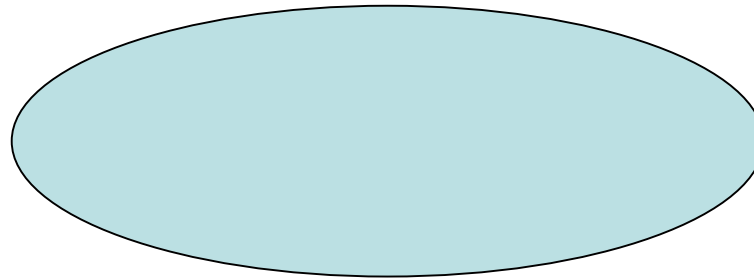
- 角色



有两种符号表示。

用例图图元（用例）

- 用例



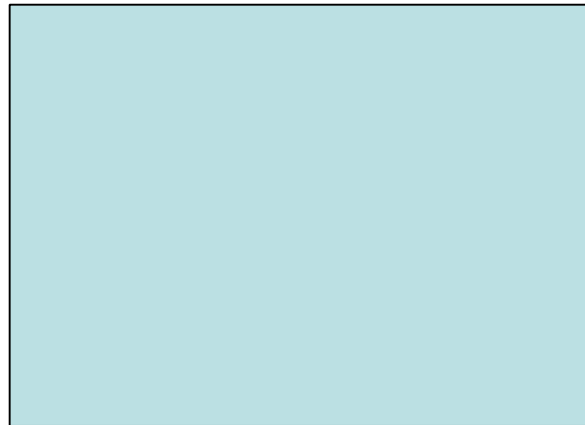
用例名称

实线椭圆表示用例，用例名称写在椭圆下面。

用例图图元（系统边界）

- 系统边界，系统边界可以省略

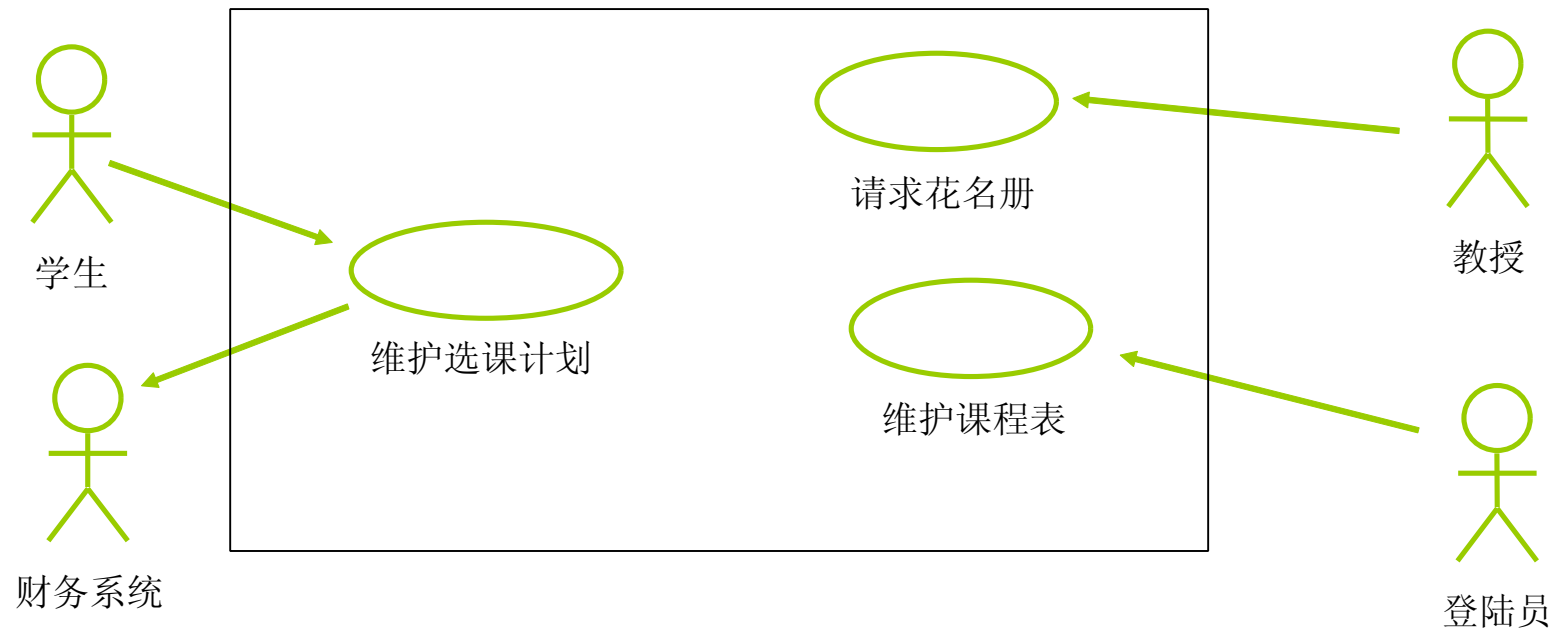
系统名称



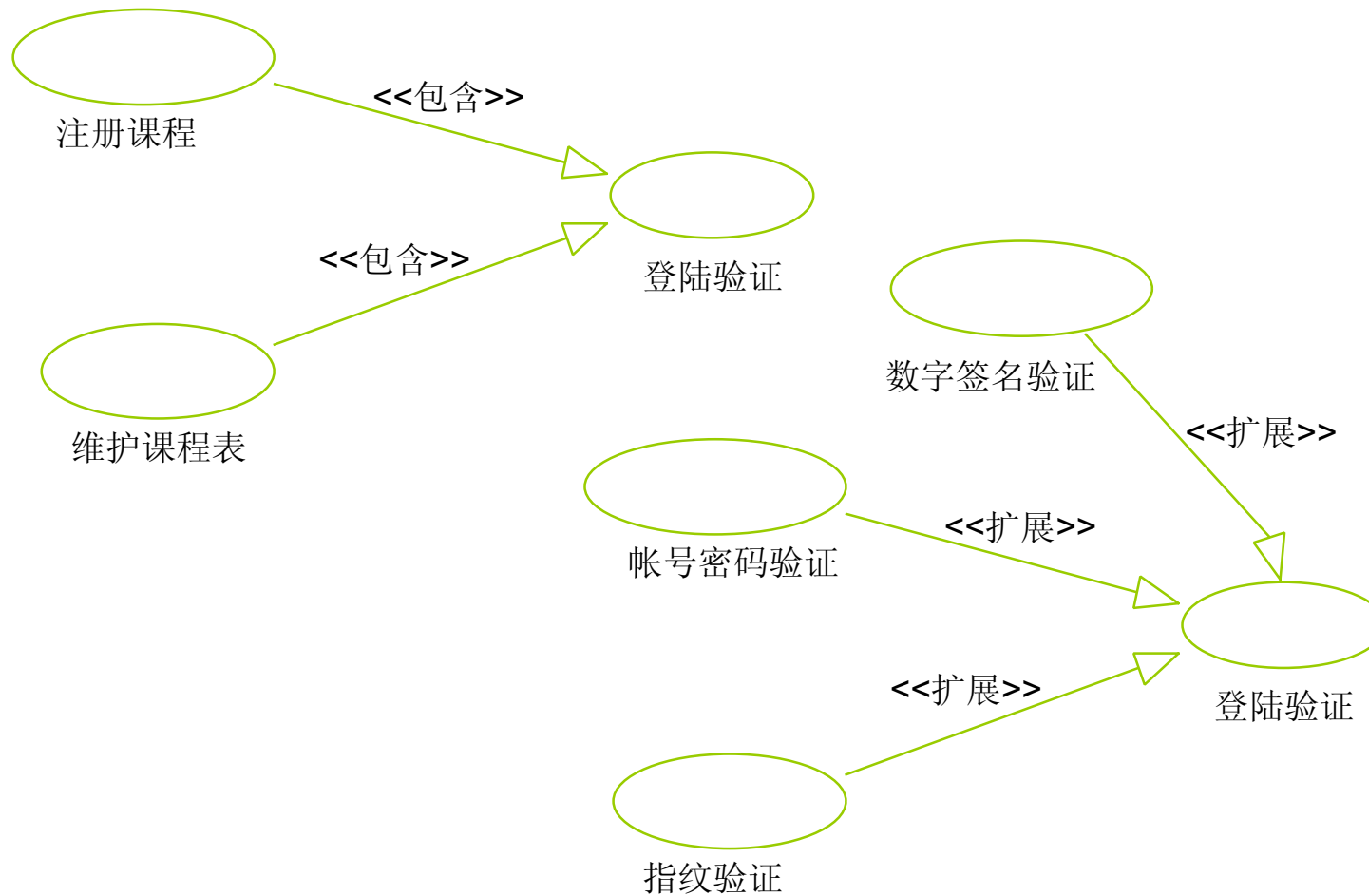
箭头表示参与

用例图

学生注册系统



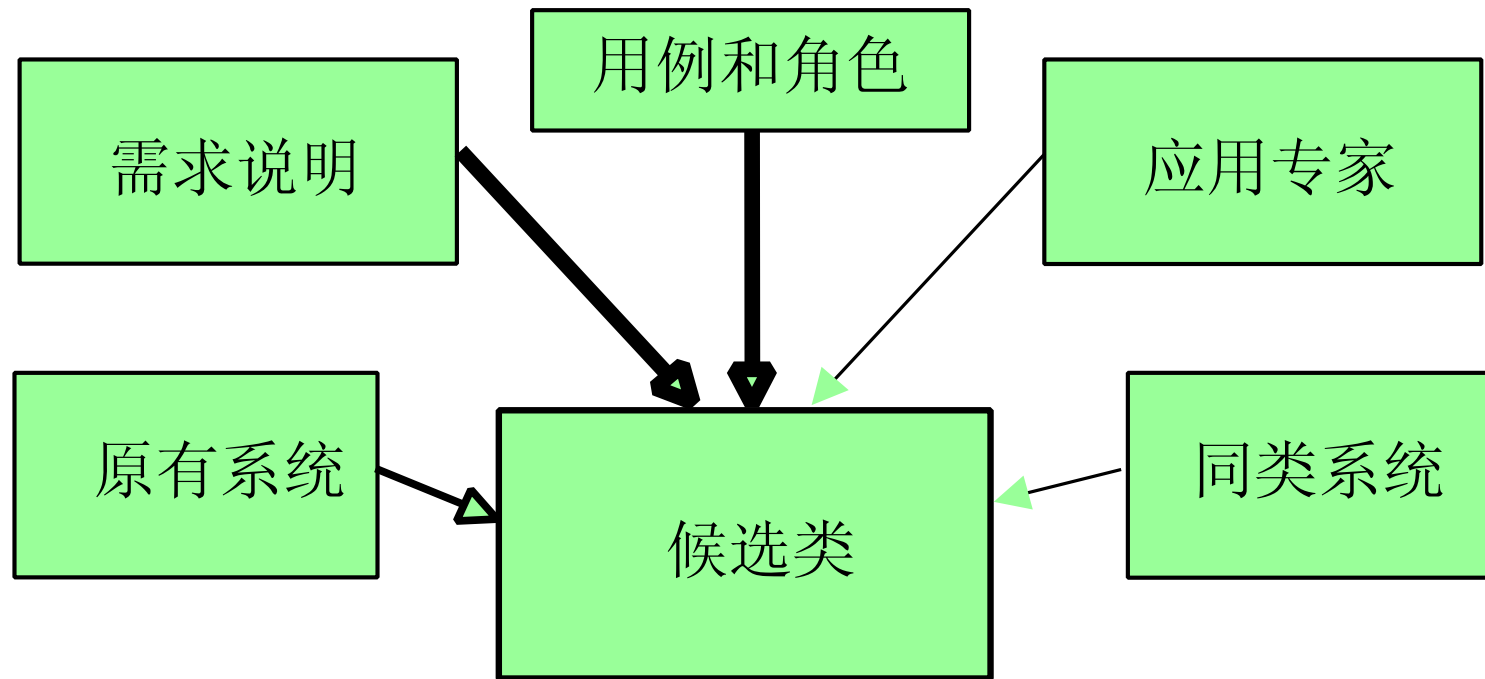
用例之间的关系



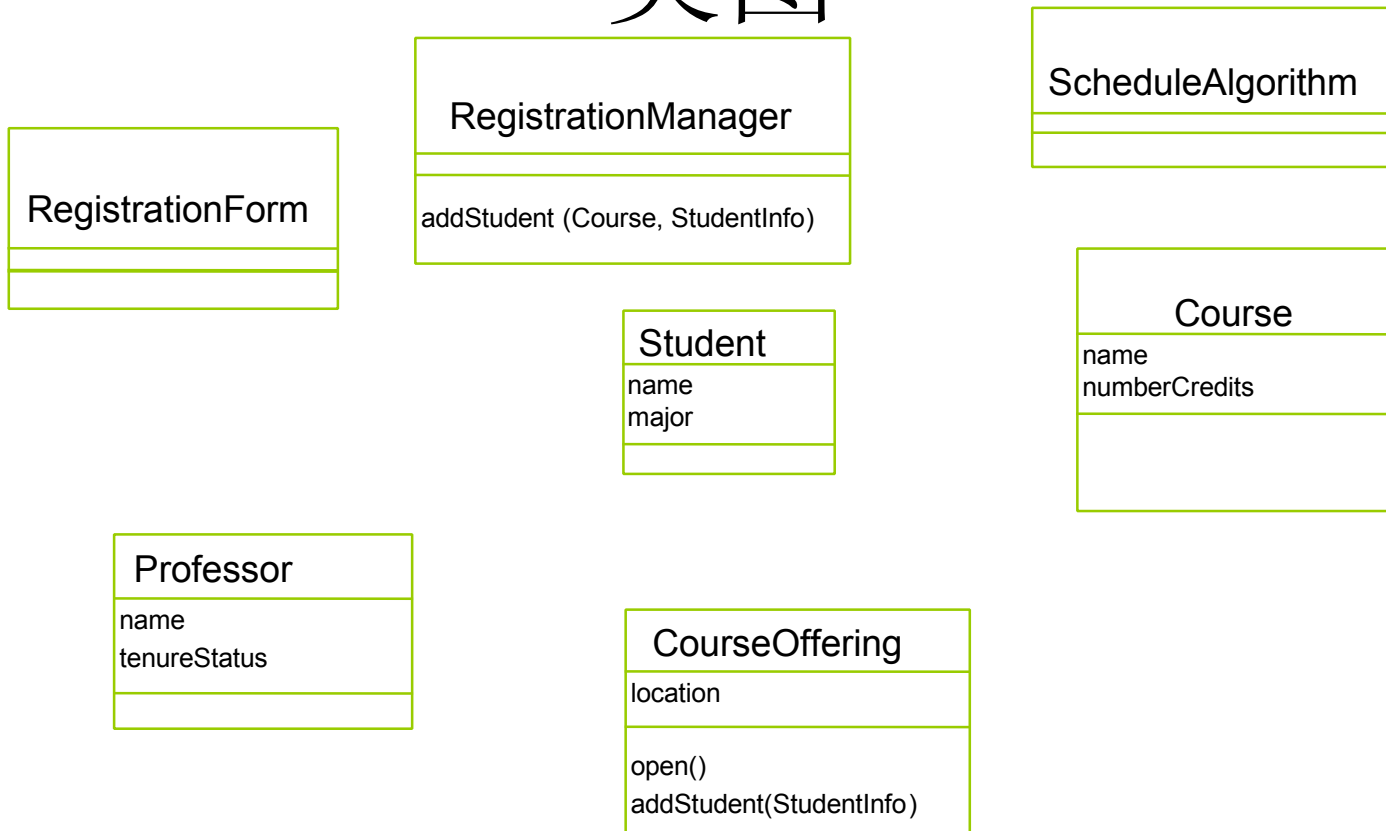
发现类

- 分两步
- 先发现候选类
- 然后净化候选类
- 净化原则：
- 摒弃冗余
- 通过继承合并重组类
- 提取接口或者抽象类重组类
- 通过拆分类，保证类对应的概念清晰

发现候选类



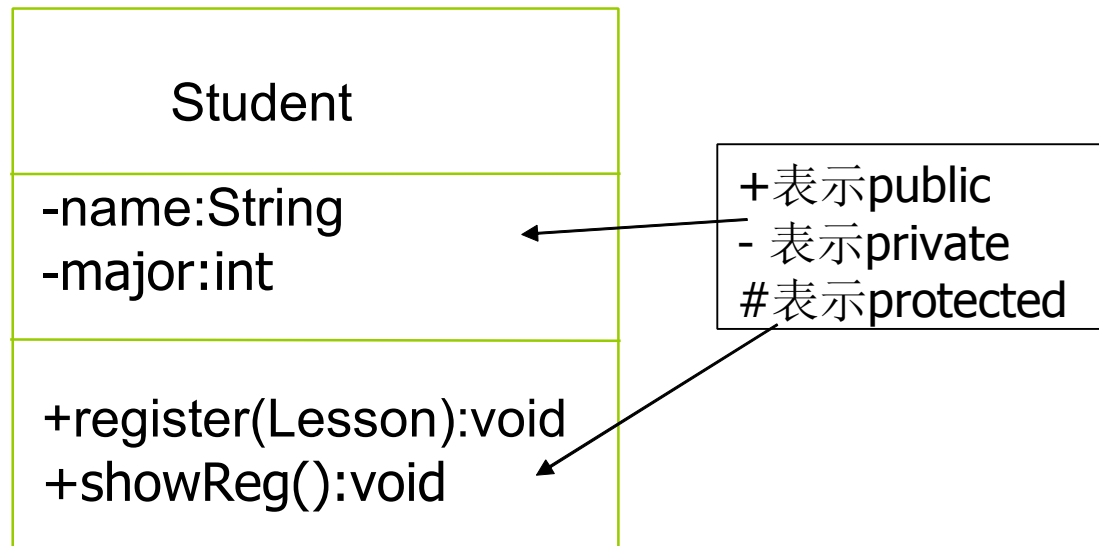
类图



类图

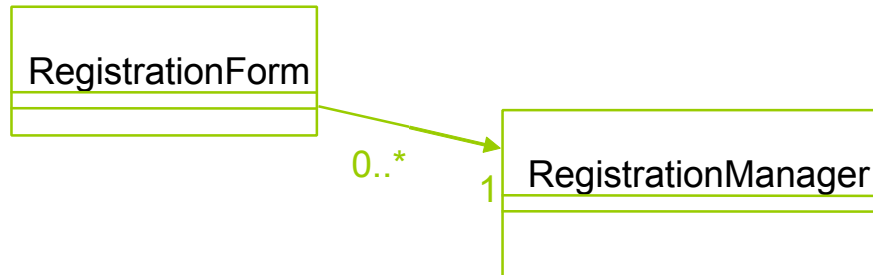
- 类图由水平线分割为三个部分

| |
|-----|
| 类名称 |
| 属性 |
| 方法 |

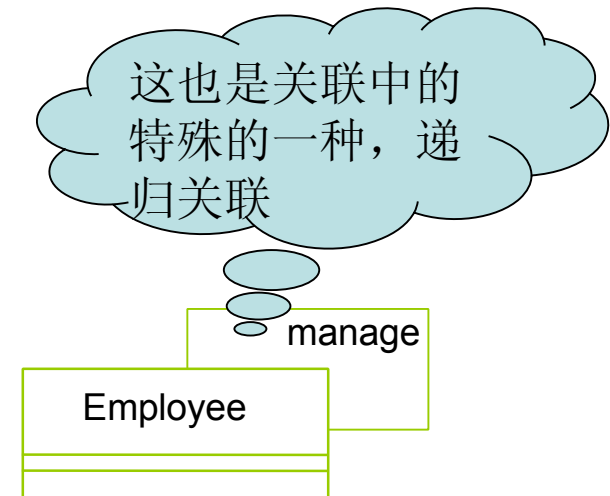


关联

- 关联分单向或双向

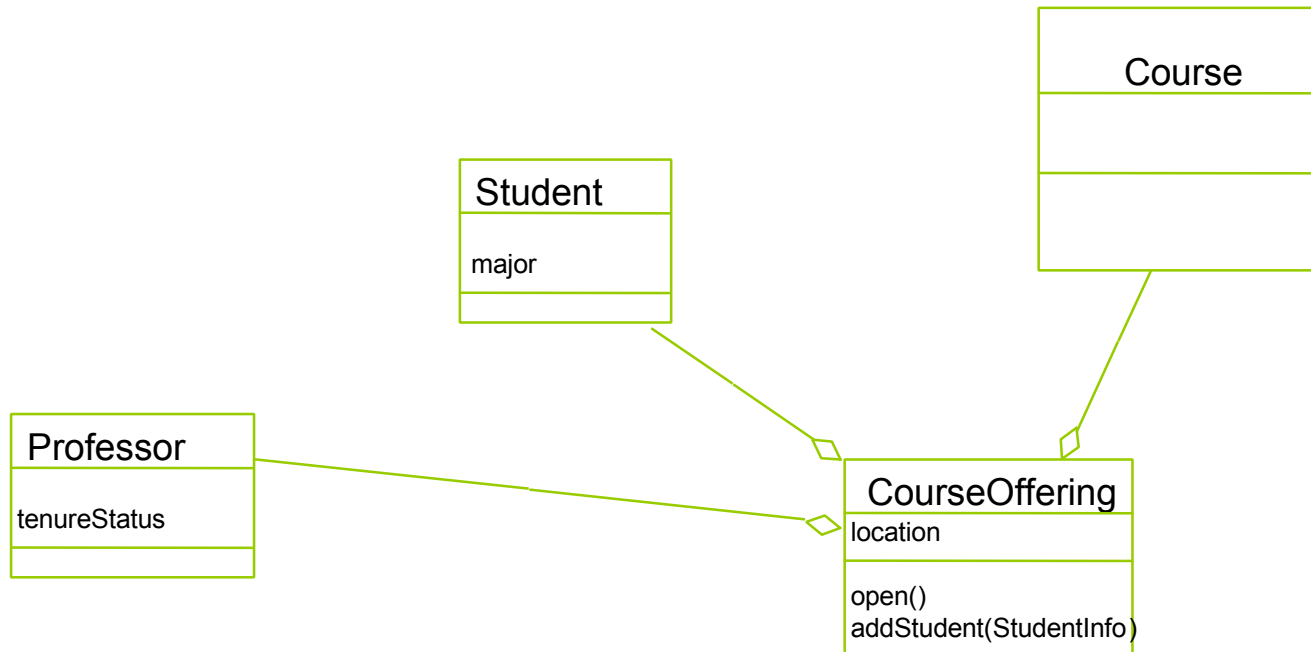


单向关联



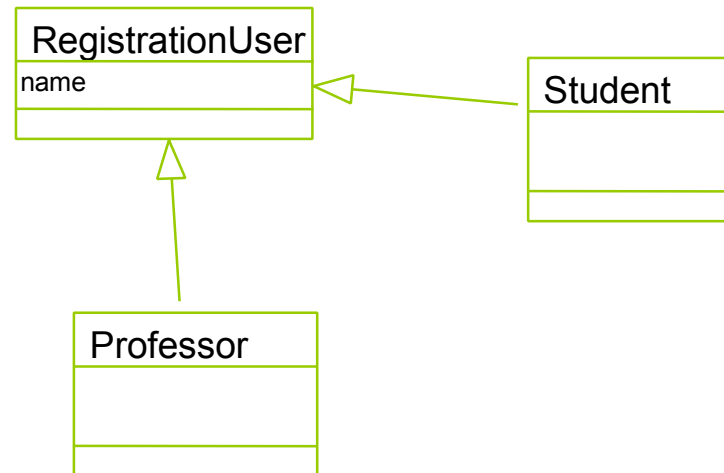
双向关联

聚合



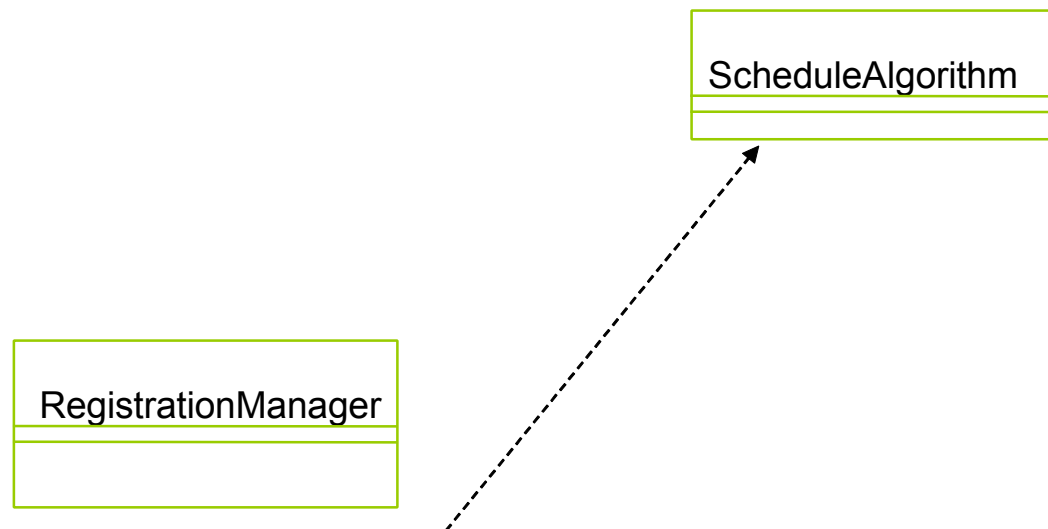
继承

- 学生和教授用《学生注册管理系统》的第一步要登陆。所以他们都有登录用户的含义

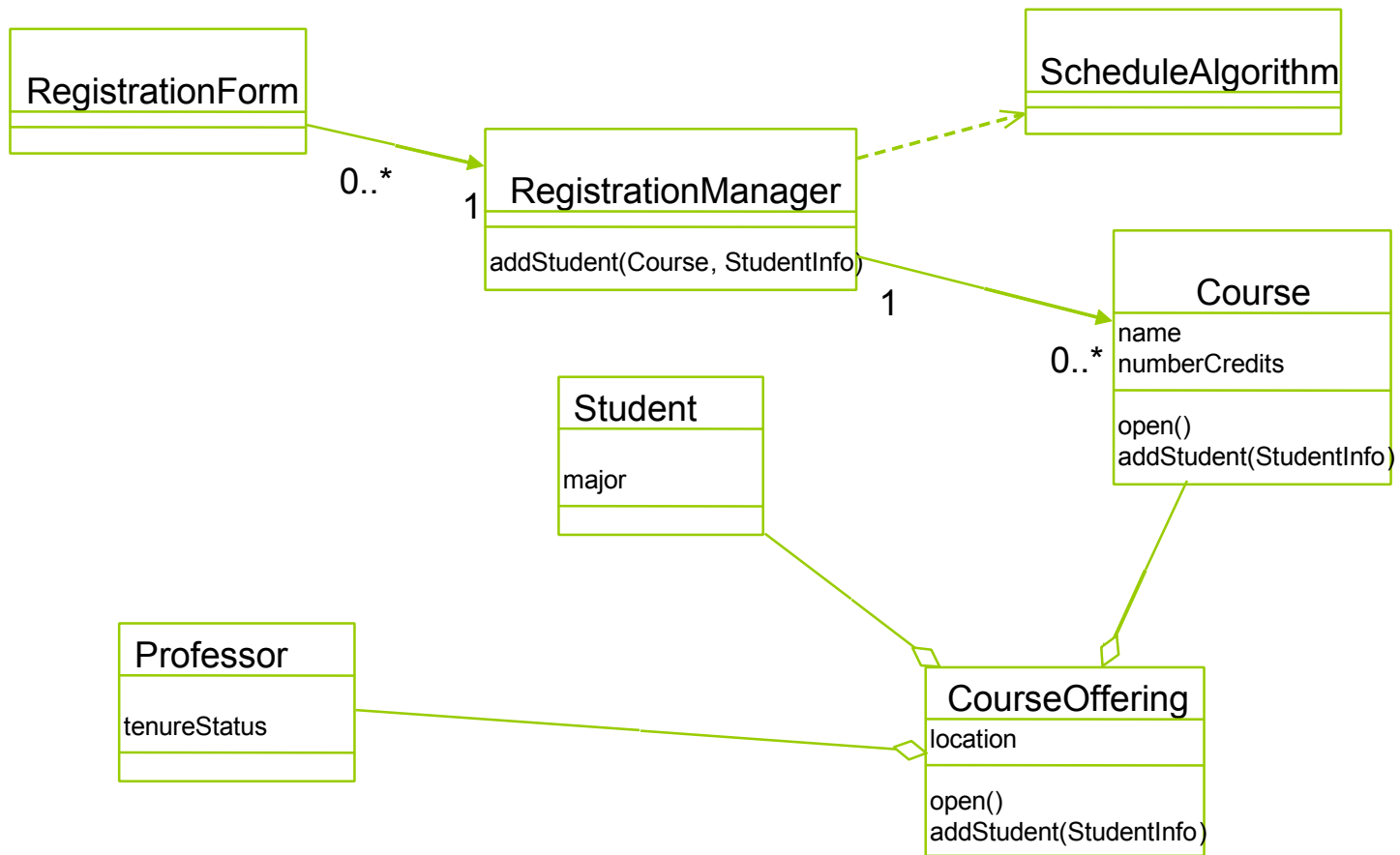


依赖关系

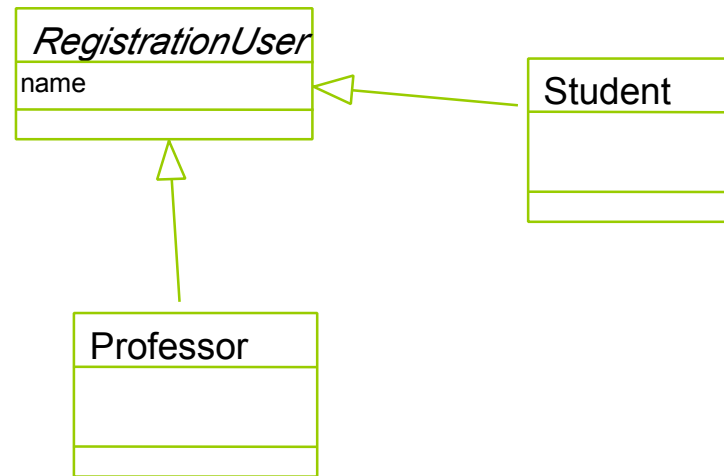
依赖关系表达了概念之间的依赖性。比如类R中需要调用类S的算法，因此表现为依赖，表示方法虚箭头线。



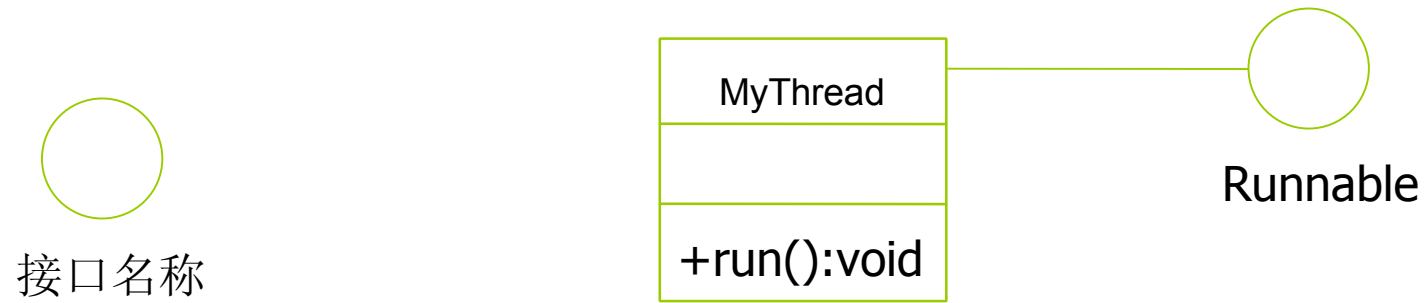
类关系图



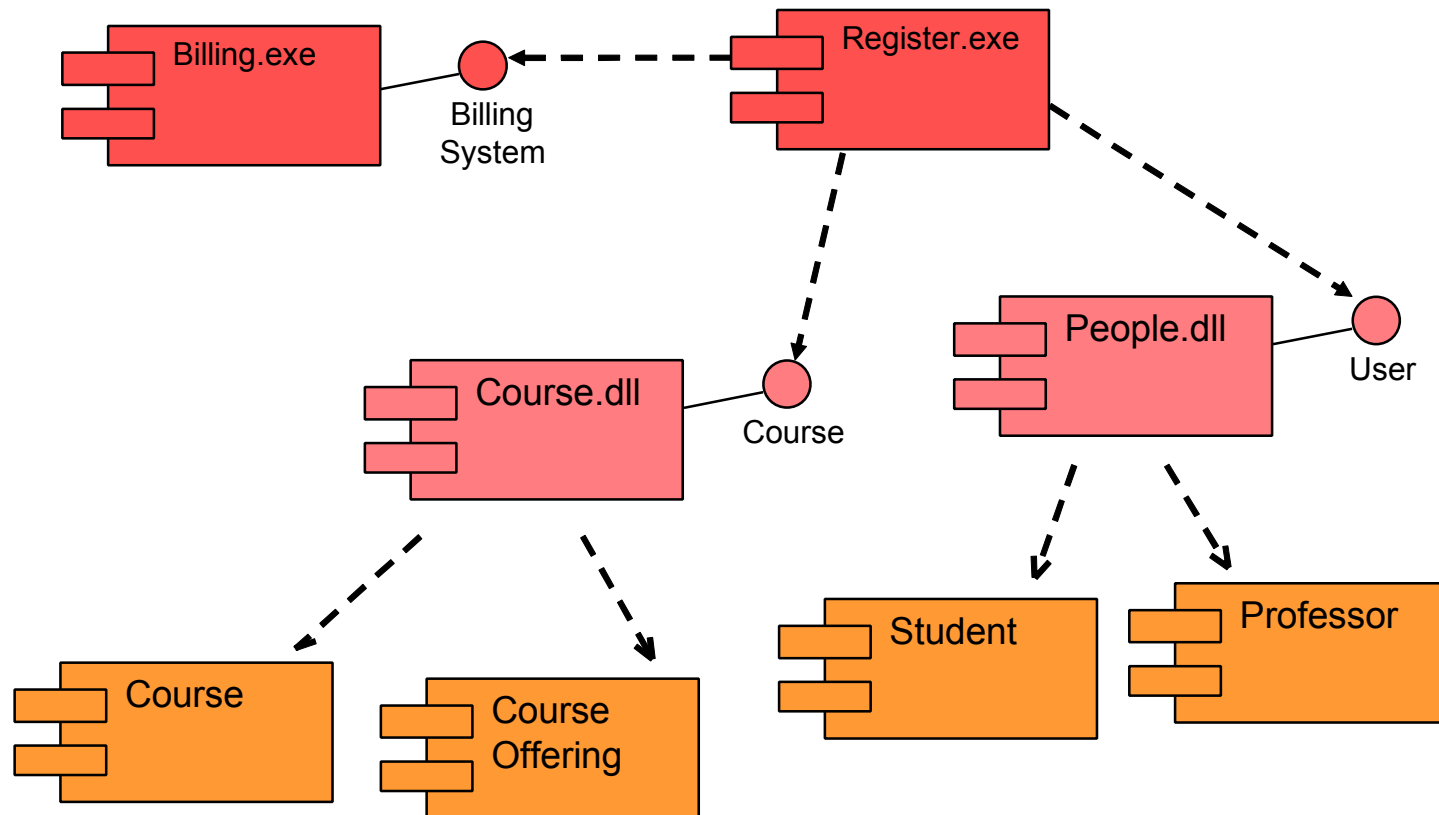
抽象类表示



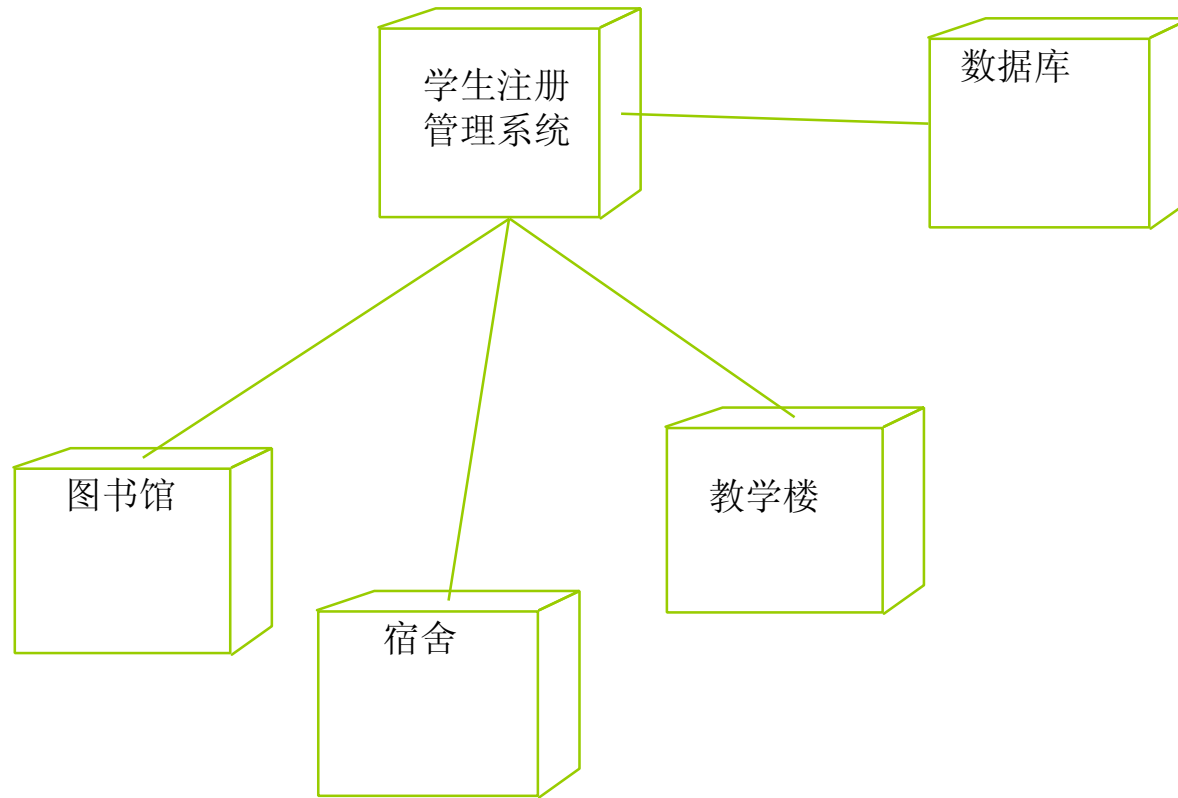
接口表示



组件图



部署图



第五章

UML：系统动态

本章要点

- 动态模型
- 序列图
- 协作图
- 状态图
- 活动图
- UML图分类

动态模型

- 动态模型描述系统随时间演化细节
- 描述系统动态的UML图有：
 - 交互图
 - 序列图
 - 协作图
 - 演化图
 - 状态图
 - 活动图






交互图

- 序列图
 - 描述了对对象之间的交互，重点在于描述消息及其时间顺序
- 协作图
 - 和序列图一样描述了对对象之间的交互，但是重点在于描述消息及其实现

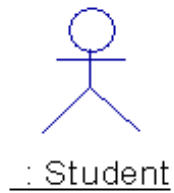
交互图的关键→消息

- 对象之间的的信息传递就是所谓的消息发送
- 消息就是对象调用另外一个对象的方法时所传递的参数
- 发送者和接收者之间的箭头表示消息

消息的类型

- 简单 
- 同步 
- 异步 
- 计时消息 
- 返回消息 

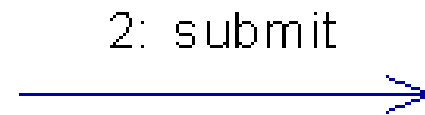
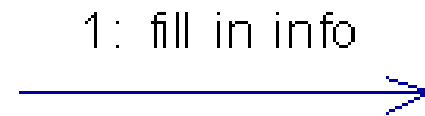
序列图元素



角色类对象



普通对象

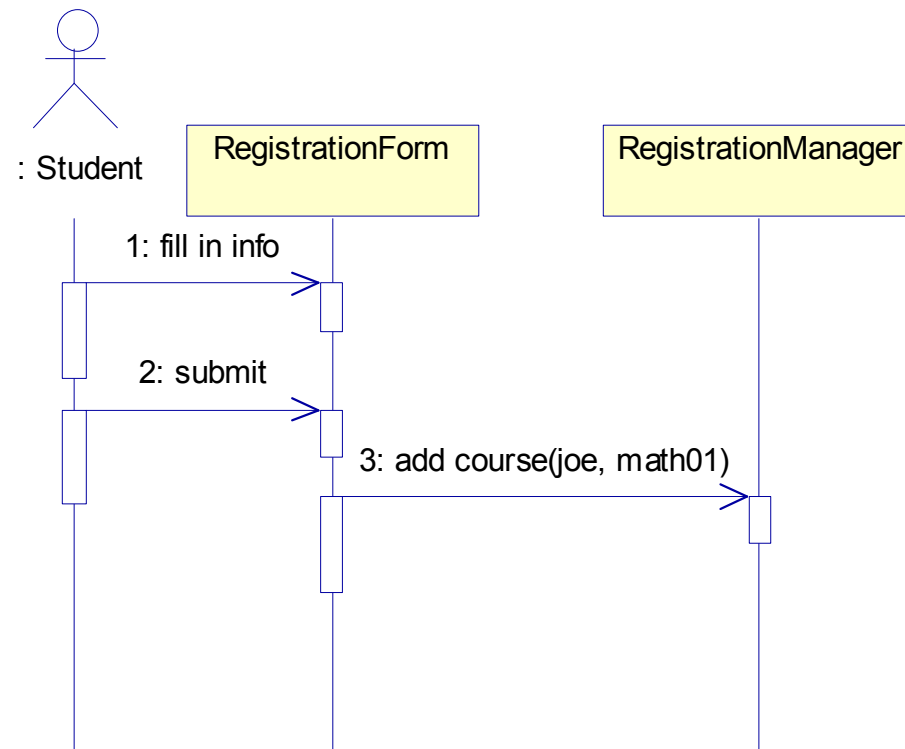


带有消息编号
的先后消息



对象的
生存期

序列图案例



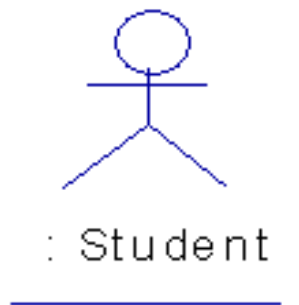
序列图组成

- 两个方向的扩展
- 消息穿梭在序列图中
- 一系列对象
- 消息编号

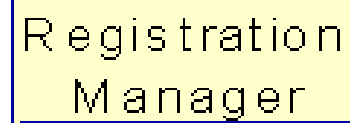
绘制序列图

- 相邻摆放
- 生存期要精确
- 顺序和编号
- 系统运行时的微观图

协作图元素

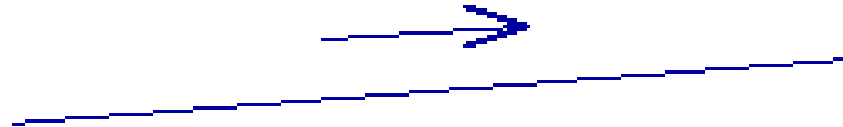


角色类对象



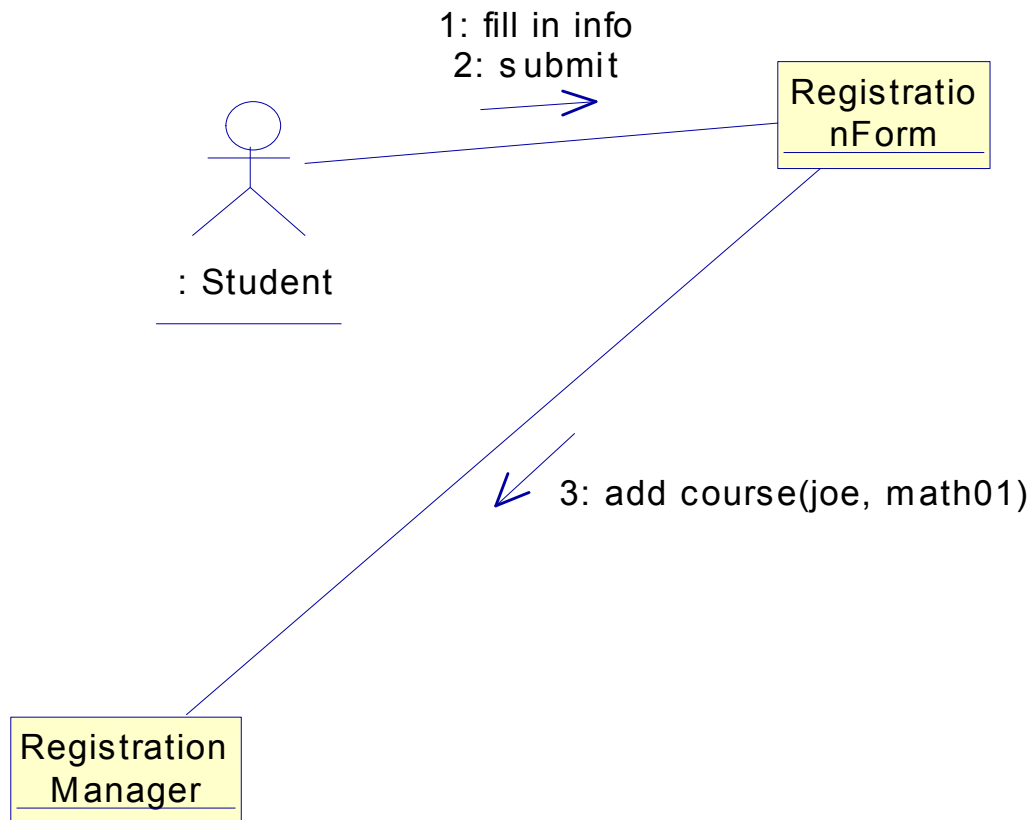
普通对象

1: fill in info
2: submit



带有编号的消息传递

协作图案例



协作图组成

- 对象节点
- 消息链接
- 网络布局
- 消息编号

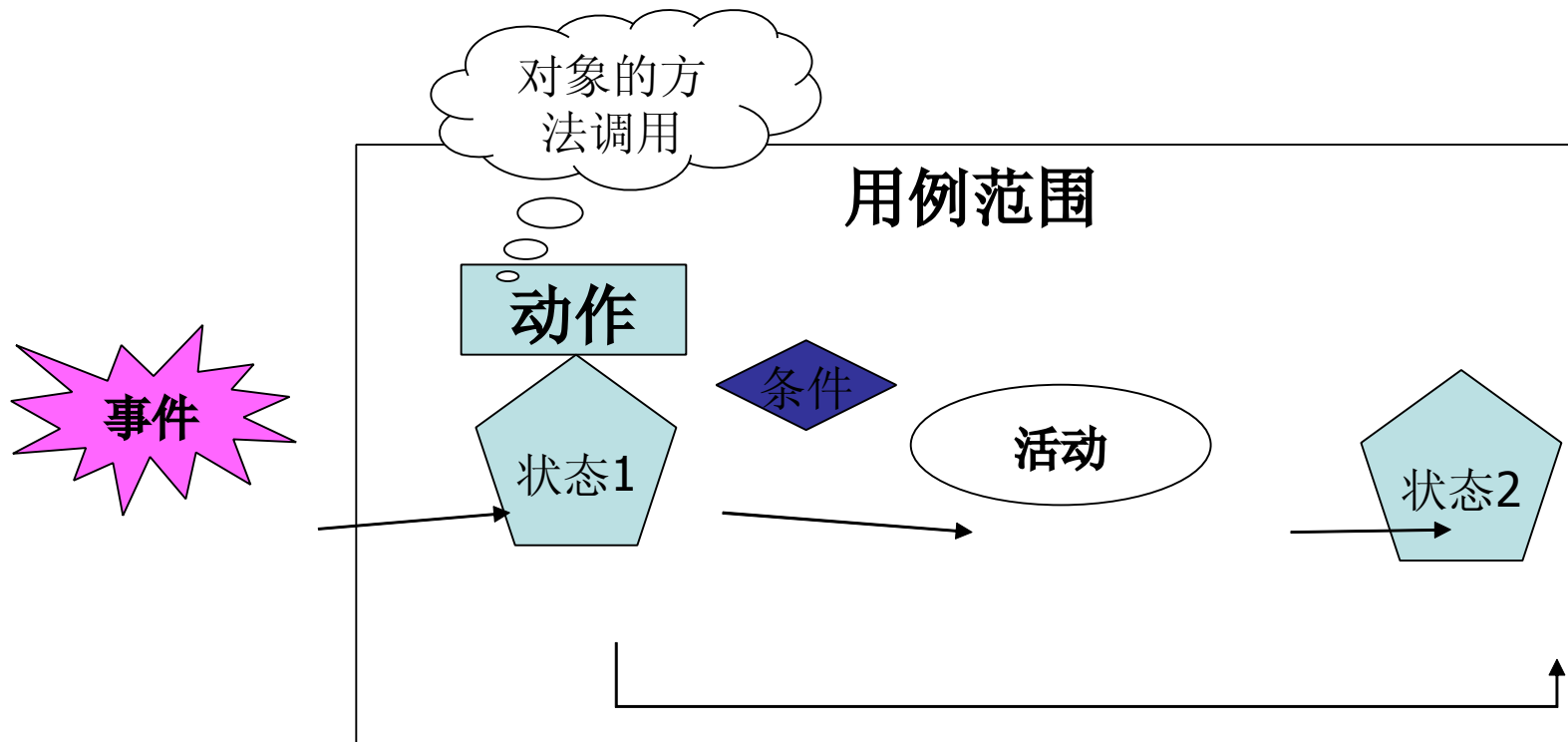
绘制协作图

- 重点描述消息
- 网状的图
- 两个对象之间的多条消息
- 核心对象放在中间
- 编号要准确

演化图

- 状态图
 - 描述某一对象的生命周期中需要关注的不同状态
 - 刺激状态转移的事件和对象采取的动作也要被描述
- 活动图
 - 描述用例内部的活动或方法的流程
 - 多了对并行活动的描述以外它就是流程图
 - **Rose2002**中活动图中可以增加对状态的描述

演化图5个要素



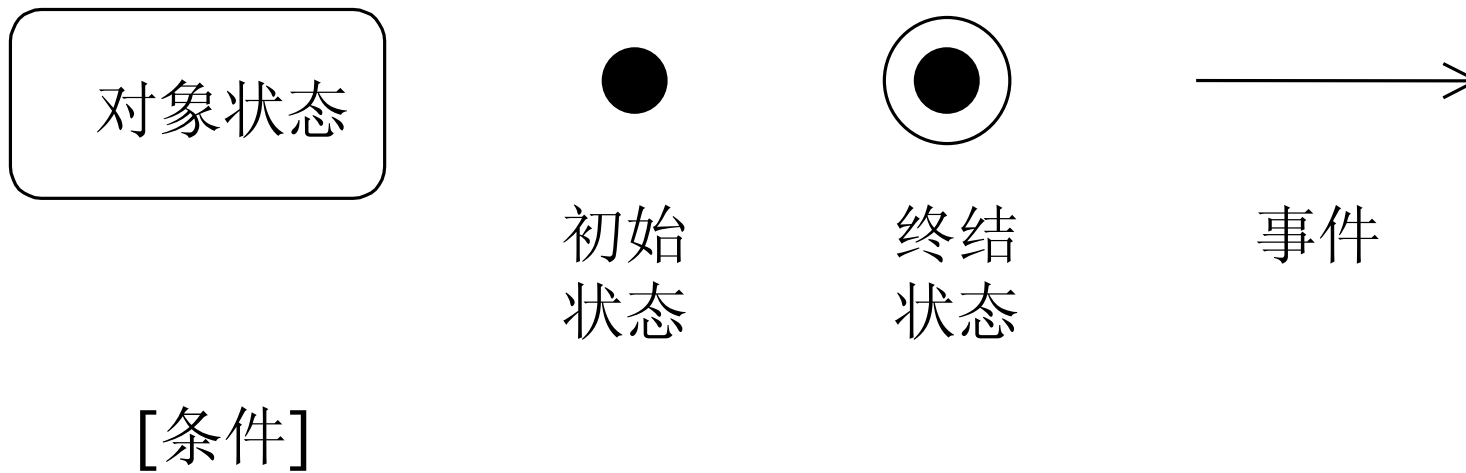
事件类型

- 有两种类型事件
 - 内部事件
 - 为从系统内部激发的事件，一个对象的动作通过事件激活另一个对象动作
 - 外部事件
 - 为从系统边界外的激发的事件

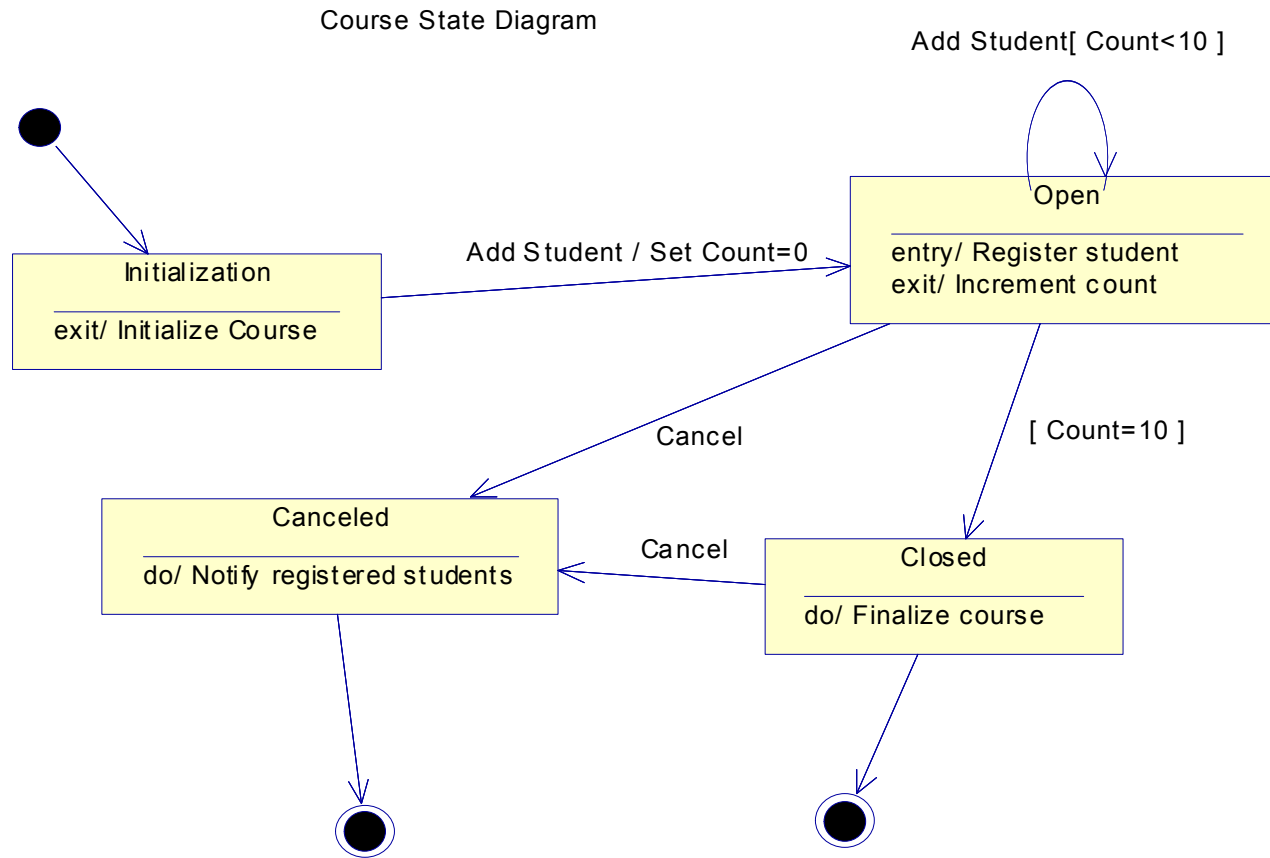
状态和事件

- 状态是某一时间段内对象所保持的稳定态
 - 一个对象的状态一般是有限的
 - 状态可以想象成对象演化当中的照片
- 事件是来自对象外部的刺激
 - 事件是对象演化的动力
 - 事件可以想象成电击对象的行为

状态图元素



状态图案例



状态图的组成

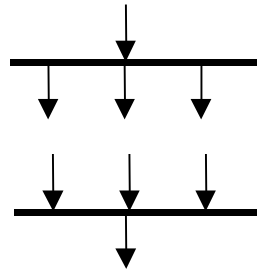
- 初始状态
- 终结状态
- 生存期状态
- 事件
- 动作
- 条件

状态图绘制

- 保证一个初始状态
- 可以有多个终结状态
- 状态是对象演化中的离散快照
- 状态要表示对象的关键快照，有重要的实际意义
- 事件和动作要明确

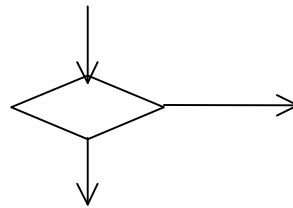
活动图元素

- 并行：

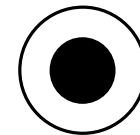


- 条件：[]

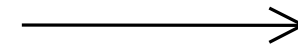
- 分支点：



开始

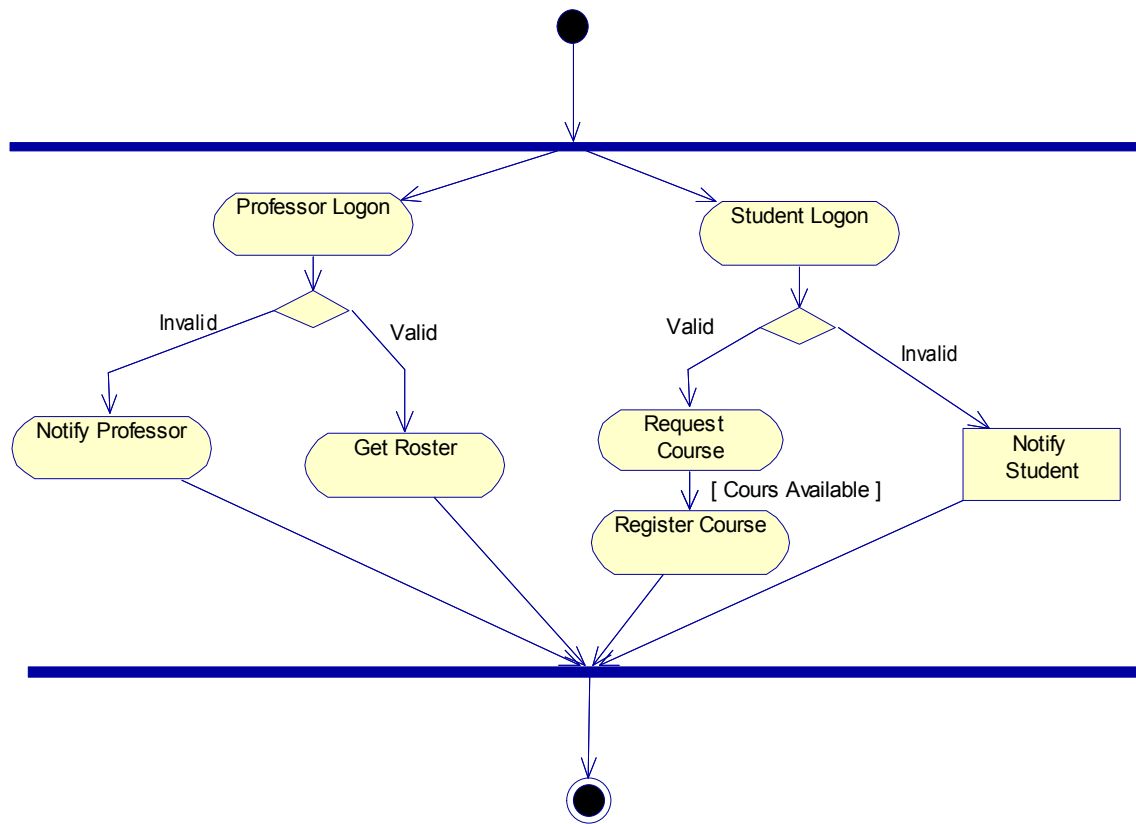


结束



事件

活动图案例



活动图的组成

- 开始活动和结束活动
- 其他活动
- 并行活动
- 分支和循环活动
- 事件
- 条件

活动图绘制

- 一个开始活动和一到多个结束活动
- 活动为中心
- 并行活动和串行活动的分离
- 分支逻辑和循环逻辑都可以表示
- 分支和循环条件最好明确表示

UML图分类

- 结构
- 动态
- 模型管理

第六章

设计模式

内容简介

- 设计模式背景知识
- 设计模式的种类
- Factory Method
- Abstract Factory
- Adapter (Object)
- Observer
- MVC

设计模式背景知识

- 模式的出现
- 软件设计中的模式
- 设计模式的定义

模式的出现

- 20世纪70年代
- Alexander
- 建筑设计模式

软件设计中的模式

- Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides
- 《Design Patterns—Elements of Reusable Software》
- GOF 23

设计模式的定义

- 解决重复设计问题
- 反复出现的设计问题的解决方案
- 模式被用来在特定场景下解决一般设计问题的类和相互通信的对象的描述
- 处理同一类软件分析结果的典型设计结构

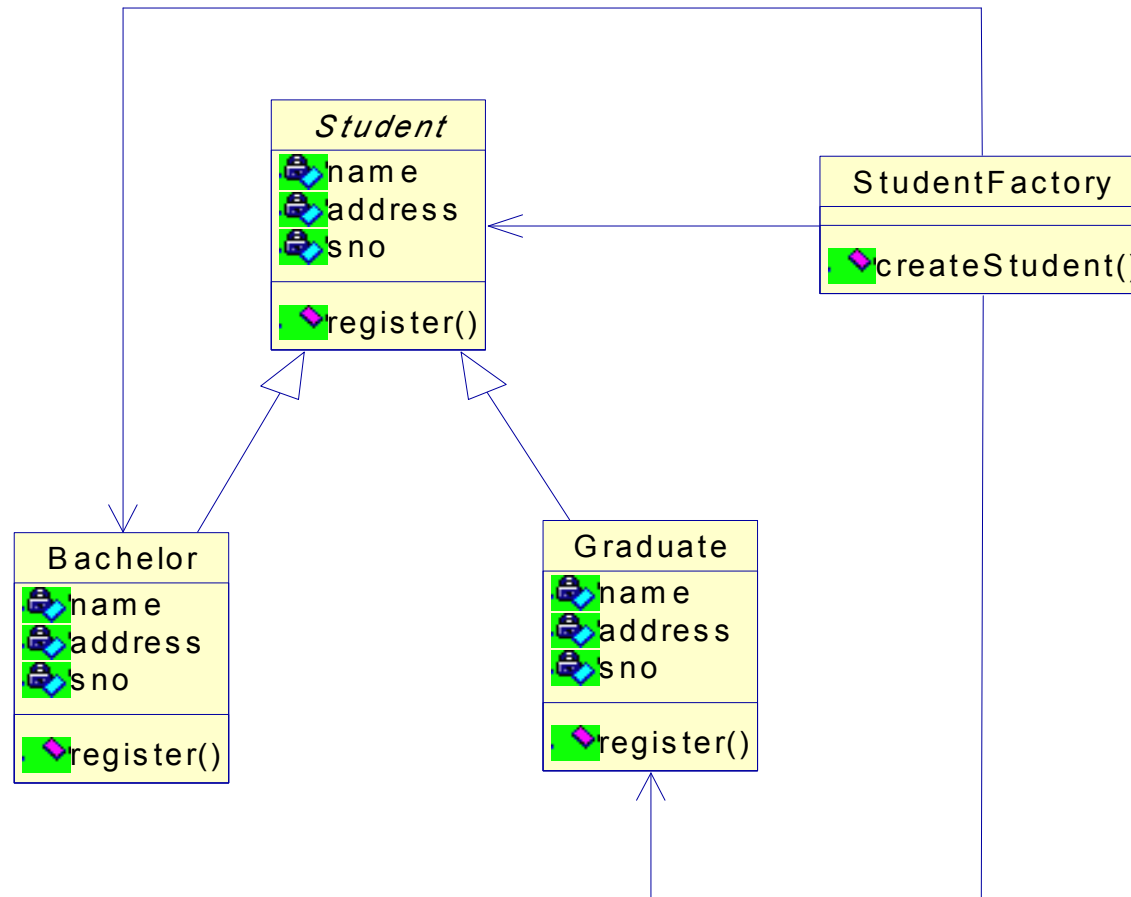
设计模式的种类

- 类范畴
 - 创建型
 - 结构型
 - 行为型
- 对象范畴
 - 创建型
 - 结构型
 - 行为型

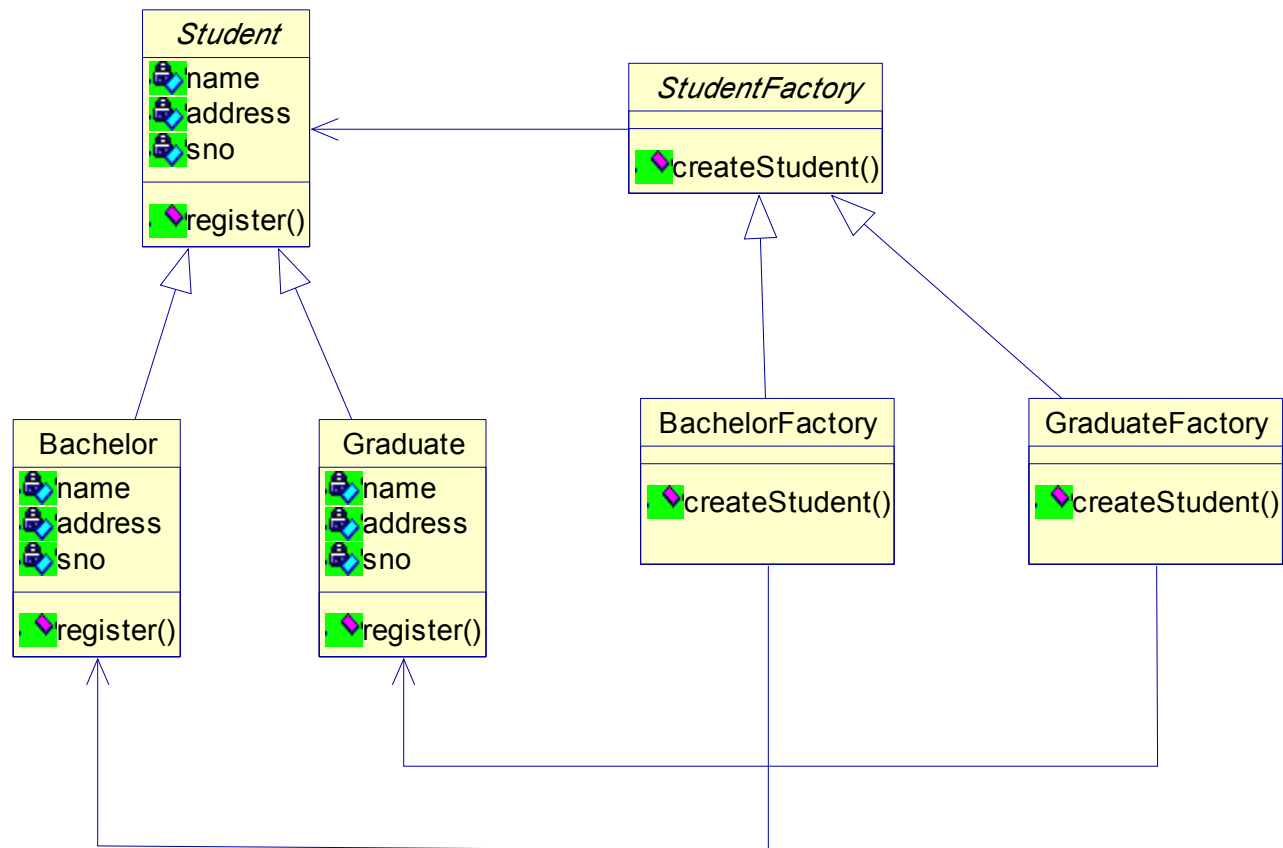
模式讲解思路

- 模式提出
- 模式案例
- 模式原理分析

Factory Method案例1



Factory Method案例2



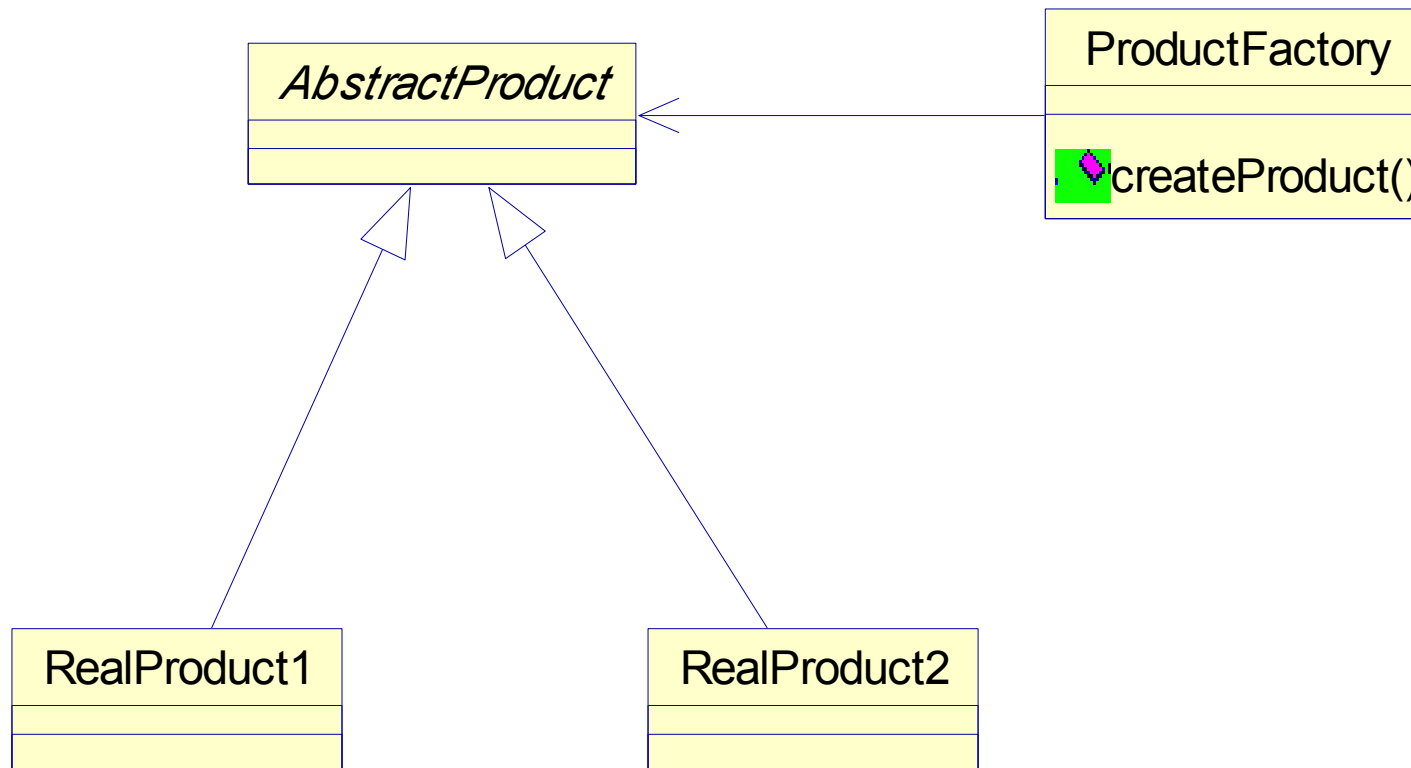
Factory Method(工厂方法)

- 对象创建型模式
- 意图：定义一个用于创建对象的接口，让子类决定实例化哪一个类。使一个类的实例化延迟到其子类。
- 别名：虚构造器
- 动机：框架使用抽象类**定义和维护**对象之间的关系，对象的创建由框架负责。但它只知道要创建的对象抽象父类，而不知道具体哪一种子类将被创建。
- 工厂方法：它将负责“生产”一个对象。
- 本质：用一个**virtual method**完成创建过程。

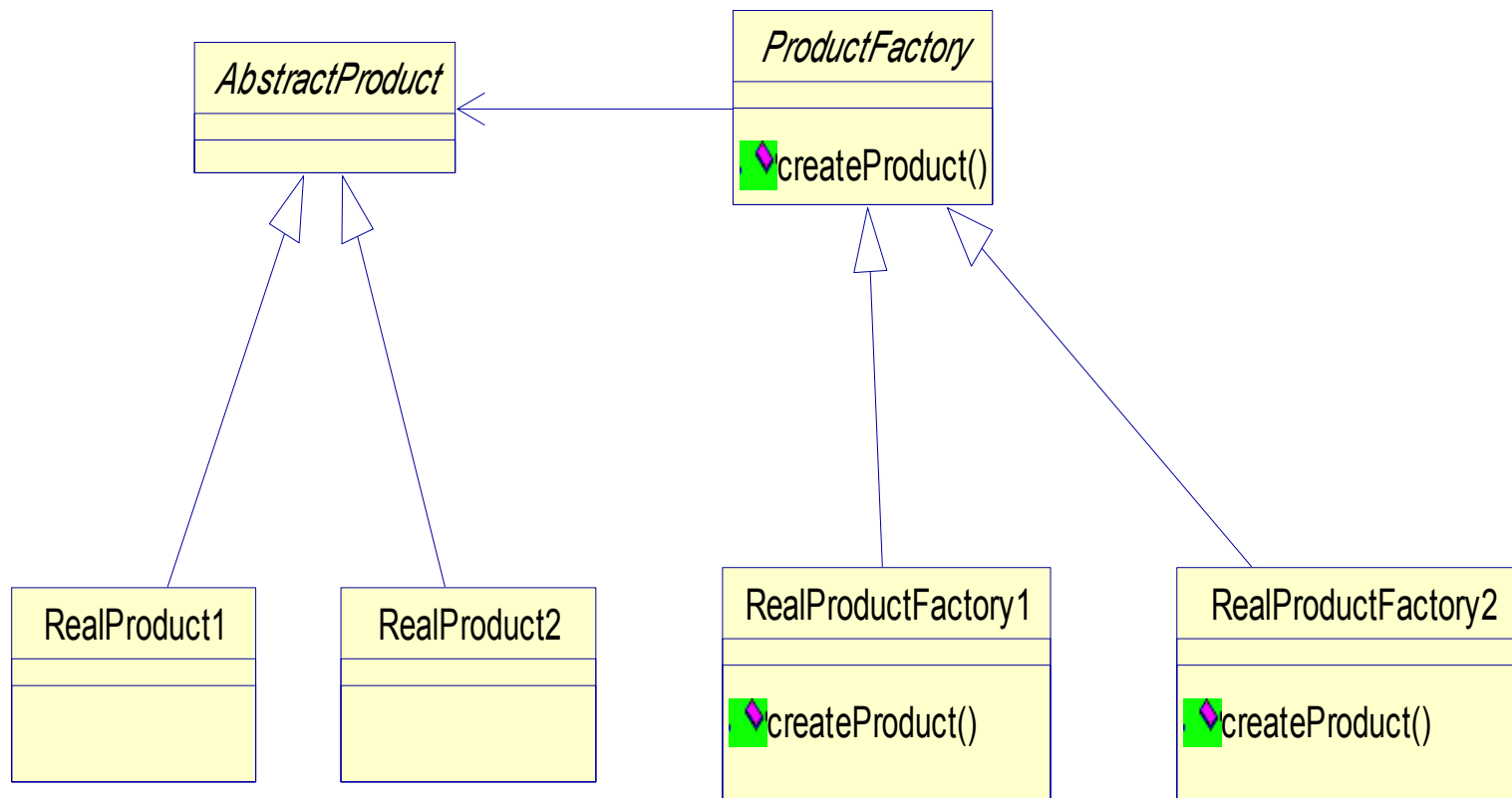
Factory Method(工厂方法)

- 适用性：
 - 当一个类不知道它所必须创建的对象类的类的时候；
 - 当一个类希望由它的子类来指定它所创建的具体对象的时候；

Factory Method原理1



Factory Method原理2



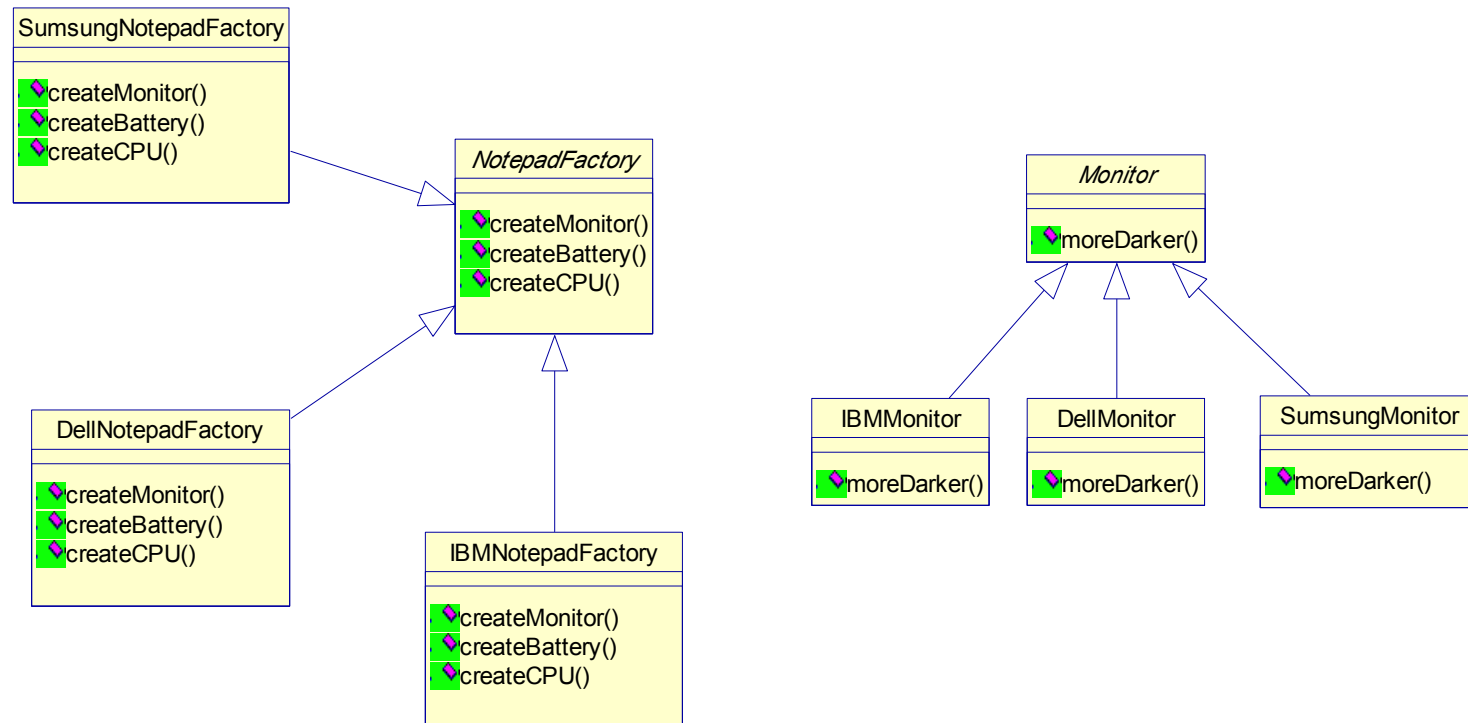
Factory Method(工厂方法) 参与者

- **AbstractProduct**
 - 定义工厂方法所创建的对象接口
- **RealProduct**
 - 实现AbstractProduct接口
- **ProductFactory**
 - 声明工厂方法，该方法返回一个AbstractProduct类型的对象。
- **RealProductFactory**
 - 重定义工厂方法以返回一个RealProduct对象。

Factory Method(工厂方法)

- 协作过程
 - **ProductFactory**依赖于它的子类来定义工厂方法，所以它返回一个适当的**RealProduct**对象。
- 效果
 - 工厂方法不再将与特定应用有关的类绑定到代码中。代码仅处理**AbstractProduct**接口，因此它可以与用户定义的任何**RealProduct**一起使用。

Abstract Factory 案例



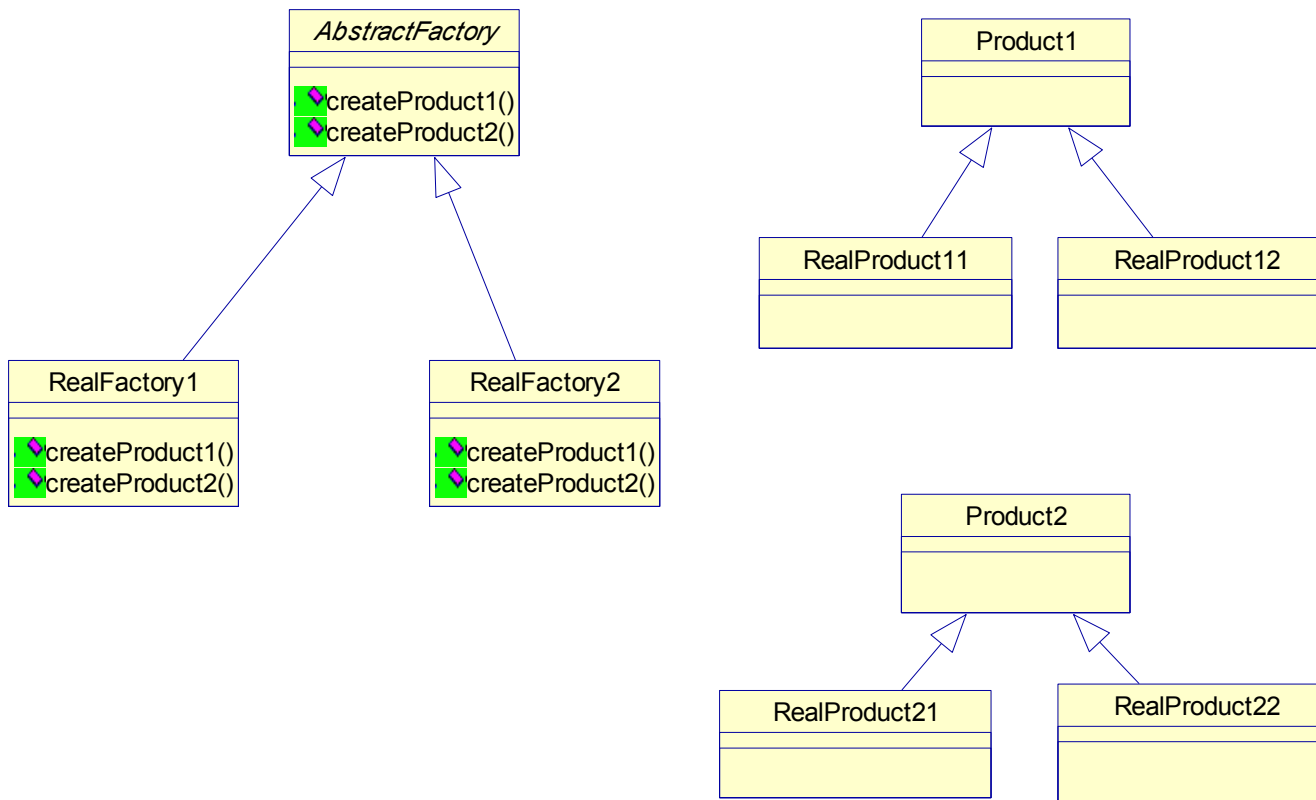
Abstract Factory

- 对象创建型模式
- 意图：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。
- 别名：Kit
- 动机：客户仅通过NotepadFactory接口创建Notepad的组件monitor/battery/cpu，它们并不知道哪些类创建了这些组件。即：客户仅与抽象类定义的接口交互，而不使用特定的具体类的接口。

Abstract Factory

- 适用性：
 - 一个系统要独立于它的产品的创建、组合和表示时；
 - 一个系统要有多个产品系列中的一个来配置时；
 - 当你要强调一系列相关的产品对象的设计以便进行联合使用时；
 - 当你要提供一个产品类库，而只想显示它们的接口而不是实现时。

Abstract Factory 原理



Abstract Factory 参与者

- **AbstractFactory**
 - 声明一个创建抽象产品对象的操作接口；
- **RealFactory**
 - 实现创建具体产品对象的操作；
- **AbstractProduct**
 - 为一类产品对象声明一个接口；
- **RealProduct**
 - 定义一个将被相应的具体工厂创建的产品对象；
 - 实现**AbstractProduct**接口；

Abstract Factory

- 协作过程
 - 通常在运行时刻创建一个**RealFactory**类的实例，这一具体的**factory**创建具有特定实现的产品对象。为创建不同的产品对象，客户应使用不同的具体工厂。
 - **AbstractFactory**将产品对象的创建延迟到它的**RealFactory**子类。

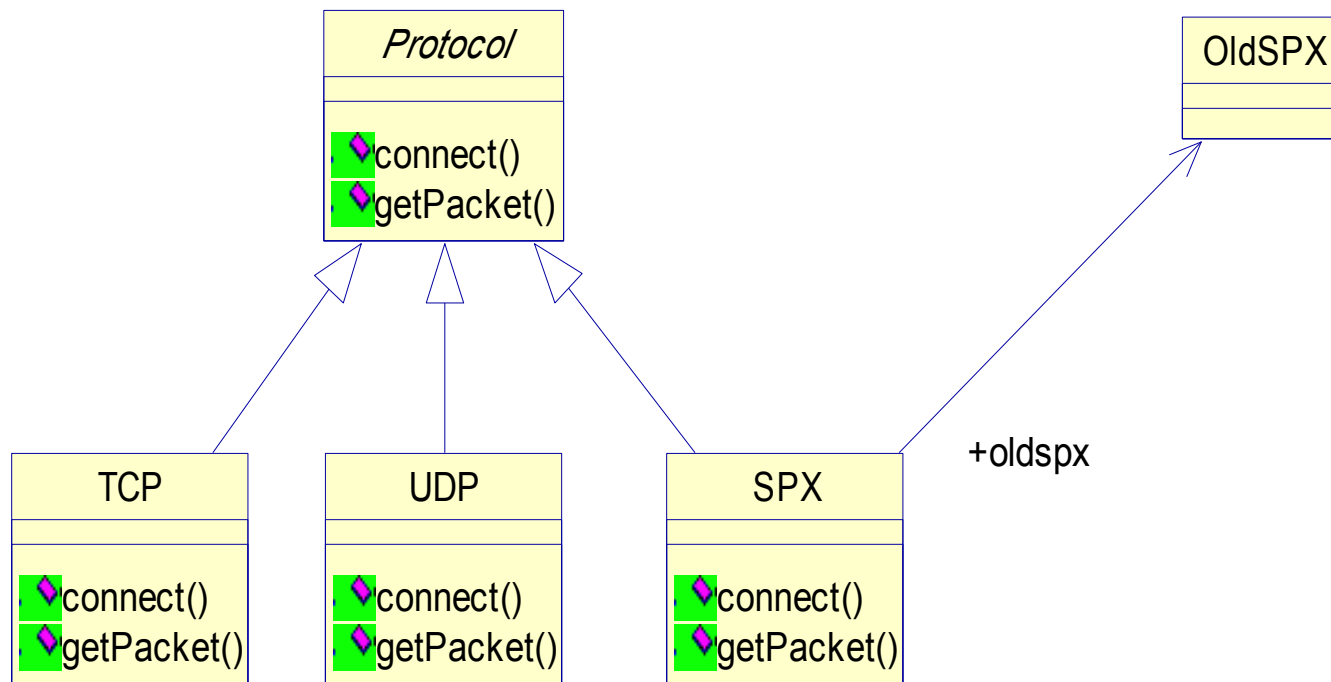
Abstract Factory

- 效果
 - 它分离了具体的类；
 - 它使得易于交换产品系列；
 - 它有利于产品的一致性；
 - 难以支持新种类的产品；

工厂方法及抽象工厂 总结

- 了解每一种模式的实质
 - 具体实现的时候可能会有变化情况，或者扩展，或者退化
- **factory method**是基础，**abstract factory**是它的扩展
- **factory method**、**abstract factory**都涉及到类层次结构中对对象的创建过程，有所取舍
 - **factory method**需要依附一个**creator**类
 - **abstract factory**需要一个平行的类层次
 - 根据应用的其他需求，以及语言提供的便利来决定使用哪种模式

Adapter(Object)案例



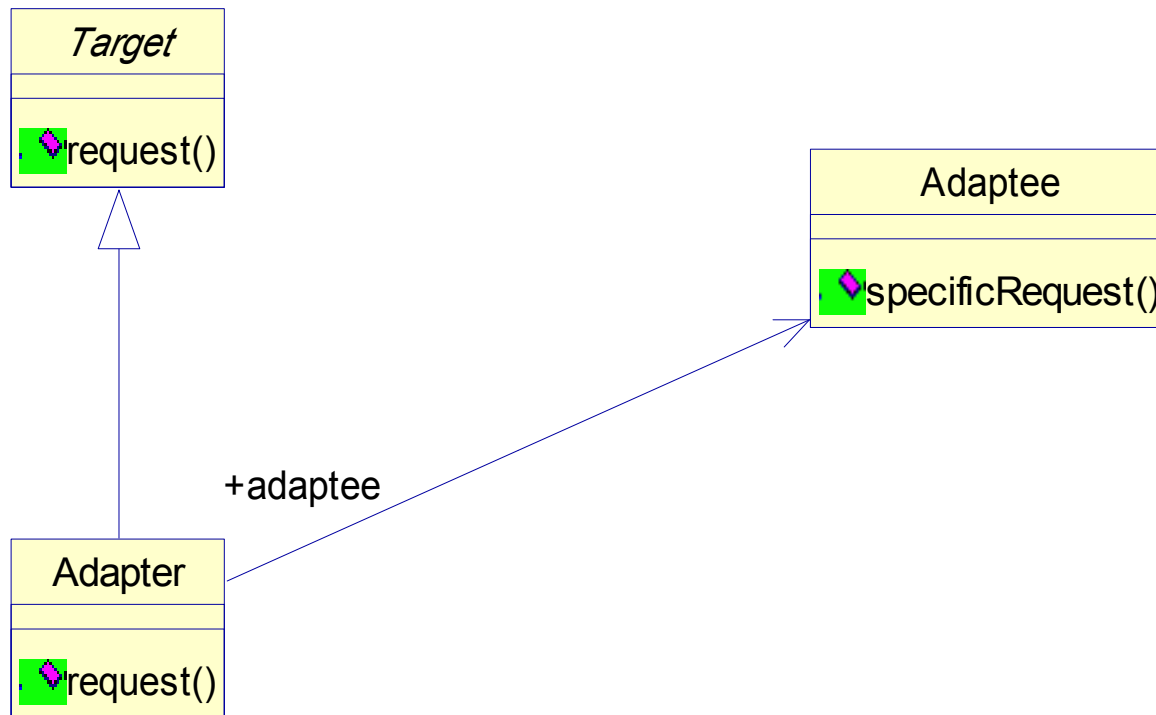
Adapter 适配器

- 意图：将一个类的接口转换为客户希望的另外一个接口。**Adapter** 模式使原本由于接口不兼容而不能一起工作的类可以一起工作。
- 动机：为复用而设计的工具箱类不能被复用的原因仅仅因为它的接口与专业应用领域所需要的接口不匹配。

Adapter 适配器

- 适用性：
 - 你希望使用一个已经存在的类，而它的接口不符合你的需求。
 - 你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类协同工作。

Adapter(Object)原理



Adapter(Object) 参与者

- **Target:** 定义客户端使用的与特定领域相关的接口。
- **Adaptee:** 定义一个已经存在的接口，这个接口需要适配
- **Adapter:** 对Adaptee的接口与Target接口进行适配。

Adapter(Object)

- 协作过程：
 - 客户端在**Adapter**实例上调用一些操作，接着适配器调用**Adaptee**的操作实现这个请求。

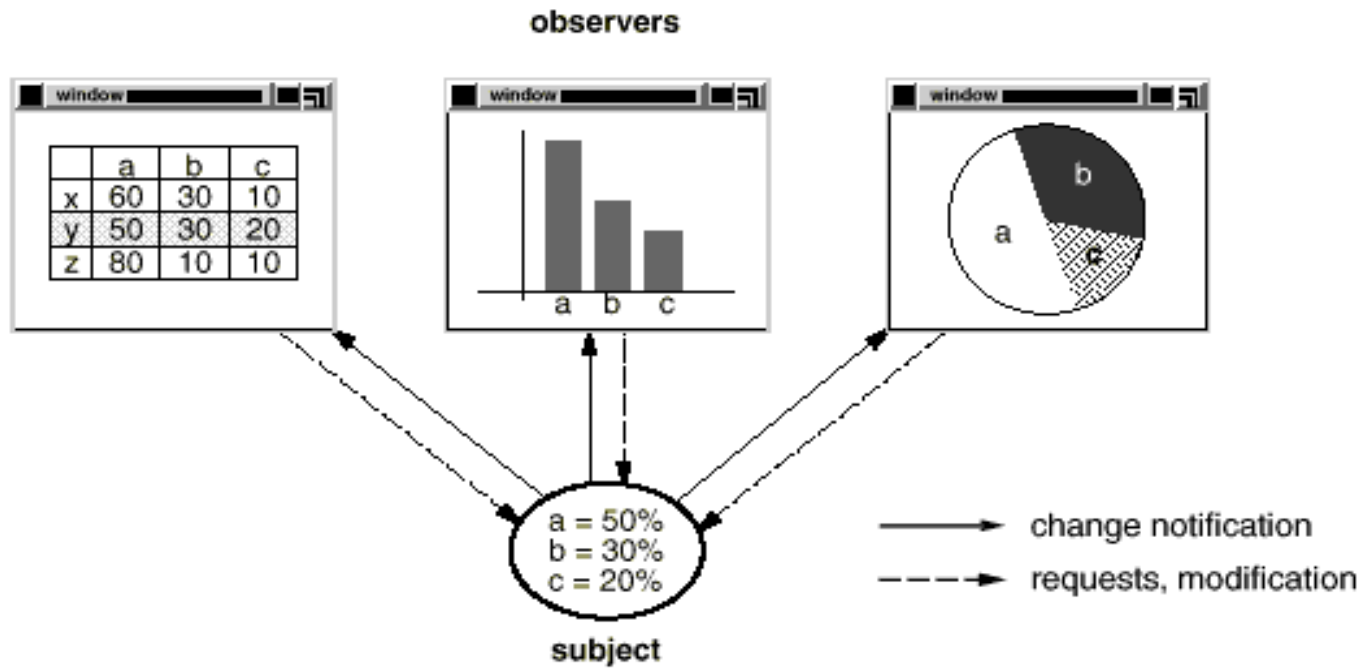
Adapter(Object)

- 效果（对象适配器）：
 - 允许一个Adapter与多个Adaptee——即Adaptee本身以及它的所有子类同时工作。
 - 使得重定义Adaptee的行为比较困难。
- 需要考虑的问题：
 - Adapter的匹配程度：Adapter的工作量取决于Target接口与Adaptee接口的相似程度。

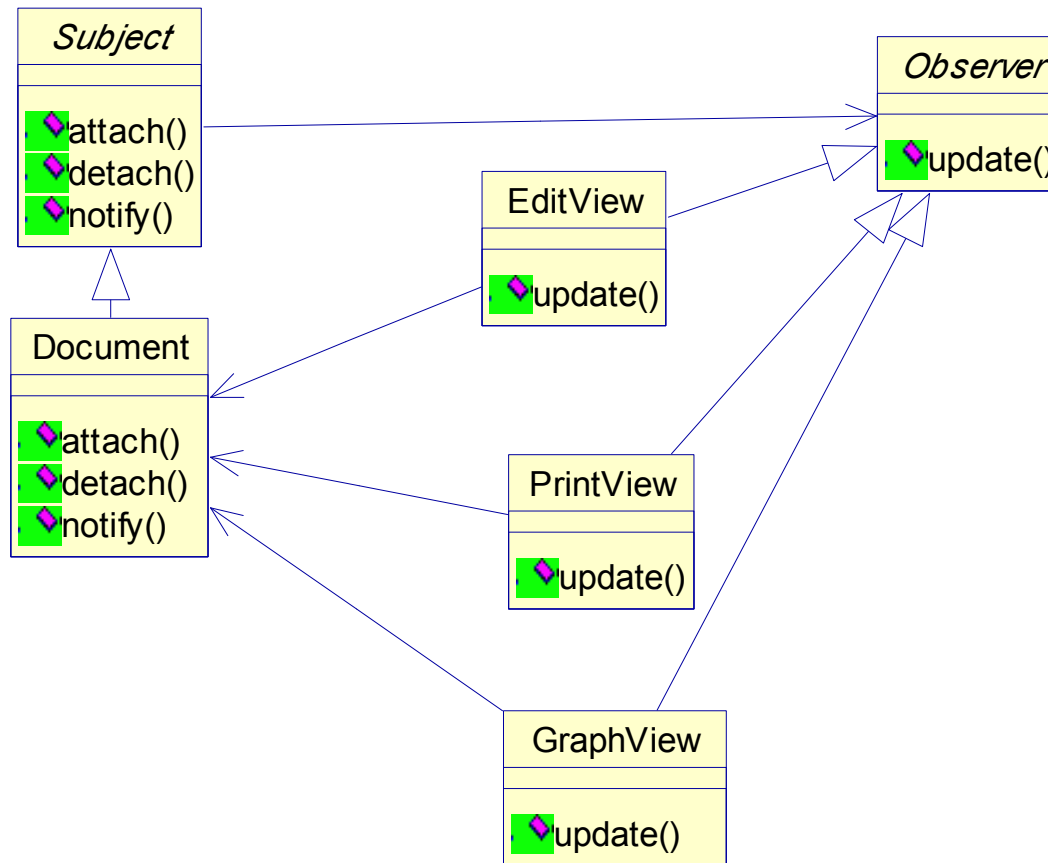
Observer

- 意图：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。
- 动机：需要维护相关对象间的一致性。且不希望为了维持一致性而使各类紧密耦合，这样会降低它们的可重用性。

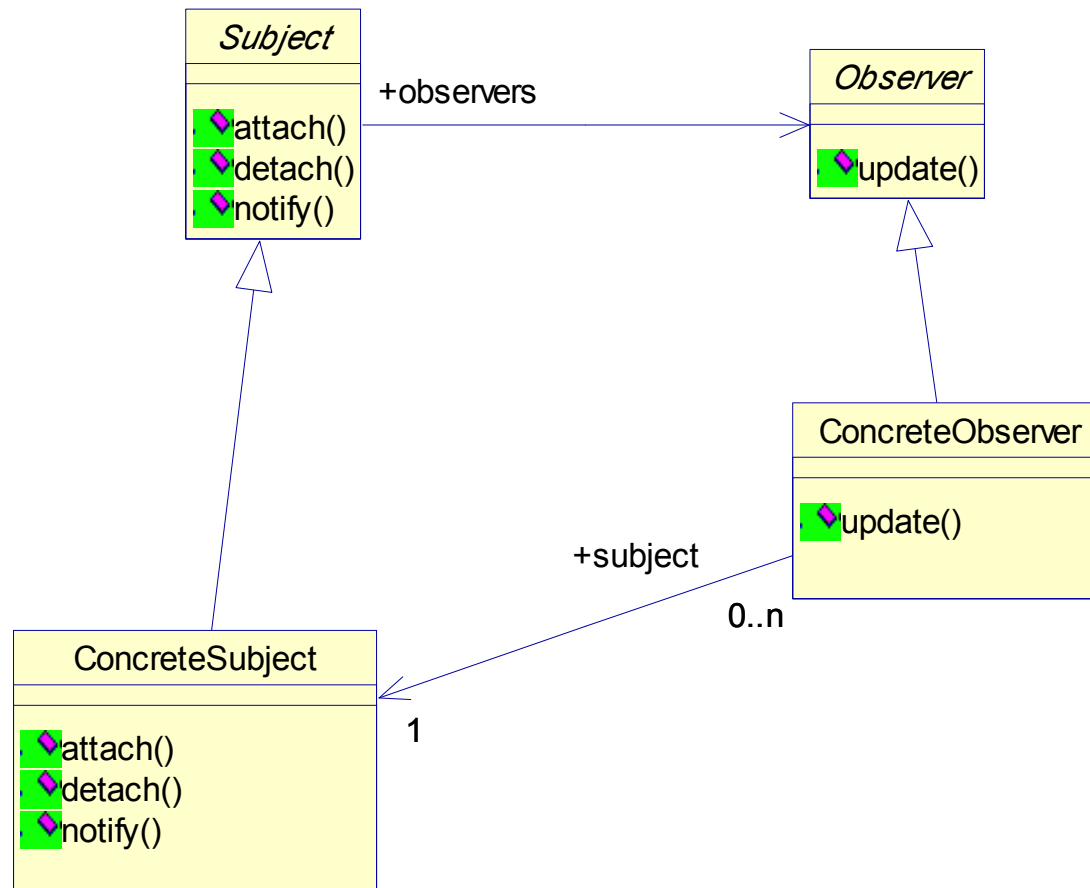
Observer案例



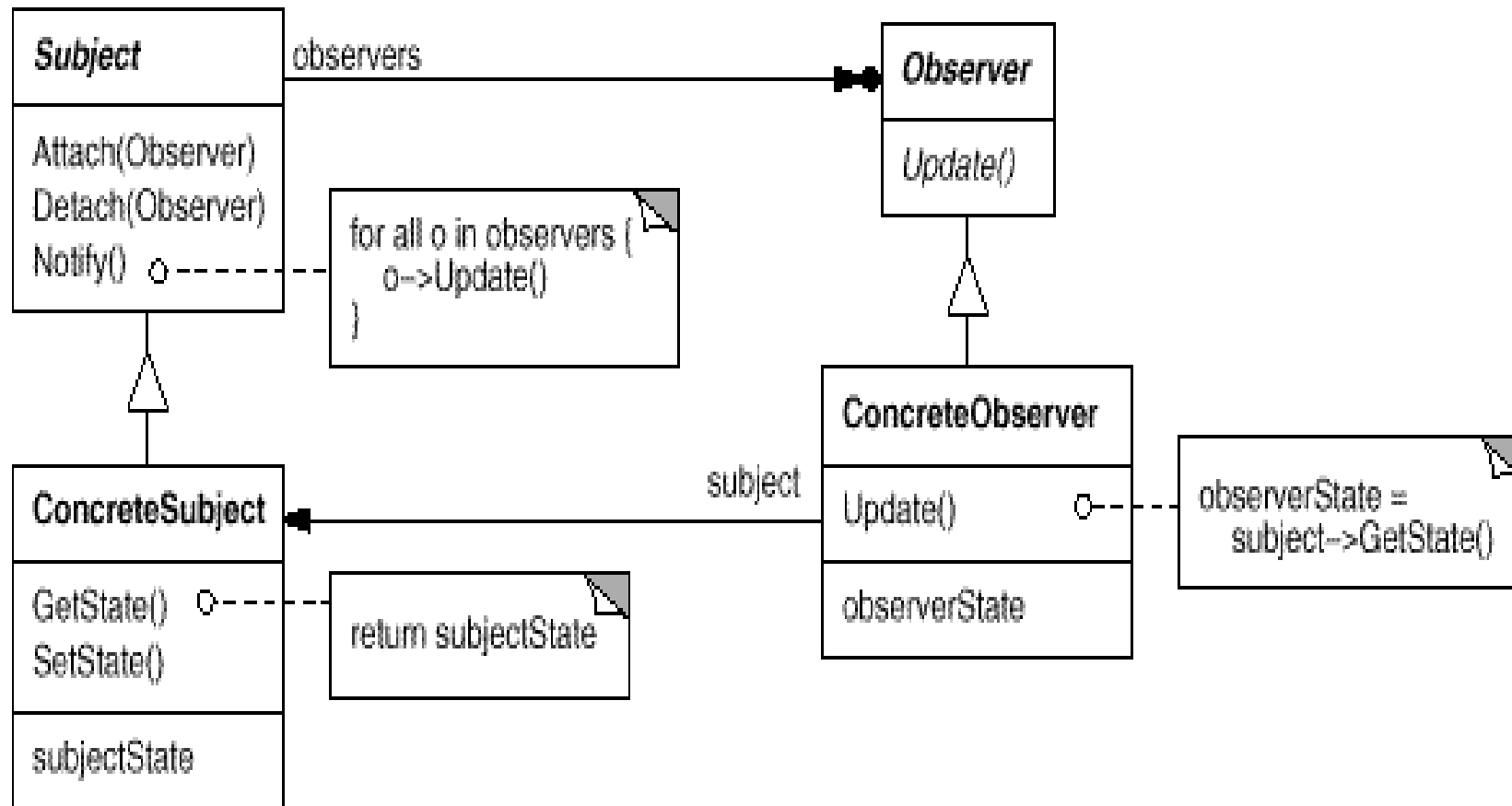
Observer案例



Observer原理



Observer原理



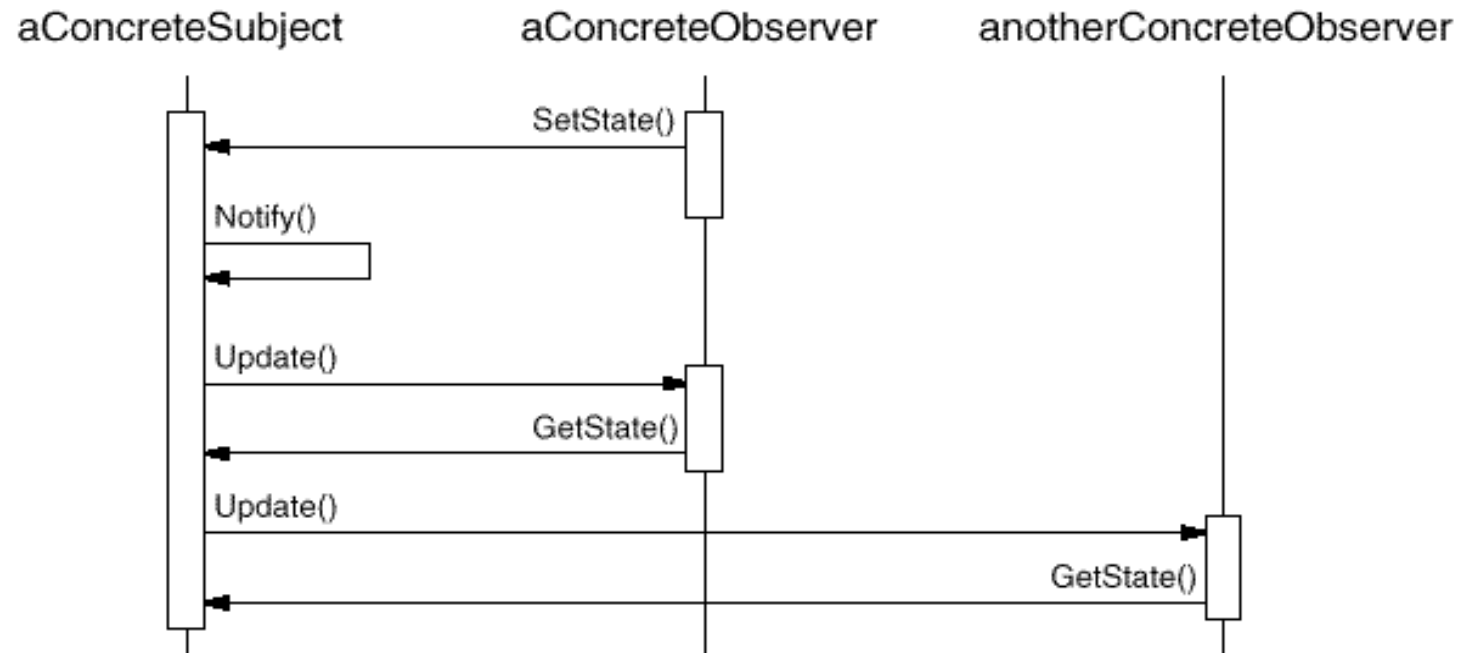
Observer参与者

- **Subject(目标):**
 - 目标知道它的观察者。可以有任意多个观察者观察同一个目标。
 - 提供注册和删除观察者对象的接口。
- **Observer（观察者）**
 - 为那些在目标发生改变时需获得通知的对象定义一个更新接口。
- **ConcreteSubject（具体目标）**
 - 将有关状态存入各ConcreteObserver对象。
 - 当它的状态发生改变时，向它的各个观察者发出通知。
- **ConcreteObserver（具体观察者）**
 - 维护一个指向ConcreteSubject对象的引用。
 - 存储有关状态，这些状态应与目标的状态保持一致。
 - 实现Observer的更新接口以使自身状态与目标状态保持一致。

Observer

- 协作过程：
 - 当**ConcreteSubject**发生任何可能导致其观察者与其本身状态不一致的改变时，它将通知它的各个观察者。
 - 在得到一个具体目标的改变通知时，**ConcreteObserver**对象可向目标对象查询信息，**ConcreteObserver**使用这些信息以使它的状态与目标对象的状态一致。

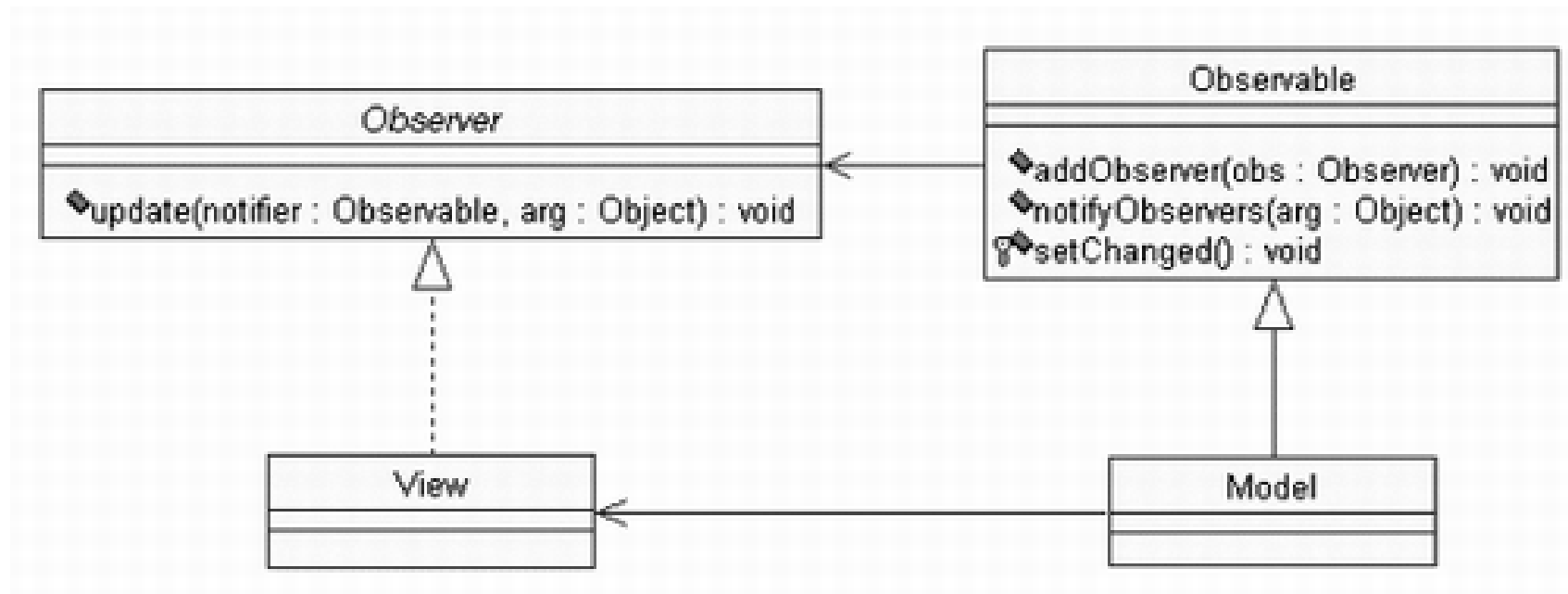
Observer – 观察者模式序列图



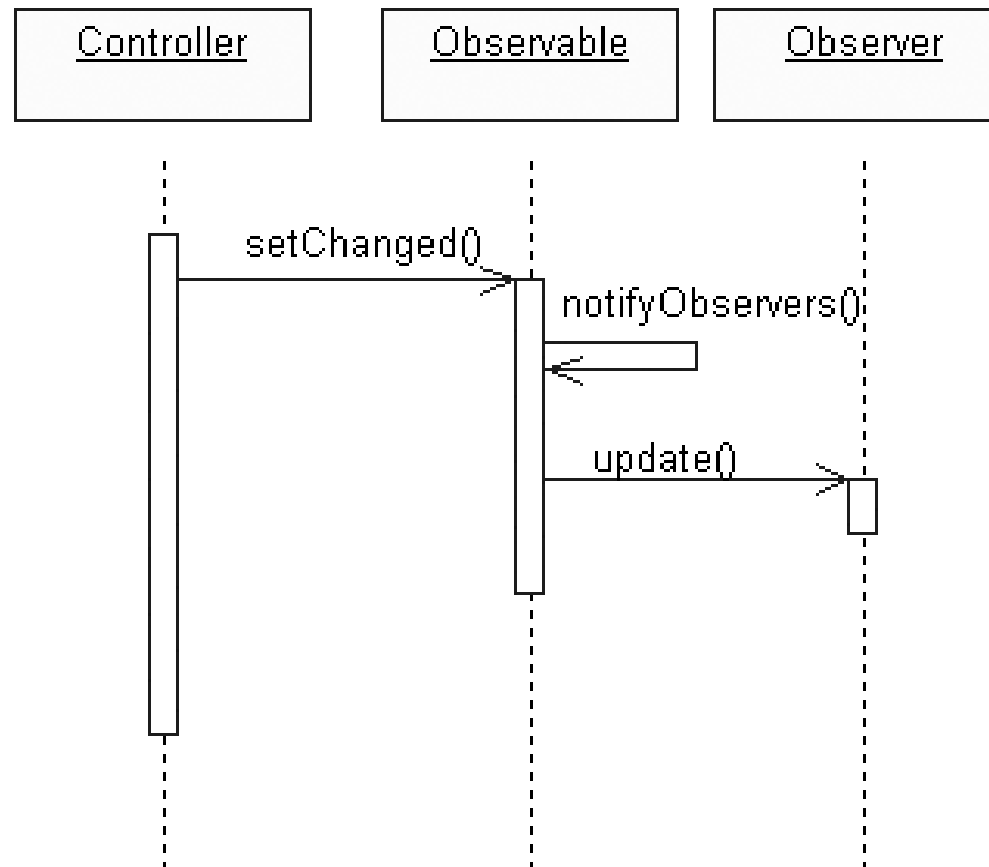
MVC

- Model 模型
- View 视图
- Controller 控制器

MVC模式原理图-MVC模式的UML类图



MVC模式原理图-MVC模式的UML 序列图



MVC关系图

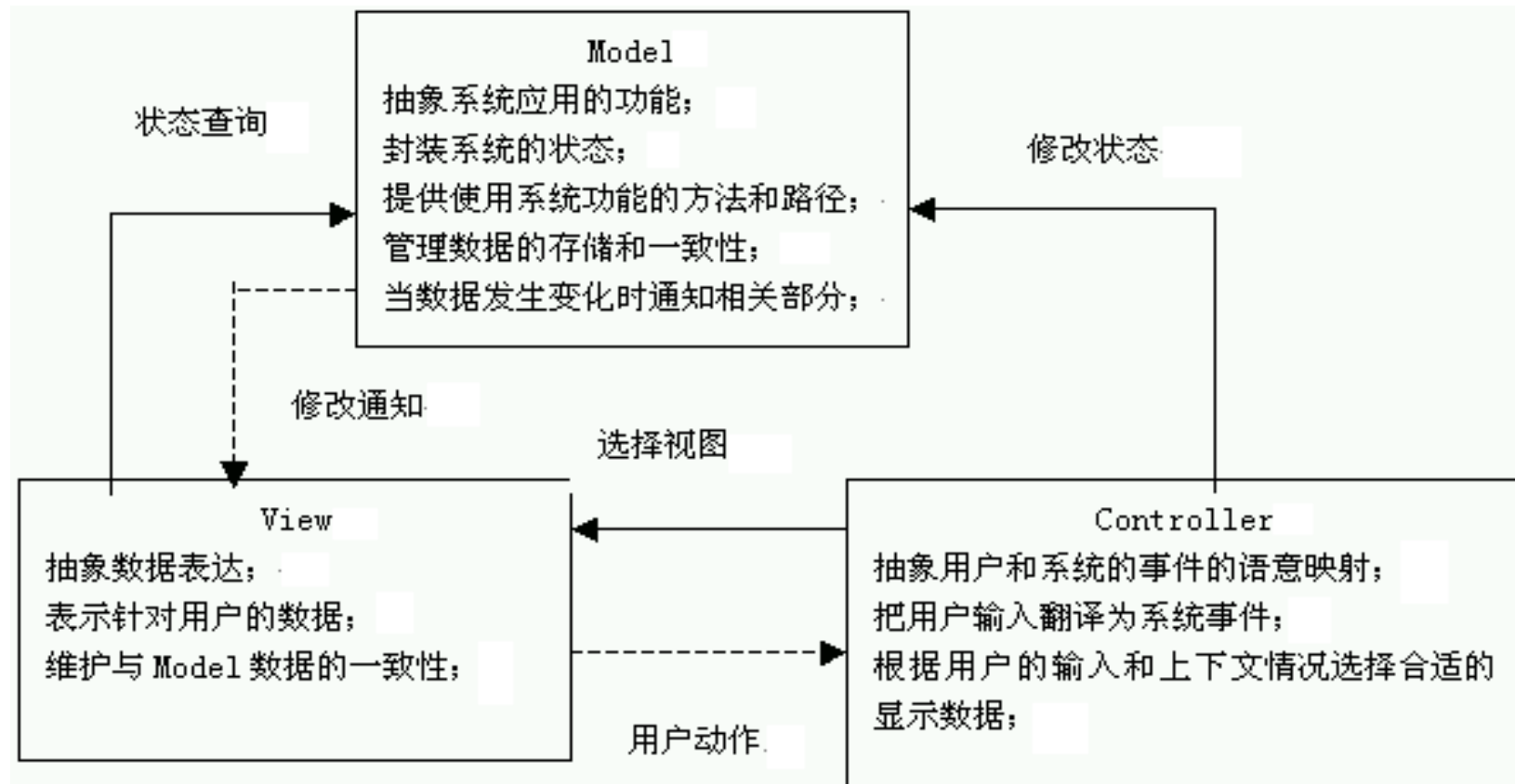


图 1. MVC 关系图

对MVC关系图的理解

| | 模型 M | 视图 V | 控制器 C |
|----|---|---|---|
| 分工 | <ul style="list-style-type: none">● 抽象系统应用的功能● 封装系统的状态● 提供使用系统功能的方法和路径● 管理数据的存储和一致性● 当数据发生变化时通知相关部分 | <ul style="list-style-type: none">● 抽象数据表达● 表示针对用户的数据● 维护与 Model 数据的一致性 | <ul style="list-style-type: none">● 抽象用户和系统的事件的语意映射● 把用户输入翻译为系统事件● 根据用户的输入和上下文情况选择合适的显示数据 |
| 协作 | <ul style="list-style-type: none">● 当他改变系统数据时通知 View● 能够被 View 检索数据● 提供对 Controller 的操作途径 | <ul style="list-style-type: none">● 把 Model 表征给用户● 当数据被相关 Model 改变时更新表示的数据● 把用户输入提交给 Controller | <ul style="list-style-type: none">● 把用户输入转成对 Model 的系统行为● 根据用户输入和 Model 的动作结果选择合适的 View |

Iterator(迭代器)

- 意图：
 - 提供一种方法顺序访问一个聚合对象中的各个元素，而又不需暴露该对象的内部表示。
- 动机：
 - 一个聚合对象（如**list**），应提供一种方法让别人可以访问其中的元素，又不需暴露其内部结构。针对不同的需要，可以不同的方式遍历这个列表。

Iterator(迭代器)

- 关键思想：
 - 将对列表的访问和遍历从列表对象中分离出来并放入一个**迭代器(iterator)**对象中。由迭代器定义一个访问该列表元素的接口。迭代器负责跟踪当前的元素，即：它知道哪些元素被遍历过了。

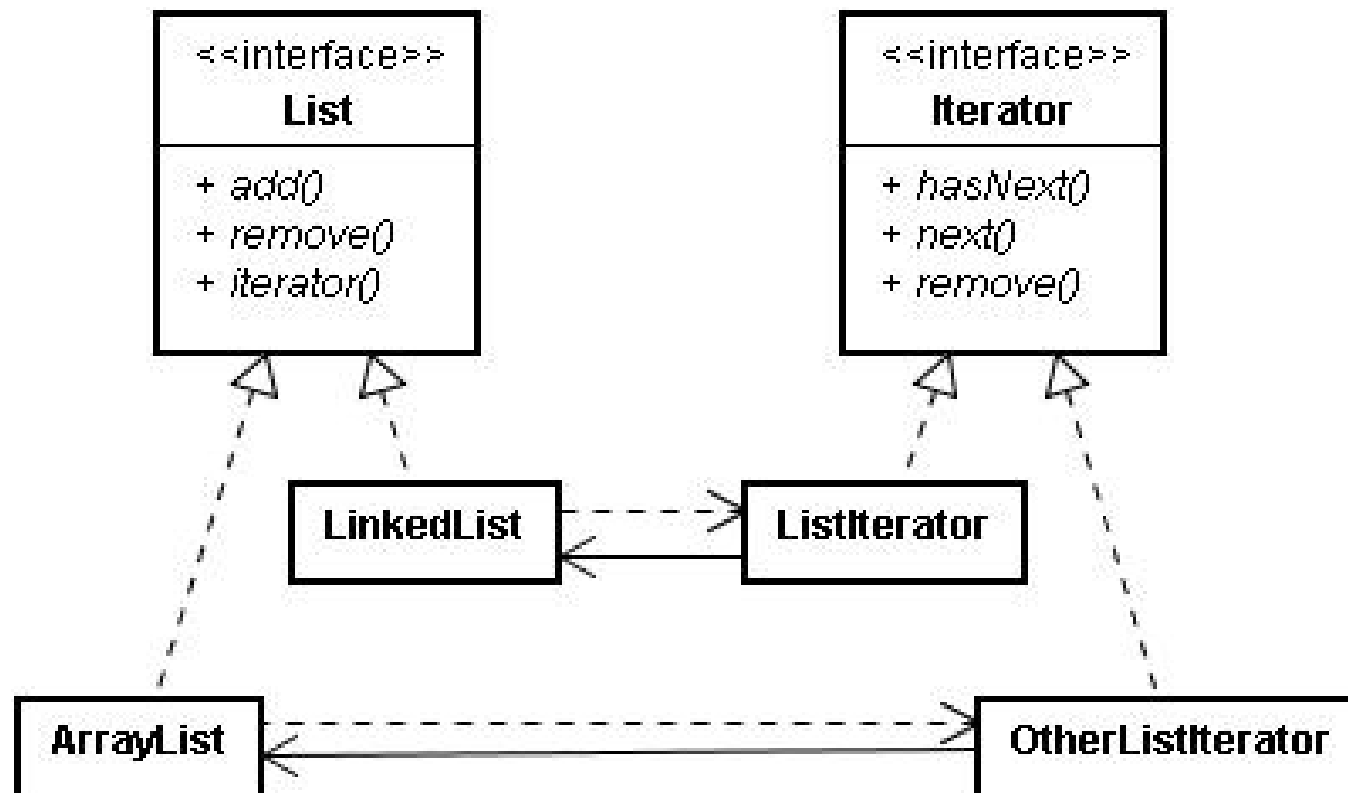
Iterator(迭代器)案例

- 不使用迭代器：
 - `for(int i=0; i<array.size(); i++) { ... get(i) ... }`
- 缺点：
 - 必须事先知道集合的内部结构。
 - 访问代码和集合本身是紧耦合，无法将访问逻辑从集合类和客户端代码中分离出来
 - 每一种集合对应一种遍历方法，客户端代码无法复用。即一旦改变集合类型，程序代码需重写。
- 使用迭代器：
 - `for(Iterator it = c.iterator(); it.hasNext();) { ... }`

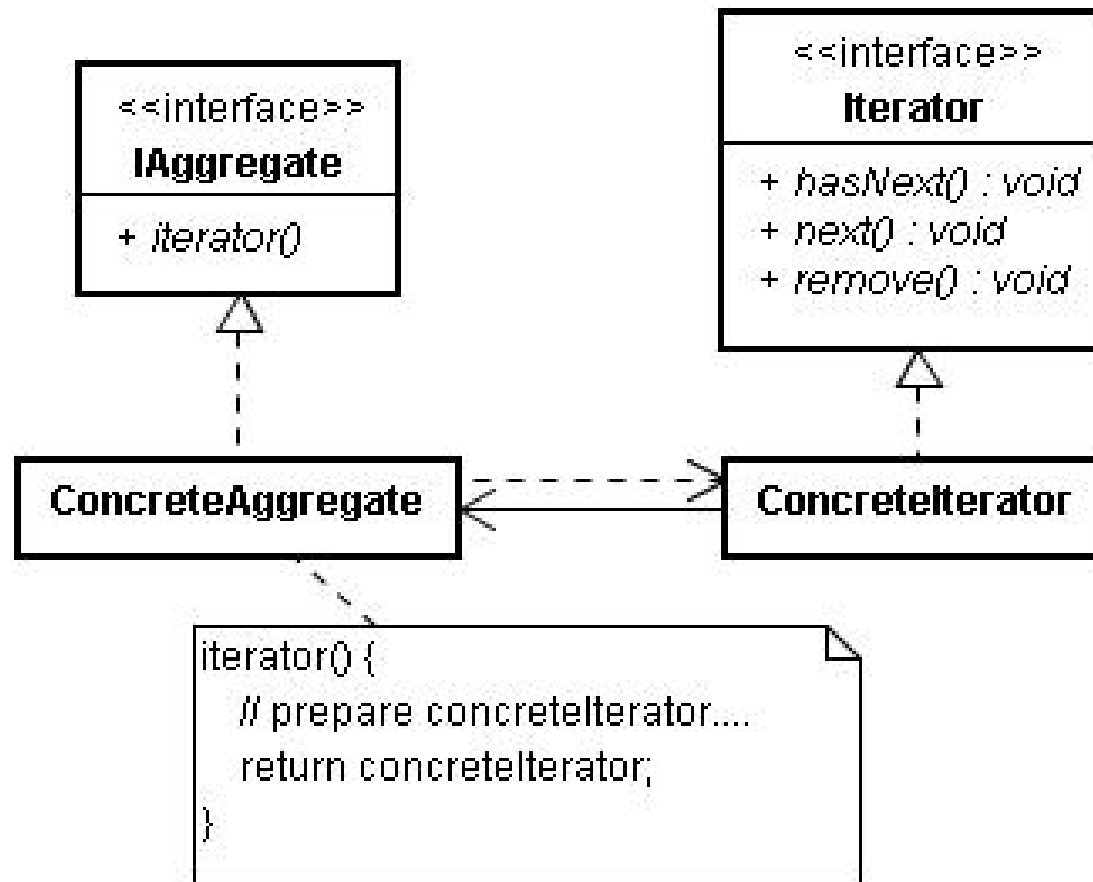
Iterator(迭代器)案例

- 迭代器接口定义：
 - ```
public interface Iterator {
 boolean hasNext();
 Object next();
 void remove();
}
```
- 使用迭代器完成遍历：
  - ```
for(Iterator it = c.iterator(); it.hasNext(); ) {  
    Object o = it.next();  
    // 对o的操作...  
}
```

Iterator(迭代器)案例



Iterator(迭代器)原理



Iterator(迭代器)参与者

- **Iterator**（迭代器）
 - 迭代器定义访问和遍历元素的接口；
- **Concreteliterator**（具体迭代器）
 - 具体层迭代器实现迭代器接口；
 - 对该聚合遍历时跟踪当前位置；
- **Aggregate**（聚合）
 - 定义创建相应迭代器对象的接口。
- **ConcreteAggregate**（具体聚合）
 - 实现创建相应迭代器的接口，该操作返回 **Concreteliterator** 的一个适当实例。

Iterator(迭代器)

- 协作过程：
 - ConcreteIterator跟踪ConcreteAggregate聚合对象中的当前对象，并能够计算出待遍历的后继对象。
- 效果：
 - 支持以不同的方式遍历一个聚合对象；也可以自己定义迭代器的子类以支持新的遍历。
 - 迭代器简化了聚合的接口；
 - 在同一个聚合对象上可以有多个遍历，即可以同时进行多个遍历。

第七章

配置管理

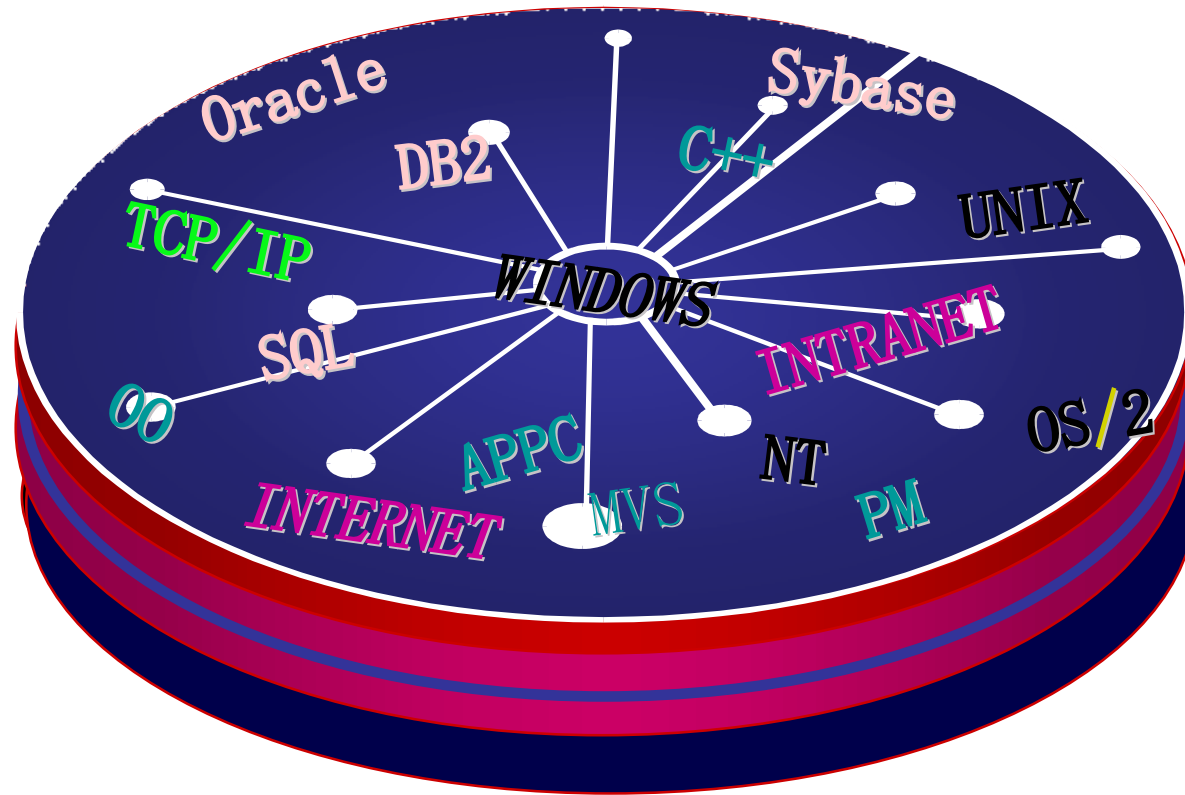
本章要点

- 为什么做配置管理
- 什么是软件配置管理
- 配置管理中的关键概念
- **SCM**工具的功能
- 检入检出

为什么做配置管理

- 开发环境的复杂性
- 软件开发过程中的很多困境

开发环境的复杂性



软件开发过程中面临的困境

- ◆ 需求管理
- ◆ 程序和文档维护
- ◆ 代码可重用性
- ◆ 人员流动
- ◆ 构建管理



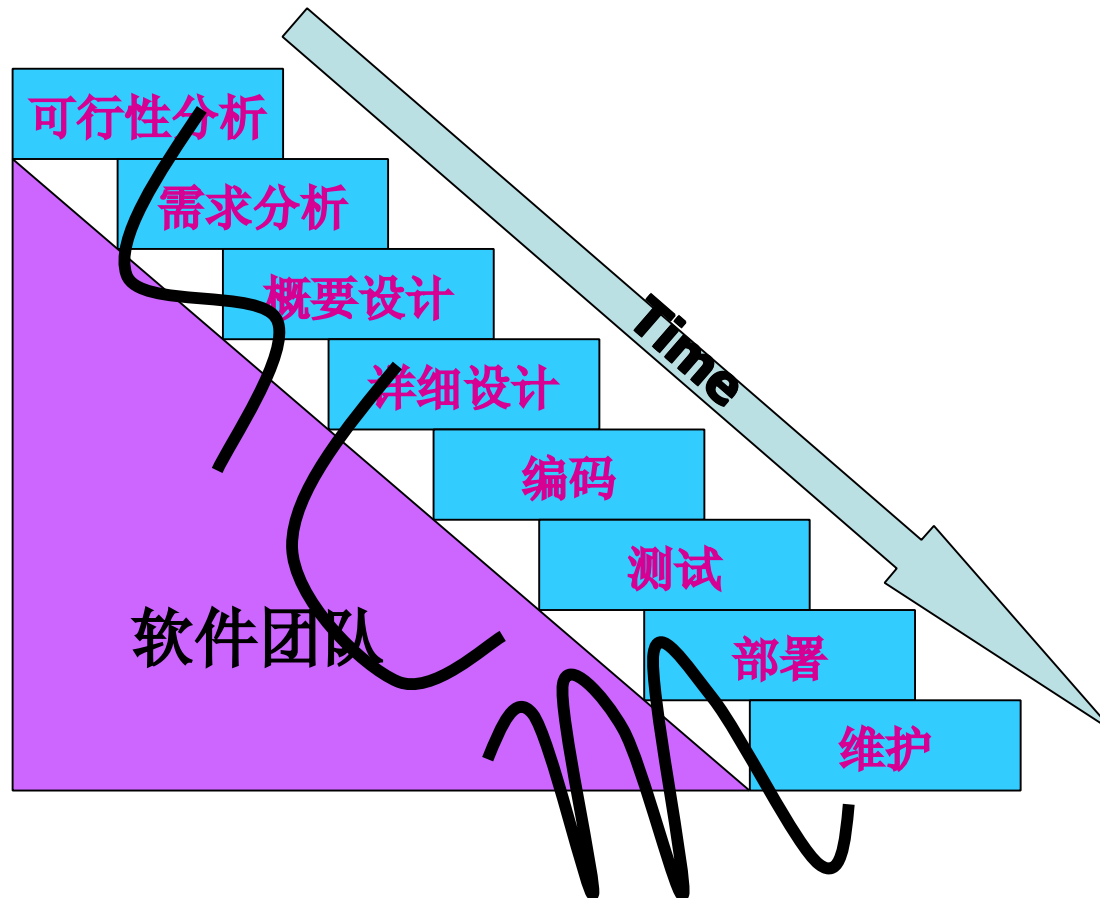
什么是软件配置管理

- 一般定义
 - 软件配置管理就是管理变更的过程
 - 变更是指软件开发活动中的文档、源码和数据的更改。
- 标准定义
 - IEEE标准729-1983
 - 软件配置管理（Software Configuration Management，简称SCM）是一门应用技术管理和监督相结合的学科，通过表示和文档标记配置项的功能和物理特性，控制这些特性的变更，记录和报告变更的状态和过程，并验证它们与需求是否一致。

配置管理的标准定义

- 标识
- 控制
- 状态统计
- 审计和审查
- 生产
- 过程管理
- 小组协作

SCM贯穿软件工程整个生命期



软件配置

- 软件配置定义
- 配置项（CI, Configuration Item）
- **配置管理**是标识和控制配置项。

配置管理常规内容

- 标识和版本控制
- 变更控制
- 状态报告
- 配置审核

配置管理逻辑

- 记录配置项的五个方面：
 - 谁创建的？
 - 从哪里来？
 - 什么时间创建的？
 - 为什么创建此配置项？
 - 当前在哪里？
 - 将到哪里去？

配置项粒度

- 文件内容级
- 文件级
- 文件夹级
- 其他

SCM工具核心功能

- 版本控制
- 变更控制
- 配置控制
- 状态统计
- 配置审计

版本控制

- 定义
 - 版本控制是配置管理的基本要求，它可以保证在任何时刻恢复任何一个配置项的任何一个版本。
 - 版本控制主要是对变更配置项的软件行为及变更结果提供一个可跟踪的手段，便于软件开发工作以基线渐进的方式完成。
- 版本
- 配置标识
- 软件组成单元

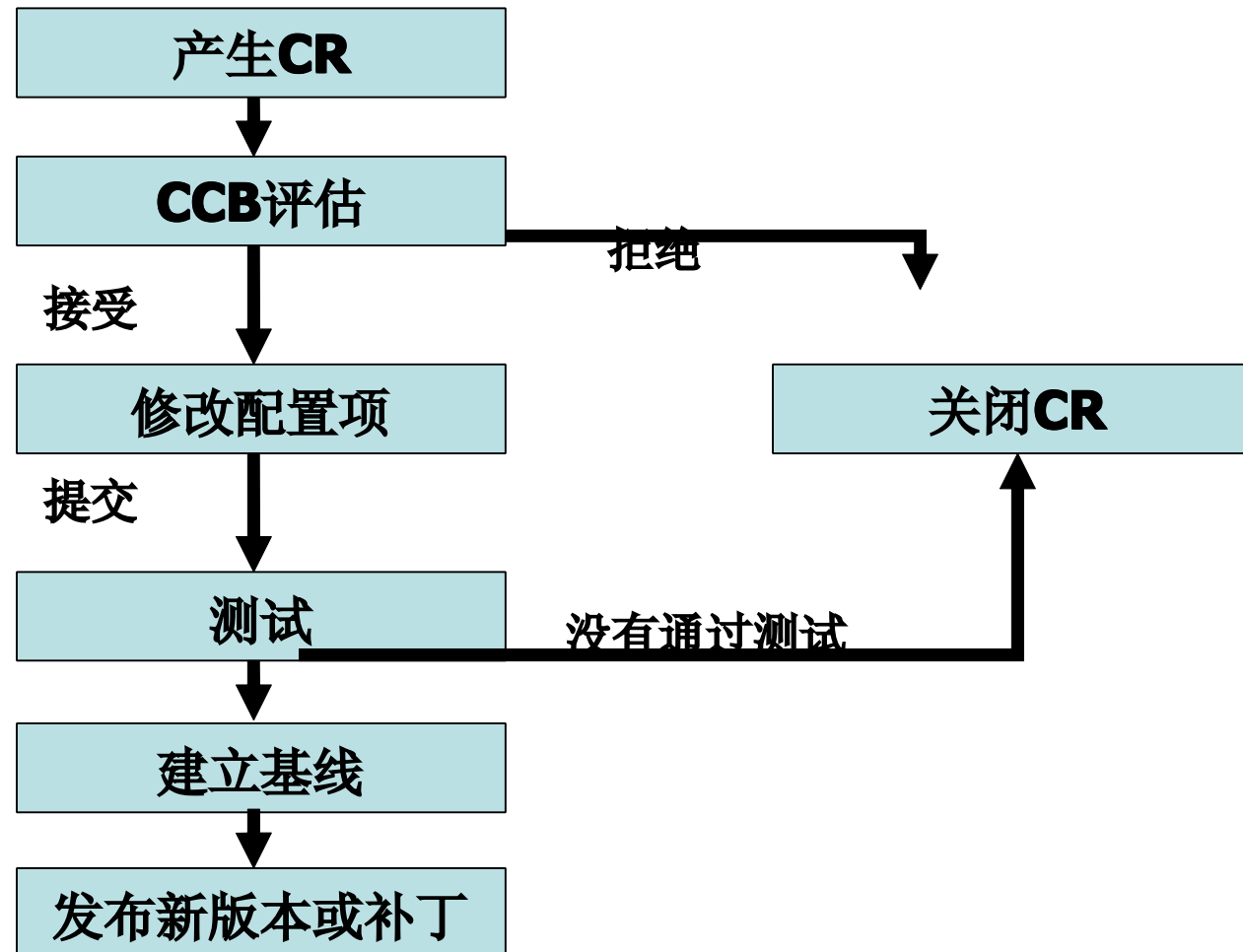
变更控制

- 定义
 - 变更管理主要是控制和协调不同责任的软件开发人员有效的交流。软件过程中某一阶段的变更，均要引起软件配置的变更，这种变更必须严格加以控制和管理，保持修改信息，并把精确、清晰的信息传递到软件工程过程的下一步。
- 变更请求（**Change Request**，简称**CR**）
- 变更的起源
- 对变更进行控制的机构

变更控制规范

- 所有的变更处理都要**经过统一规范**的处理，程序员不能直接处理用户的请求、修改缺陷而没有任何记录的情况。
- **变更审核**：由变更控制负责人审核变更内容是否合理。
- 变更完毕**提交变更报告**，报告包括变更的内容、过程、通报情况等；
- 对于用户不合理的变更需求或有危险的变更需求**予以排除**，并写出原因。
- 对于工程现场的变更，可**先变更后补办变更手续**，但必须是在项目负责人同意之后才能变更。
- 变更之后，**通知**与之相关的部门和人员，与之相关的配置项也应做相应的修改。

变更机制



配置控制

- 定义

主要是以用户和开发团队均认可的衡量尺度（如：与用户签定的软件合同），通过正式配置审核和软件配置审核两种方式，对软件实施过程和软件功能的完整性、正确性审核。

正式技术复审主要是进行配置项的功能性审核，复审者评估SCI（软件配置项）以确定它与其他SCI的一致性、遗漏、及潜在的副作用，正式复审应该对所有变化进行。软件配置审核通常是在软件开发生命周期上每个阶段的末尾开始。

- 构建管理

- 里程碑

- 配置项基线

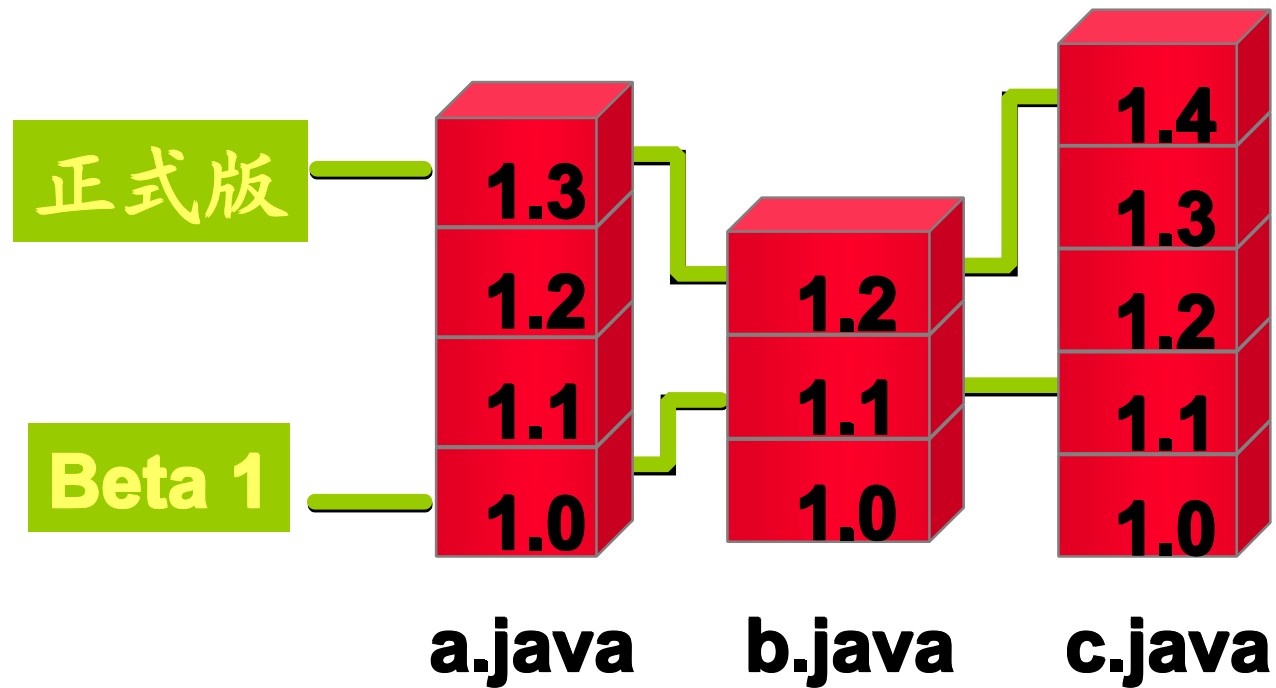
软件配置审核的内容

- 审核产品功能是否能满足用户要求。
- 审核产品功能是否与需求文档一致。
- 审核发布出去的产品是否有帮助文档，文档内容是否与应用程序一致。
- 审核发布产品的功能是否与用户签定的合同一致。

基线

- 基线（**BaseLine**）是已经通过复审和批准的某规约或产品，可以作为进一步开发的基础，并且只能通过正式的变化控制过程的改变。
- 基线是软件生命周期中各开发阶段的一个特定点（也称里程碑），它的作用是把开发各阶段工作的划分更明确化，使本来连续的工作在这些点上断开，以便于检查与肯定阶段成果。
- 基线可以作为一个检查点，在开发过程中，当采用的基线发生错误时，可以知道其所在的位置，返回到最近和最恰当的基线上。

基线的定义



状态报告

Configuration status reporting, 简称CSR

- 发生了什么？
- 为什么要发生？
- 谁做的？
- 什么时候发生的？
- 在哪儿改变的？
- 将影响别的什么吗？

每次当配置审计进行时，其结果作为CSR任务的一部分被报告。

配置审计

- 配置是否符合规范
- 配置项是否记录正确
- 变更控制记录的完整性
- **SQA**（软件质量保证）

检入检出（CICO）

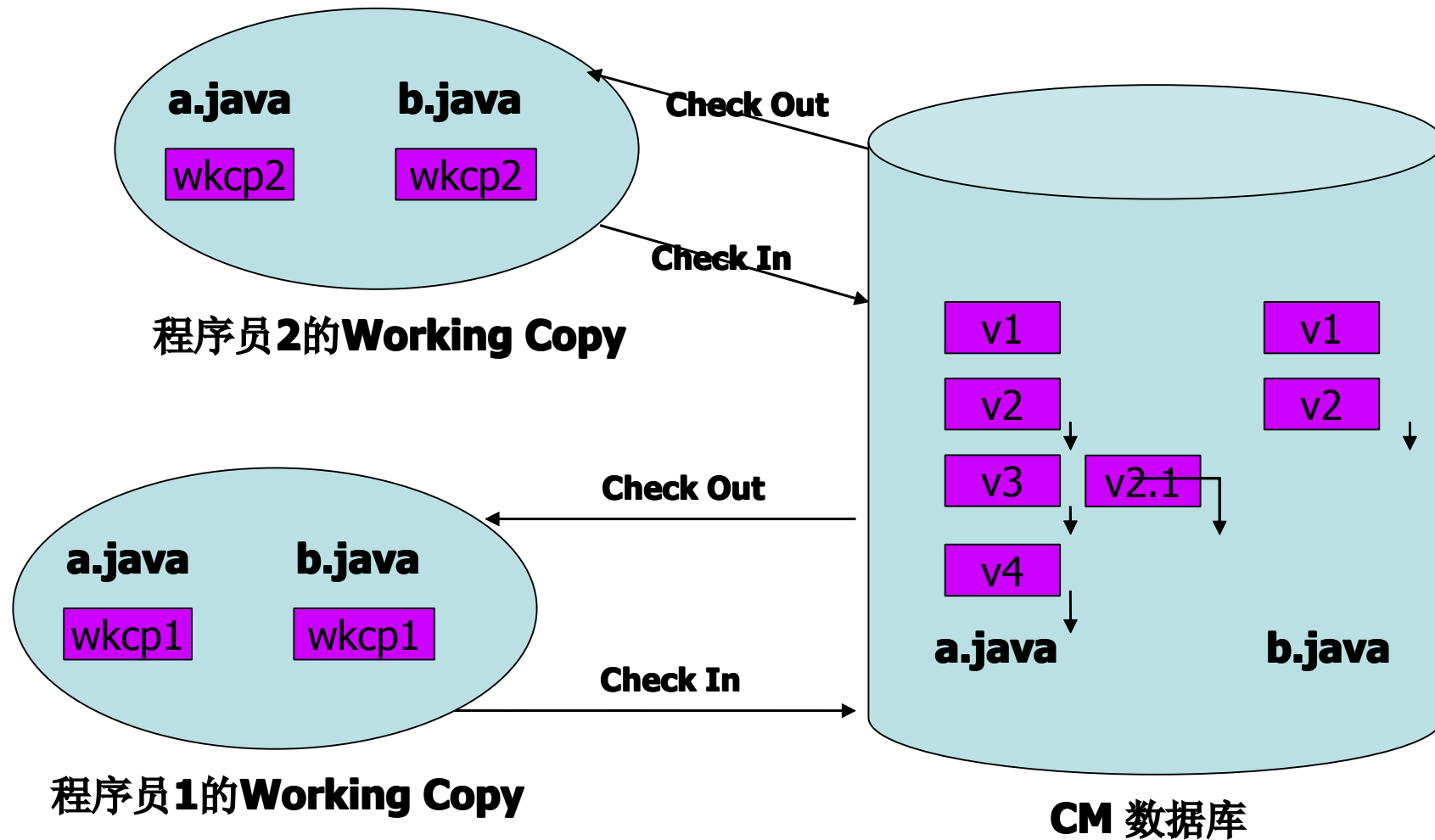
Check In:

配置管理者从程序开发人员的工作空间（目录）中拷贝配制项到项目空间（目录）中。

Check Out:

配置管理者从配置库中检出（Check Out）相应的配置项，拷贝到程序开发人员的工作空间中。

CICO模型



第八章

配置管理工具

本章要点

- 基础知识
- WinCVS登陆
- WinCVS操作
- WinCVS高级版本控制技术

软件配置管理的工具支持

- 开源工具CVS
- 购置配置管理工具进行软件配置管理，如基于团队开发的配置管理工具，提供以过程为基础的包含版本管理，过程控制等功能；
- 利用Visual Studio6.0中所提供的VSS6.0(Visual SourceSafe)工具进行软件配置管理。
- 另外还有butter fly， Rational Clear Case等配制管理工具。

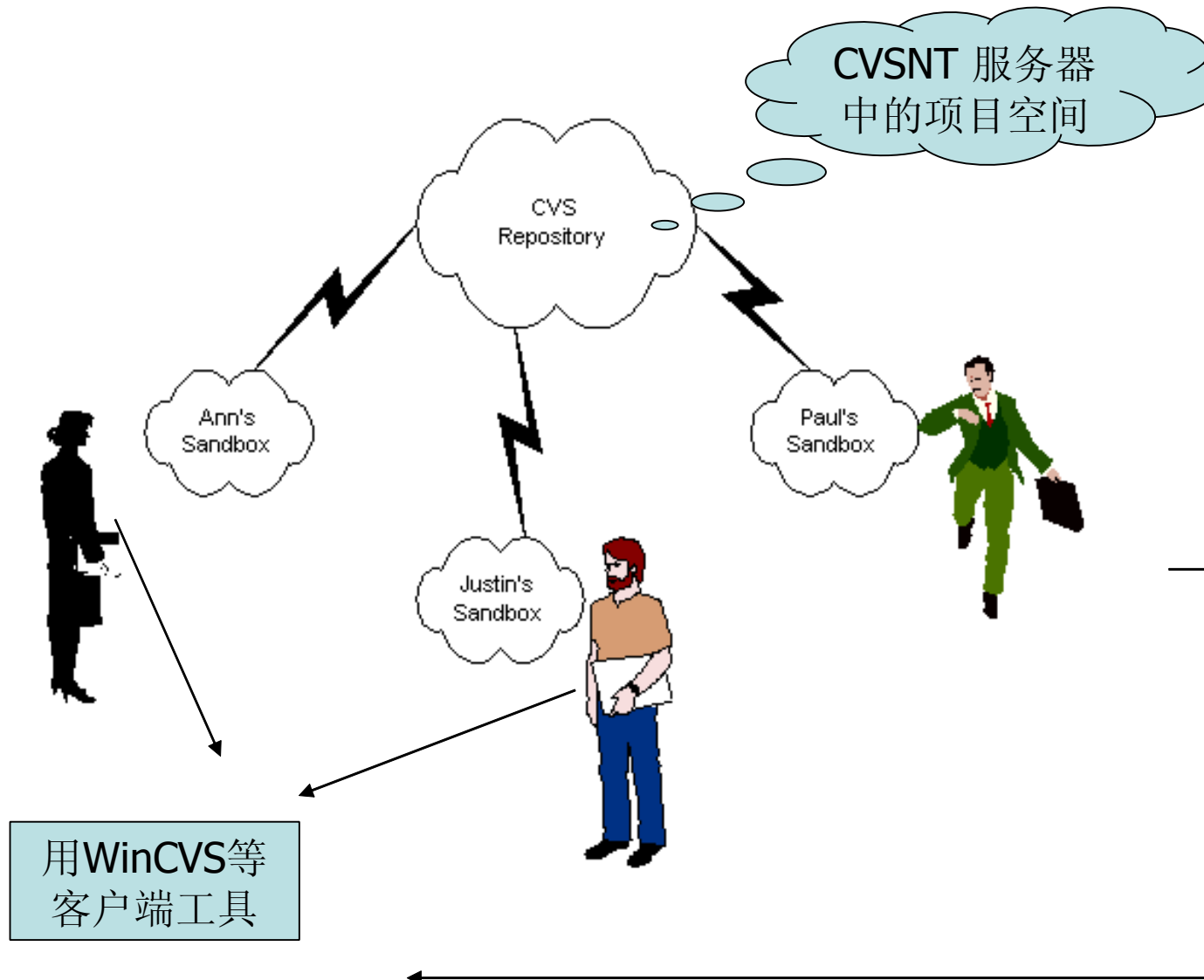
CVS

- Concurrent Versions System
- GNU协议
- C/S
- CVSNT

WinCVS

- CVS客户端工具
- 图形化
- CheckOut和CheckIn
- 开发人员可以配置项的旧版本、查看更改日志等操作

CVS的使用



CVS术语

- CVSROOT: 配置库
- Repository: 项目空间
- WorkSpace: 工作空间
- SourceSpace: 资源空间
- Version: 版本
- Branch: 版本分支
- Tag: 标签
- Update: 更新
- Commit: 提交

配置CVSNT

- 指定临时工作目录
- 创建CVSRoot
- 创建Repository

WinCVS登陆

- 配置登陆选项
- **login**到特定的**Repository**
- 选择工作空间
- **logout**

WinCVS操作

- Import Module
- Checkout Module
- Update Selection
- View Selection
- Graph Selection
- Commit Selection

Import Module

- 目的是资源空间拷贝一个Module到项目空间里面去
- **Create->Import Module**
- 选择本地目录

Checkout module

- 目的是从项目空间选择一个Module检出到工作空间中，以备后续修改
- Create->Checkout module
- Checkout settings

Update selection

- 目的是将工作空间和项目空间中的配置项同步起来
- Update selection 原因
- Modify->Update selection
- Update settings

View Selection

- 目的是查看或修改工作空间中的配置项
- 选择文件，点击鼠标右键，选择菜单 **View Selection**
- 修改文件
- 保存文件

Graph Selection

- 目的是查看配置项的版本树
- **Ctrl+G**选择
- Log settings

Commit selection

- 目的是将配置项从工作空间检入到项目空间中
- 鼠标右键点击配置项，选择“Commit Selection”
- Commit settings

WinCVS高级版本控制技术

- 标签的使用
 - 标签是给配置项贴的文字标记
 - 不同的配置项可以贴相同的标签
- 版本分支
 - 版本分支是让配置项的版本树分叉的动作
 - 版本分支的目的是让配置项沿着不同的方向进化

标签

- 创建标签
- 删除标签
- 按标签检出

版本分支

- 创建分支
- 按分支检出
- 分支上的新版本

第九章

软件测试

本章要点

- 软件测试概述
- 软件测试分类
- 几种单元测试方法
- 组织软件测试
- 软件测试文档

软件测试的定义

- 软件测试定义（**1983，IEEE**）：
“使用人工或自动手段来进行或测定某个系统的过程，其目的在于检验它是否满足规定的需求或是弄清预期结果与实际结果之间的差别。软件测试以检验是否满足需求为目标”。
- 软件质量保证和软件测试不同

测试的目的

- 寻找缺陷。
- 发现新的缺陷。
- 不能保证没有缺陷。

软件测试的原则

- 追溯根源
- 计划在先
- 从小到大
- 不能穷举

正确认识软件测试

- 软件测试不是程序测试
- 测试心态
- 测试的分量

软件缺陷的来源

- 编程错误
- 软件复杂度
- 沟通
- 不断变更的需求
- 时间的压力
- 人员风险意识
- 缺乏文档
- 软件开发工具

总体分类

- 静态测试
 - 代码审查
 - 代码分析
 - 文档检查
- 动态测试
 - 结构测试（白盒）
 - 功能测试（黑盒）

工程分类

- 单元测试
- 集成测试
- 系统测试
- 用户测试



单元测试

是一种白盒测试

非合理输入
系统异常

通过接口的调用参数

数据定义、
使用

模块接口

出错处理

局部数据结构

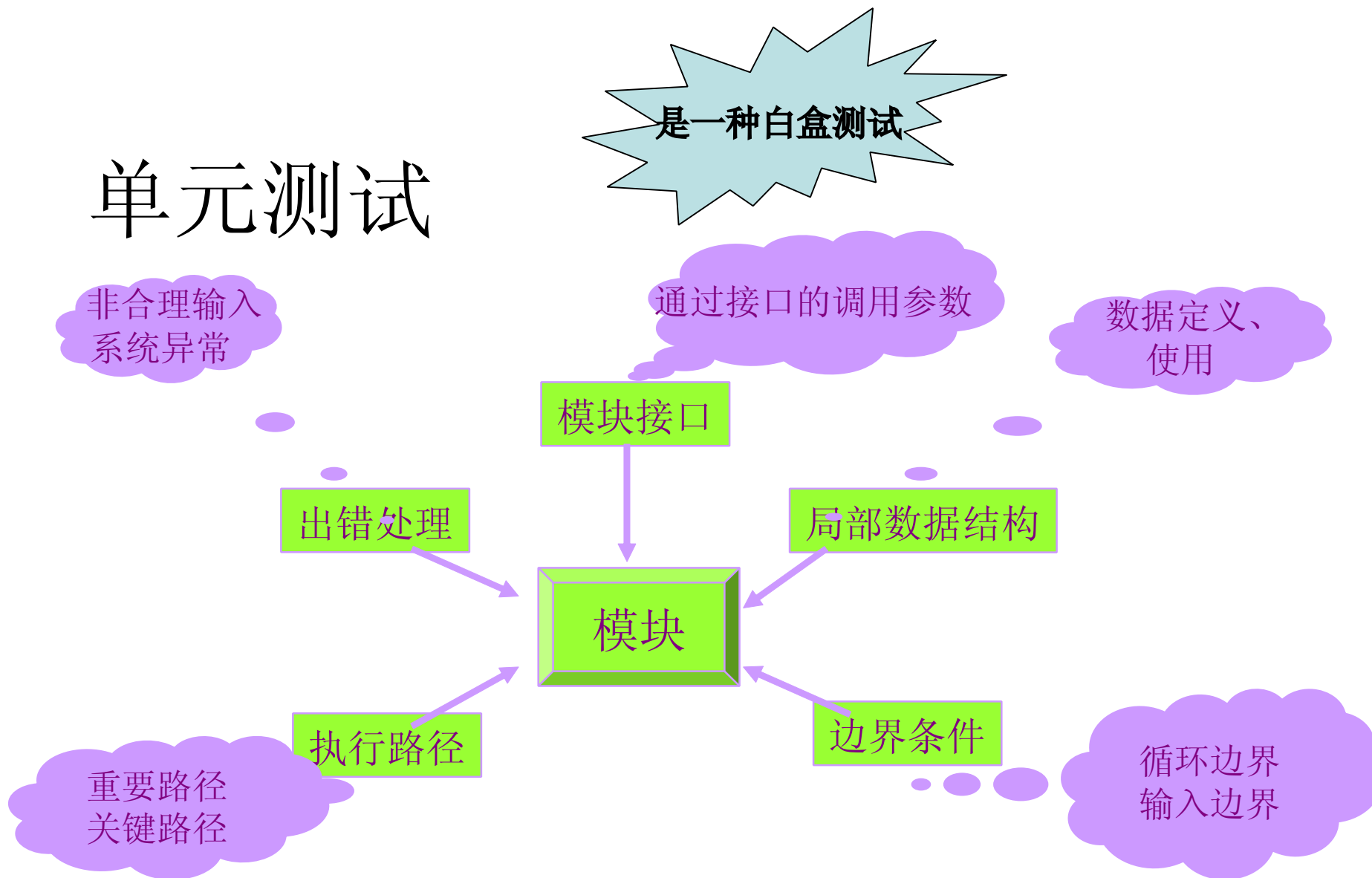
模块

执行路径

边界条件

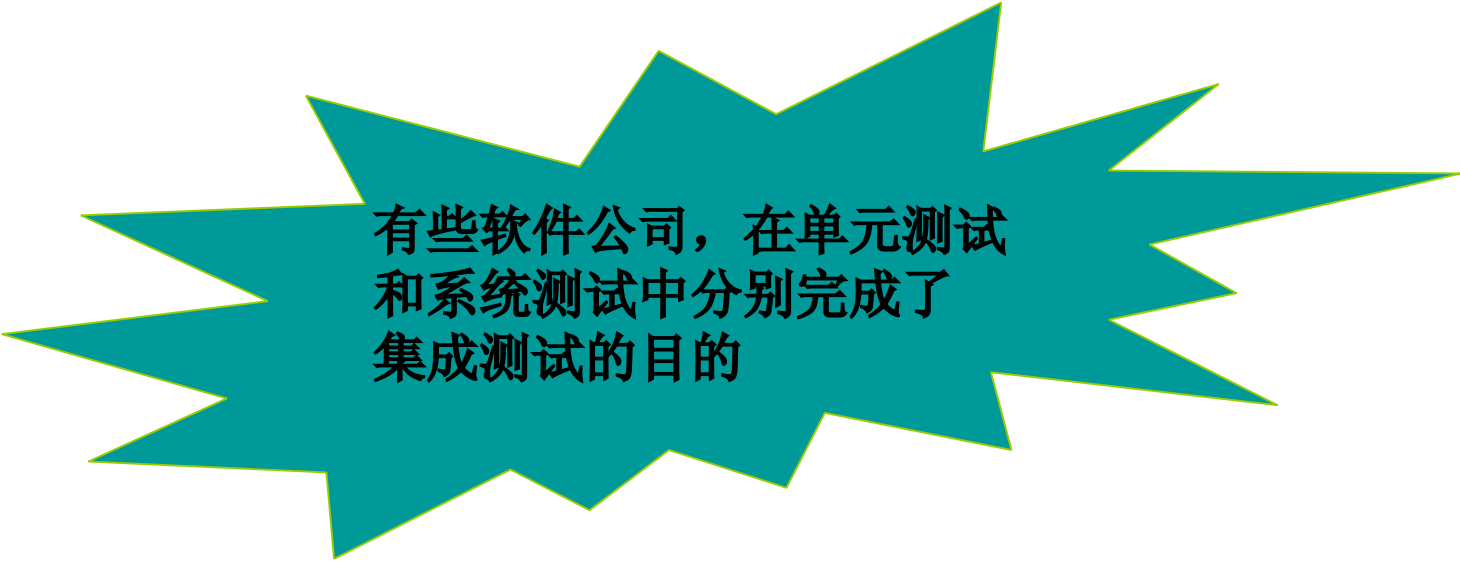
重要路径
关键路径

循环边界
输入边界



集成测试

- n 全局数据结构是否正确
- n 通过模块接口的数据是否正确
- n 模块间的耦合影响性能了吗



有些软件公司，在单元测试和系统测试中分别完成了集成测试的目的



也叫功能测试
是一种黑盒测试

系统测试

◆ 有效性测试

在模拟的环境下，运用黑盒测试的方法，验证所测软件是否满足需求规格说明书列出的要求

◆ 软件设计复查

保证软件配置的所有成分都齐全，各方面的质量都符合要求，具有维护阶段所必需的细节，而且已经编排好分类的目录

用户测试

◆ α 测试

测试人员模拟最终用户测试

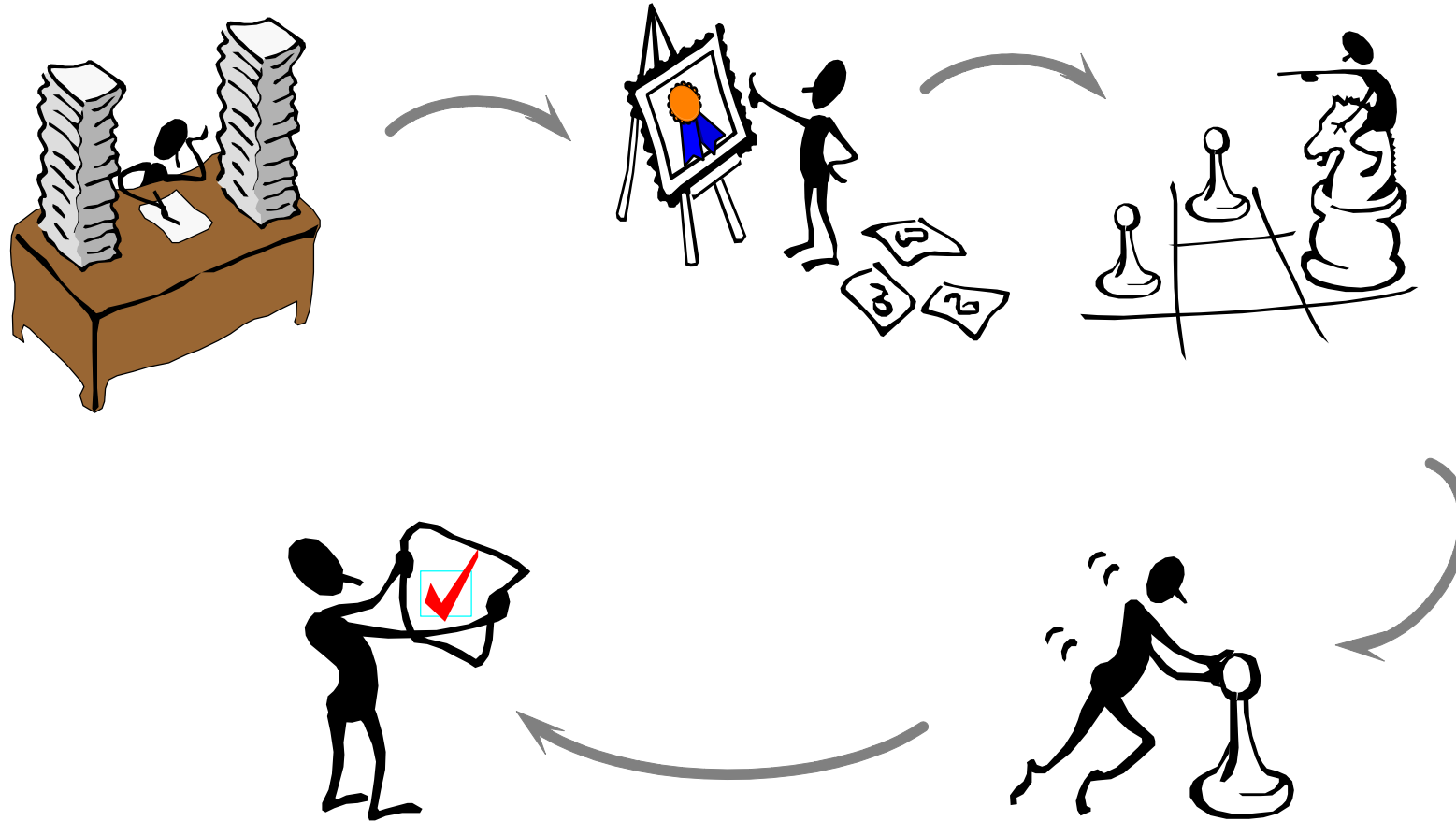
◆ β 测试

最终用户的实际测试

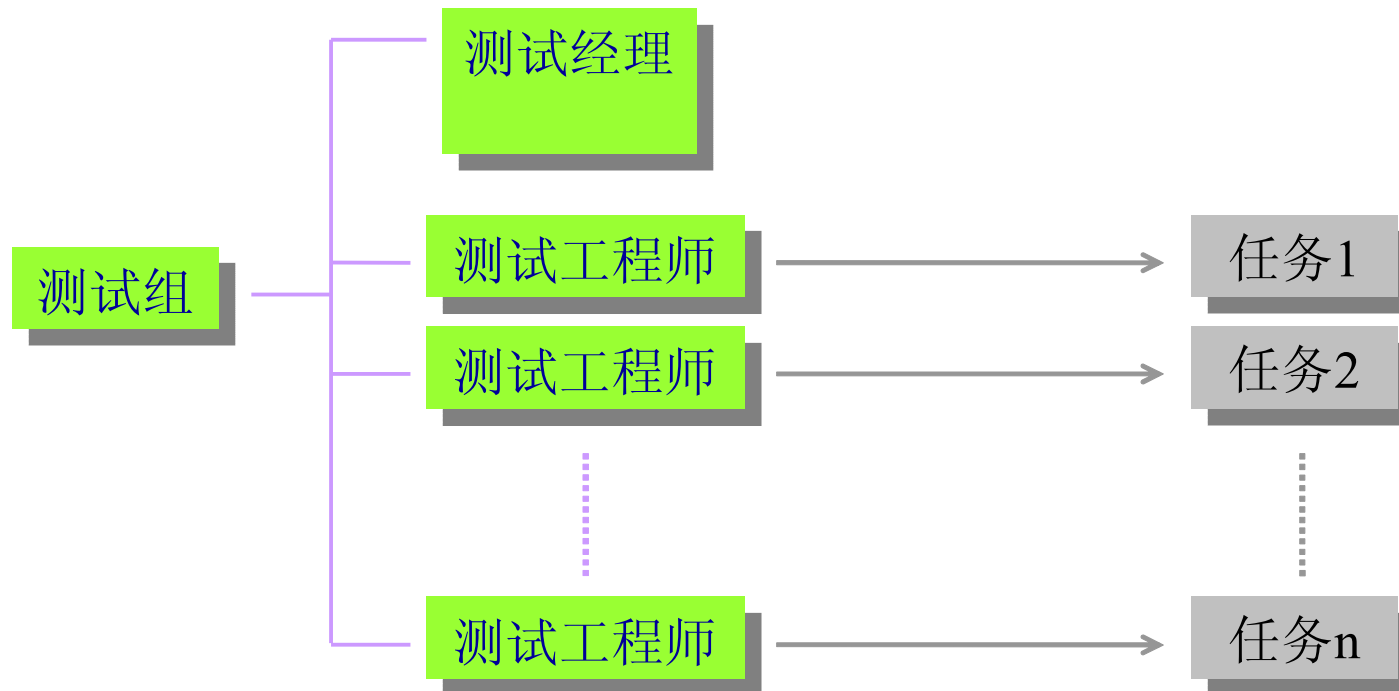
几种单元测试方法

- 边界值分析
- 语句覆盖
- 判定覆盖
- 条件覆盖
- 判定-条件覆盖
- 路径覆盖

测试活动的生命周期



软件测试小组



测试文档

- 测试计划
- 测试用例

第十章

软件测试工具

内容简介

- JUnit 背景
- 用TestCase测试一个类
- TestRunner
- TestCase原理
- 用TestSuite一次测试多各类
- TestSuite原理

JUnit是什么？



- 开源的
- 单元测试框架
- xUnit
- JUnit 的特点
- www.junit.org

JUnit与XP

- Extreme Programming
- 测试驱动开发
- 单元测试用例
- JUnit与XP
- Kent Beck和Erich Gamma

用TestCase测试一个类

- 新建JBuilder9 项目
- 给项目添加一个新类
- 全手工写的测试代码
- 创建测试用例
- 执行测试用例

新建JBuilder 项目

- 点击菜单**File->New Project**
- **Step1:** 设立项目名称
- **Step2:** 点击**Next**到下一步
- **Step3:** 点击**Next**到下一步
- 项目建好

给项目添加一个新类

- **Ctrl+N->General Tab->Class**
- 填写类名称
- 类的原型生成
- 给类添加方法
- 类创建完毕

创建测试用例

- Ctrl+N->Test Tab->TestClass
- Step1: 选择方法
- Step2: 测试用例类名称
- Step3: 点击Next
- Step4: 设置SwingUI
- 测试案例类原型生成

执行测试用例

- 执行测试用例类**TestCompute**
- 测试结果没发现错误
- 更改**testAdd**方法在执行一次
- 发现了断言失败

TestRunner

- Text 版
- Awt 版
- Swing 版

TestCase 原理

- TestCase类继承结构
- TestCase生命周期
- 断言

用TestSuite一次测试多各类

- 再给项目添加一个新类
- 给新类创建测试用例类
- 创建测试套件类
- 执行测试套间类

TestSuite原理

- 静态suite方法
- addTestSuite方法
- addTest(Test)方法
- run(TestResult)方法

JUnit基本框架

