



IBM Software Group

软件架构原理

Rational software

IBM 软件部 王家欣

A horizontal decorative bar spanning the width of the slide, featuring a series of colored squares (cyan, green, yellow, red, purple) followed by a row of icons: a crane, a circular arrow, a person, a document, a globe, a starburst, and a four-way arrow.

ON DEMAND BUSINESS

© 2006 IBM Corporation

内容

- ➡ ■ 什么是软件架构？
- 架构带来什么好处？
- 以架构为核心的软件开发过程
 - ▶ 用例与架构
 - ▶ 架构的步骤
 - ▶ 架构的描述与架构基线
 - ▶ 架构开发实现



架构一词的来源

- 建筑行业：
 - ▶ 建筑中包含多个层面的技术
 - 外观、挖掘、地基、结构、墙、地板、电梯、电气、空调、水、卫生
 - 其它专门为居住者提供服务的设施
 - ▶ 架构师需要把所有的层次结合起来：
 - 使客户理解
 - 在建造的过程中为施工者提供指导
- 架构相关于所有事情
 - ▶ 架构为所有人提供一个共同的远景目标
- 架构不包括每个部分的细节



Architecture: 三种不同的解释

- Architecture as a *process* or *discipline*
 - ▶ 建造供所有类型的人使用的大厦的艺术或科学The art or science of building or constructing edifices of any kind for human use
 - ▶ 建造的活动或过程The action or process of building
 - ▶ 依据建筑物的详细的结构和装饰的组织而采用的特殊的方法或风格The special method or style in accordance with which the details of the structure and ornamentation of a building are arranged
- Architecture as an *artifact*
 - ▶ 架构设计工作: 结构, 建筑物
 - ▶ 总体的构造或结构

Webster's Ninth Collegiate Dictionary



什么是软件架构？

- 期望其与建筑架构起到相同的作用：
 - ▶ 将软件的所有层次组合在一起
 - ▶ 便于客户理解
 - ▶ 为建造过程提供指导
- 软件架构包含了过于下列方面的重要决定：
 - ▶ 软件系统的组成
 - ▶ 对所包括的系统及其接口的结构元素的选择，以及元素间的协作行为
 - ▶ 结构和行为元素如何组成不断增长的更大的子系统
 - ▶ 架构风格：组成元素与接口、相互协作、相互组合
- 架构元素不仅与结构和行为有关，也和用法、功能、性能、适应性、重用、可理解性、经济和技术限制、折中、美学等有关



软件架构定义的发展

Perry and Wolf, 1992

A set of architectural (or design) elements that have a particular form. Perry and Wolf further distinguish between processing elements, data elements, and connecting elements.

Boehm et al., 1995

A software system architecture comprises

- A collection of software and system components, connections, and constraints
- A collection of system stakeholders' need statements
- A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements.

Clements et al., 1997

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Paul Clement's page @ <http://www.sei.cmu.edu/architecture/definitions.html>



所有定义中的公共元素

- 架构定义了关键 组件
- 架构定义了组件之间的 关系 (结构) 和 交互
- 架构忽略了组件中与组件之间交互无关的内容的信息
- 从另外一个组件的视点能够观察到的组件的行为是架构的一个部分
- 每个系统都有一个架构 (即便这个系统只有一个组件组成)
- 架构定义了组件和结构背后的 基本原理
- 在架构定义中不包含组件的定义
- 架构不是简单的结构定义 – 在架构定义中不只是单一的结构定义



RATIONAL 对架构的定义

- 软件架构将关于软件系统的组织的正确的决定组合在一起
 - ▶ 系统组成所需要的结构元素以及它们之间的结构的选择
 - ▶ 在这些元素之间的特定的协作行为
 - ▶ 将这些结构元素和行为元素组合成为一个更大的子系统
 - ▶ 用于制导系统组织的架构风格

*Grady Booch, Philippe Kruchten, Rich Reitman, Kurt Bittner; Rational
(derived from Mary Shaw)*



RATIONAL 对架构的定义 (2)

- 软件架构也关系到
 - ▶ 功能性Functionality
 - ▶ 可用性Usability
 - ▶ 系统弹性Resilience
 - ▶ 性能Performance
 - ▶ 重用Reuse
 - ▶ 可理解性Comprehensibility
 - ▶ 经济和技术的约束及相关折中Economic and technology constraints and tradeoffs
 - ▶ 美学的考虑Aesthetic concerns



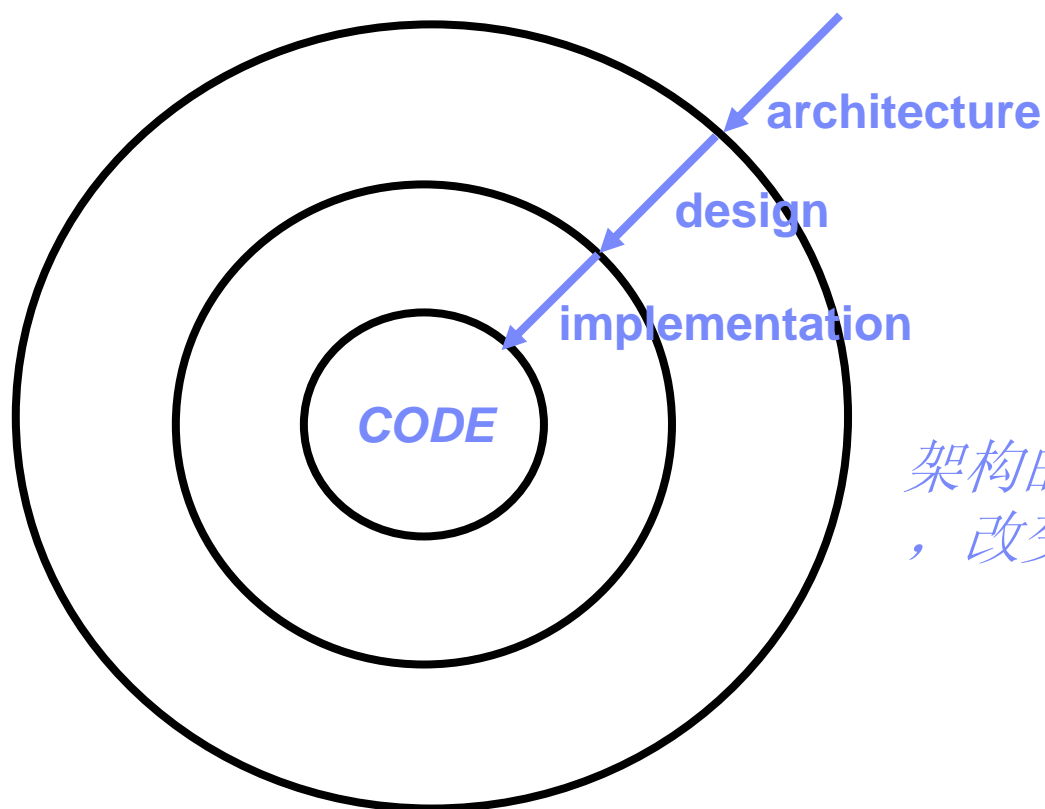
(软件)组件是什么？

- 可以被用于分布、集成、交付、替换的单元
- 实现关键的功能
- 由组件所提供的和它所要求的（需要的）服务定义
 - ▶ 服务只展现集成所必需的组件行为
- 在下列的上下文关系中有意义：
 - ▶ 架构
 - ▶ 组件模型和框架
- 不一定是一个单一的代码单元或单一的二进制文件



架构约束设计和实施

- 架构涉及到一系列对设计和建造起约束作用的策略性的设计决定、规则和模式



架构的决定是最为基本的决定
，改变他们将掀起宣然巨波

常见的错误概念

- 架构和设计是相同的事情
- 架构和基础架构(infrastructure)是相同的事情
- 架构就是结构
- 架构是平面化的，一个蓝图就足够了
- 架构是不能够被度量和验证的
- 架构是艺术或架构是科学



错误概念: 架构 = 设计

- 架构是设计的一个表象，它关注于：
 - ▶ 对于结构来讲关键的元素
 - ▶ 对于性能、可靠性、成本、适用性等有重大影响的元素
- 架构捕获了一组重要的设计决定
- 架构不关心每个独立元素的详细设计



错误概念: 架构 = 基础架构Infrastructure

- 架构比基础架构包含的内容更多
- 基础架构是一个完整的、重要的架构的组成部分
- 架构也定义了也定义了应用在基础架构上的运行
- 架构定义了基础架构和应用组件之间的互操作性



错误概念: 架构 = 结构

- 架构相关于结构、分解、接口等
- 架构比结构包含的更多:
 - ▶ 动态表象
 - ▶ 基本原理
 - ▶ 适合于上下文关系:
 - 业务上下文
 - 开发上下文



错误概念: 架构是平面化的

- 只有在那些非常细小的案例中，架构才是平面化的
- 架构有很多的维度，分别表述不同的涉众的不同的关注点
- 利用一个单一的蓝图来表达全部（大部分的）的维度将导致语义的过度使用和不完整
- 架构需要多个视图



错误概念: 架构是不能够被度量和验证的

- 架构不是一个粗略的、用草纸和铅笔进行的顶层的设计
- 对于架构能够就功能和质量的需求、风险、和其他的系统关键属性进行系统化地评估
 - ▶ 评审架构的工件
 - ▶ 测试架构原型



错误概念: 架构是艺术或架构是科学

- 两者都不是/两者都是
- 难于应用分析方法
- 非常大的问题空间，却只有非常少的可选择的量化的度量标准
- 好的架构师拷贝过去成功地解决方案并与当前增加的改进结合
- 架构定义的过程有明确定义的步骤和明确描述的工件
- 架构相关的知识体系开始被编成法典
 - ▶ 模式, 框架, 服务, 启发模式, ...
- 直觉和创造性仍然重要

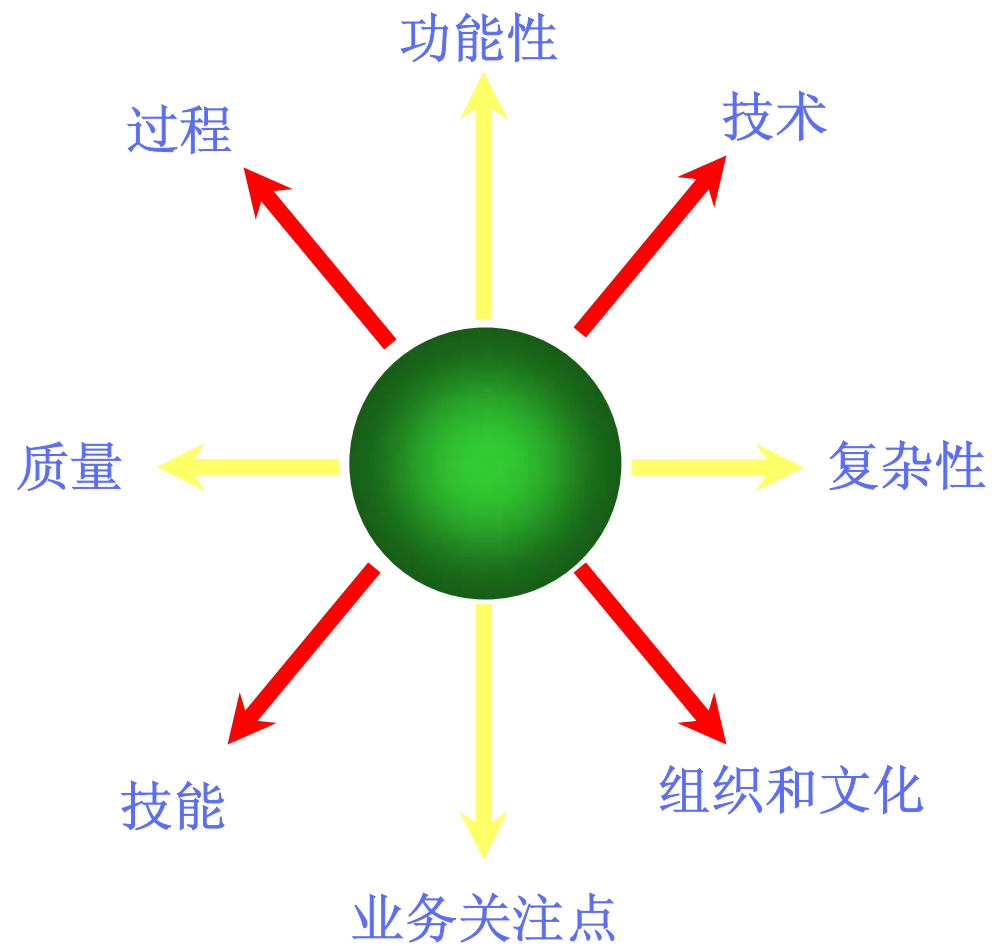


内容

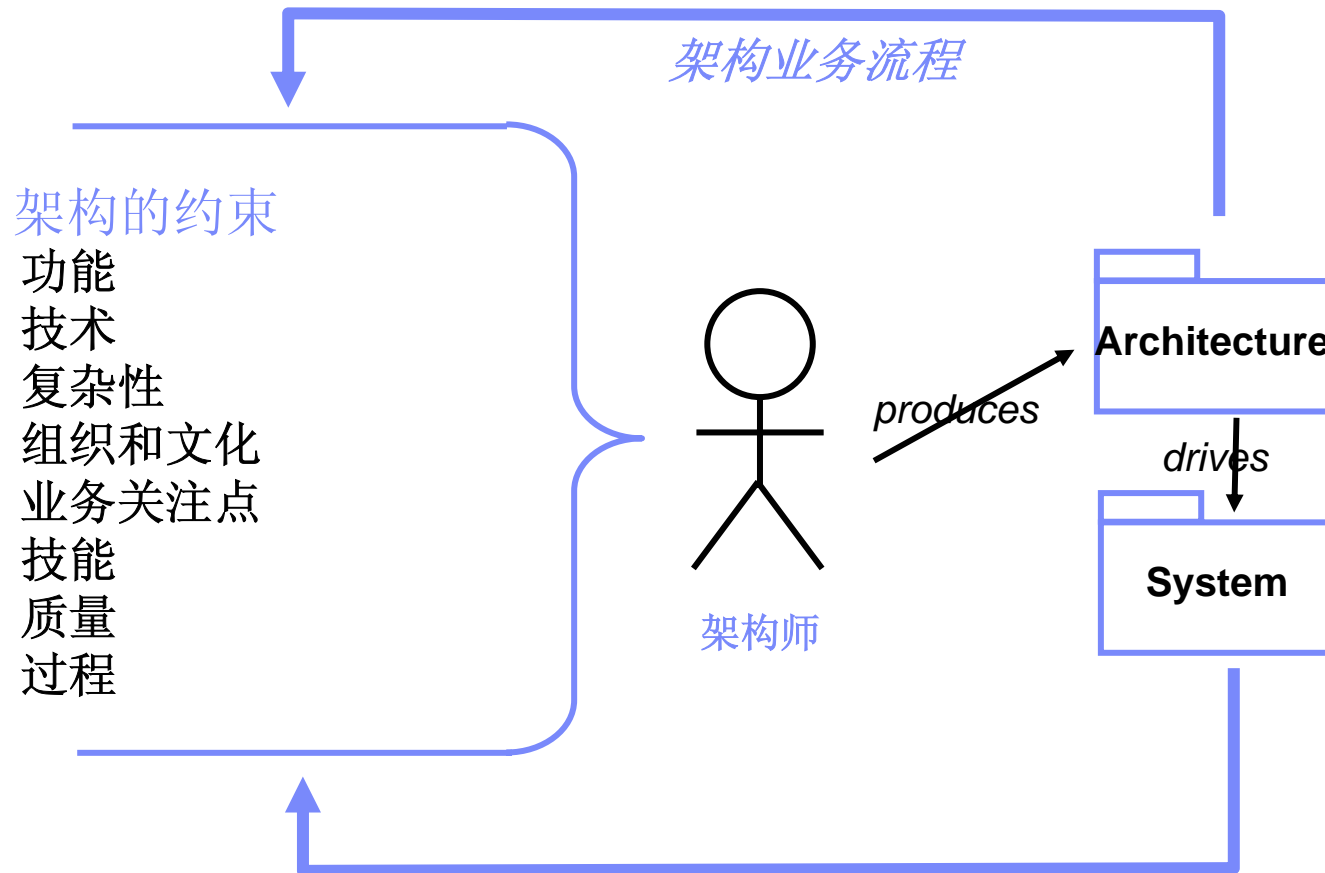
- 什么是软件架构？
- ➡ ■ 架构带来什么好处？
- 以架构为核心的软件开发过程
 - ▶ 用例与架构
 - ▶ 架构的步骤
 - ▶ 架构的描述与架构基线
 - ▶ 架构开发实现



架构是权衡



架构的业务流程



Software Architecture in Practice: L. Bass, P. Clements, R. Kazman, Addison-Wesley, 1998



好处: 系统完整性和质量

- 架构能够在变更中保持概念的完整性和系统质量
 - ▶ 所要求的功能
 - ▶ 质量属性
 - ▶ 新需求
 - ▶ 变更的需求
 - ▶ 融合技术
- 架构延长系统的寿命
 - ▶ 容易进化
 - ▶ 系统弹性
 - ▶ 弹性化的应变能力



好处: 控制复杂度

- “排除与克服”
 - ▶ 分解到组件
 - ▶ 隐藏实施的细节
- 关注点的分离
 - ▶ 组件 (层次) 封装细节
 - ▶ 不同组件可以由掌握不同专业技能的人员来实施



好处: 可预见

- 过程可预见性
 - ▶ 架构原型允许你收集度量指标
 - 开发成本的度量指标
 - 进度的度量指标
- 行为可预见性
 - ▶ 架构迭代排除关键风险



好处: 可测试性

- 良好构件化了的系统支持更好的、更容易的
 - ▶ 诊断
 - ▶ 跟踪能力
 - ▶ 发现错误
- 测试是用例驱动的



好处: 重用

- 架构定义了替换规则
- 组件接口定义替换物的边界
- 架构使得各种粒度的重用成为可能
 - ▶ 组建级别的小范围的重用
 - ▶ 大范围的重用
 - 子系统
 - 产品
 - 框架



好处: 沟通

- 架构支持涉众间的沟通
- 不同的视图定位于不同涉众的关注点
- 架构沟通的是关键的设计决定
- 架构设计的基本原理沟通折中



好处: 组织和项目管理

- 组织的结构与架构协同
 - ▶ 开发团队
 - ▶ 分包商
- 组件/子系统作为下列单元使用:
 - ▶ 开发
 - ▶ 配置管理
 - ▶ 测试
 - ▶ 交付与升级
- 组件/子系统被并行开发



架构的成本是先期支付的

- 架构的投资发生在系统开发的早期阶段
 - ▶ 启动阶段
 - ▶ 精化阶段
- 在前期阶段的成本估算模型是不精确的
- 架构所带来的收益发生在实施和维护阶段

Software Project Management: A Unified Framework (Appendix B), Walker Royce, Addison Wesley, 1995



好处: 开销避免/节省

- 最小化返工带来的开销
- 重用带来的节省
 - ▶ 组件的重用
 - ▶ 开发可重用的组件
- 通过高效的资源利用带来的节省
- 通过准确的成本/进度估计带来的节省
- 从改进的维护和支持能力带来的节省



架构很昂贵 (?)

- 管理通常总是忽略先期支付的项目成本
- 健壮的架构对于复杂的、关键性的、尺寸增大的项目尤为关键
- 好的架构帮助估算和控制开发成本

*从长期的角度看，，没有架构
的系统是十分昂贵的！*



内容

- 什么是软件架构？
- 架构带来什么好处？
- ➡ ■ 以架构为核心的软件开发过程
 - ▶ 用例与架构
 - ▶ 架构的步骤
 - ▶ 架构的描述与架构基线
 - ▶ 架构开发实现



成功地开发

- 只有正确地开始，才能正确地结束
 - ▶ 过早地仓促地进入系统的构建是错误的开始
 - ▶ 正确地开始：确保需求正确，并使其处于良好状态，以驱动开发
- 成功开发的步骤
 - ▶ 把需求组织成用例的形式
 - ▶ 制定出稳定的架构，作为可执行的基线知道建造工作
 - ▶ 迭代的过程，按照事情的优先顺序渐进完成



内容

- 什么是软件架构？
- 架构带来什么好处？
- 以架构为核心的软件开发过程



- ▶ 用例与架构
- ▶ 架构的步骤
- ▶ 架构的描述与架构基线
- ▶ 架构开发实现



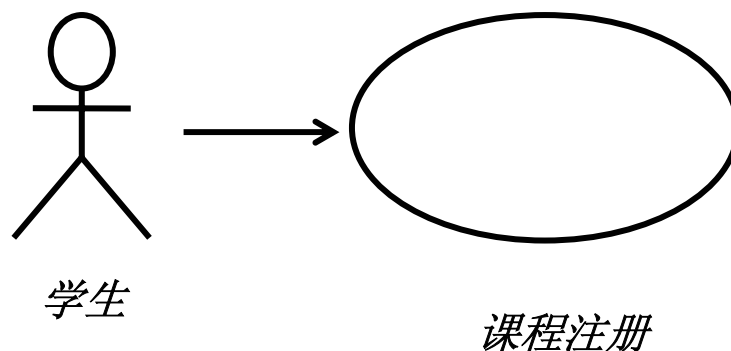
用例驱动开发

- 需求常见困难：
 - ▶ 传统的需求说明书是困难的根源
 - 主要需求和次要需求混杂
 - 缺乏明显的到可用性的跟踪能力
- 用例解决这一困难
 - ▶ 用例纪录几乎所有需求（功能性，非功能性）
 - ▶ 每个用例纪录一部分功能性需求，并组织到用例模型中
 - ▶ 非功能性需求附加在特定用例上
 - ▶ 不单纯描述系统应该做什么，描述系统针对每个用户做什么



什么是用例？

- 用例在UML的图中展现



- 用例需要利用文本描述

Use-Case Specification – Register for Courses

Brief Description

This use case allows a Student to register for course offerings in the current semester. The Student can also modify or delete course selections if changes are made within the add/drop period at the beginning of the semester. The Course Catalog System provides a list of all the course offerings for the current semester.

Actors

1. *Primary Actor – Student*
2. *Secondary Actor - Course Catalog System*

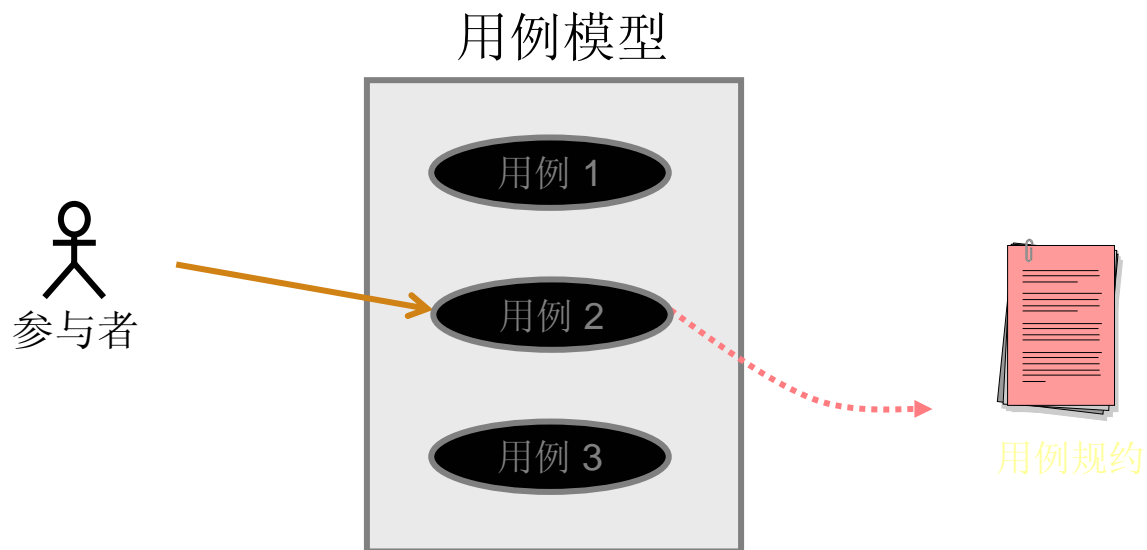
Flow of Events

1. Basic Flow

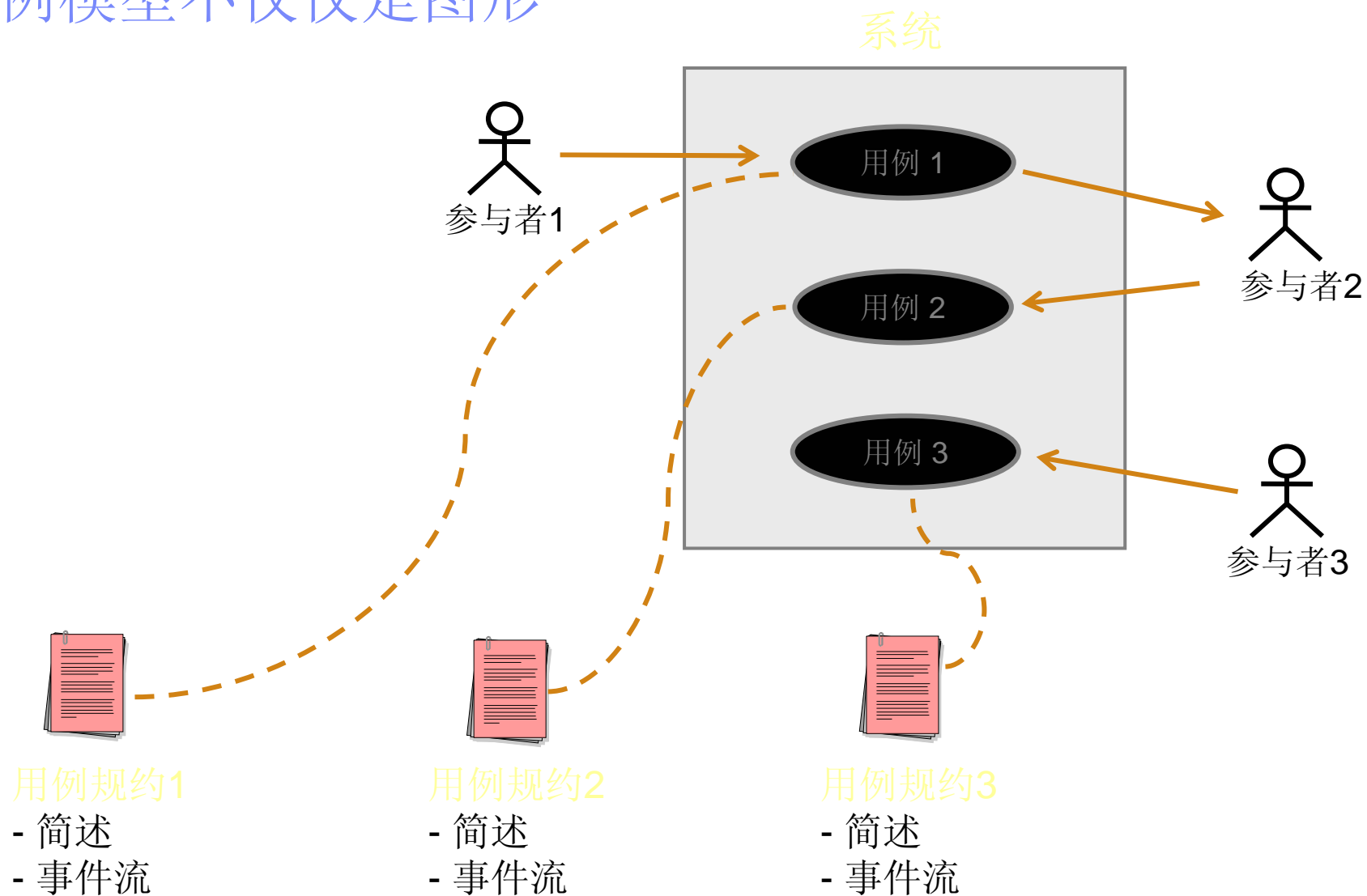
- 1.1. LOG ON.
This use case starts when a student accesses the Course Registration System. The student enters a student ID and password and the system validates the student.
- 1.2. CREATE SCHEDULE.
The system displays the functions available to the student. These functions are: Create A Schedule, Modify a Schedule and Delete a Schedule. The student selects 'Create a Schedule'.
- 1.3. SELECT COURSES
The system retrieves a list of available course offerings from the Course Catalog System and displays the list to the student. The Student selects up to 4 primary course offerings and 2 alternate course offerings from the list of available offerings. The student can add and delete courses as desired until choosing to submit the schedule.
- 1.4. SUBMIT SCHEDULE.
The student indicates that the schedule is complete. The system validates the courses selected and displays the schedule to the student. The system displays the confirmation number for the schedule. The system saves the student's schedule information. The use case ends.

什么是用例建模?

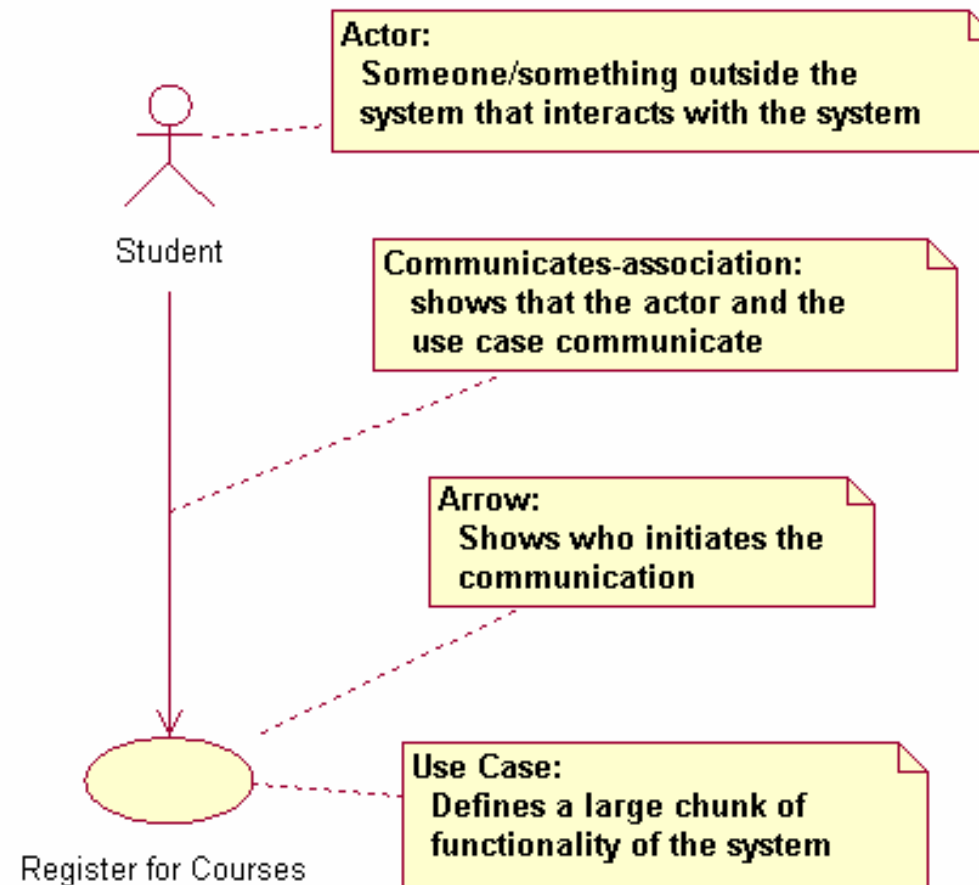
- 一种描述功能性需求的方法
- 清晰地定义系统的边界
- 详细描述系统和外部环境的交互



用例模型不仅仅是图形

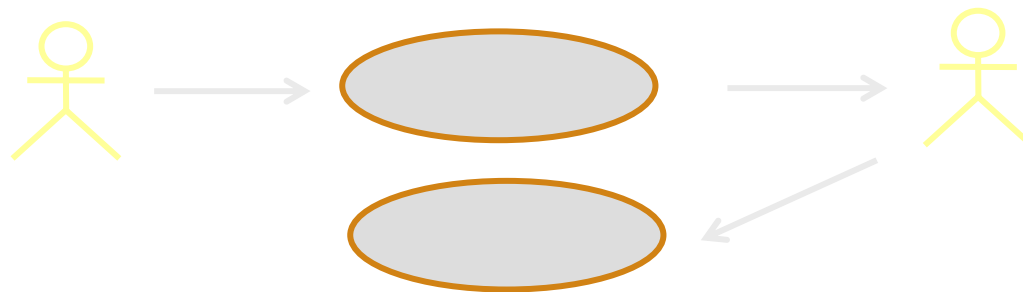


UML图形符号: 参与者（Actor）及关联关系

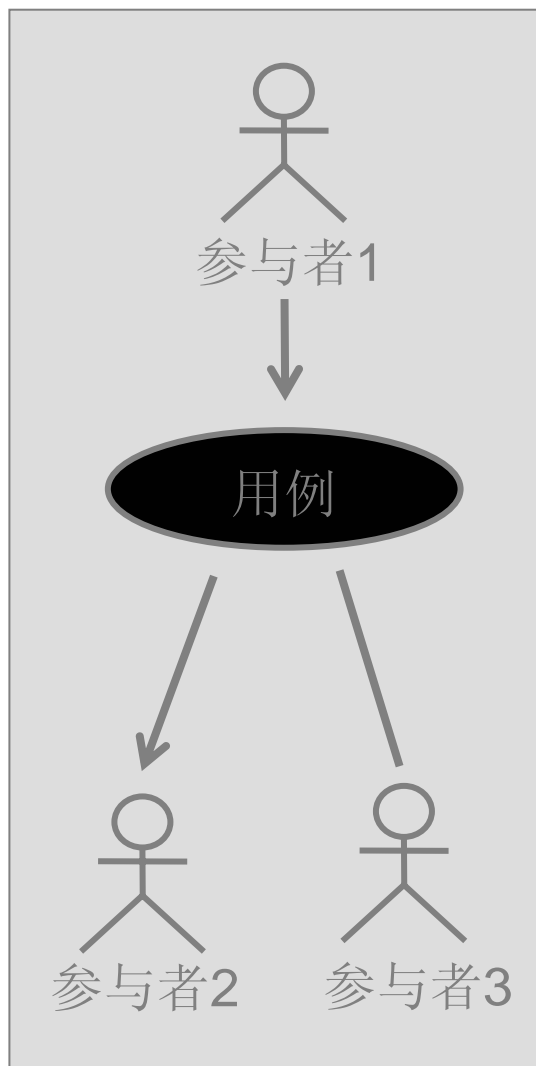


用例描述了系统需求

- 每一个用例
 - ▶ 表述了系统向参与者所提供的某种服务
 - ▶ 也表述了参与者是如何使用系统的
 - ▶ 描述了系统和参与者之间的一段对话



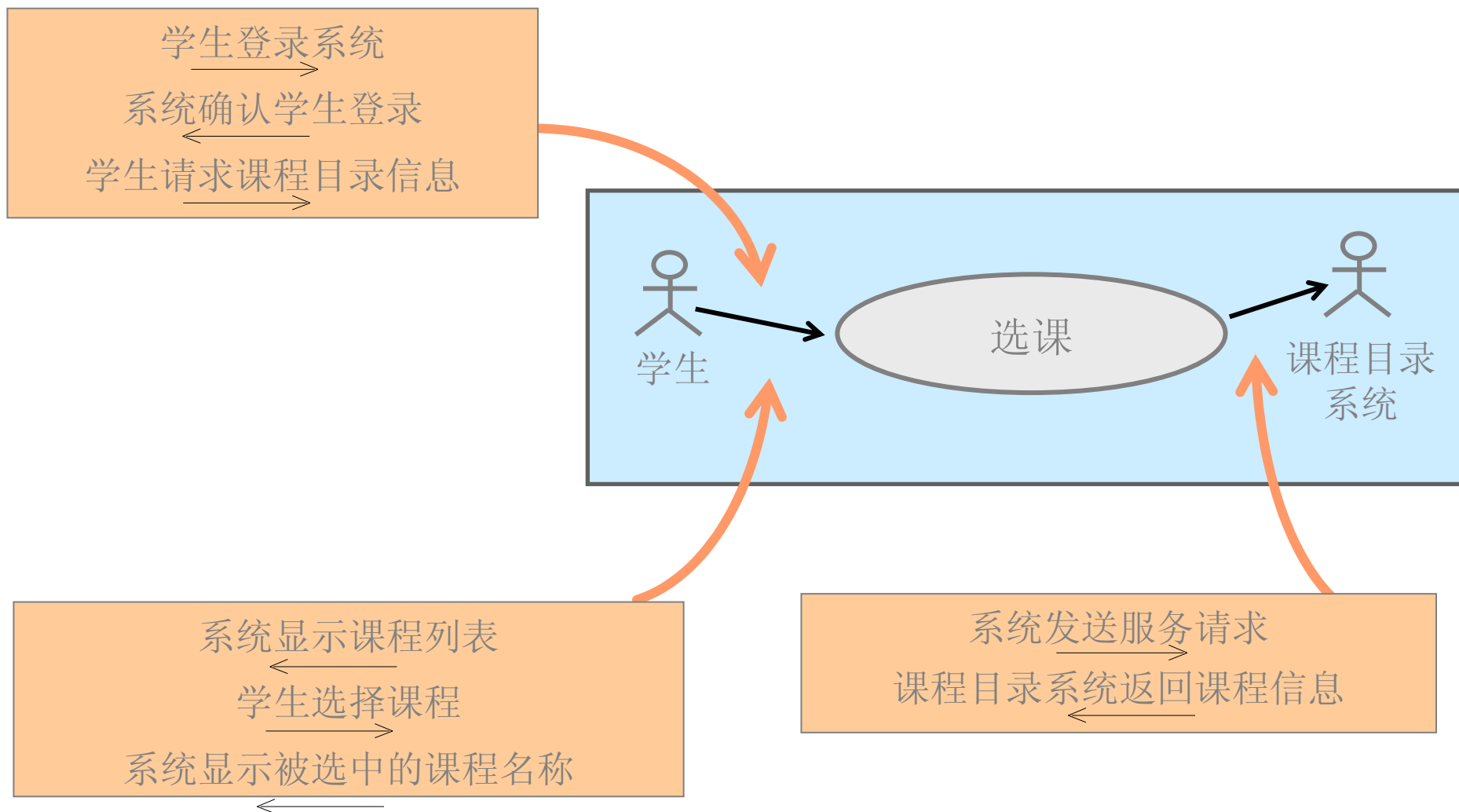
关联 (Association)



- 参与者和用例之间进行对话的一个渠道
- 用一条带或不带箭头的线来表示
 - ▶ 箭头表示是谁发起了这次对话
 - ▶ 没有箭头表示任何一方都可以发起对话
 - ▶ 箭头并不表示数据的流向，数据流向总是双向的



每一个通讯关联都代表了一段对话



用例的定义

A use case describes a sequence of actions a system performs that yields an observable **result of value** to a particular actor (UML 1.4)

A use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, **of value** for one or more actors or other stakeholders of the system (UML 2.0)

- 注意：UML并没有明确规定一个用例的文本描述应该是什么结构，内容怎样组织，如何编写
- 如何编写测试用例将很大程度上影响依据他进行设计和对它进行测试是否容易。



一个用例的内容

Use Case Name

1 Brief Description

2 Actors

3 Flows of Events

3.1 Main (basic) Flow

3.1.1 Step 1

3.1.2 Step 2

3.1.3 Step ...

3.2 Alternative Flows

3.2.1 Alternative flow 1

3.2.1.1 Step 1

3.2.1.2 Step 2

3.2.1.3 Step ...

3.2.2 Alternative flow 2

3.2.3 Alternative flow ..

4 Special Requirements

4.1 Usability requirements

4.2 Business rules

4.x Other non-functional requirements...

5 Pre-conditions

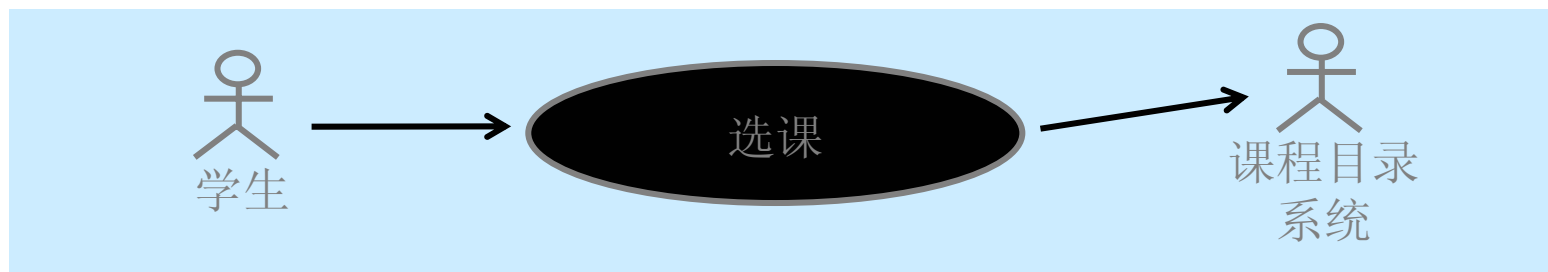
6 Post-conditions

7 Extension Points

- 一个基本事件流
 - ▶ 正常的场景
- 很多备选事件流
 - ▶ 常规选择
 - ▶ 极端（边界）情况
 - ▶ 例外（错误）处理流



场景是用例的一个实例



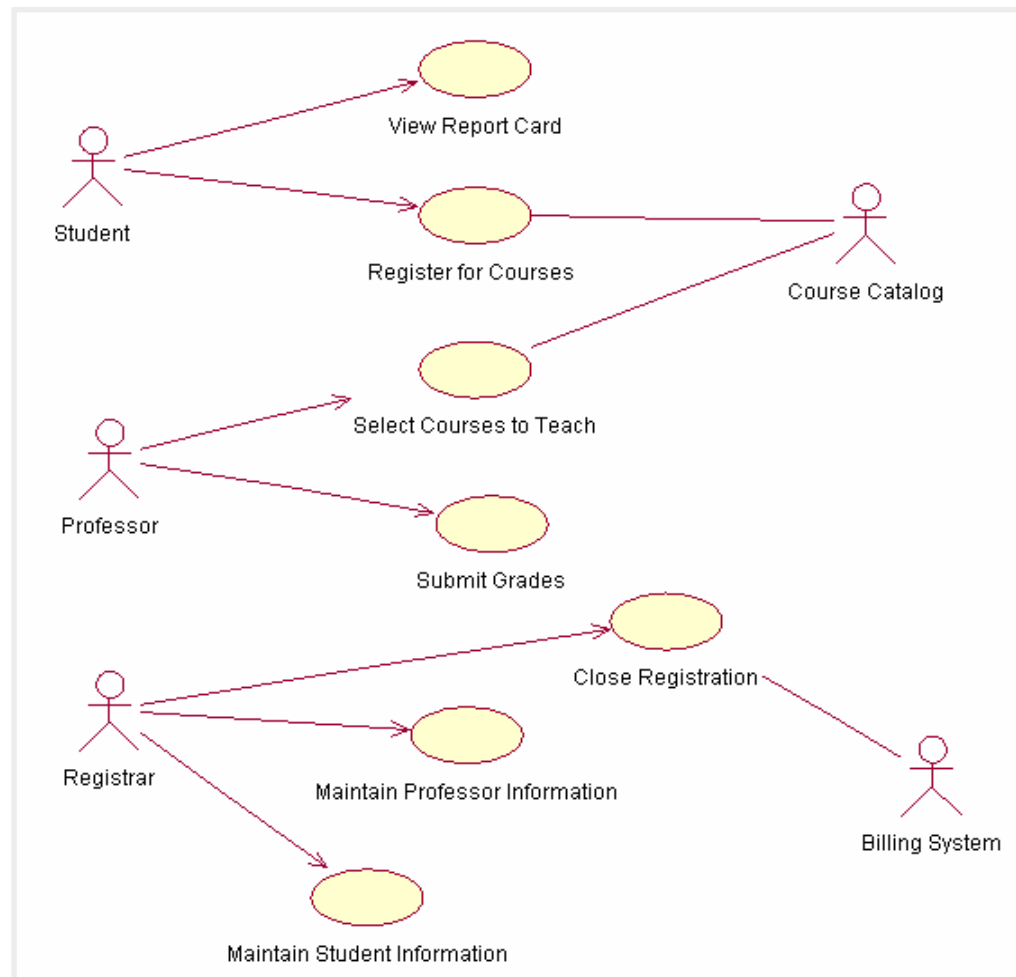
场景 1

登录系统
批准登录
输入搜索主题
获得课程列表
显示课程信息
选择课程
确认课程可选
显示最终课表

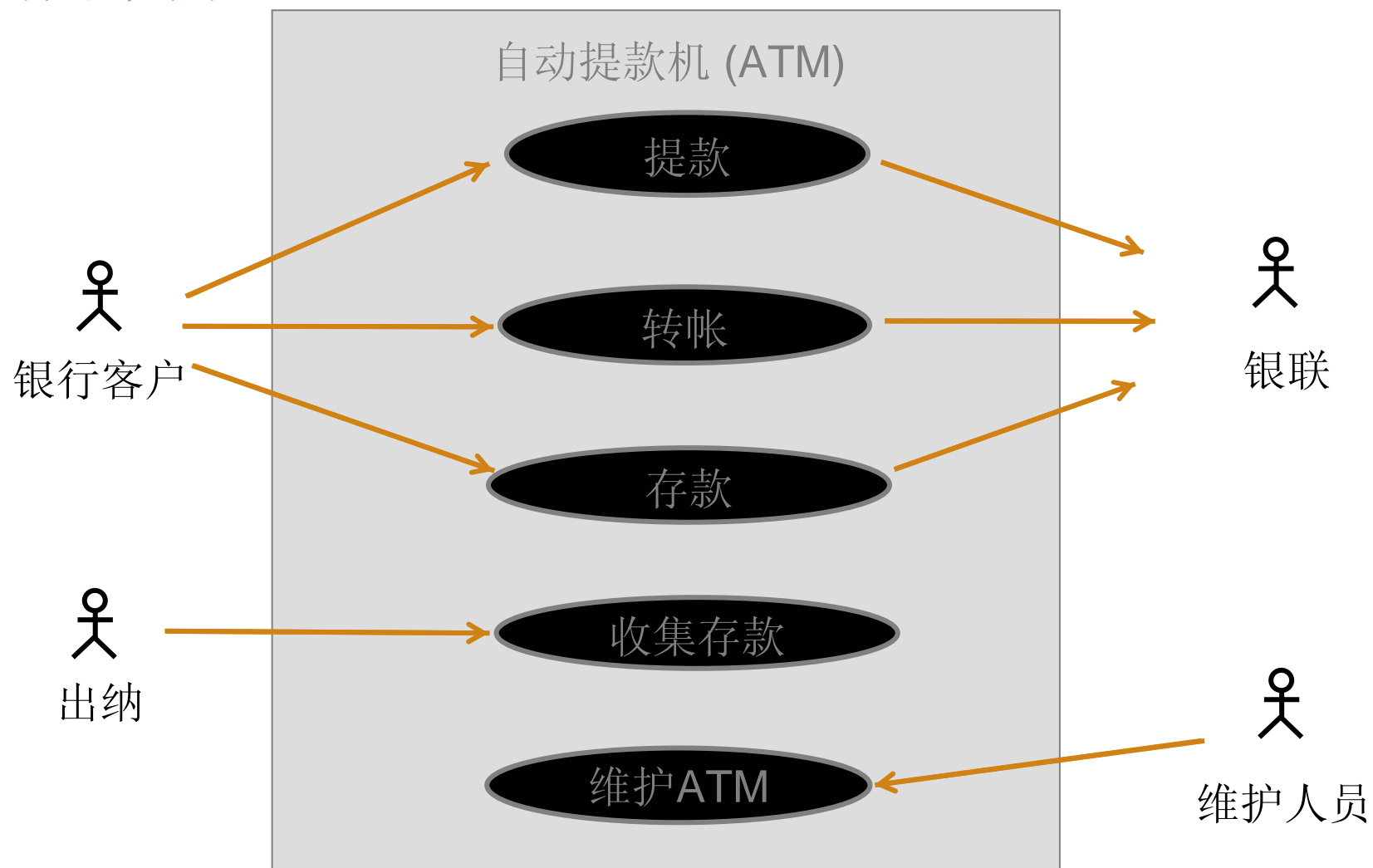
场景 2

登录系统
批准登录
输入搜索主题
无效的搜索主题
重新输入主题
获得课程列表
显示课程信息
选择课程
确认课程可选
显示最终课表

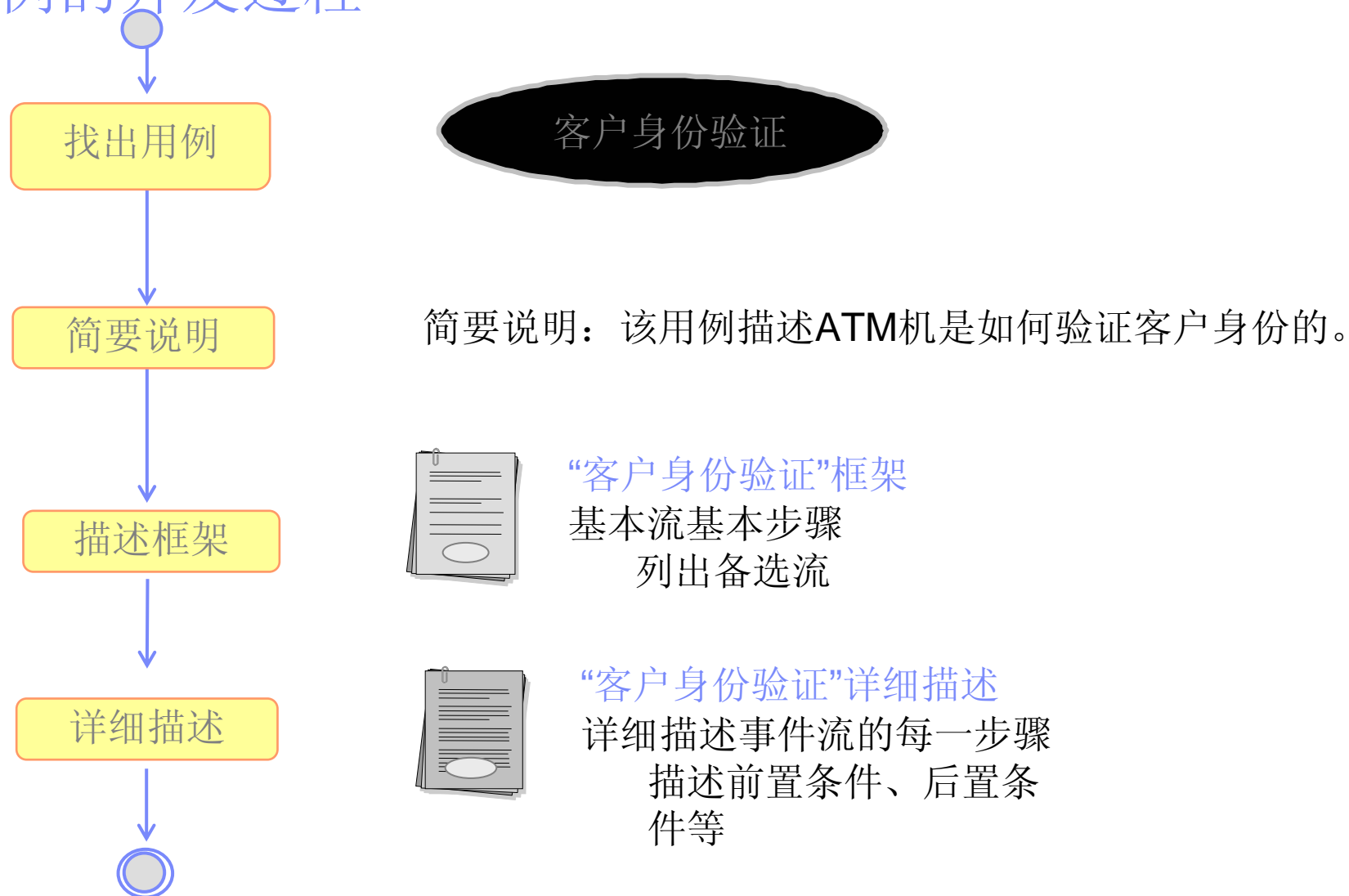
用例图 – 建立系统完整视图



用例图举例



用例的开发过程

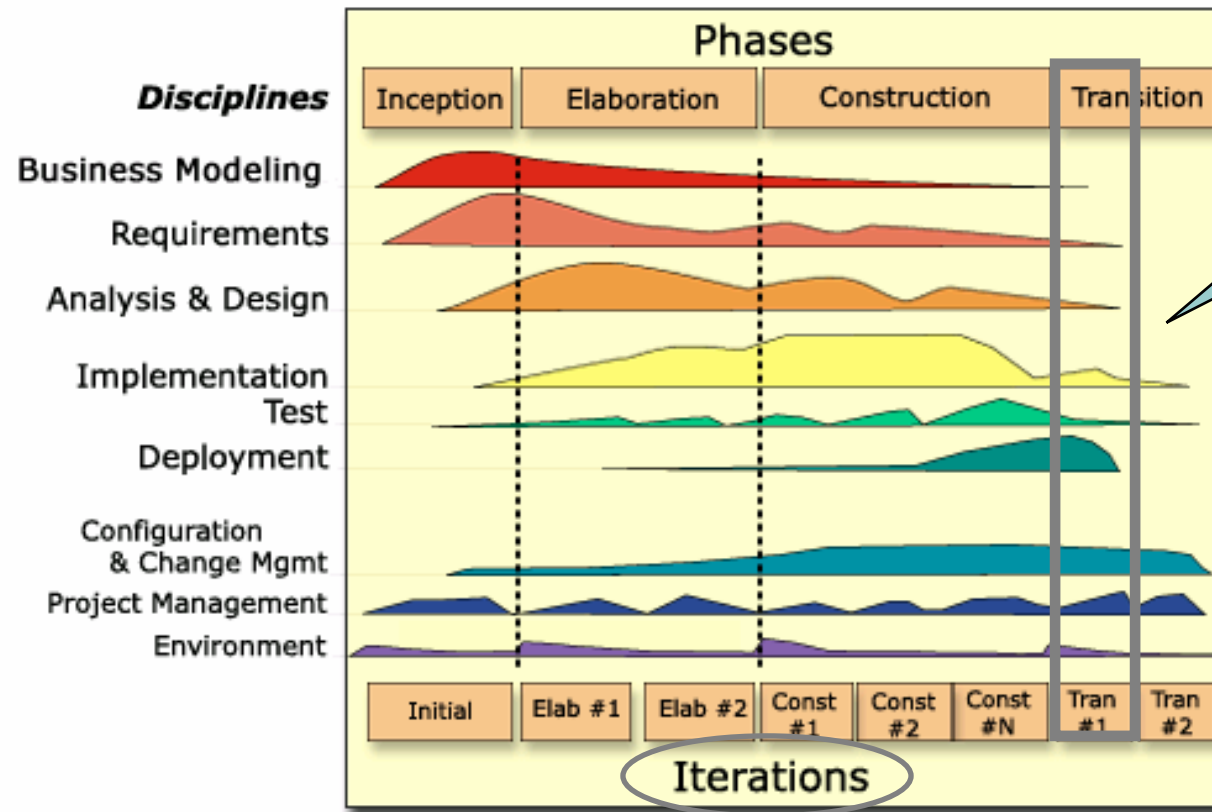


内容

- 什么是软件架构？
- 架构带来什么好处？
- 以架构为核心的软件开发过程
 - ▶ 用例与架构
 - ▶ 架构的步骤
 - ▶ 架构的描述与架构基线
 - ▶ 架构开发实现

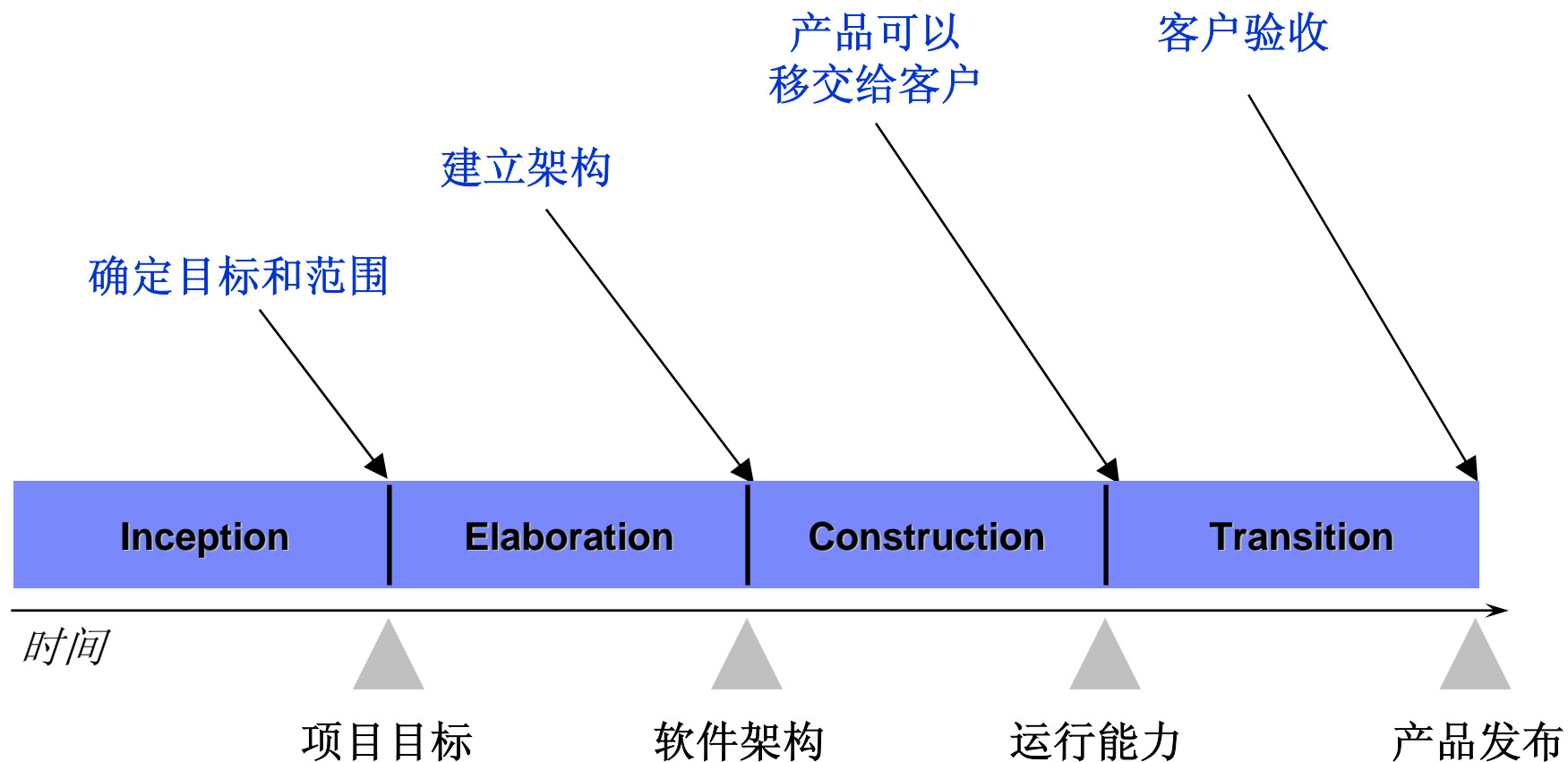


什么是迭代(iteration)?



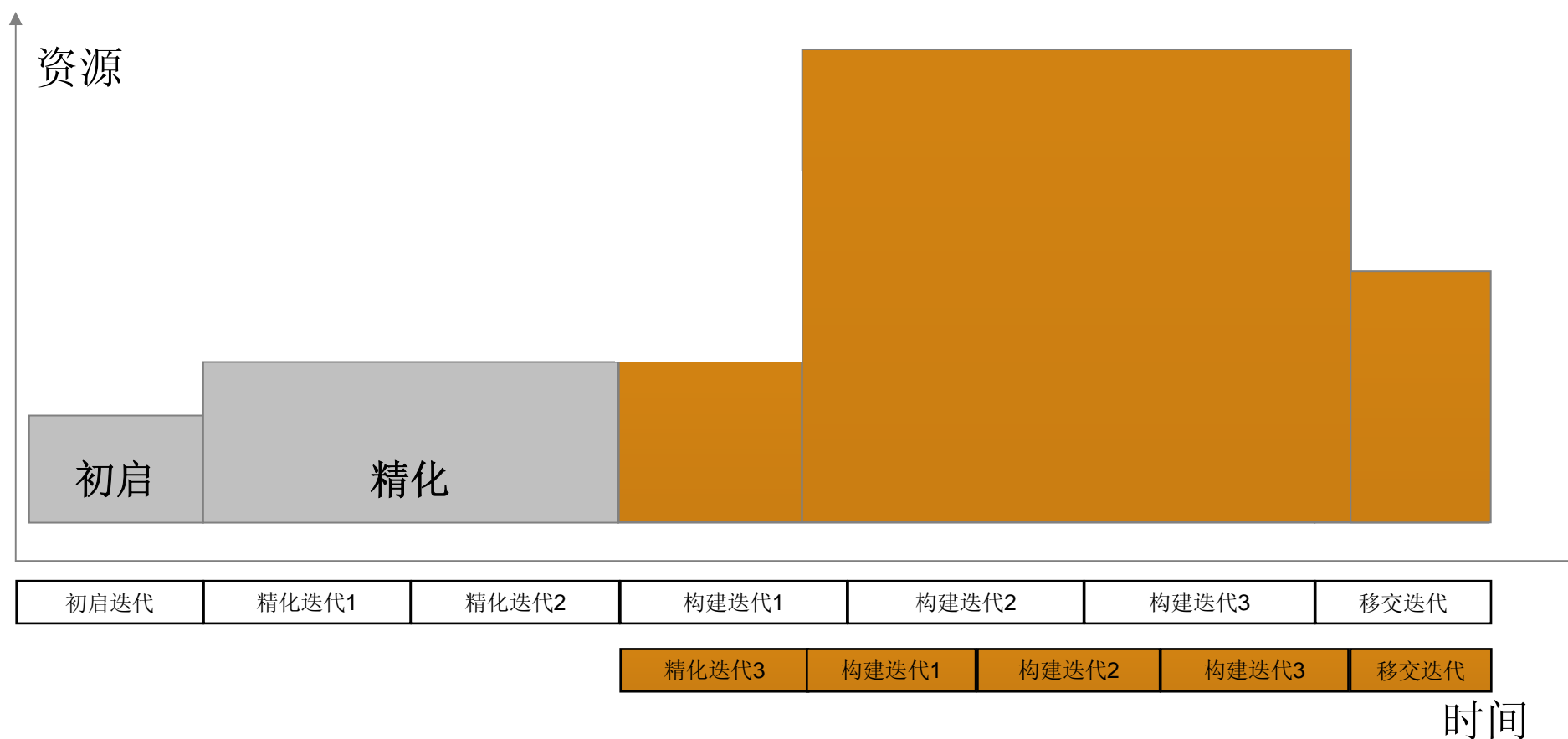
迭代是按预先计划所进行的一系列开发活动，通过迭代会产生一个软件发布结果（内部或外部的），并且根据预先制定的标准来对该结果进行评估

软件项目的四个阶段



把项目风险集中在前两个阶段优先解决

- 有更多的时间来及时调整项目计划，保证项目进度



架构开发步骤

- 在项目的早期阶段开发架构
 - ▶ 在项目的启动阶段开始着手定义架构
 - ▶ 在项目的精化阶段稳定架构
- 架构由用例驱动
 - ▶ 对架构重要的用例来自两个地方
 - 帮助降低最严重风险的用例
 - 对系统用户来讲最重要的用例
- 形成架构基线——稳固的架构

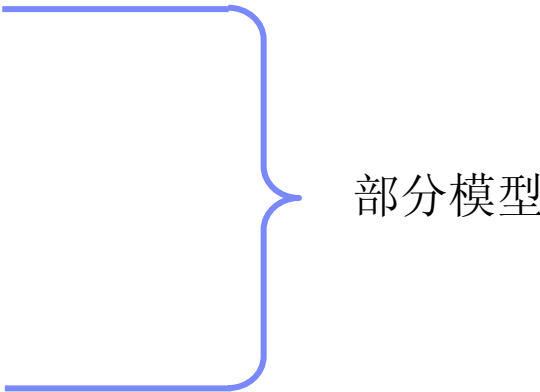


内容

- 什么是软件架构？
- 架构带来什么好处？
- 以架构为核心的软件开发过程
 - ▶ 用例与架构
 - ▶ 架构的步骤
 - ▶ 架构基线与架构的描述
 - ▶ 架构开发实现



架构基线

- 架构基线的实质——一组描述最重要的用例及其实现的模型
- 包括：
 - ▶ 用例模型
 - ▶ 分析模型
 - ▶ 设计模型
 - ▶ 实施模型
 - ▶ 实现模型
 - ▶ 测试模型

部分模型
- 瘦小的，可运转的

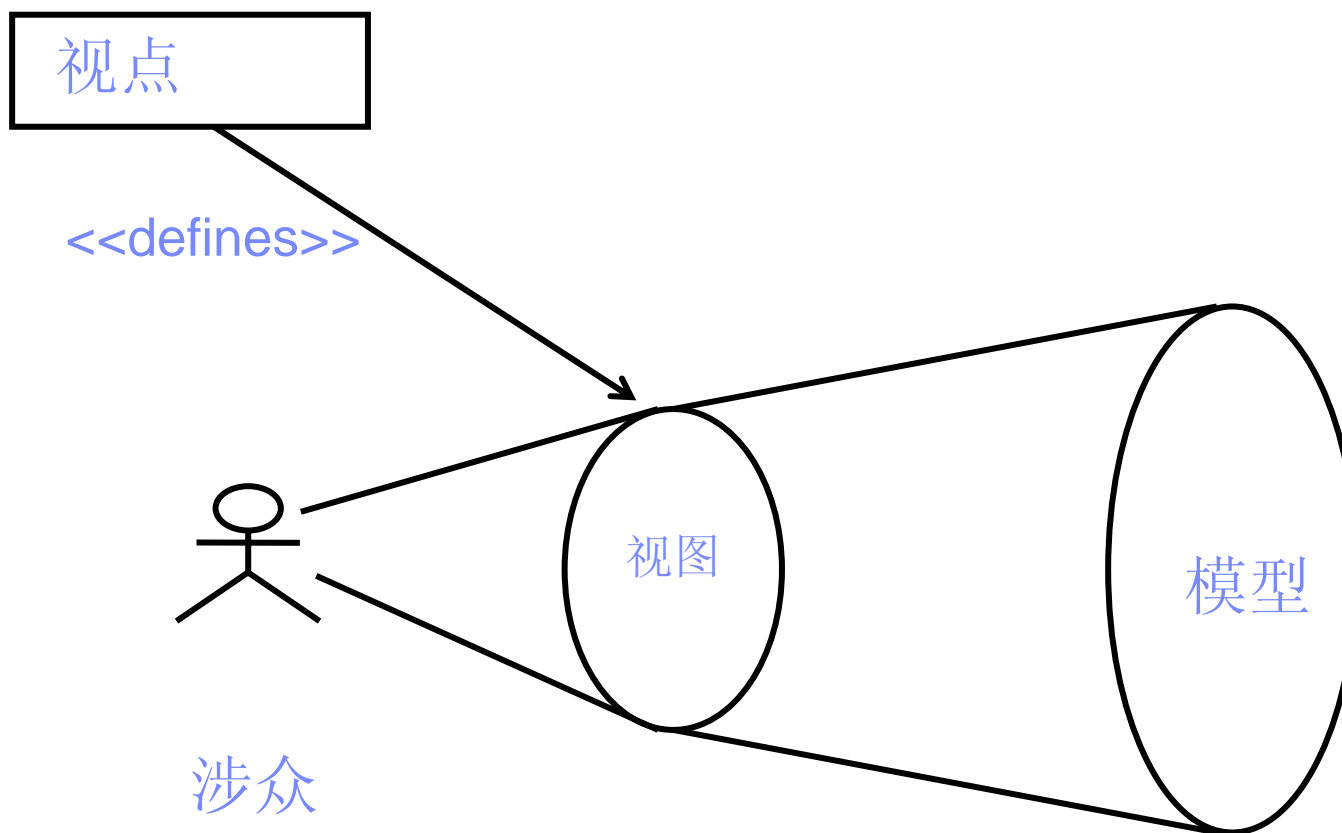


架构描述

- 模型
 - ▶ 从一个实际的透视、在一个特定的抽象层次上给出的一个完整的对于系统的描述
- 视图
 - ▶ 模型的一个投影。隐藏了与透视无关的实体，从一个特定的透视角度能够观察到的
- 视点
 - ▶ 视图的定义（或描述）
 - ▶ 内容，含义和表达（标记法，建模技术）

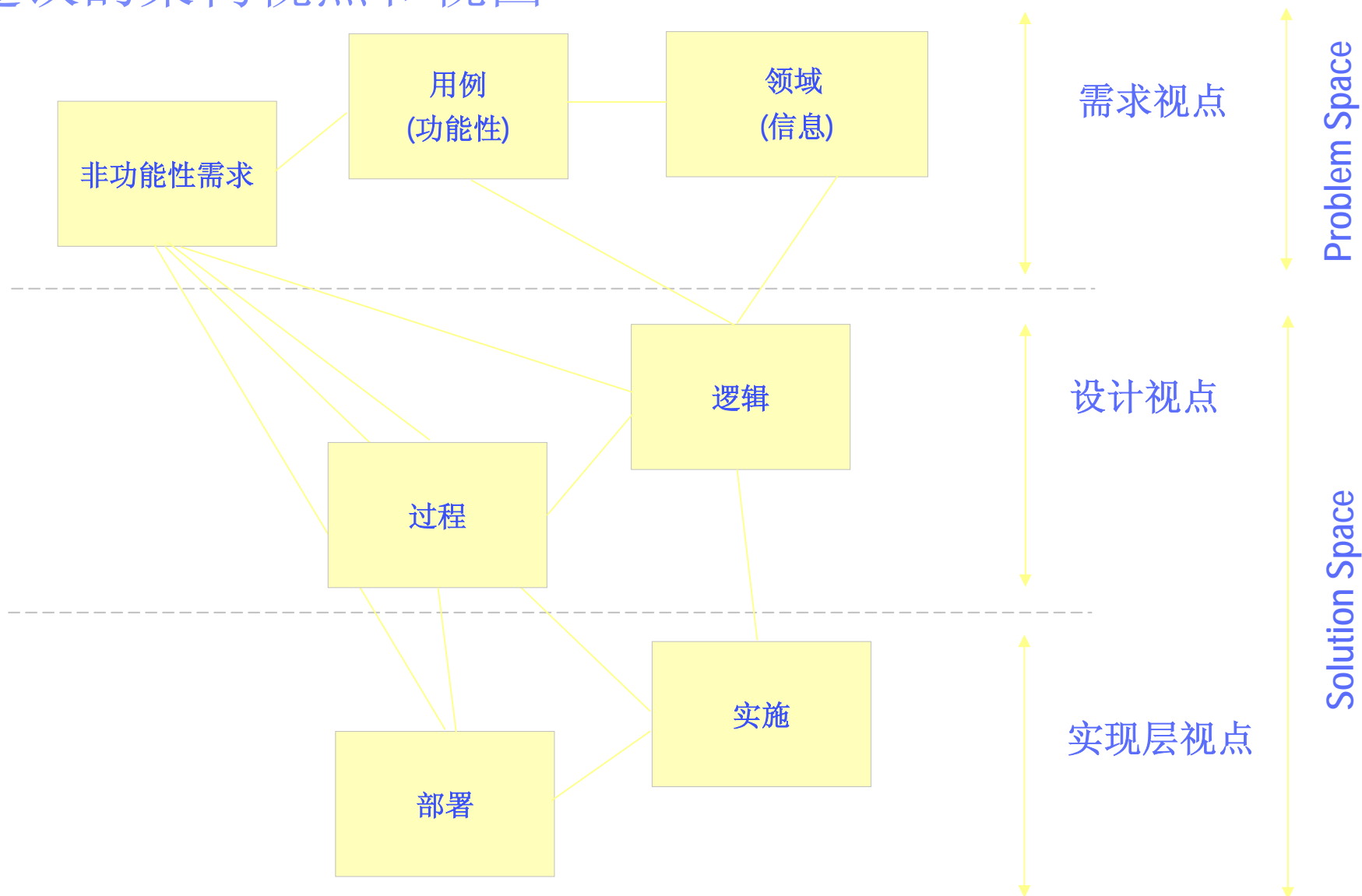


架构视图作为模型的投影

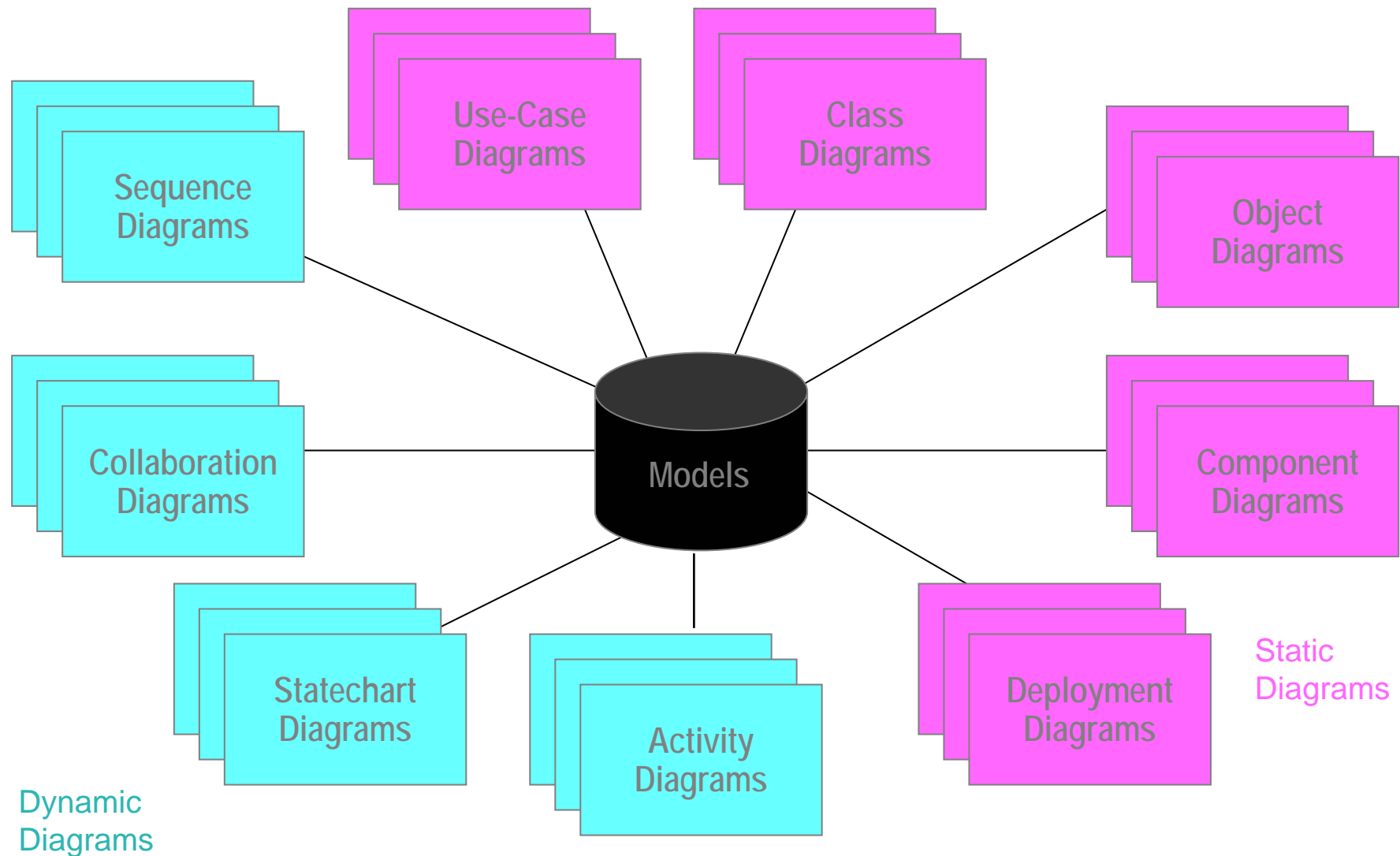


Views are projections of a model for a particular stakeholder

建议的架构视点和视图



Visual Modeling with Unified Modeling Language



内容

- 什么是软件架构？
- 架构带来什么好处？
- 以架构为核心的软件开发过程
 - ▶ 用例与架构
 - ▶ 架构的步骤
 - ▶ 架构的描述与架构基线
 - ▶ 架构开发的实现

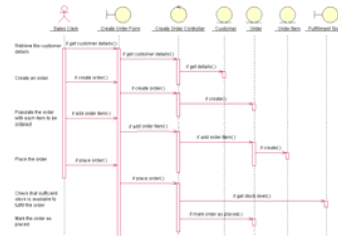
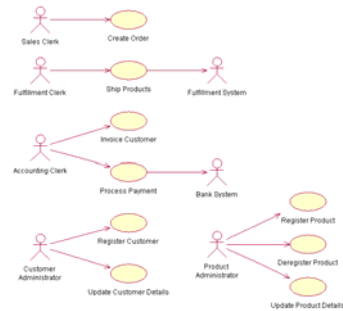


以架构为中心的开发的实现

- 迭代化的开发过程
- 以架构为中心的分析设计（UML, Visual Modeling）
- MDA/MDD加速架构开发过程
- IBM SDP支持更好地以架构为核心



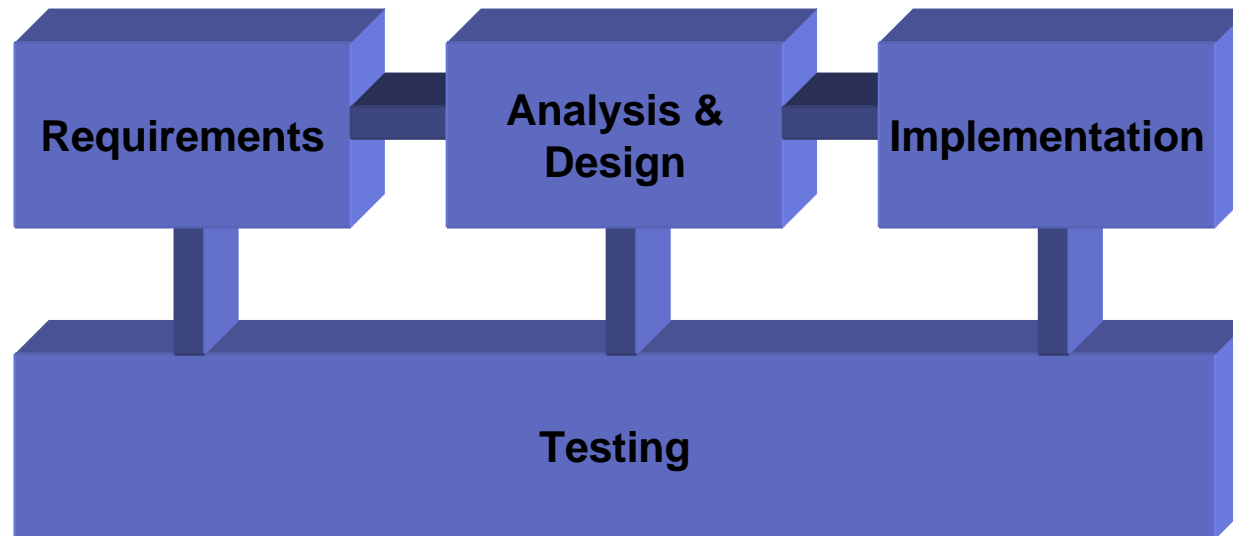
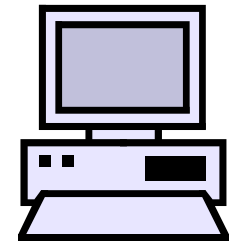
软件的MDD/MDA



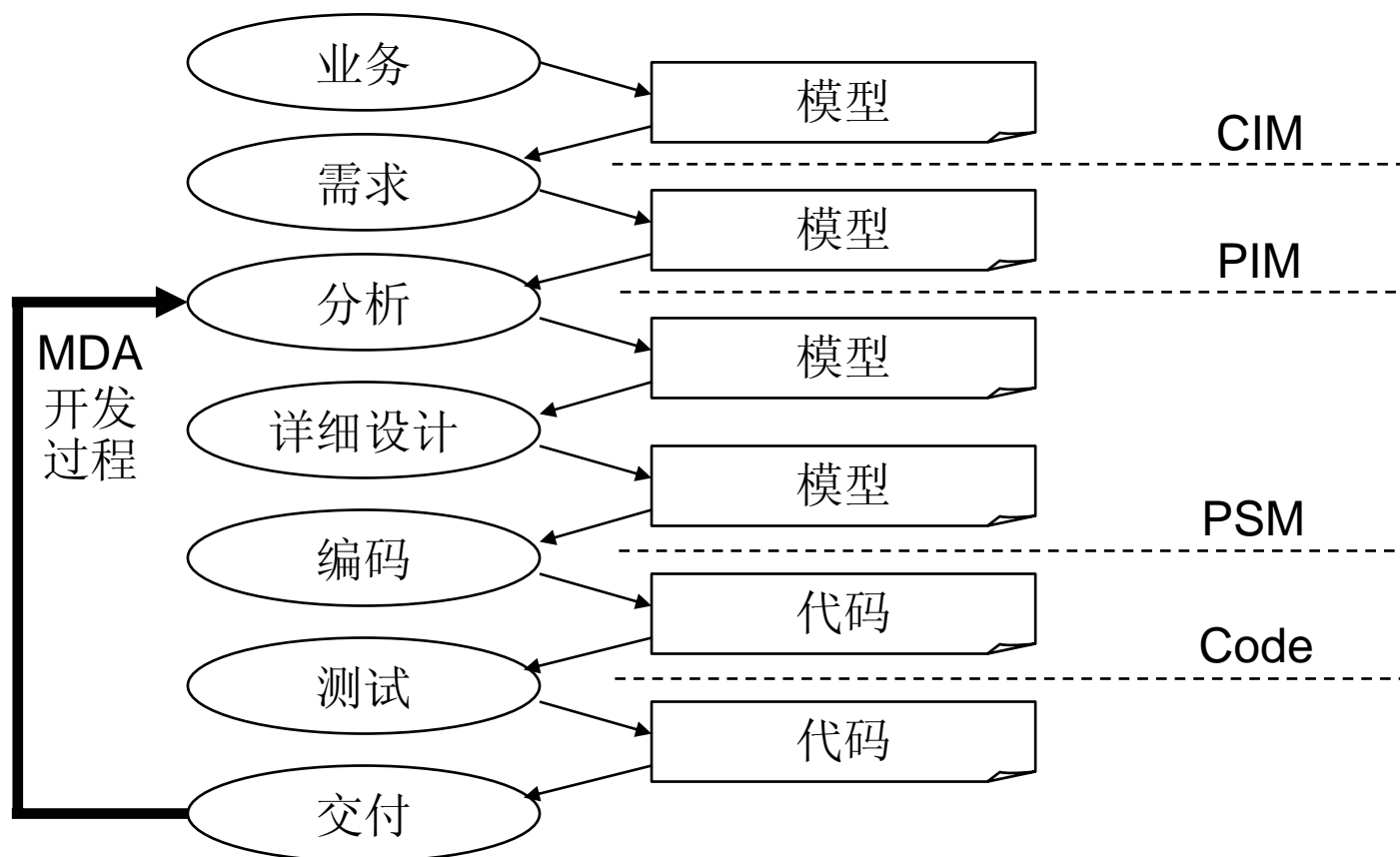
```

Function GetSubClasses (theSuperClass As Class, AllClasses As ClassCollection) As
    Dim theSubClasses As ClassCollection
    Dim theSubClass As Class
    Dim theSubClasses As New ClassCollection

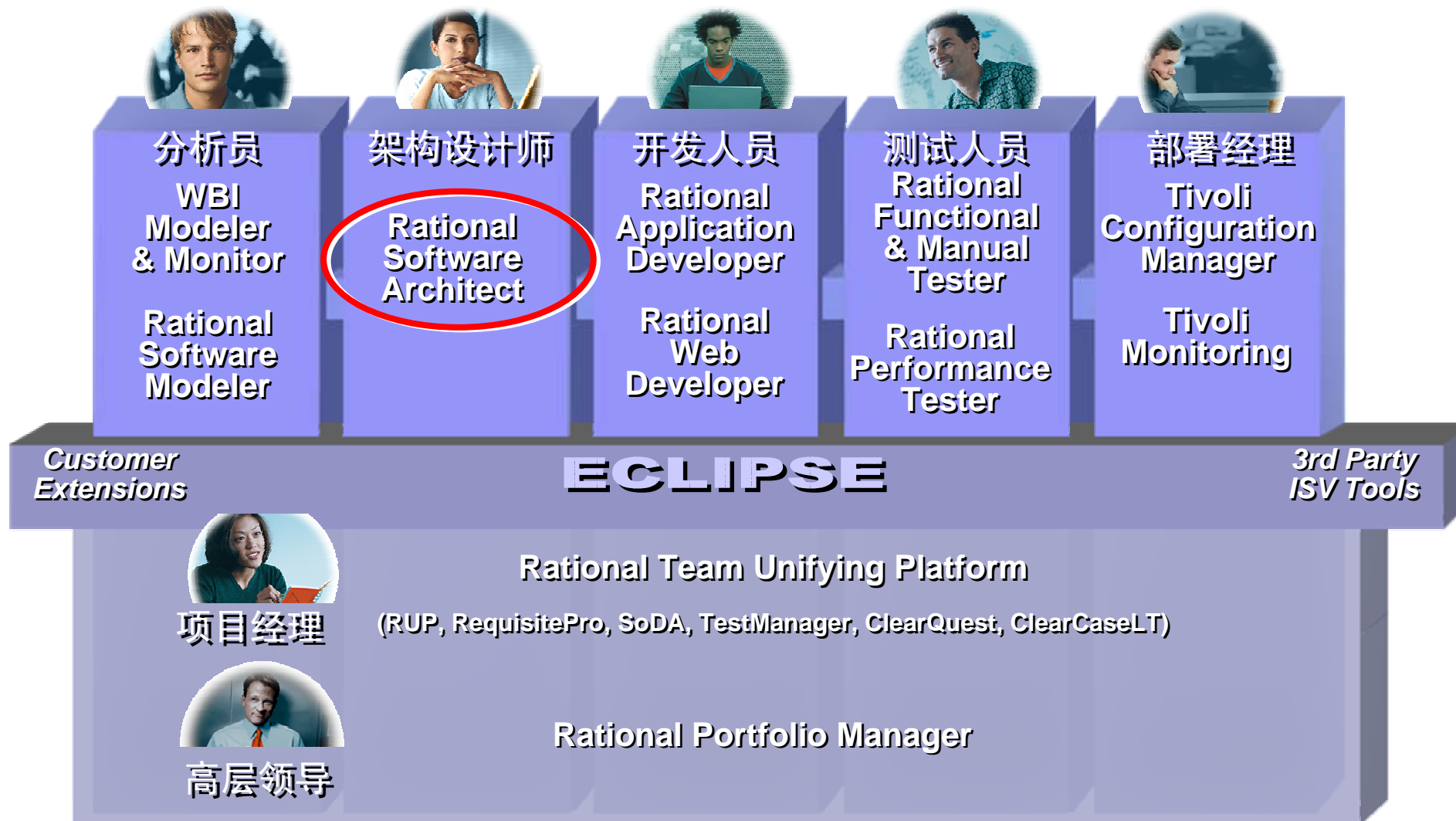
    Print "...Checking subClasses for " & theSuperClass.Name
    For i = 1 To AllClasses.Count
        Set theSubClass = AllClasses.GetItem (i)
        Print "...theSubClass " & theSubClass.Name
        Set theSubClasses = theSubClasses.Concat (i)
        For j = 1 To theSubClasses.Count
            Print "...theSubClasses " & theSubClasses.GetItem (j).Name
        Next j
        If theSubClasses.Exists (theSuperClass) Then
            Print "...Found One!"
            theSubClasses.Add theSubClass
        End If
    Next i
End Function
    
```



MDA的开发过程



IBM 软件开发平台



第三代建模工具 RSA

Rational Software Architecture

- 基于 Eclipse 3.0 的新一代软件开发
- RWD : Rational Web Developer
 - ▶ Web 页面开发工具
- RSM : Rational Software Modeler
 - ▶ 可视化建模工具，支持 UML2.0
- RAD : Rational Application Developer
 - ▶ 前身为 WSAD (Websphere Studio Application Developer)
 - ▶ 支持 J2EE 开发的 IDE 环境
- RSA : Rational Software Architecture
 - ▶ 新一代软件建模平台

Rational software



WebSphere software

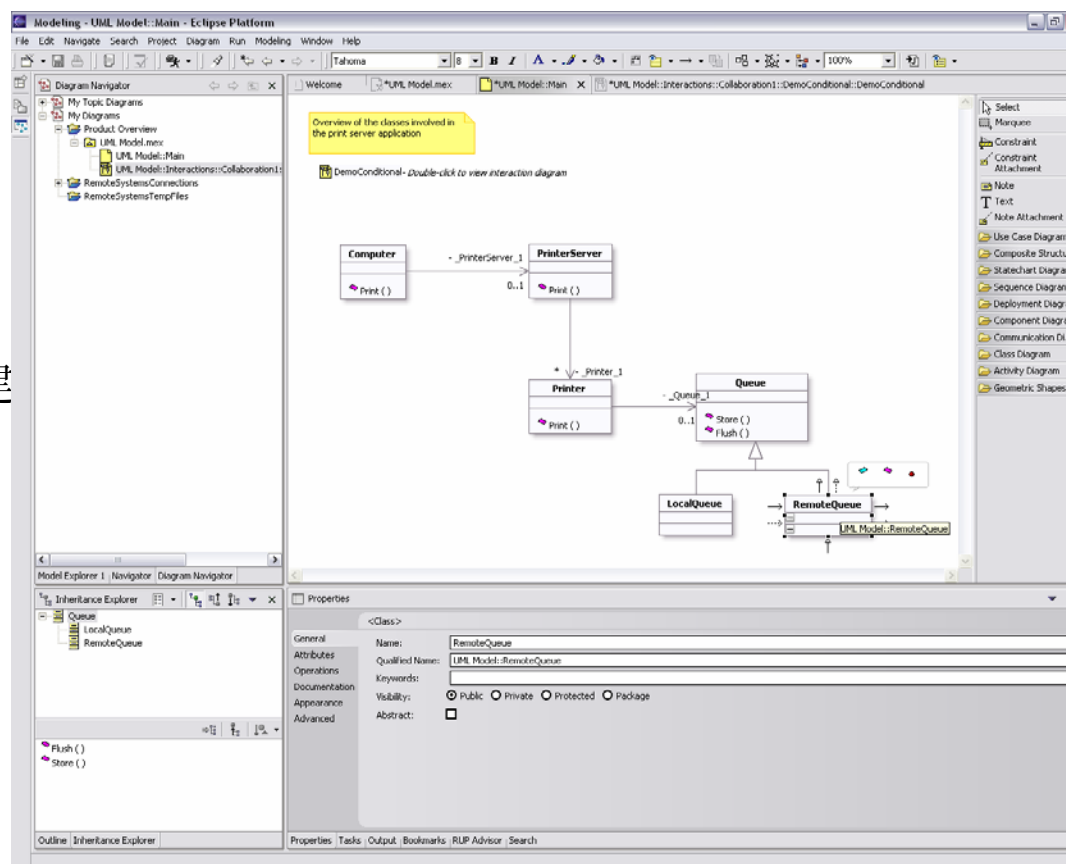


Rational Software Architect 主旋律



关键特性: 快速建模

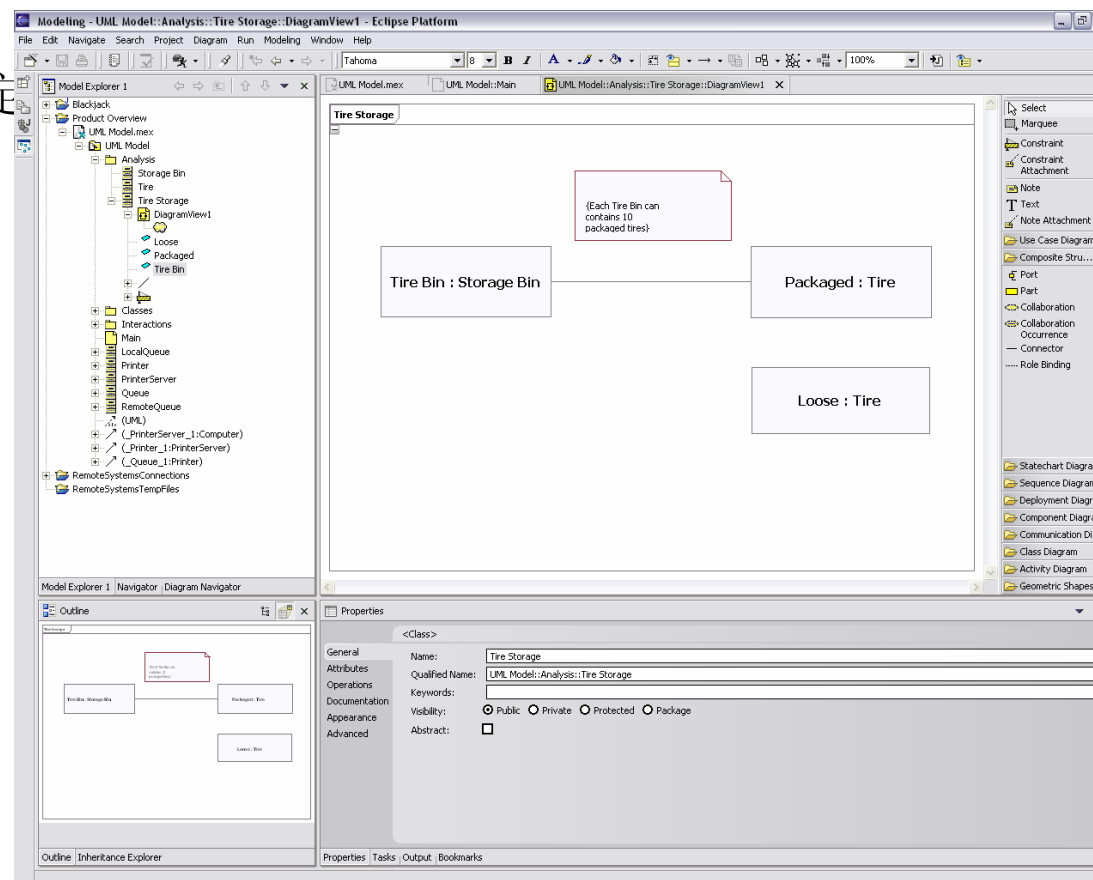
- 简化分析设计过程中UML模型的捕获
 - ▶ 为更多的用户提供方便、快捷的建模功能
- 新的定制视图，改进建模体验



突出特点: 组合结构图支持



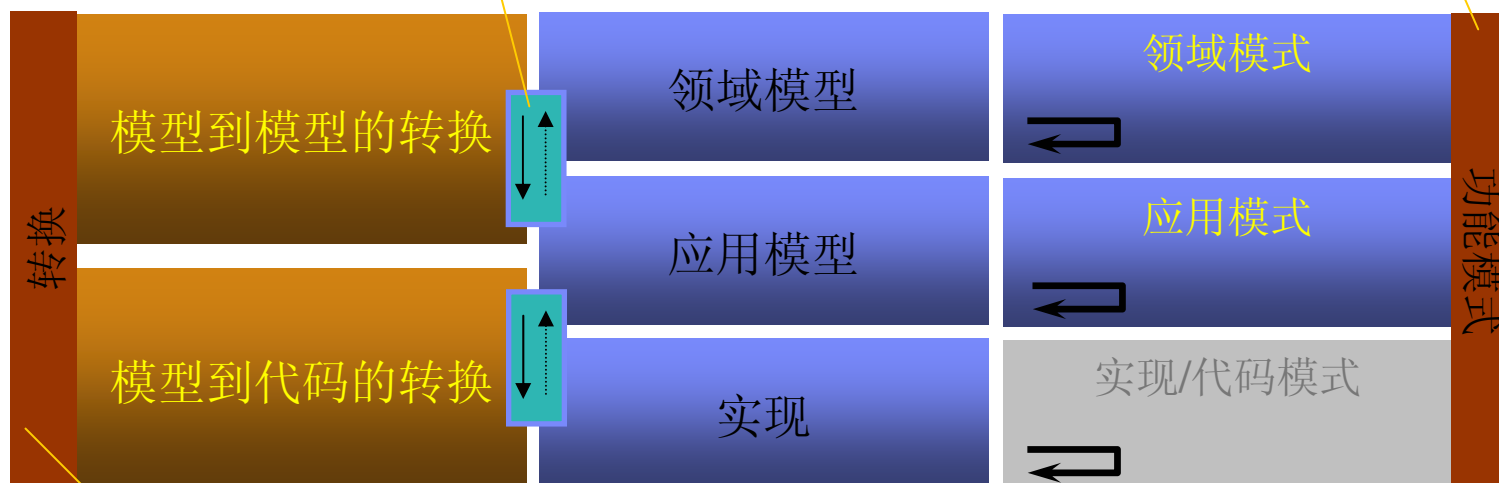
- 对模型的角色层面进行定义, 使模型更加完整
- 捕获应用的逻辑结构
 - ▶ 如SOA应用开发



模式与转换

通过编写转换，可以在转换的源与目标之间建立追踪关系

通过编写和应用模式，可以将模式作为基本的构造来精化模型或实现



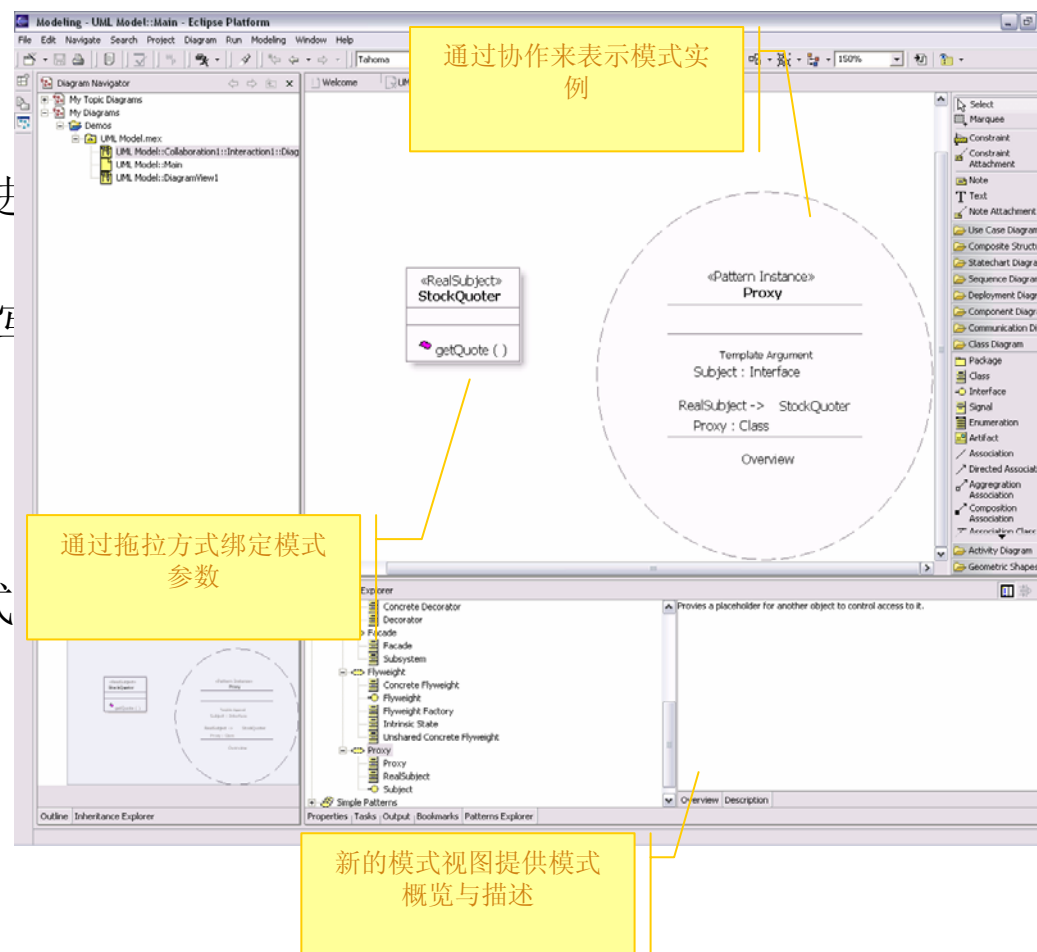
通过编写基于规则/约束的转换，可以以批量的方式进行模型的变换

通过Java和Eclipse插件开发环境（PDE）编写转换的定义

转换执行时可利用功能模式

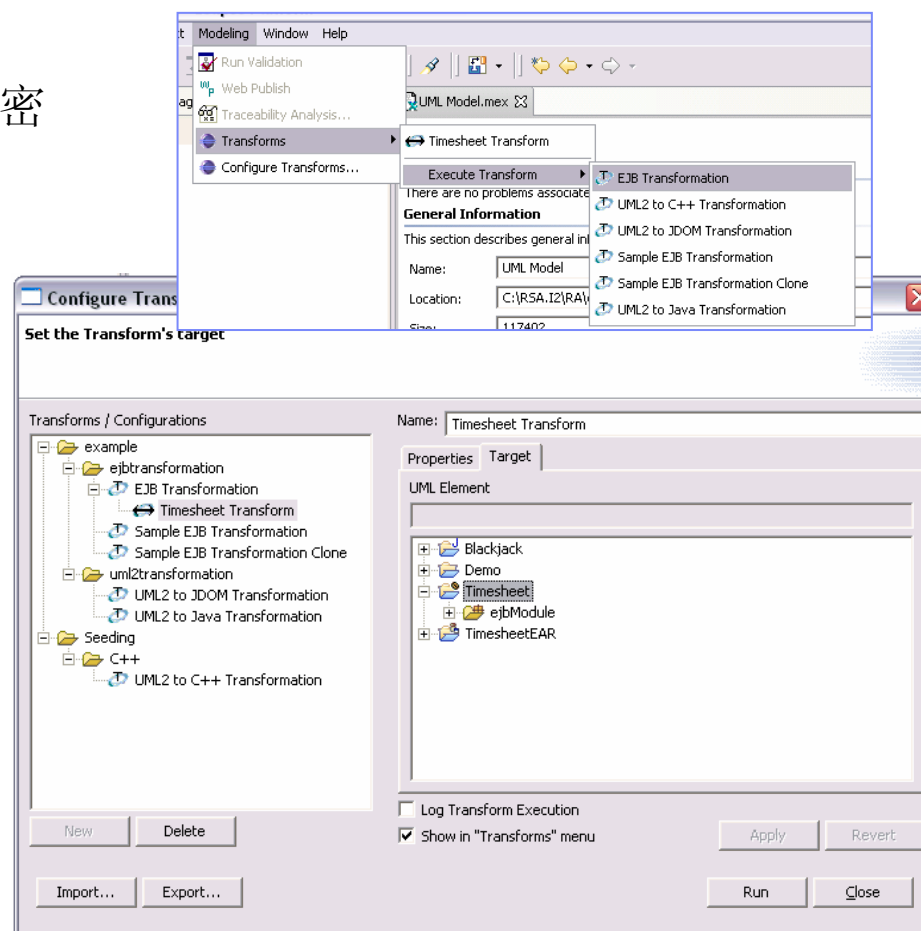
关键特性: 模式

- 模式应用更加简单
 - ▶ 基于已有经验, 进一步改进模式体验
- 通过开放的API, 为模式编写提供更大的灵活性
- 支持全部GOF设计模式
- 通过IBM Developer Works上的RAS库提供更多的模式



关键特性:转换

- 通过转换实现批处理式的、计算密集的操作，如
 - ▶ 模型到模型的转换
 - ▶ 模型到代码的转换
- RSA内置以下代码转换功能
 - ▶ UML-to-J2EE/Java
 - ▶ UML-to-C++
 - ▶ 以及模型到模型的转换示例
- 通过IBM Developer Works上的RAS库提供更多的转换，如
 - ▶ Web Service转换
 - ▶ XSD转换



关键特性: C++ 开发环境

Perspective for C++ Development

C/C++ project hierarchical tree view

C/C++ editor with syntax highlighting, code completion, and advanced search

UML class diagram visualization of C/C++ classes and structs

```
#include "HelloWorld.h"
#include <stream.h>;

HelloWorld::HelloWorld()
{
}

HelloWorld::~HelloWorld()
{
}

// Welcome to the Eclipse CDT
//
static int HelloWorld::main(int argc, char ** argv)
{
    cout << "Hello World!!!" << endl;
}

// eof
```

```
classDiagram
    class HelloWorld {
        ~HelloWorld()
        HelloWorld()
        HelloWorld()
        main(int, char**) int
    }
    class HelloWorld {
        ~HelloWorld()
        HelloWorld()
        HelloWorld()
        main(int, char**) int
    }
    class HelloWorld {
        ~HelloWorld()
        HelloWorld()
        HelloWorld()
        main(int, char**) int
    }
```


关键特性: Java 方法可视化

- 通过UML序列图对代码进行可视化，帮助理解系统的行为
- 可以将可视化的UML图嵌入到Javadoc文档报告中

Integrated with the Java Package view

Leverages UML 2.0 sequence diagram constructs for loops, conditionals, etc...

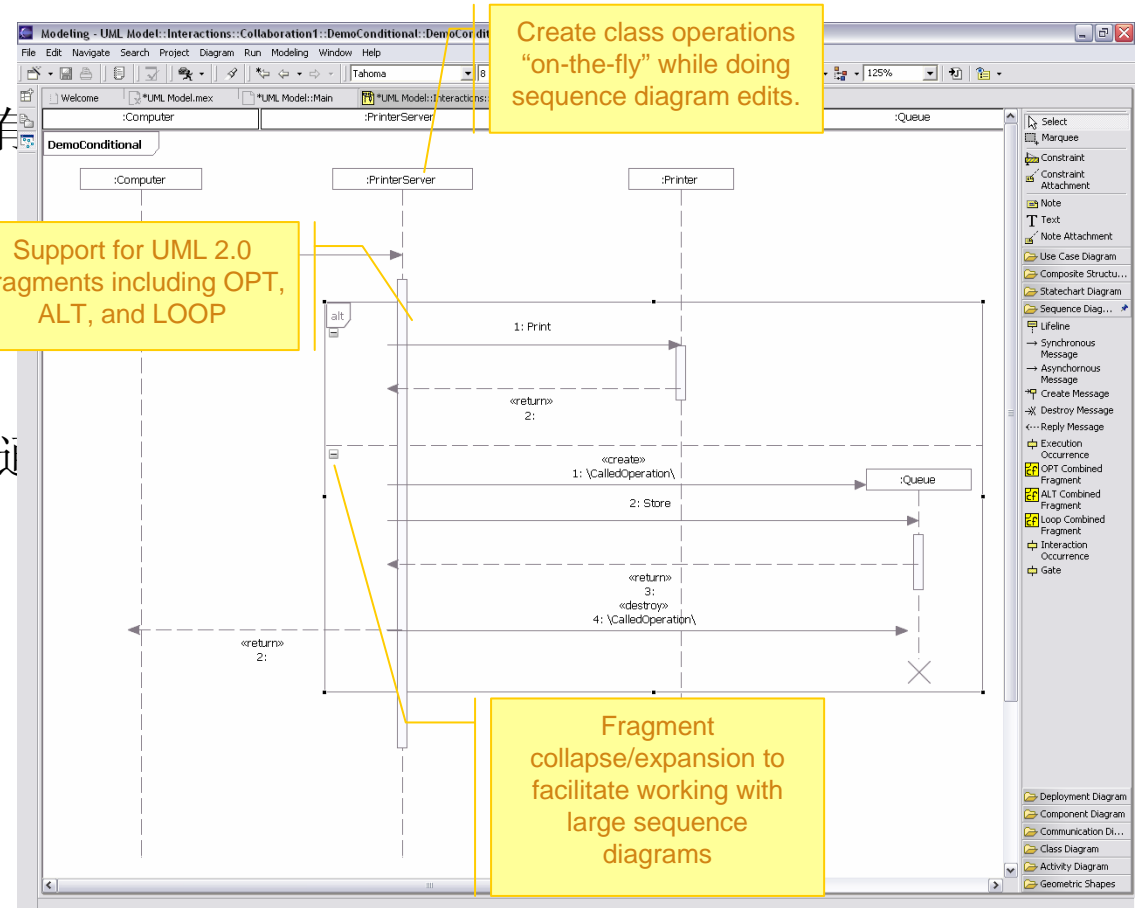
Alternate abstract view of method behavior

"Topic" diagram for method is automatically updated/refreshed when method is updated

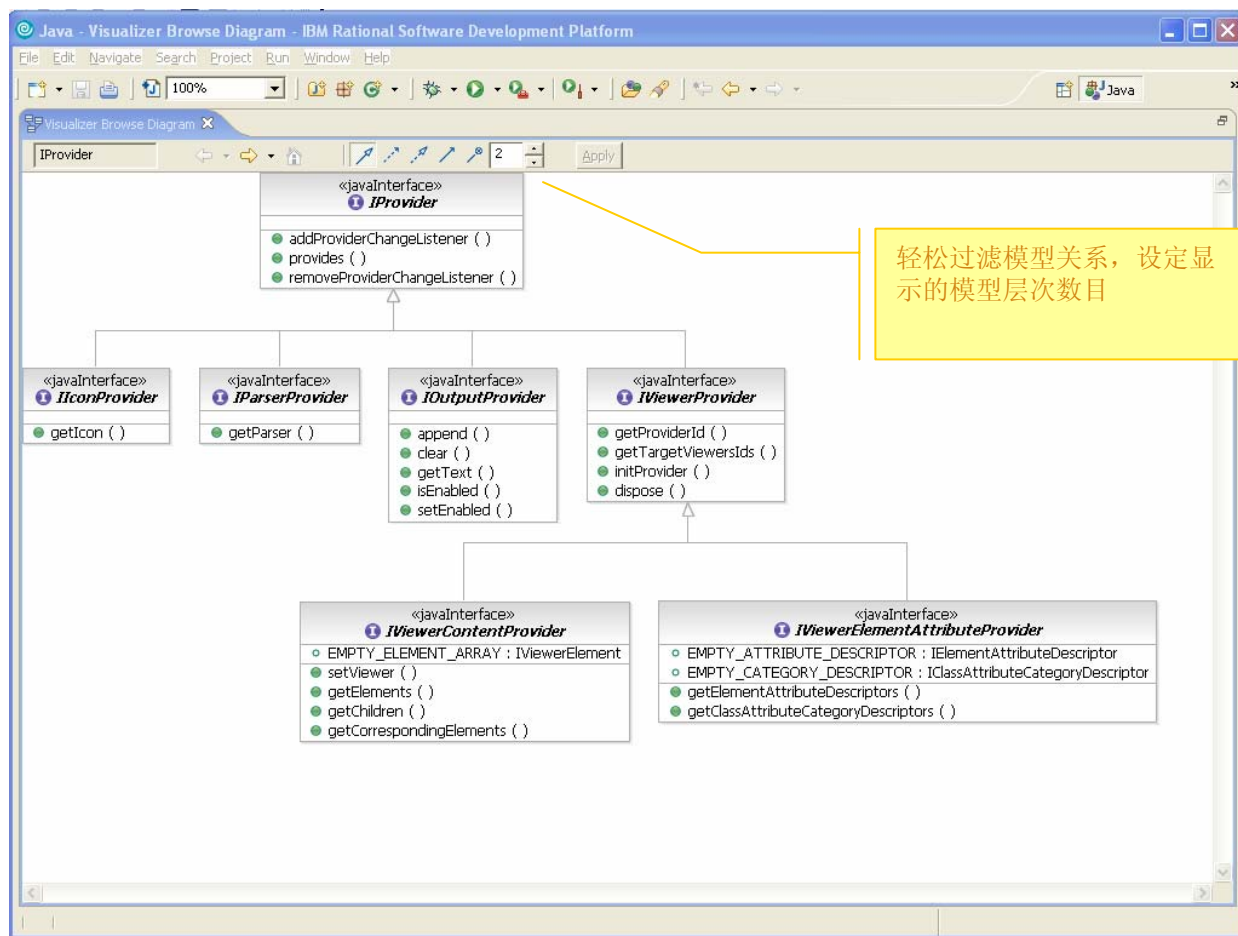
Select method to be visualized using UML

UML 增强功能: 交互建模

- 使用UML 2.0可以更加有效地表示交互
 - ▶ 支持对测试路径的描述
 - Loop, alt, opt
 - 交互发生
- 交互可以通过序列图或通信图表示
- 改进的序列图编辑功能
 - ▶ 排序/重新排序



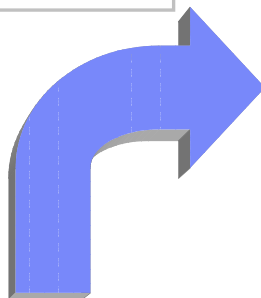
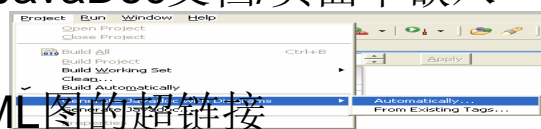
UML 增强: 浏览图



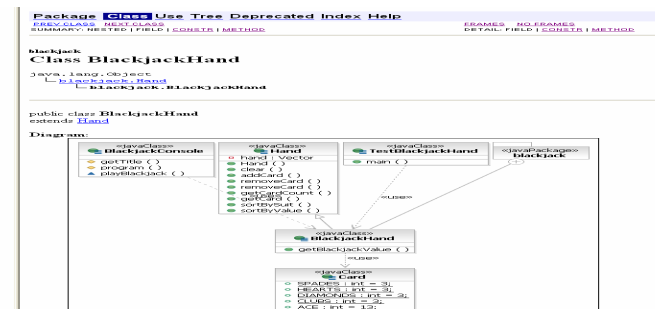
即时生成/更新浏览图，不需要对图形进行单独的创建或维护。帮助用户更好地理解 and 发现模型以及应用

UML 增强: 在JavaDoc中嵌入UML图

- 生成更加丰富的 JavaDoc文档
 - ▶ 直接在JavaDoc文档/页面中嵌入UML图
 - ▶ 支持UML图的超链接



All Classes
BlackjackGame
Card
ConsoleApplet
ConsoleApplet
Deck
Hand
TestBlackjackHand
TestDeck

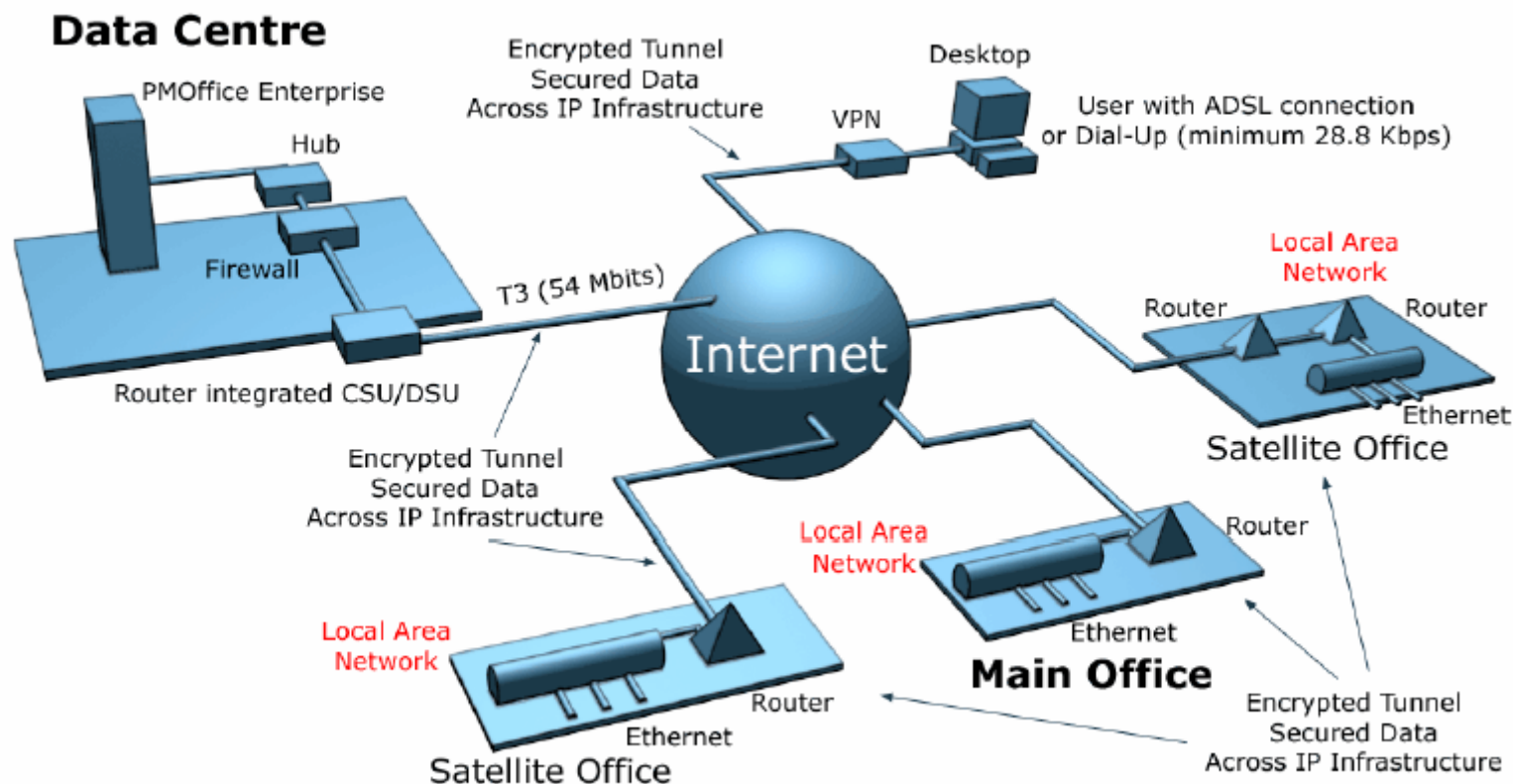


架构问题总结

- 结构化的分析设计方法导致没有架构
 - ▶ 以功能及数据为核心导致软件处理与数据耦合过于紧密
 - ▶ 没有架构的应用没有办法适应需求的快速变化
- 应用OOP不代表架构
 - ▶ 优于结构化编程，但同样带来架构问题
- Architecture代表高质量
 - ▶ Pattern——复用设计的最佳实践
 - ▶ J2EE 13 Core Patterns
 - ▶ AntiPattern——应用性能、可靠性的主要问题根源
 - Butterfly, Hub, Breakable, Tangle
- RSx提供全面的质量保证方案
 - ▶ Code Review, jUnit, Component Test, Runtime Analysis (PPlus)
 - ▶ Functional Testing, Performance Testing



Portfolio Manager 典型系统架构



Questions

Thank
You