UML系统分析与架构设计实战

康凯

1

目录

2

第一单元：用UML辅助系统分析与设计

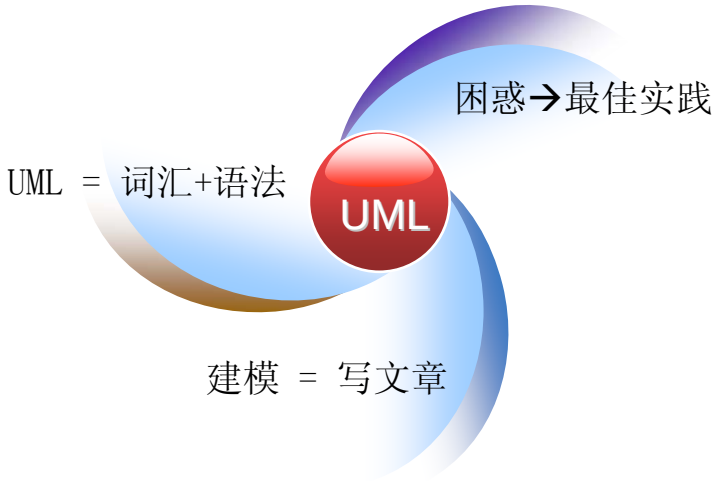
3

UML实效建模

——让设计建模更明白、更有效

4

UML建模是什么



5

内容提要

UML图概览	建模案例	实效提示
• 应用时机 • 建模目的	• 如何选用 • 模型解读 • 建模要点	• 常见误区 • 改进建议

6

UML图概览

应用时机、建模目的

7

开发过程解析

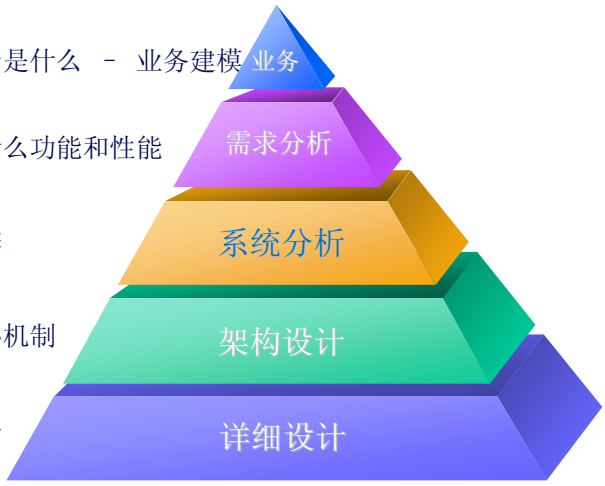
目前的现实和愿景是什么 - 业务建模

系统应对外提供什么功能和性能

需求到设计的桥梁

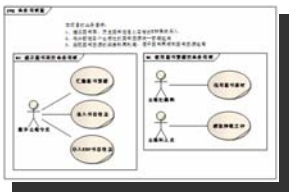
体系、框架、核心机制

根据架构进行设计

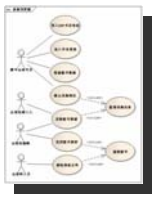


8

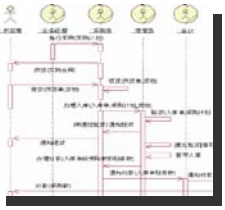
需求分析阶段



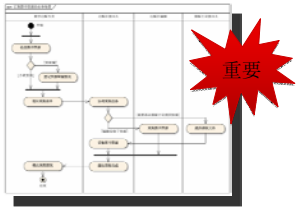
业务用例图



系统用例图



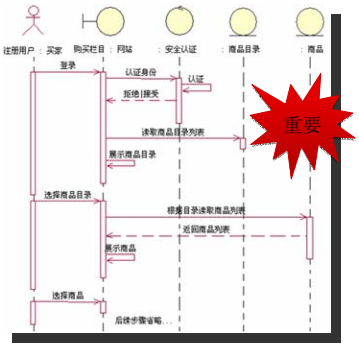
用例场景序列图



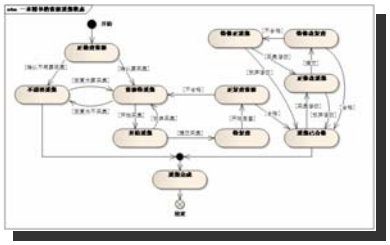
用例场景活动图

9

系统分析阶段



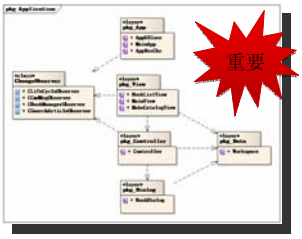
用例实现序列图



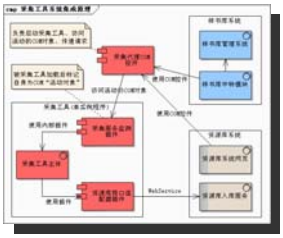
流程分析用的状态图

10

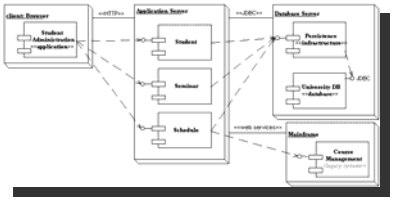
架构设计阶段



架构包图



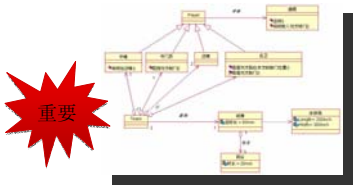
组件图



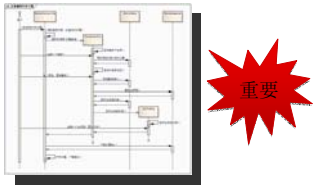
部署图

11

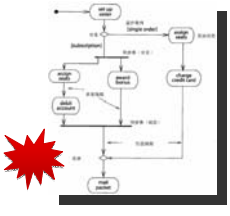
详细设计阶段



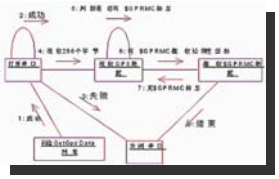
类图



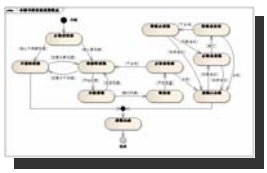
序列图



活动图



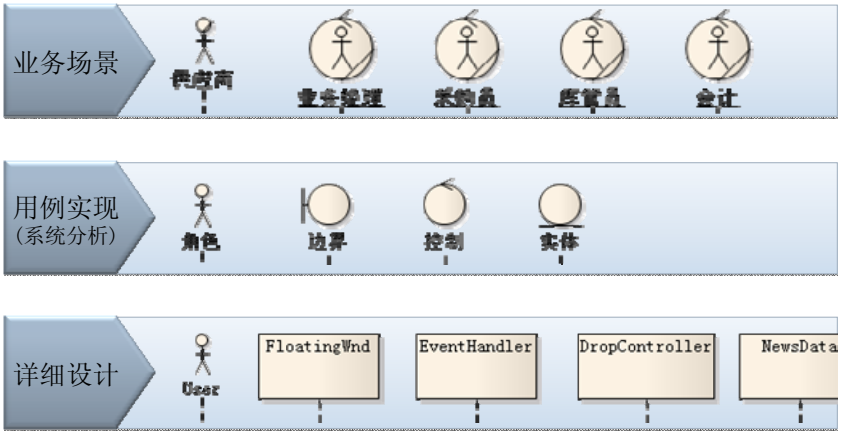
协作图



状态图

12

序列图的区别



13

常用UML图汇总表

	需求分析	系统分析	架构设计	详细设计
用例图	★			
活动图	★			★
序列图	★	★		★
状态图		☆		★
架构包图			★	
组件图			★	
部署图			☆	
协作图				★
类图				★

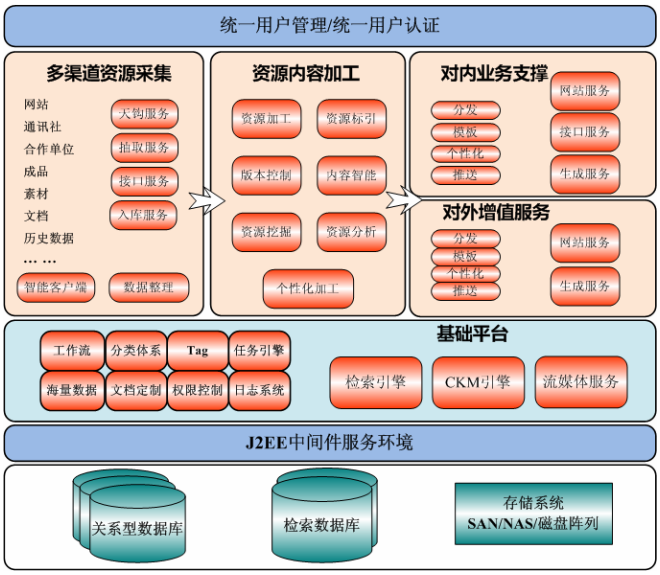
14

UML建模案例分析

如何选用、模型解读、要点

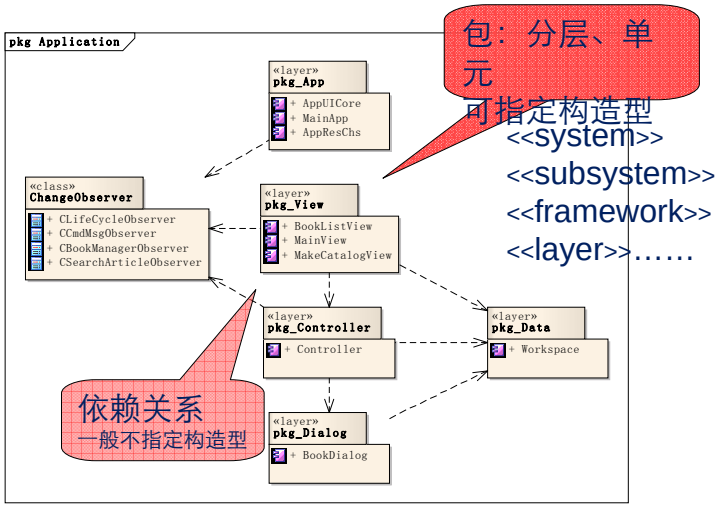
15

下面是“系统架构图”吗



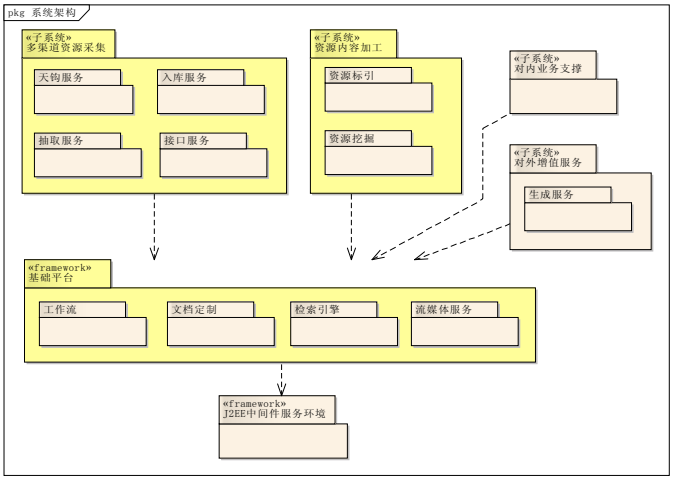
16

用包图描述系统架构



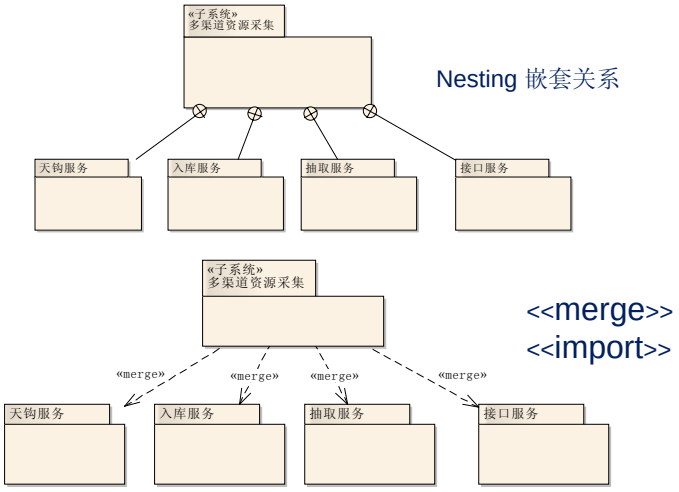
17

包图可嵌套



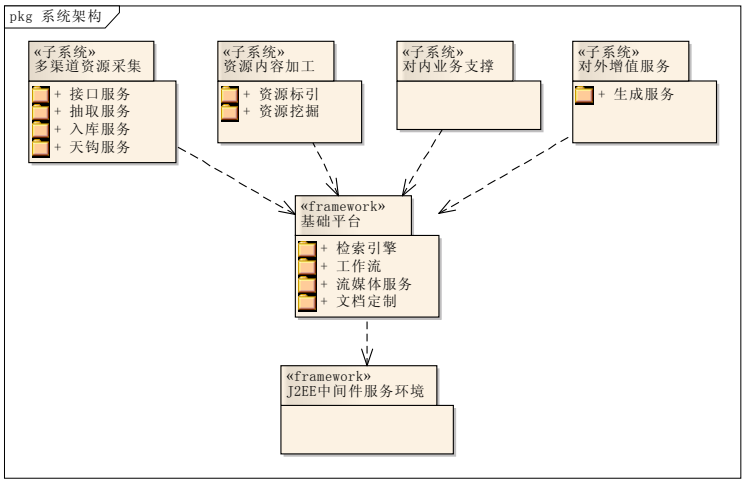
18

嵌套包图的替代画法



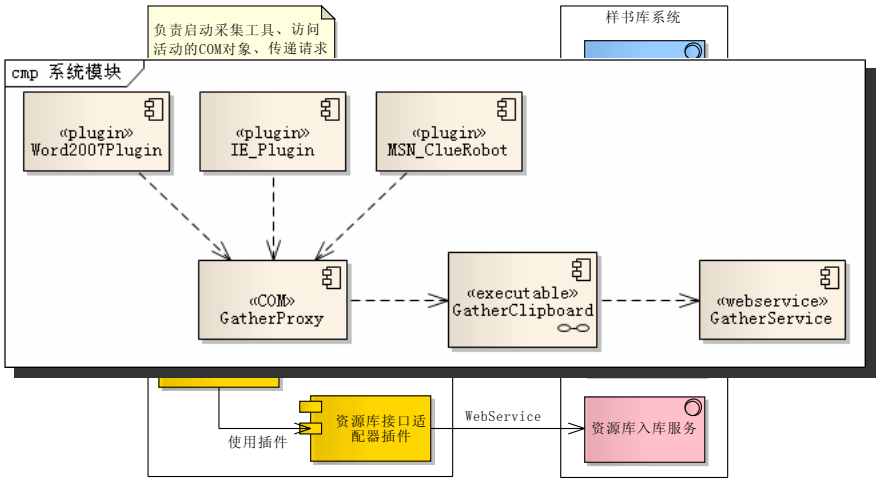
19

嵌套包图的替代画法

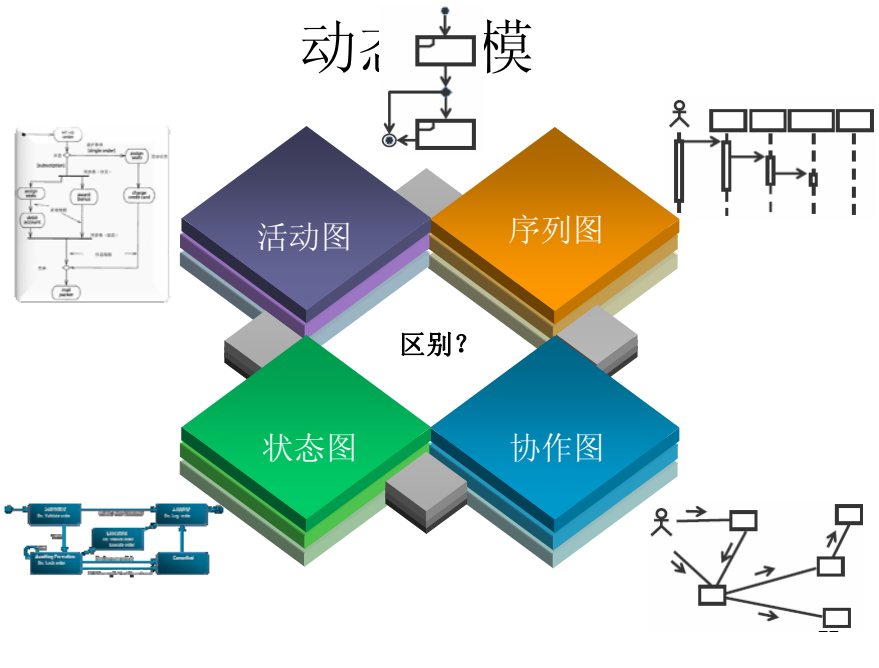


20

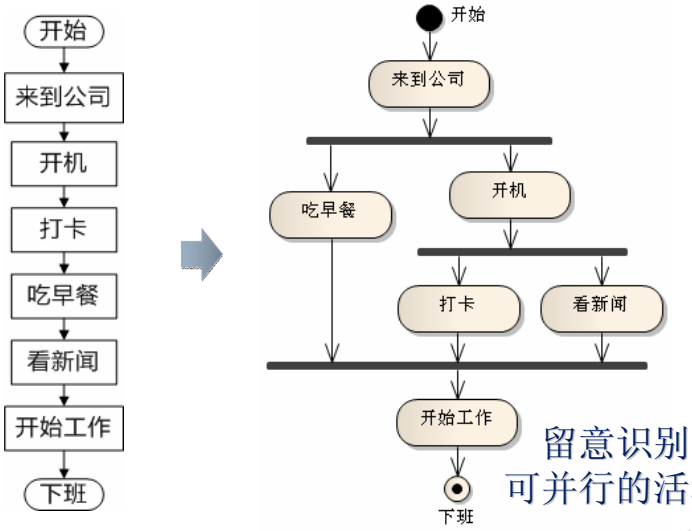
组件图→系统架构


负责启动采集工具、访问
活动的COM对象、传递请求
样书库系统
cmp 系统模块
«plugin»
Word2007Plugin
«plugin»
IE_Plugin
«plugin»
MSN_ClueRobot
«COM»
GatherProxy
«executable»
GatherClipboard
«webservice»
GatherService
使用插件
资源库接口适
配器插件
WebService
资源库入库服务

21

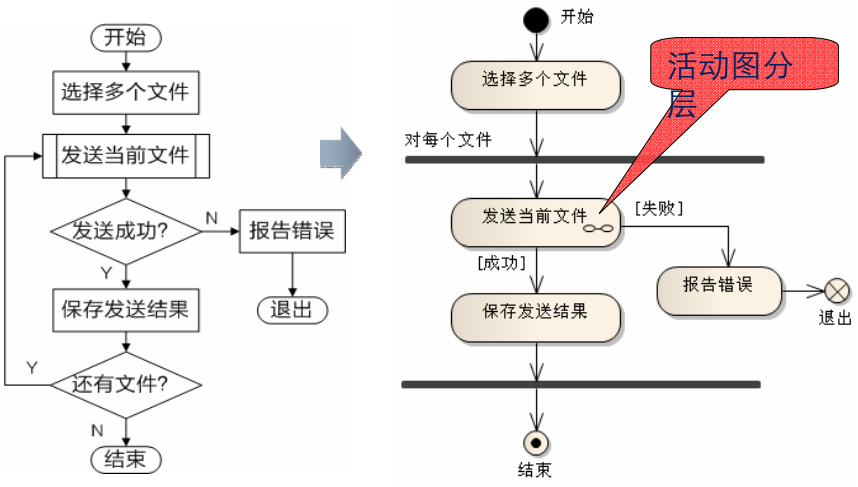

模
活动图
序列图
区别？
状态图
协作图

从流程图到活动图



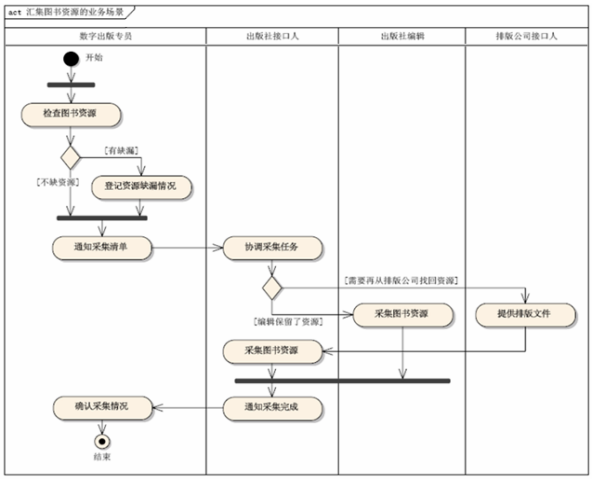
23

从流程图到活动图



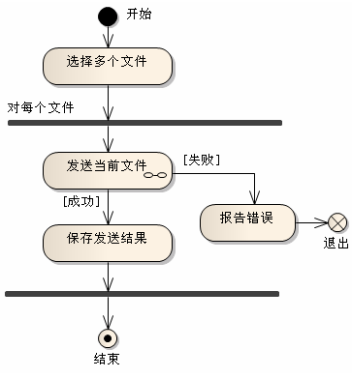
24

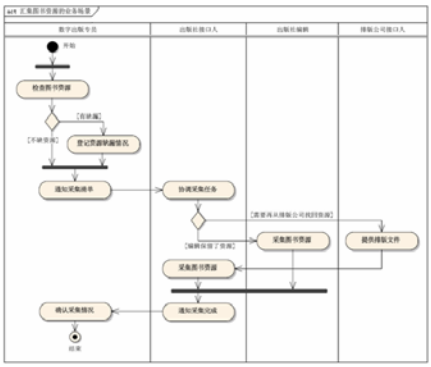
泳道→单元相互关系和职责



25

活动图→序列图

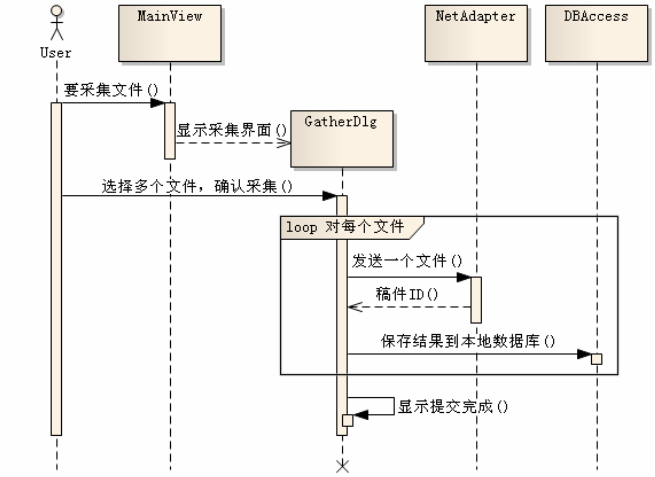




分支判断、并行汇合、单向流转
重在活动本身，表达流程

26

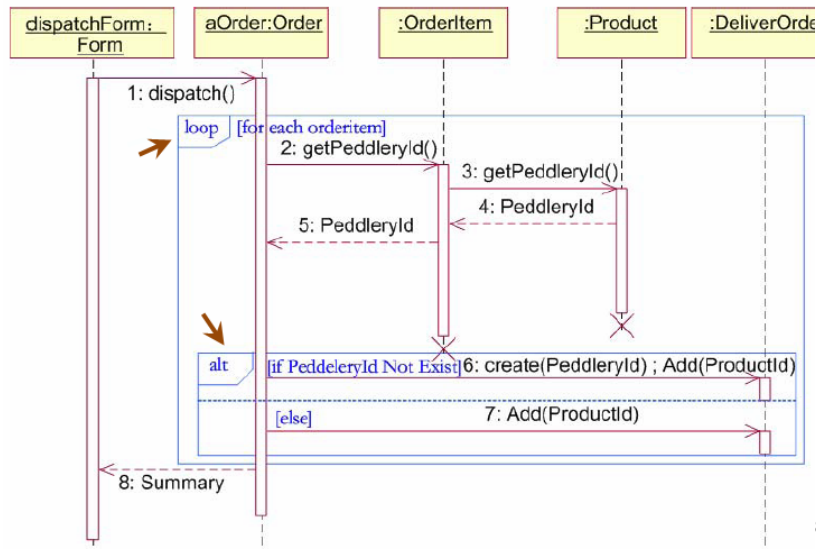
活动图→序列图



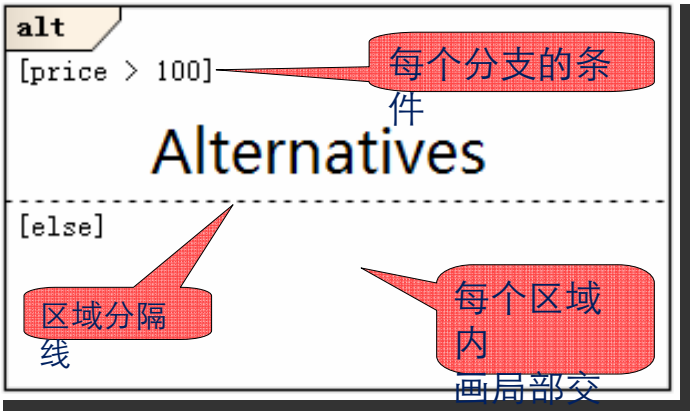
返回、流转、生命期、细节
重在时序，表达交互细节

27

交互片断→分支流程

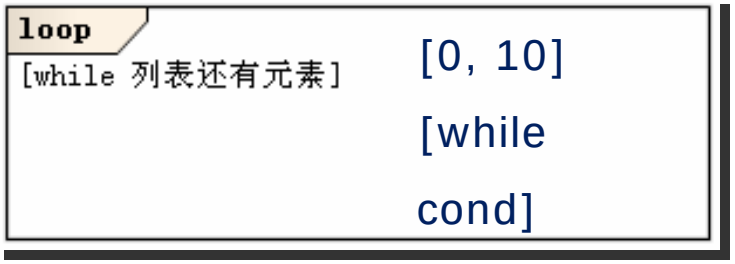
- 

交互片断的用法—条件分支



29

交互片断的用法—循环



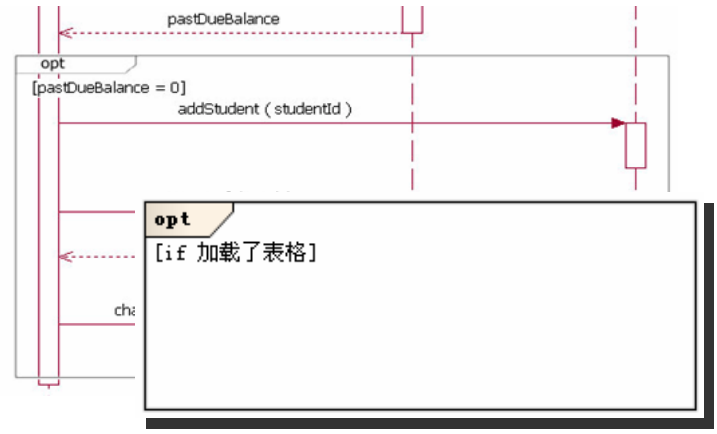
[0, 10]

[while cond]

[for each ...]

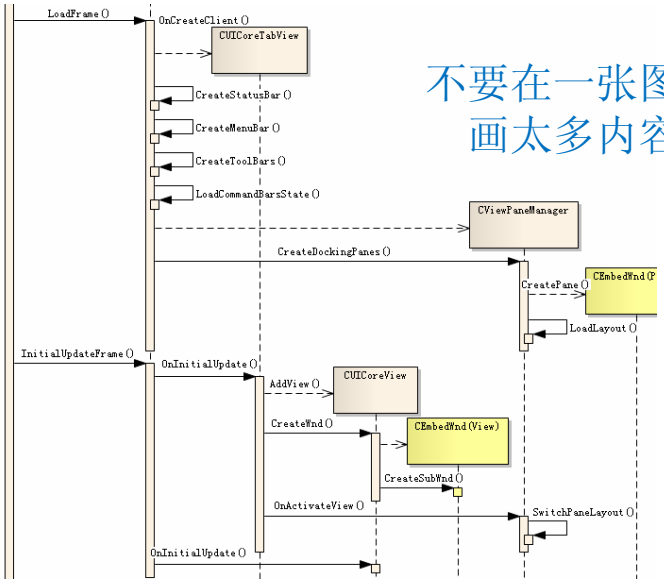
30

交互片断的用法—可选



31

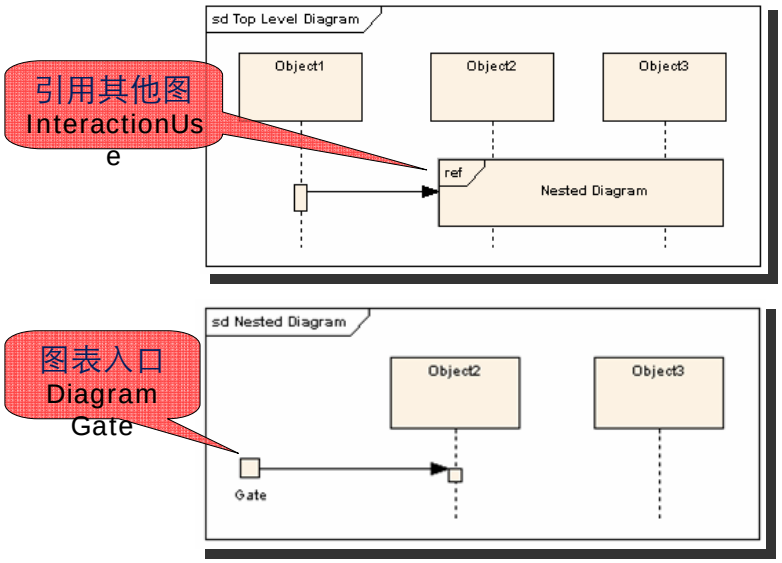
下面的序列图能看明白吗



不要在一张图上
画太多内容

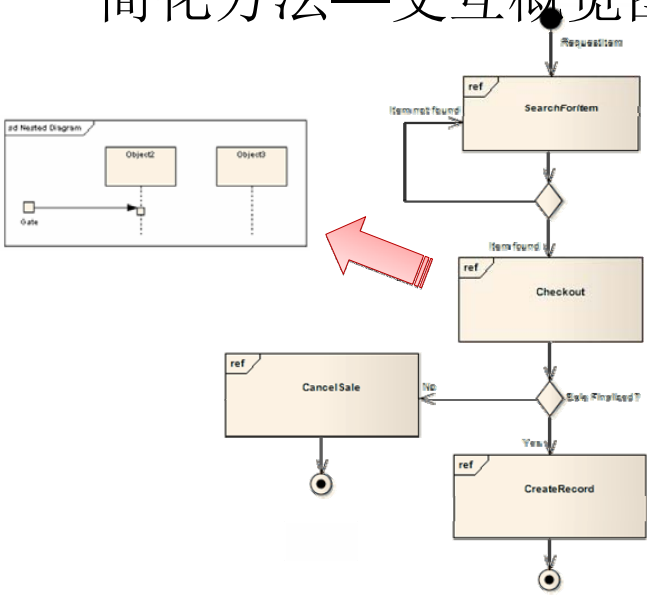
32

序列图分层（引用子图）



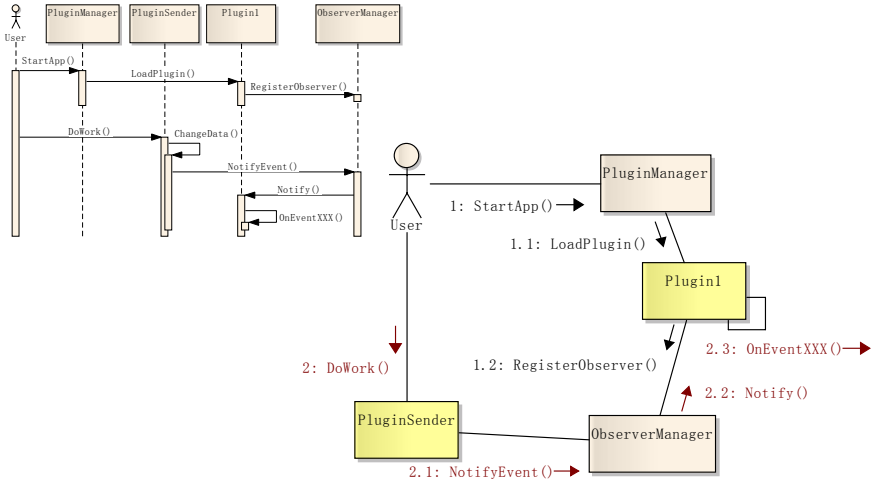
33

简化方法—交互概览图



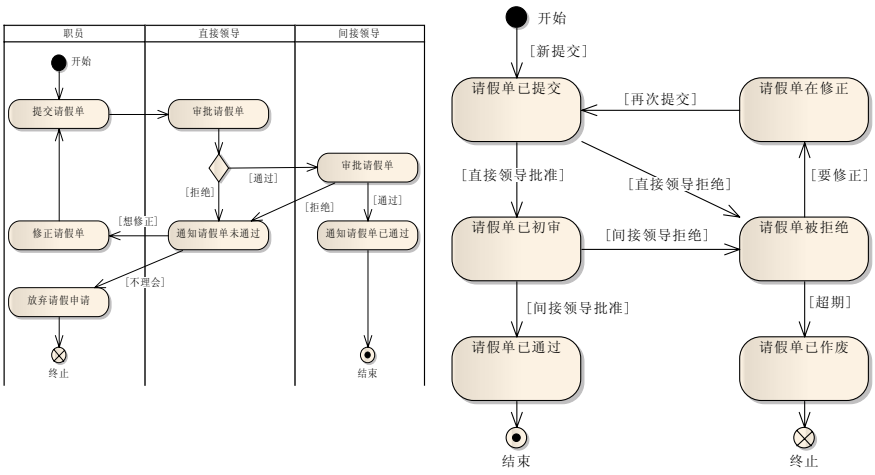
34

序列图 vs 协作图



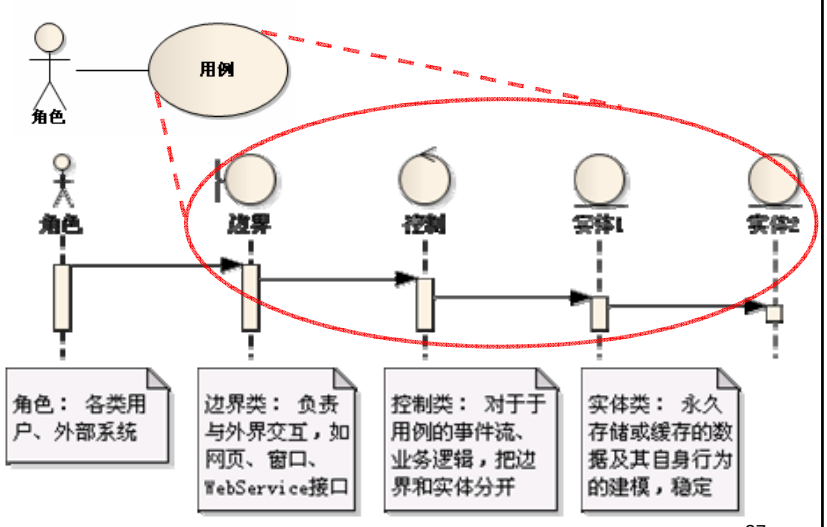
35

活动图 vs 状态图



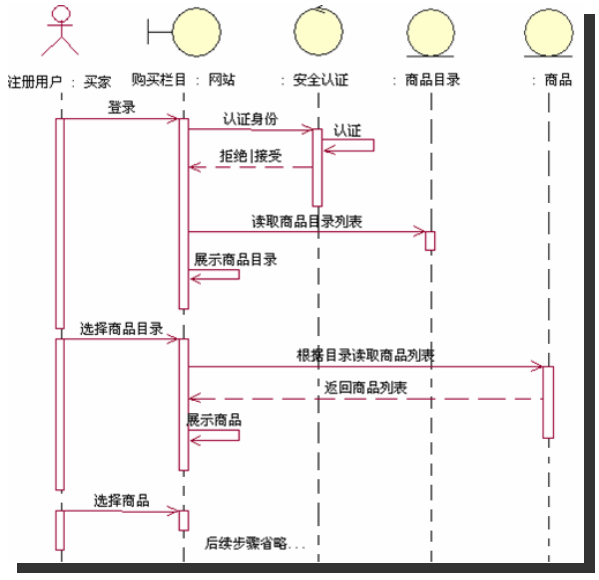
36

系统分析模式



37

系统分析模式

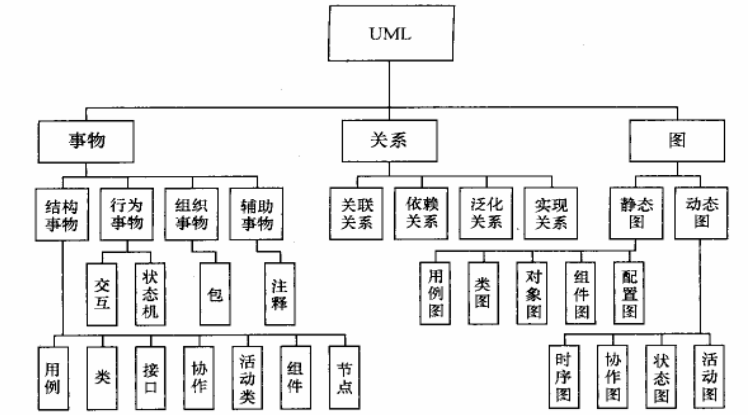


38

UML简介及常见疑难问题辨析

39

UML用来描述模型的内容有3种,分别是事物(Things)、关系 Relationships)和图(Diagrams)。



40

UML中的事物

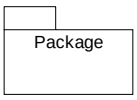
- UML中的事物包括结构事物、行为事物、组织事物和辅助事物(也称注释事物)。
- 结构事物(Structure Things)
  - 结构事物主要包括7种,分别是类、接口、协作、用例、活动类、组件和节点:

41

- 行为事物（Behavior Things）
  - 行为事物也称动作事物,是UML模型中的动态部分,代表时间和空间上的动作。行为事物主要有两种:交互和状态机。它们是UML模型中最基本的两个动态事物元素,通常和其他的结构元素、主要的类、对象连接在一起。
- (1)交互(Interaction)
  - 在UML图中,交互的消息通常画成带箭头的直线。
- (2)状态机(State Machine)
  - 状态机是对象的一个或多个状态的集合。

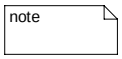
42

- 3、组织事物(Grouping Things)
  - 组织事物也称分组事物,是UML模型中组织的部分,可以把它看作一个个的盒子,每个盒子里面的对象关系相对复杂,而盒子与盒子之间的关系相对简单。组织事物只有一种,称为包（Package）
  - 包是一种有组织地将一系列元素分组的机制。包与组件的最大区别在于,包纯粹是一种概念上的东西,仅仅存在于开发阶段结束之前,而组件是一种物理元素,存在于运行时。在UML图中,包通常表示为一个类似文件夹的符号。



43

- 4、辅助事物(Annotation Things)
  - 辅助事物也称注释事物,属于这一类的只有注释(Annotation)。
  - 注释就是UML模型的解释部分。在UML图中,一般表示为折起一角的矩形。



44

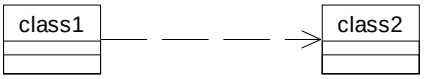
UML中的关系

- UML中的关系(Relationships)主要包括4种:关联关系、依赖关系、泛化关系和实现关系。
- 1、关联关系是一种结构化的关系,指一种对象和另一种对象有联系。给定关联的两个类,可以从其中的一个类的对象访问到另一个类的相关对象。在UML图中,关联关系用一条实线表示。

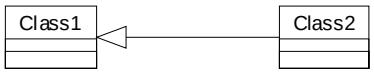
- 另外,关联可以有方向,表示该关联在某方向被使用。只在一个方向上存在的关联,称作单向关联(Unidirectional Association),在两个方向上都存在的关联,称作双向关联(Bidirectional Association)。

45

- 2、依赖(Dependency)关系
- 对于两个对象X、Y,如果对象X发生变化,可能会引起对另一个对象Y的变化,则称Y依赖于X。在UML图中,依赖关系用一条带有箭头的虚线来表示。



- 3、泛化(Generalization)关系
- UML中的泛化关系定义了一般元素和特殊元素之间的分类关系,与和C++及Java中的继承关系有些类似。在UML图中,泛化关系用一条带有空心箭头的实线来表示。

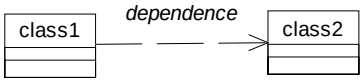


46

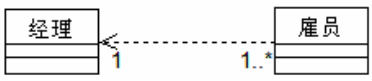
- 4、实现(Realization)关系
- 实现关系将一种模型元素(如类)与另一种模型元素(如接口)连接起来,其中接口只是行为的说明而不是结构或者实现。真正的实现由前一个模型元素来完成。在UML图中,实现关系一般用一条带有空心箭头的虚线来表示。

47

- 以上讲述了UML中的4种关系,除了需要注意各种关系的区别与联系以外,还要了解对关系的修饰。最常见的,对关系可以做两种修饰。
- 第1种是命名,即可以为关系取名。



- 第2种是数字,可以表示不同对应情况的关系,比如一对多、多对一、一对一和多对多等。



48

UML中的视图

- UML中的各种组件和概念之间没有明显的划分界限,但为方便起见,用视图来划分这些概念和组件。视图只是表达系统某一方面特征的UML建模组件的子集。视图的划分带有一定的随意性。在每一类视图中使用一种或两种特定的图来可视化地表示视图中的各种概念。
- 在最上一层,视图被划分成3个视图域:结构分类、动态行为和模型管理。

域	视图	图	主要概念
结构	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
	实现视图	构件图	构件、接口、依赖关系、实现
	部署视图	部署图	节点、构件、依赖关系、位置
动态	状态机视图	状态机图	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
模型	模型管理视图	类图	报、子系统、模型
扩展性	所有	所有	约束、构造型、标记值

49

“4+1”视图

- Use Case View (End-user: Functionality)
- Logical View (Analysts/Designers: Structure)
- Process View (System integrators: Performance, Scalability, Throughput)
- Implementation View (Programmers:Software management)
- Deployment View (System engineering: System Topology, Delivery, installation, communication)

50

UML图示

- Use Case Diagram
- Sequence Diagram
- Class Diagram
- Collaboration Digram
- State Diagram
- Activity Diagram
- Component Diagram
- Deployment Diagram

51

- 静态视图
  - 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。这护视图之所以被称之为静态的,是因为它不描述与时间有关的系统行为。
- 用例视图
  - 用例视图描述系统应该具备的功能,也就是被称为参与者的外部用户所能观察到的功能。用例是系统的一个功能单元,可以被描述为参与者与系统之间的一次交互作用。参与者可以是一个用户或者是另一个系统。客户对系统要求的功能被当作多个用例在用例视图中进行描述,一个用例就是对系统的一个用法的通用描述。用例模型的用途是列出系统中的用例和参与者,并显示哪个参与者参与了哪个用例的执行。

52

- 配置视图
  - 配置视图显示系统的物理部署,它描述位于节点上的运行实例的部署情况。
- 活动视图
  - 活动图是状态机的一个变体,用来描述执行算法的工作流程中涉及的活动。活动状态代表了一个活动:一个工作流步骤或一个操作的执行。活动图描述了一组顺序的或并发的活动。活动视图用活动图来体现。
- 状态视图
  - 状态视图是一个类对象所可能经历的所有历程的模型图。状态机由对象的各个状态和连接这些状态的转换组成。

53

- 交互视图
  - 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。类元是对在系统内交互关系中起特定作用的一个对象的描述,这使它区别于同类的其他对象。
- 模型管理视图
  - 模型管理视图对模型自身组织建模。一系列由模型元素(如类、状态机和用例)构成的包组成了模型。

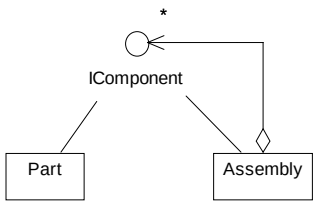
54

一些常见问题辨析

- 类的层次结构表示
- 属性与聚合
- 关联角色
- 关联类

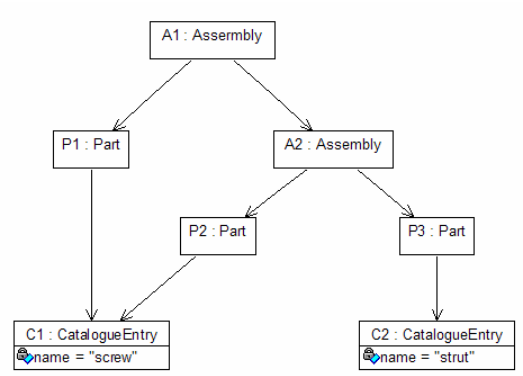
55

层次结构



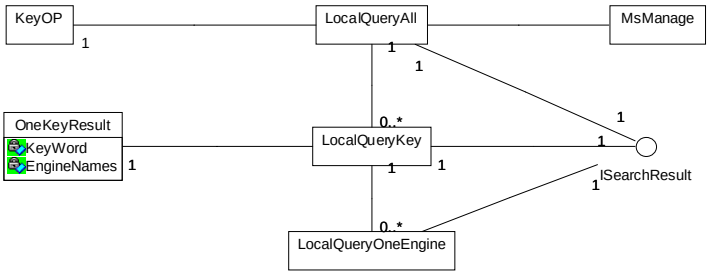
56

对象图



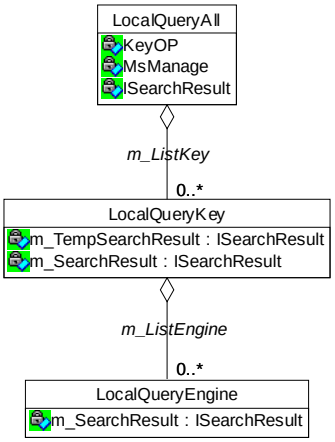
57

领域建模—重数

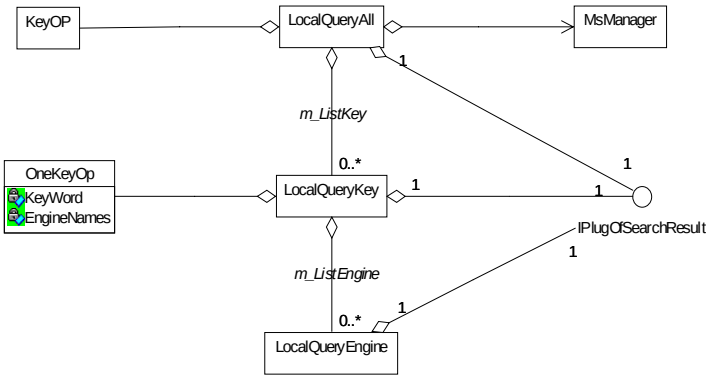


58

细化类模型


LocalQueryAll
KeyOP
MsManage
ISearchResult
m_ListKey
0..*
LocalQueryKey
m_TempSearchResult : ISearchResult
m_SearchResult : ISearchResult
m_ListEngine
0..*
LocalQueryEngine
m_SearchResult : ISearchResult

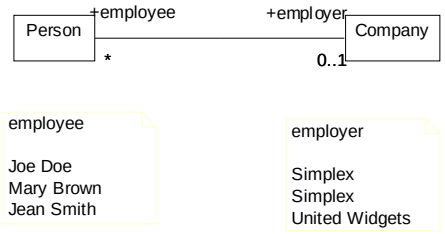
59


KeyOP
LocalQueryAll
MsManager
1
m_ListKey
0..*
1
1
OneKeyOp
KeyWord
EngineNames
LocalQueryKey
1
IPlugOfSearchResult
1
m_ListEngine
1
0..*
LocalQueryEngine

60

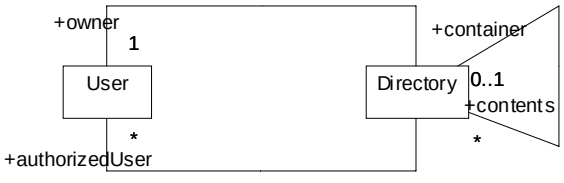
关联角色

- 关联角色名出现在关联终端的旁边。当仅仅使用关联名不足够表达清楚后，可以用关联角色名来加强表达。
- 可以把每个名称都当成伪属性，关联角色名提供了一种可以遍历关联的方法。



61

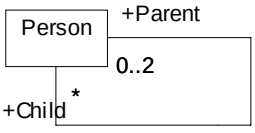
- 对同一个类的两个对象之间的关联来说，角色名是必需的。
- 例：一条目录可以包含很多子目录，子目录下还能嵌套包含其它下层子目录。每个目录有一个用户是目录的拥有者，还有多个得到授权可以访问此目录的用户。



62

- 在创建类图时，应该为使用正确的角色名，而不是为每个引用引入一个独立的类。
- 因为角色名可以区分对象，所以附在一个类上的关联名必须唯一（可以把角色名想象成类的伪属性）。同样，角色名不应该与类的属性名重复。

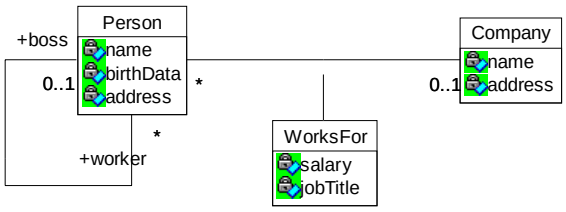
63



64

关联类

- 正如可以用属性描述类的对象一样，也可以用属性来描述关联的链接，可以把这样的一组属性组成关联类。



65

用例视图

- 用例图是由软件需求到最终实现的第一步,在UML中用例图用于对系统、子系统或类的行为的可视化,以便使系统的用户更容易理解这些元素的用途,也便利软件开发人员最终实现这些元素。
- UML中的用例图描述了一组用例、参与者以及它们之间的关系，包括以下内容:
  - 用例 (Use Case)
  - 参与者 (Actor)
  - 参与者之间的关系,泛化关系、包含关系、扩展关系等

66

参与者(Actor)

- 参与者(Actor)是系统外部的一个实体(可以是任何的事物或人),它以某种方式参与了用例的执行过程。参与者通过向系统输入或请求系统输入某些事件来触发系统的执行。参与者由他们参与用例时所担当的角色来表示。
- 参与者包括人参与者 (Human Actor)和外部系统参与者 (System Actor)。系统的用户是参与者,用户通过与系统的交互,操纵系统,完成所需要的工作。但参与者不一定是人,也可以是一个外部系统,该系统与本系统相互作用,交换信息外部系统可以是软件系统,也可以是个硬设备,例如在实时监控系统中的数据采集器,自动化生产系统上的数控机床等。

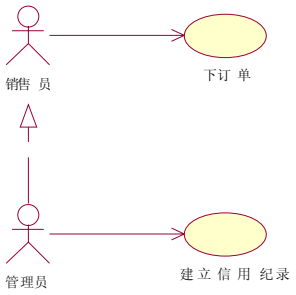
67

Actor的一些注意事项

- 包括人与系统，总是外部的。
- 定义了边界。
- Actor是“角色”，不是特定的人或特定的事。
- 要有恰当的名字。
- 可以泛化
- 不是可有可无的东西。另一方面它可以划分系统与外部实体的界限，是系统设计的起点。

68

Actor Generalization

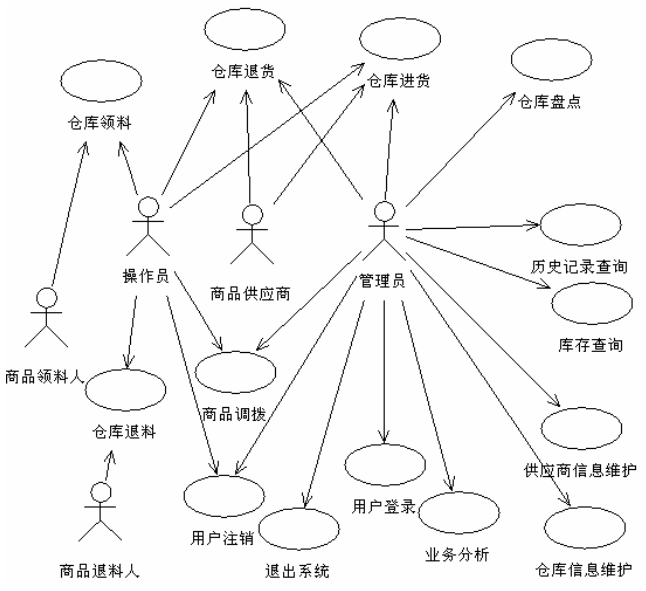


69

用例（ Use Case）

- 理解用例
  - 用例是对一个系统或一个应用的一种单一的使用方式所作的描述,是关于单个活动者在与系统对话中所执行的处理行为的陈述序列。
  - 用例是一个叙述型的文档,用来描述参与者(Actor)使用系统完成某个事件时的事情发生顺序。用例是系统的使用过程,更确切的说,用例不是需求或者功能的规格说明,但用例也展示和体现出了其所描述的过程中的需求情况。
- 识别用例
  - 识别用例最好的办法就是从分析系统的参与者开始,考虑每个参与者是怎样使用系统。使用这种策略的过程中可能会找出一个新的参与者,这对完善整个系统建模很有帮助。

70



仓库信息系统的用例图

71

- 用例与事件流：
  - 用例分析处于系统的需求分析阶段,这个阶段应该尽量避免考虑系统实现的细节问题。但是要实际建立系统,则需要更加具体的细节,这些细节写在事件流文件中。事件流的目的是为用例的逻辑流程建立文档,这个文档详细描述系统用户的工作和系统本身的工作。

- 用例间的关系：
  - 用例除了与其参与者发生关联外,还可以具有系统中的多个关系,这些关系包括:泛化关系、包含关系和扩充关系。应用这些关系是为了抽取出系统的公共行为和变体。

72

用例的一些注意事项

- 是需求分析的第一步。用户首先关心功能。
- 用例是对一个系统或一个应用的一种单一的使用方式所作的描述,是关于单个活动者在与系统对话中所执行的处理行为的陈述序列。
- 不是事件流。
- 不是需求规格说明，但反映了主要的功能性需求。
- 识别用例最好是从分析流开始。
- 每个用例都是独立的。
- 用例名用动宾结构描述，不要写成一个名词。
- 用例是分层的，一般而言，高层/中层用例更有实际意义。
- 不要将不同的用例混合在一起。
- 用例与编码：低层的用例可以辅助编码，高层的不能。
- 扩展用例：基础用例不必知道扩展用例的细节，只提供扩展点。

73

借鉴RUP的UML建模与分析

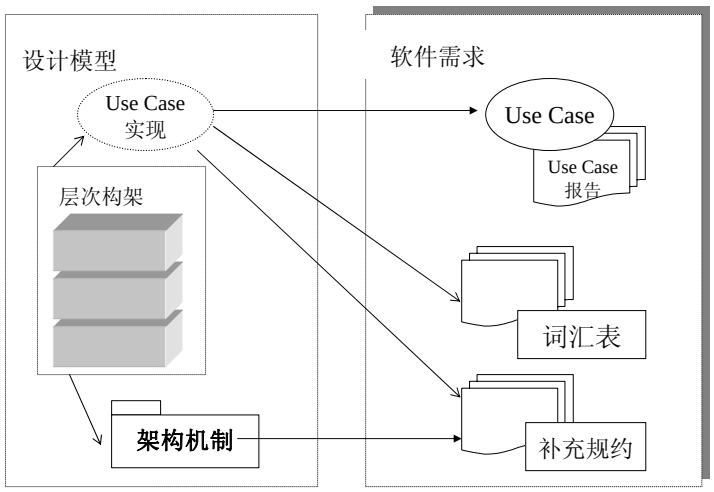
74

应用UML建模过程

- 任务和活动：
  有步骤、分层次地演进系统构架
  将软件需求逐渐转变为软件的设计方案
  保障软件的设计方案能够适应实施环境
- 总体框架：
  全局分析-局部分析-全局设计-局部设计-细节设计

75

应用UML建模过程



76

应用UML建模过程

设计模型的内容与演进

广义的设计模型包括在分析和设计活动中得到的所有结果。设计模型由3部分组成：

- “Use Case实现”：反映软件需求对设计内容的驱动
- 层次构架(Layers)：一种典型的构架模式，它将分析和设计的结果按照特殊到一般的等级分组，层次构架中的内容是后续开发活动的直接依据。
- 构架机制(Mechanism)描述可复用的设计经验

77

应用UML建模过程

- 需求规范

  需求规范涉及到为了获取、确认和维护一个系统的需求制品而执行的一系列活动。实际上，可以把需求看作客户和软件组织的一个合同。需求为软件经理提供了一种参考基准，是系统的发展可控。它们还可以作为设计团队、软件测试人员、质量保证小组以及准备用户文档的人们的输入。

78

- 用例的拆分
- 用例分包
  分包的目的：
  1.为了让系统用例图能更清晰地表现出系统的业务逻辑关系和层次，让用户能够通过开发人员的描述看到他们自己的业务结构，并对此认可。
  2.为了对系统进行模块的分割，这种分割将影响到系统今后的开发及系统的最终表现形式。

79

应用UML建模过程

- 分包的基本原则：
  1. 业务相关性：根据业务所属关系对用例进行分割划分，形成系统的业务包和子系统
  2. 功能相似性：根据用例的功能关联关系，将功能实现相似的用例组合在一起形成系统的业务包和子系统。
- 分包的过程：
  第一步先考虑业务相关性原则，进行第一层分包。
  对于较大系统，可以考虑功能相似性原则进行第二层分包。
  在完成了业务相关性分包之后，在进行同一业务的功能相似性划分可以提高系统某一模块的内聚性，同时，对于较大系统才可以划出比较有效的子系统，提高开发效率和成果。

80

全局分析

- 全局分析侧重于定义拟建系统所采用的构架以及影响构架的要素。全局分析充分利用相似或问题中的经验，避免在确定构架上浪费人力和物力。
- 选用架构模式
- 识别关键抽象：寻找那些无论在问题域或方案领域都具有普遍意义的概念点。
- 标识分析机制：将那些问题领域（应用逻辑）没有直接关联的计算机概念及相应的复杂行为表述为支撑分析工作的“占位符”。
- 选定分析局部：针对拟建系统的整体架构，找出那些蕴含相对高风险的局部作为工作内容。

81

全局分析

- 识别“关键抽象”活动的主要依据是词汇表、Use Case报告和既往的经验。
- 关键抽象的含义：业务需求和软件需求中通常会揭示拟建系统必须处理的核心概念，这些概念同样将成为设计模型中的核心要素。关键抽象，就是那些能够始终贯穿分析和设计的类及相应对象。关键抽象往往对应重要的实体信息。

82

全局分析

- 确定分析机制
  确定分析机制的主要依据是“补充规约”、既往的经验以及关键抽象（概念模型）；结果是一组被识别出来的“分析机制”。
- 什么是分析机制?
  在分析过程中，“分析机制”向设计人员提供复杂行为的简明表述，降低（全局）分析活动的复杂性并提高（局部）分析活动的一致性。通过这些机制，可以使分析工作更有重点。借助分析机制，可以姑且不深究那些用于支撑核心功能（应用逻辑）但其自身并非核心功能的复杂行为，这些内容通常是解决特定软件技术问题的设计经验。
  分析机制提供概念化的服务模式，“分析类”将使用这些服务模式的实例。分析机制的实例在系统架构中充当某些复杂行为的“占位符”。运用分析机制，可以避免分散全局分析的任务工作重点。一个比较典型

83

全局分析

- 例子是数据存取问题，事实上这并不是一个简单的问题，但却是一个很普遍的问题，如果在分析和设计的初期就陷入数据存取功能的细节，势必严重影响设计人员对整体架构的把握。在全局分析任务中，可以利用一个称作“留存”（Persistency）的分析机制封装相关的技术细节。分析机制通常与应用逻辑中的特定内容关系松散，分析机制大多只涉及计算机软件技术的概念和要素。分析机制为那些与应用逻辑密切相关的要素提供必要的软件技术支撑，根据层次架构的基本原则，用于实现分析机制的涉及内容大多位于构架的中低层。

84

全局分析

- 常见的分析机制
  留存
  分布式处理
  安全性
  进程间通信
  消息路由
  进程控制与同步
  交易事务管理
  信息交换
  信息的冗余
  错误检测、处理和报告
  数据格式转换

85

全局分析

	安全	留存	分布处理	信息交换
报销单	∨	∨	∨	
报销记录	∨	∨	∨	
员工		∨		
经理		∨		
工资户头	∨	∨		∨

86

Use Case 优先级 风险		调整策略	结算当月报销费用	批复报账申请	审核报账申请	提交报销申请	提交借款申请	完成日常维护	制作报表	登录系统
1	数据库的响应速度			★	★	★	★			
2	数据库的容量					★	★			
3	与内部邮件系统的连接	★	★	★	★	★	★			
4	与银行系统连接		★		★					
5	与人事管理数据库连接					★	★			★
6	内部网有足够的带宽			★	★	★	★			★
7	数据备份							★		

87

局部分析

- 提取分析类
- 转述需求场景
- 整理分析类

88

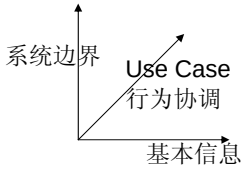
局部分析

- 提取分析类
  针对用例模型中每一个Use Case定义系统的分析元素如何为Use Case 提供相应的行为。
  分析类的定义和描述与Use Case 流建模是同时进行的。
- 确定分析类
  分析类是概念层面的内容，与应用逻辑直接相关。分析类的实例所具备的行为，用于捕获拟建系统对象模型的雏形。
  分析类直接针对软件的功能需求，因而分析类的实例的行为来自于对软件功能需求的描述(Use Case)。在使用分析类的时候，姑且不必关注某些与应用逻辑不直接相关的细节，特别是那些纯粹的软件技术问题。在某种意义上，可以认为分析类是技术解耦的。

89

局部分析

- 分析类的类型划分
  众多实践表明，如果立足于软件功能需求，拟建系统往往在三个维度易于发生变化：一、拟建系统和外部要素之间交互的边界；第二，拟建系统要记录和维护的信息；第三，拟建系统在运行中的控制逻辑。通常按照这三个变化因素的维度，将分析类划分为三种类型：边界类、控制类、实体类



90

局部分析

- 边界类的含义
  用于描述拟建系统外部环境与内部运作之间的交互，主要负责内容的翻译和形式的转换，并表达相应的结果。
  边界类主要用于描述三种类型的内容：
  拟建系统和用户的界面，拟建系统和外部系统的接口以及拟建系统与设备的接口。不同类型边界的建模工作有所不同。如与外部系统接口主要关注通信协议；用户界面的建模主要关注交互内容，在概念上表达整个界面，而不是具体窗口体构件。
- 控制类含义：
  通常控制类用于描述一个Use Case 所特有的事件流控制行为。控制类相当于协调人，被那些提出具体任务要求的类所共知：它自己通常不处理具体的任务，但它知道哪些类有能力完成具体的任务。
  控制类将Use Case 所特有的行为进行封装，具有良好的隔离作用概念上，拟建系统的其他部分（边界类和实体类）将与use Case 具体执行逻辑形成松散耦合。

91

局部分析

- 实体类的含义
  实体类用于描述必须存储的信息，同时描述相关的行为。实体类代表拟建系统的核心信息，是拟建系统的重要部分，通常需要长期保存。
- 提取分析类
  就是确定一组备选的、能够执行Use Case 中行为的分析类。
  从文字说明的软件需求过渡到图形描述的设计内容是一个渐进的过程，提取一组备选的分析类是这个过程的第一步。
  为了获得高内聚、低耦合的设计，使用三种不同的构造型识别和划分候选的分析类
  边界类：通常一个Actor和Use Case之间的通信关联对应一个边界类
  控制类：通常一个Use Case 对应一个控制类
  实体类：在全局分析中的概念模型（关键抽象）集合

92

局部分析

- 分析类在模型中的位置
  在全局分析任务中，“关键抽象”位于一般应用层的关键抽象包；在局部分析任务中，分析类通常分布在特定应用层和一般应用层
  -控制类通常放在特定应用层
  -从特定的分析局部中发掘的实体类通常放在特定应用层
  -表述用户界面的边界类放在特定应用层
  -表述外部系统接口的边界类放在一般应用层

93

局部分析

- 边界类的复用
  边界类实例的适用范围和生命周期可能超越特定的Use Case的时间流内容。如果两个Use Case同时和一个外部因素(Actor)交互，它们往往可以共用同一边界类，因为它们表现的行为和承担的责任有可能存在很多复用的内容

94

局部分析

- 控制类的变通：控制类实例的适用范围和生命周期通常和特定Use Case的事件流内容匹配。一个特定的 Use Case Realization对应一个控制类是典型的情况。
  - 如果不同Use Case包含的任务之间有比较紧密的联系，某些控制类可以参与多个Use Case Realization。
    - 换言之，不同的控制类可以参与同一个Use Case Realization。注意，这只是一种可能性，目的是重复利用相似部分从而降低整体的复杂性。通常，在明确多个Use Case Realization之后，才有可能发现它们之间的相通之处，在此基础上，这种做法才可能带来积极效果。不应该以此为目标，否则会适得其反。
    - Use Case Realization未必一定需要控制对象。在Use Case 事件流的逻辑结构非常简单的情况，边界类有可能在相关实体类的协助下实现相应Use Case的行为。简言之，如果Use Case Realization中的控制逻辑过于简单，那么（承担这类责任）控制类的必要性明显降低。

95

局部分析

- 实体类的建议
  实体类实例的适用范围和生命周期可能超越特定Use Case 的事件流内容。实体类通常不是某一特定Use Case Realization所专有。以下是针对提取实体类的一些建议：
  -充分利用关键抽象和已经被识别出来的分析类，可能是以往迭代中识别出的分析类，或者是当前迭代中从其它分析局部识别出的分析类
- 有时，和Use Case相关的Actor会在系统内部有一个实体类与之对应，尤其当该Actor的相关信息（属性、操作和关联）对实现拟建系统行为有直接贡献的时候。当然，Actor和与之相应的实体类在概念上截然不同，通俗讲，就是“内外有别”。
- 实体类不仅仅包含数据信息，还包含面向数据操作的行为，甚至是相当复杂的行为。

96

局部分析

- 如果有一个拟建的实体类A仅仅被另一个类B引用，并且实体类A不具有明显的行为特征；那么可以考虑将实体类A作为类B的属性；相反，如果需求中的某一实体信息有可能被多个类引用，或者该实体信息具有显著的行为特征，通常将其建模为一个独立的实体类。

97

局部分析

- 构造型的可选性
  分析类的构造型有助于更加清晰地展现分析活动的结果，目的是显式地提高团队对模型内容的管理和理解效率。但是，这并不意味着在局部分析活动中一定要使用分析类的构造型。从逻辑上讲，每个分析类都会归属于边界、控制或者实体中的某一维度。但是，如果从事分析的人员对不同分析类所属维度有清楚的理解，分析类构造型可以被隐含，不做显式表达。客观上，顺利地进入设计阶段后，分析类的构造造型将不再起实质作用。

98

局部分析

- 转述需求场景
  转述需求场景活动的主要依据是Use Case报告中的用文字描述的事件序列（需求场景）；该活动的结果是基于面向对象概念，用UML交互图(主要是序列图)转述的需求场景。
- 消息与责任

99

局部分析

- 局部分析的目的是用面向对象的方法转述需求中的应用逻辑。提取分析类的活动中找出了用于转述需求的分析元素。用面向对象的方法转述Use Case中的内容，就是用分布在一组分析类中的责任分担Use Case所要求的行为。
  描述分析类实例之间的消息转递过程过程就是将这些责任指派到分析类的过程。
- 这个过程是从软件需求过渡到设计内容的关键环节，其中核心概念是消息和责任的对应关系。Use Case的事件序列通常能用一组具有逻辑连续性的、介于分析类实例之间的消息传递加以表述。消息的发出者要求消息的接收者通过承担相应的责任作为对消息发出者的回应。一个分析类的实例在事件序列中接受的消息集合就是该分析类应承担责任的依据

100

局部分析

- 消息在概念上具有显著的动态特征，和分析类的实例相关联；而责任在概念上具有显著的静态特征，和分析类的定义相关联，消息在客观上是有次序的，但是责任并没有次序的概念。一种类型的责任往往能够响应多种消息。
  通俗地讲，责任的集合定义了分析类所具有的能力，之所以要具备这些能力是为了满足消息发出者的要求。消息的有序组合本质上表达出软件需求中的应用逻辑，分析类承担的责任集合本质上将驱动系统的设计，需求和设计在微观层面就这样被面向对象地关联在一起。
- 鉴于系统内部要素之间通过消息驱动，整体上，要素之间具有显著的弱耦合特征，这是面向对象带来的益处。

- 责任的沿用
  分析类的责任约定了分析类的行为，即分析类实例响应消息的能力。分析类在后续的设计活动中将逐步地演变为具体的设计元素；相应的，分析类的责任也将逐步地演化为设计元素的行为。具体讲就是设计类的操作和子系统接口的行为规约。

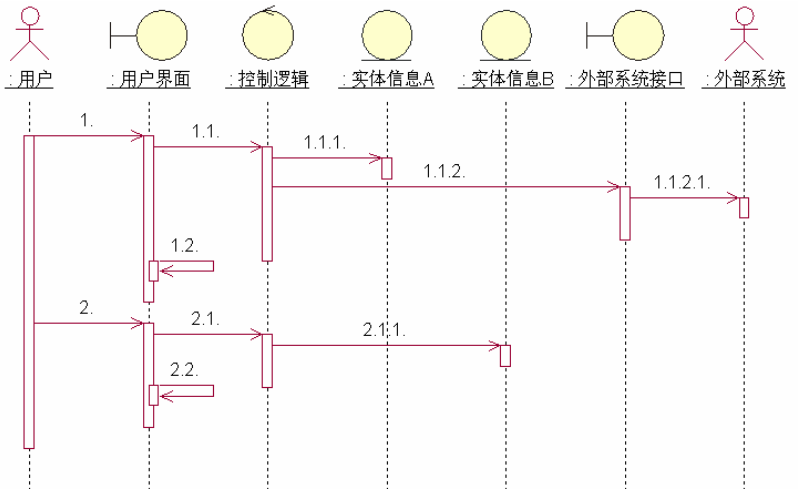
101

局部分析

- 序列图中的Actor实例
  序列图是表述消息传递的直接途径，是最常用的一种交互图。在转述Use Case场景的序列图中，通常会出现Actor的实例。Actor的实例表述系统外部的要素。而分析类的实例表述系统内部的要素，二者在概念上“内外有别”。对应一个Use Case有一个主导Actor作为交互时序的发起者，主导Actor的实例通常位于序列图的左侧。同一序列图中有可能出现多个Actor的实例，除了主导Actor的实例外，其他（被动）Actor的实例应尽量位于序列图的右侧。

102

局部分析



103

局部分析

- 描述Use Case 事件序列
  Use Case的事件流中包含多个事件序列，其中有一个基本事件序列和若干备选事件序列。简单讲，绘制序列图的工作就是在对象之间用消息传递的方式将事件序列的内容复述出来。
  概念上，Use Case的每一个备选事件序列的内容需要用一张独立的序列图来描述。

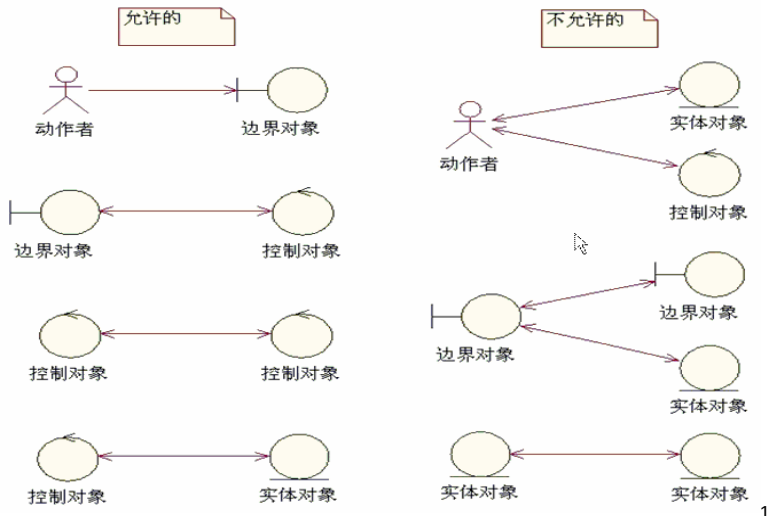
104

局部分析

- 找出对象传递消息的通道
  序列图中强调事件序列在时间维度上的轨迹，在轨迹的沿展过程中，我们了解到不同对象之间交互的丰富信息，具体讲就是“消息”。与此同时，我们发掘出消息赖以传递的通道，或者称之为对象之间的“连接”（Link）。笼统地讲，对象之间的连接意味着相应的类之间存在关联关系，这是后续设计活动的重要依据。在序列图中，对象之间只有显式的消息传递，连接的语义表述是隐含的，为了更明确地反映这层语义，可以绘制于序列图对应的的协作图，从而显式地表达“连接”。

105

局部分析



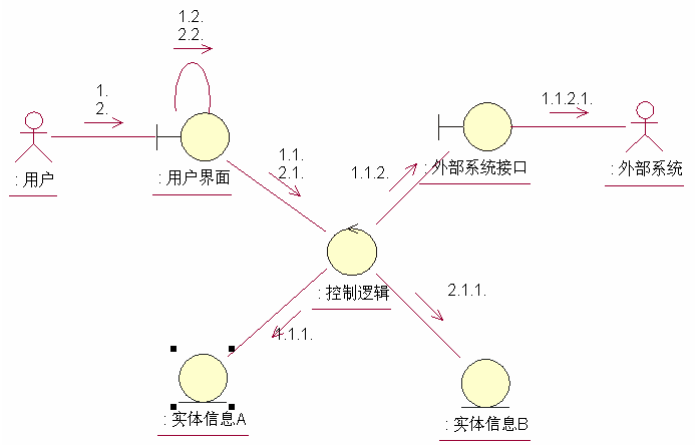
106

局部分析

换言之，如果存在跨越控制类实例生命线的消息，则有必要引起注意并做出相应的调整。在序列图中，控制类实例的生命线好像消息的一条分水岭。基于前端界面设计者的角度，这种序列图有显著的优点，即前端的设计和后端的处理备很好地隔离。
在对应的协作图中，如果将边界类实例及相应的Actor放在协作图的上方，实体类实例放在协作图的下方，控制类实例放在协作图的中央，那么控制类实例看起来很像一个贯通南北的交通枢纽。

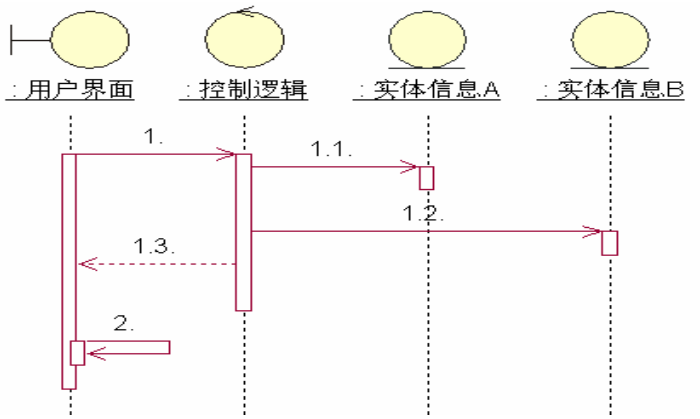
107

局部分析



108

局部分析



109

局部分析

- 整理分析类
  分析类是这样的类：它代表问题域中的简洁抽象；

- 分析类映射到真实世界的业务概念（并且据此仔细命名）。 问题域是首先产生软件系统（以及从此而来的软件开发活动）需求的域。通常，这是特定的业务领域，如在线销售或者客户关系管理。然而，务必注意，问题域可能根本不是任何特定业务活动，但 是可能产生需要软件在其上运转的一块物理硬件—这是嵌入式系统。基本上，所有业务软件开发服 务于某种业务需求，自动化一个已有业务过程或者开发具有有意义的软件组件的新产品。

110

- 分析类的最重要方面是，应该使用清晰和无歧义的方法映射到某个真实世界业务概念，如客户、 产品或账户。
- 这些工作的大部分是在需求工作流中捕获需求的活 动、创建用例模型和项目词汇表的活动中完成的。然而，进一步的澄清工作发生在创建分析类的用例 实现的过程中。
- 分析模型中的所有类都是分析类，而不是从设计考虑而产生的类（问题域），这一点是很重要的。 当你详细设计时，可能发现一个分析类被精化为一个或多个设计类。

111

局部分析

- 分析类应该展示非常“高级层次”的属性集合，它们表示最终设计类可能具有的属性。我们可以 说分析类为设计类捕获候选属性准备了条件。
- 分析类操作，在高级层次上，说明类必须提供的关键服务。在设计中，它们将成为实际的、可实 现的操作。然而，一个分析级的操作常常分解成多于一种的设计级的操作。
- 分析类的最小形式由以下部分组成：
  - 名称—这是强制的。
  - 属性—尽管只有候选属性的重要子集在此时建模，但是属性的名称是强制的。属性类型被认 为是可选的。
  - 操作— 在分析中，操作仅是类职责的高级层次的陈述。只在它们对于理解该模型很重要时， 显示操作的参数和返回类型。

112

分析类的职责

- 职责是类和它的客户之间的契约或者是类对它的客户的义务。本质上，职责是类提供给其他类的 服务。分析类具有直接同类（由它的名称所表达）的目的相一致的以及直接同该类正在建模的真实世 界“事物”相一致的非常内聚的职责集合，这一点是至关重要的。例如ShoppingBasket示例，你将期 望该类具有如下职责：
  - 向购物篮添加商品；
  - 从购物篮删除商品；
  - 显示购物篮中的商品。 这是内聚的职责集合，一切都是为了维护客户选择的商品集合。它是内聚的，因为所有的职责都朝着相同的目标—维护客户已经选择的商品集合。实际上，我们能够把这些职责概括为非常高级层 次的职责“维护购物篮”。
  - 同样，向 ShoppingBasket 中添加如下职责：

113

分析类的职责

  - 验证信用卡；
  - 接收付款；
  - 打印收据。 但这些职责似乎同购物篮的目的或直觉语法不匹配。它们不是内聚的，显然应该赋予其他什么类— 可能是类CreditCardCompany、类Checkout以及类ReceiptPrinter。把职责适当地分配给分析类 以最大化每个类中的内聚性，是很重要的。

- 最后，良好的类与其他类之间具有最低数目的耦合。我们以给定类与其他类具有关系的数目来度量类间的耦合度。类间职责的平均分布趋向于产生低耦合度。把控制或职责局限于单一的类趋向于增 加到该类的耦合度。

114

分析类经验法则

- 以下是创建形式良好的分析类的一些经验法则。
  - 每个类大约3~5个职责—典型地，类应该保持尽可能简单，这通常限制类能够支持的3~5个职 责的数目。先前ShoppingBasket 的示例是带有小的和可管理数目职责的专注的类的好的示例。
  - 不存在独立的类—好的OO分析和设计的精华是，类相互协作让用户受益。同样，每个类应该 同小部分类协作以提供所期望的功能。类可以把它们的一些职责托付给专注于特定功能的其他“辅助”类。
  - 当心很多非常小的类— 有时很难取得正确的平衡。如果模型看起来具有大量的非常小的类， 每个类都具有一个或者两个职责，那么你应该仔细查看以把一些小的类整合成更大的类。

115

- 当心少数几个非常庞大的类—上述的反面是，模型具有很少的类，但每个类都是具有职责数 量（>5）的庞大的类。解决问题的策略是依次查看这些类，看看是否每个类都能够被分解成为 两个或者多个能够承担恰当数目职责的、更小的类。
- 当心“伪类”—伪类其实是一般的过程函数，它伪装成类。
- 当心万能类— 存在似乎能够承担任何工作的类。看看名称为“system”和“controller”的 类！处理这个问题的策略是看看万能类的职责是否能够分解成内聚的子集。如果是，每个这些 内聚职责集合可能独立成类。这些更小的类协作以实现由原始万能类所提供的行为。

116

分析类经验法则

– 避免深度继承树—设计良好的继承层次的本质是继承层次中每个抽象层次应该具有良好定义 的目的。容易添加很多实际上不能服务于任何目的层次。

– 实际上，通常的错误是使用继承来实 现一种功能分解，其中每个抽象层次恰有一个职责！ 无论从哪个方面来讲，这都是无意义的，是会导致复杂的、难以理解的模型。

– 在分析中，类代表业务事物，而业务事物趋向于形成更宽（不超过三层）的继承层次。 我们把三层或者更多层次的继承认为是“深度”继承。

117

局部分析

- 实体类与属性的差异
  类的属性和独立的实体类有所不同。但是客观地讲，它们之间的界限并不非常清晰。在以下两种情况，通常用独立的实体类为特定客体（信息）建模。
  -客体具有比较复杂的自身行为
  -客体具有独立标识(Identification)，有可能被多类对象共享或者传递。

118

局部分析

相反，在以下情形，通常用类的属性为特定客体建模。
-客体除了非常简单的取/赋值操作(get,set等)，不具备更多其他行为。
-客体不需要独立标识，仅供一类对象使用。
有些时候，建模形式的选择不容易一步到位。例如，在分析和设计的演进过程中，如果发现之前定义的实体类A几乎没有行为并且仅仅被另外一个类B使用，那么可以将类A转换成类B的属性a。相反，如果类X的某个属性y越来越显现出复杂的行为特征，则可以考虑将该属性建模为一个独立实体类Y.

119

局部分析

- 不同分析类的同名责任
  在实践中，不同的分析类通常会包含相通名称的责任。因为承担责任的分析类有两种选择，一是自己负责把问题解决，二是将问题转交出去。作为发送相应消息的对象并不关心责任者如何解决问题，这是封装带来的益处。

120

局部分析

- 复用已有的责任、属性和关联关系
  在局部分析任务中，应当充分利用面向对象基本原理提供的天然优势，尽可能地复用已经存在的责任、属性和关联关系。一方面减少重复性建设，使得模型更加简单；更重要的是，避免相似内容的存在，提高对变化的后续响应效率，讲地维护成本。

121

全局设计

全局设计任务将在拟建系统的全局范围内，以分析活动的结果为出发点，将现有的分析类映射成模型中的设计元素，明确适用于拟建系统的设计机制，调整内容逐渐充实拟建系统构架。
全局设计任务中，有不同侧重的三项活动：

122

全局设计

- 确定核心设计元素。
  在系统架构的中高层，以分析类为出发点，确定相应的核心设计元素，具体讲就是设计类和子系统接口。

- 引入外围元素
  在系统构架的中低层，以分析机制为出发点，确定能够满足相关分析类要求的设计机制；根据设计机制所依赖的技术实施手段，将那些能够提供基本服务的构件所对应的逻辑内容引入设计模型。

- 优化组织结构
  按照高内聚、低耦合的基本原则，整理逐渐充实起来的层次构架内容。
  全局设计任务的总体思路是从上向下充实层次构架中的内容，然后作贯穿层次的调整。

123

设计模型和数据模型的关联

- 数据模型
  数据模型描述拟建系统中留存数据的逻辑和物理内容与结构。
  数据库设计师负责建立和维护数据模型。最初的数据模型在确定系统整体构架时建立。主要内容是那些对体系构架影响较大的留存数据，数据模型在后续活动中不断地演进和完善。

124

设计模型和数据模型的关联

- 设计模型和数据模型的映射
  -映射实体
  基于具体内容的视角，设计模型中留存类的一个实例（对象）和数据模型中表的一条纪录（行）相对应。基于组织结构的视角，留存类和表相对应，类的名称对应表的名称，类的属性名称于类型对应表的列名与类型。通常，没有必要将留存类的所有属性均映射到数据模型当中，只需映射那些需要留存的属性。

125

设计模型和数据模型的关联

-映射关系
关联关系的映射
泛化关系的映射

126

整理设计文档

- 分析和设计活动中的主要文档
  -设计指南。描述设计过程与设计向实施过渡中所遵循的原则和方法
  -Use Case 实现报告。展示分析和设计内容与需求中应用逻辑的关联
  -设计模型纵览报告。展示拟建系统构架的整体逻辑
  -设计包报告。展示特定设计包中的元素和图述
  -设计类报告。展示特定设计元素及其相关内容。

127

第二单元：UML与模式运用

128

软件系统开始坏死的症状

✓一个软件系统开始坏死时表现的症状有:
- 硬化Rigidity——系统变得越来越难以变更，修复或增添新功能的代价高昂;
- 脆弱Fragility——对系统的任何哪怕是微小的变更都可能造成四处（甚至是与变更处没有逻辑上的关联之处）崩溃;
- 绑死Immobility——抽取系统的任何部分用来复用都非常困难;
- 胶着Viscosity——以与原有设计保持一致的方式来对实施变更已经非常困难，诱使开发人员绕过它选择容易但有害的途径，其结果却使系统死的更快。

129

软件质量

✓软件系统最关键的质量特性有:
- 正确性correctness
- 健壮性robustness
- 可扩展性extensibility
- 可复用性reusability

✓软件系统其它重要的质量特性有:
- 兼容性compatibility
- 可移植性portability
- 易用性ease of use
- 高效性efficiency
- timeliness,economy and functionality

130

用GRASP模式指导设计

131

什么是GRASP模式

- General Responsibility Assignment Software Patterns——通用职责分配软件模式
- 它介于分析和设计之间，是在类被识别之后，如何向类分配职责的最基本模式
- 通常在进行用例分析与设计时，应用GRASP模式来确定类在协作中所要承担的职责，这些职责包括:
  - Knowing——掌握、或通过计算引申得到某些信息，以及相关的其它对象
  - Doing——自己做、或触发其它对象做某些事，以及控制和协调其它对象的行为
- GRASP模式的主要依据是面向对象的封装、抽象原理，和低耦合、高内聚设计原则等

132

GRASP模式列表

模式	简述
信息专家	将职责分配给信息专家（类），它掌握了为履行职责所必需的信息（数据）
创建者	如满足以下条件之一，将创建类A实例的职责指派给类B: 1）B包含了A；2）B聚合了A；3）B具有初始化A实例所需要的数据；4）B记录了A的实例；5）B紧密地使用A
高内聚	分配职责时应保持高聚合度
低耦合	分配职责时应保持低耦合度
控制者	将处理系统事件的职责分配给如下的类: 1）代表整个系统、设备、子系统的类；2）代表一个用例场景、一次会话的协调者

133

GRASP模式列表

模式	简述
多态	当相关选择或行为随着类型变化而变化时，将行为的职责—利用多态操作—指派给行为发生变化的类型
间接	将职责指派给一个中间对象，用来在其它构件或服务之间居中协调，以避免它们之间直接耦合
纯虚构	将一组高内聚的职责指派给一个虚构类，它不代表问题域中的任何概念，仅仅用来提供支持高聚合、低耦合与重用的行为
保护变量	识别预计的变量和常量，分配对应的职责以创建围绕它们的稳定接口，从而保护它们不对其它元素造成不期望的影响

134

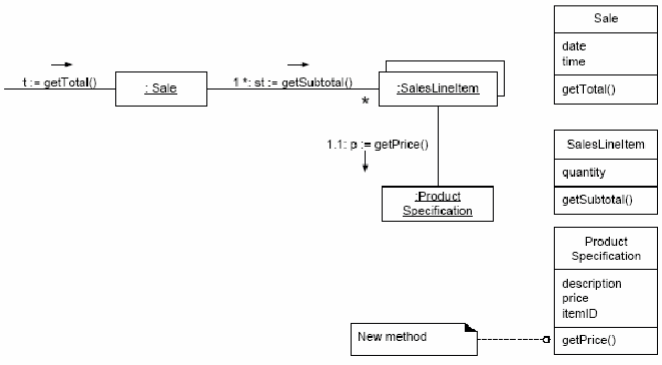
信息专家模式

- 将一个职责分配给信息专家——即掌握了为履行职责所必需的信息的类
- 对象只使用它们自己所包含的信息来完成任务，从而保持封装性，支持低耦合度
- 系统行为只分布于那些具有所需信息的类中，从而支持高聚合度

135

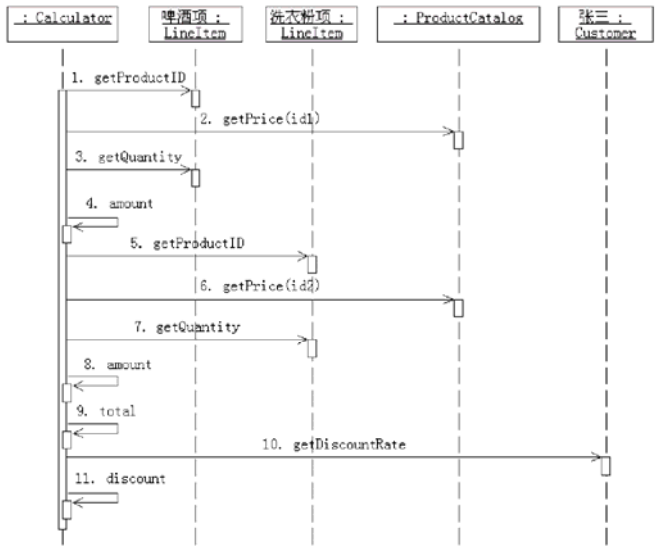
示例：信息专家模式

- 所有的类承担的职责都只依赖其本身所掌握的信息



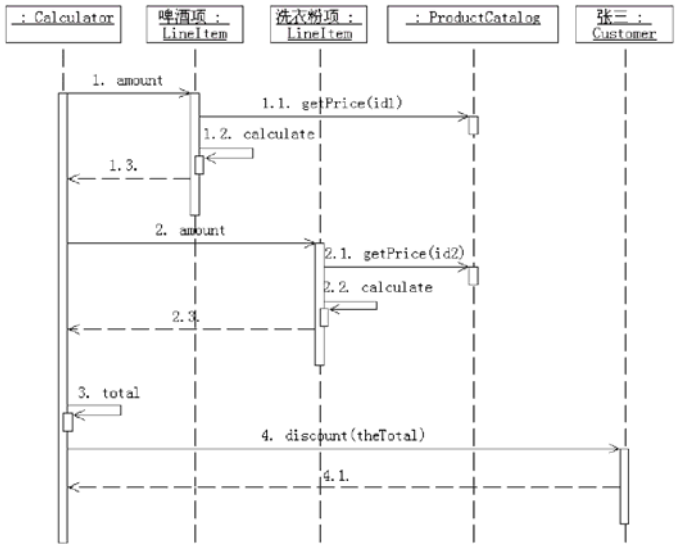
136

实例：违反信息专家模式



35

实例：应用信息专家模式



36

信息专家模式的不足

- 在某些场合下，为类分配职责时如果仅仅只考虑其掌握的信息是否满足要求，往往会得到一个并不优雅的设计
- 考察前面Sale类的例子，我们需要对其进行持久化操作（将Sale对象实例保存到数据库中），那么按照信息专家模式，我们会在Sale类上直接添加一个Save方法；然而这样做，使得Sale类严重依赖于数据库存取的细节，破坏了职责的内聚性，同时也违反了separation of concerns隔离关注面的原则
- 在GRASP等模式之上，实际上存在更为重要的东西——设计原则

139

创建者模式

- 将创建类A实例的职责指派给类B实例，如果满足以下条件之一的话：

1）B包含了A；

2）B聚合了A；

3）B具有初始化A实例所需要的数据（即B是创建A的实例这项任务的信息专家）；

4）B记录了A的实例；

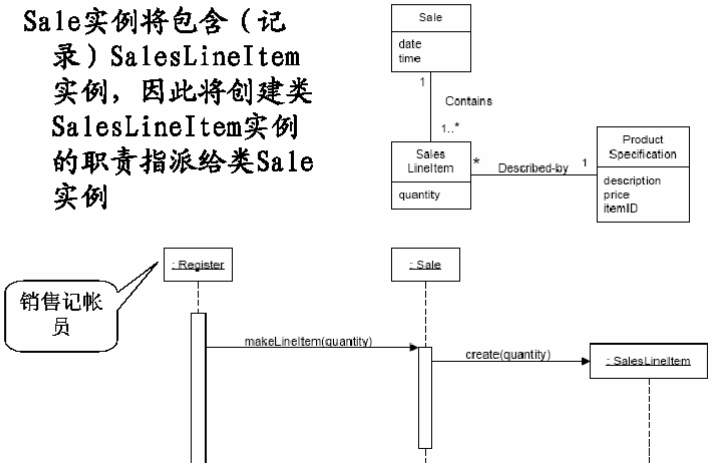
5）B紧密地使用A

- 如果满足两条以上条件的话，可以让B同时还聚合A

140

示例：创建者模式

Sale实例将包含（记录）SalesLineItem实例，因此将创建类SalesLineItem实例的职责指派给类Sale实例



141

低耦合模式

- 在为类分配职责时应保持低耦合度
- 耦合度——是对一个类与其他类关联、知道其他类的信息或者依赖其他类的强弱程度的度量
- 高耦合——过度依赖于多个其它类；它带来的问题有：相关类的变更会迫使自身也发生改变；难以独立地被理解；难以重用（需要其它被依赖的类同时使用）

142

低耦合模式

- 如果不以重用为目标，耦合就不那么重要了
- 如果类等元素是稳定的，耦合在此并不会带来负面问题；在类之间保持中等程度的耦合是正常和必要的，否则类的协作就无从谈起

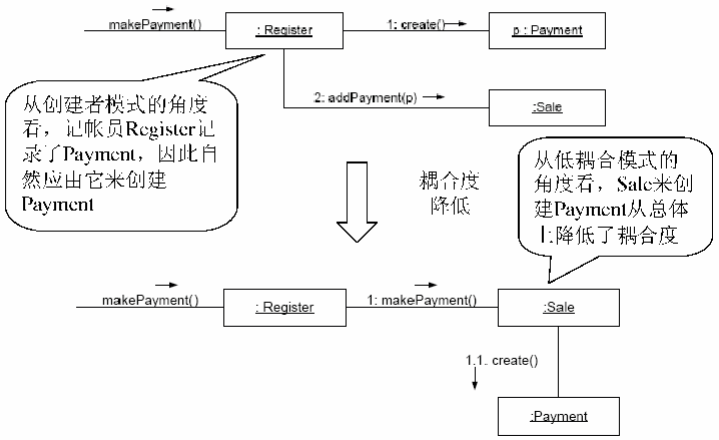
143

耦合的各种场景

- 类型X有一个属性（数据成员或实例变量）引用了类型Y的实例或类型Y本身；
- 类型X有一个方法以各种方式引用了类型Y的实例或类型Y本身；例如，类型X引用了类型Y的局部变量或参数，或者从一个消息返回的对象类型是类型Y的实例
- 类型X是类型Y的一个直接或间接的子类
- 类型Y是一个接口，类型X实现了这个接口

144

示例：低耦合模式

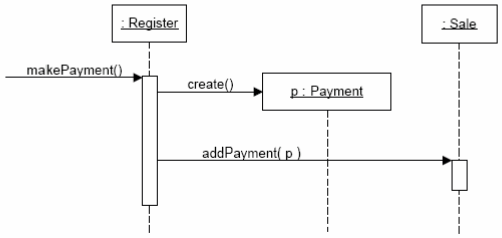


高内聚模式

- 在为类分配职责时应保持高内聚度
- （功能）聚合度——是对一个类中的各个职责之间相关程度和集中程度的度量
- 高内聚：一个类只（在一个功能领域内）承担高度相关的职责，并且不至于工作过度
- 低内聚：一个类做了许多不相关的工作，或者工作过度；它带来的问题有：难以理解；难以重用；难以维护；脆弱、易受变更的影响

示例：高内聚模式

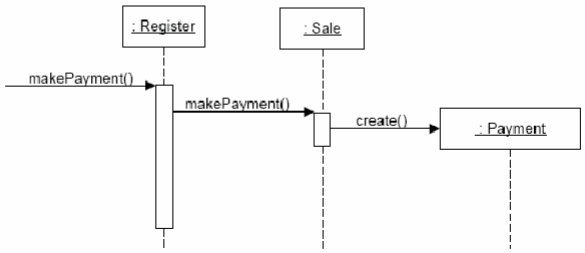
- 随着Register类增添其它的职责，让它继续来承担创建Payment的职责，将显著消弱Register行为的内聚度



147

示例：高内聚模式

- 显然由Sale来创建Payment，不但从总体上降低了耦合度，同时也提高了内聚度



148

控制者模式

- 控制者是处理系统事件的非边界类对象
- 将处理系统事件消息的职责分配给:

–代表整个系统行为的类（Façade controller 门面控制者）

–代表真实世界中自主地参与交互的类（Role Controller角色控制者，实质上对应到分析模型中的服务型控制类）

–代表一个用例中所有事件的处理者的类（Use case controller用例控制者，即对应到分析模型中的普通控制类）

149

控制者模式

- 推论:

外部接口对象和表示层（如窗口、Applet、Application、视图、文档等）不能用作控制者来承担处理系统事件的职责

使用同一个控制类处理同一个用例中的所有事件

- 本模式与RUP的分析类概念一脉相承，即与控制类的职责非常类似

150

系统操作与控制者

- 系统事件是由外部使用者发起的高层事件，是一个外部输入事件。
- 这些事件将对应于系统操作：系统为响应这些事件而执行的操作。
- 系统操作反应了企业或领域过程中的主要行为，应该由领域层的对象（即控制者）来处理，而不是在系统的接口层、表示层或应用层中被处理。

151

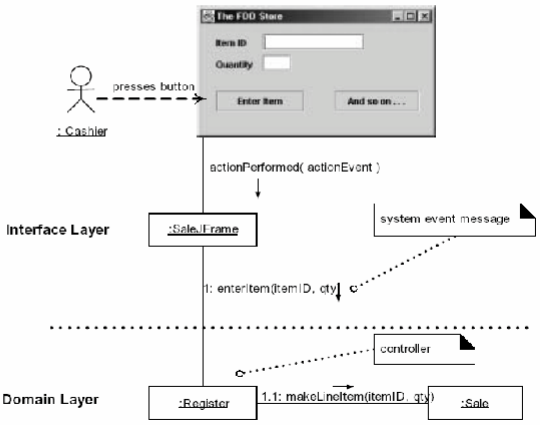
用例控制者

- 如果当把职责分配给其他种类的控制者时，将会导致低内聚或高耦合，或者当许多系统事件跨越了不同的过程时，可以选择使用用例控制者
- 增加了构件的可重用性
- 能够把握用例的状态
- 要警惕肥控制者，让它承担过多的职责

152

示例：控制者模式

- Register是用例控制者，承担了处理系统事件的职责



153

思考：用例行为分解与复用

- ✓软件复用的努力通常集中在设计层面上
- ✓用例的结构实质上是系统行为的结构，它提供了在行为层面实现复用的机会
- –调用包含用例，子用例对行为的继承，通过扩展用例扩展不同的行为组合，它们至少在描述上实现了复用
- –将用例结构在行为描述上的复用转化为实施上的复用，将把软件复用度提到更高的水平（类似于老的功能复用）

154

思考：用例行为分解与复用

– 这意味着系统的架构将反映用例的结构，当用例表达的需求发生变更时，其对构架的影响将被封闭在对应的局部范围内（针对于被包含用例、父用例、子用例、扩展用例、基用例等的实现部分）

– 可以使用UseCase Controller模式来实现这个目标:

- 每个被复用的用例都有对应的独立UseCase Controller类，这个类将在系统中依照用例结构的关系来被复用;
- 泛化关系中的子用例UseCase Controller类将继承对应父类的行为，可以在此应用Template Method模板方法模式。

155

思考：用例备选流实现

✓ 用例中的备选事件流通常会成为需求变更发生的主要场所

✓ 在用例详述中分别描述基本流和备选流，首先在需求描述工作上减少了需求变更的影响

✓ 如果能够在实施层面也能分隔基本流和备选流，则系统应对需求变更的能力将更强

– 为基本流和多个备选流分别开发各自的UseCase Controller，通过面向对象的动态绑定功能，在运行时刻依据条件逻辑判断来替换Controller对象实例，从而实现用例行为的分支执行

156

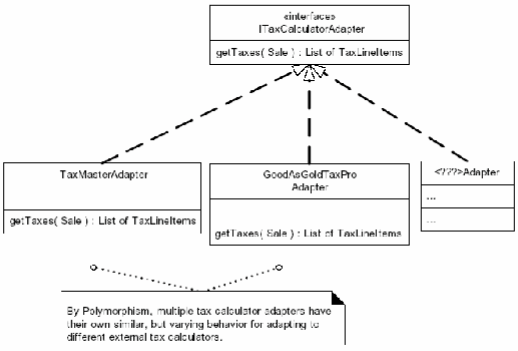
多态模式

- 当相关选择或行为随着类型变化而变化时，将行为的职责—利用多态操作—指派给行为发生变化的类型
- 推论：不要通过测试对象的类型，并使用条件逻辑判断来选择基于类型的可选执行分支
- 通过条件逻辑判断来选择可选执行分支，是程序中常见的处理场景，但这种方式在新的条件分支出现时引起的变动太大
- 一个基于通过多态来分配职责的设计可以很容易地被扩展以处理新的变化（包括上述新的执行分支）

157

示例：多态模式

- 不同的计税类实现了getTaxes()接口方法，利用多态操作，提供了不同的计税途径；而当新的计税法出现时，客户代码不受影响



158

纯虚构模式

- 将一组高内聚的职责指派给一个虚构类，它不代表问题域中的任何概念，仅仅用来提供支持高聚合、低耦合与重用的行为
- 类的设计通常通过两种途径:

（针对概念的）表示分解representational decomposition

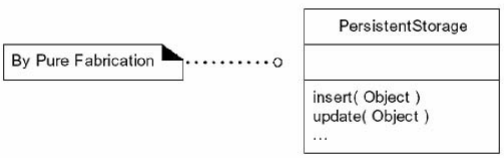
（纯粹的）行为分解behavioral decomposition

- 一个纯虚构通常基于行为分解，是一种以功能为中心的对象，而与问题域概念无关，通常为架构中高层元素提供服务

159

示例：纯虚构模式

- PersistentStorage纯粹是虚构类，它用来提供持久化的服务支持，从而帮助需要持久化的Sale类与持久化机制去耦合



160

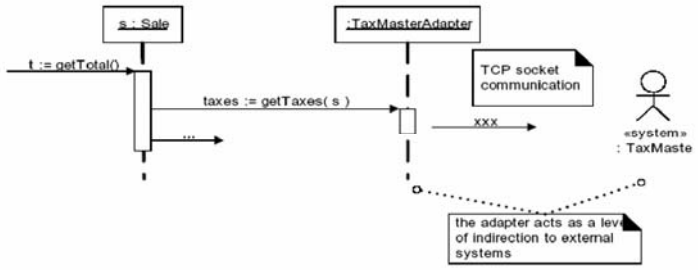
间接模式

- 当将职责指派给一个中间对象，用来在其它构件或服务之间居中协调，以避免它们之间直接耦合
- 实例：发布-订阅，观察者，适配器等

161

示例：间接模式

- TaxCalculatorAdapter充当了Sale与外部计税器之间的一个中间对象，TaxMasterAdapter通过多态重载getTaxes()接口实现，在此成为具体的中介者



162

保护变量模式

- 识别预计的变量和常量，分配对应的职责以创建围绕它们的稳定接口，从而保护它们不对其它元素造成不期望的影响
- 封装、接口、多态、间接等原则和模式是实现保护变量模式的基本途径
- 数据驱动设计、规则引擎、解释执行、反射、元数据设计等则是实现保护变量模式的更高级手段
- 不要与陌生人交谈模式是保护变量模式的一个特例

163

参考资料

<UML Distilled: A Brief Guide to the Standard Object Modeling Language>
<Applying uml and patterns 2nd>

164

领域模型

165

- 层次结构
- 领域模型
- 从EJB到轻量级框架

166

层次结构

- 表现层（present）
- 业务层
  - 业务层外观
  - 业务层核心
  - 领域对象管理/服务/仓库层
  - 领域对象层
- 持久层
  - 数据访问层
- 数据库

167

- 领域模型中的各种角色：
  - 实体--有唯一的标识,并且要有属性和行为(非GET/SET)，添加了行为，使其具有生命力。往往在设计时，实体的形为最难决断。为确定行为，我们必须识别它们的责任和协作。类的责任是指该类要做、知道、或决定的一切，由一个或多个方法完成。类中有属性和关联，协作就是为完成自己的责任所调用其它关联类。
  - 值对象--没有标识没有行为。如Address类。
  - 工厂---定义创建实体的方法，封装实例化对象并将一些关联对象注入。
  - 仓库(repository)管理实体的集合,主要有查找和删除实体的方法.实现类可以调用执久化层(如Hibernate，Ibatis)
  - 服务(Service)，实现整个应用程序的工作流(workflow)。服务包含那些无法指派的单个实体的行为，由作用于多个对象方法组成。如可以调用repository查找到实体对象，然后委派给这些对象。服务和facade很像，但不一样，它不处理以下事情：1）执行事务。2）收集返回给表现层的数据。3）脱钩对象。4）其它事情。服务可以说是业务的协调者，业务逻辑可以分散到实体对象中。

168

领域模型

- 失血模型
- 贫血模型
- 充血模型
- 胀血模型

169

失血模型

- DO只有属性及其getter/setter方法，没有任何业务逻辑。
- 缺点：行为与数据分离，很多情况导致维护与理解困难。

170

贫血模型

- DO包含不依赖于持久化的领域逻辑；依赖持久化的领域逻辑归入Service层。
  - Service(业务逻辑，事务封装)
  - DAO
  - DO
- 优点：
  - 各层单向依赖，结构清楚，易于实现和维护。
  - 设计简单易行，底层模型非常稳定。
- 缺点：
  - DO部分的持久化逻辑被放入Service层，不够OO。
  - Service层过重。

171

充血模型

- 与贫血模型类似，不同处在于如何划分业务逻辑：绝大多业务逻辑都应该放在DO里(包括持久化逻辑)，而Service层很薄，仅仅封装事务和少量逻辑，不和DAO层打交道。
  - Service(事务封装)
  - DO
  - DAO
- 优点：
  - 符合OO
  - Service层很薄，只充当Facade的角色，不和DAO打交道。

172

- 缺点：
  - DAO和DO双向依赖。
  - 如何划分Service层逻辑和Domain层逻辑没有确定的规则，取决与设计人员自己的理解。
  - Service层封装事务，须对所有的DO逻辑提供相应的事务封装方法，造成重定义所有的Domain logic。
  - Service的事务化封装的意义等于把OO的Domain logic转换为过程的Service 事务脚本。充血模型在domain层实现的OO在Service层又变成了面向过程。

173

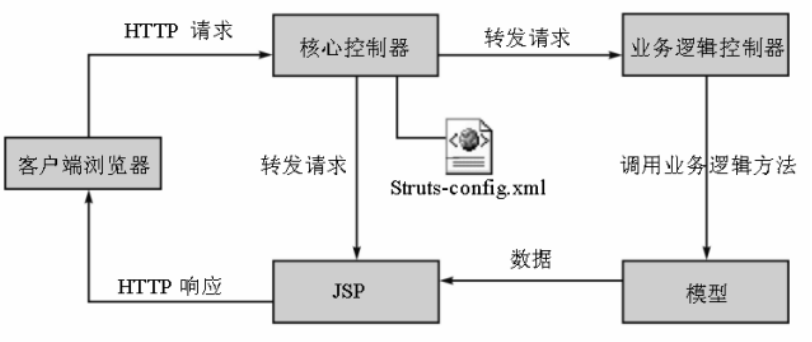
胀血模型

- 取消Service层，只剩下DO和DAO层，在DO的domain logic上面封装事务。
  - DO(事务封装，业务逻辑)
  - DAO
  - (RoR甚至合并为一层）
- 优点：
  - 分层简化
  - 符合OO
- 缺点：
  - service逻辑也强行放入DO ，引起了DO不稳定。
  - DO暴露给web层过多的信息，可能引起不必要的耦合。

174

- 原则：
  - 业务对象封装了内在的业务逻辑，而应用服务封装了外在于业务对象的业务逻辑。

175



Struts 1 的程序运行流程

176

面向对象设计的基本原则

177

面向对象设计的基本原则

✓类的设计原则:
开闭原则、依赖倒置原则、 Liskov替换原则、单一职责原则、接口分离原则、组合复用原则、所知最少原则
✓包内聚原则:
发布与复用等价原则、共同封闭原则、共同复用原则
✓包耦合原则:
无循环的依赖原则、稳定的依赖原则、稳定的抽象原则

178

liskov替换原则（LSP）

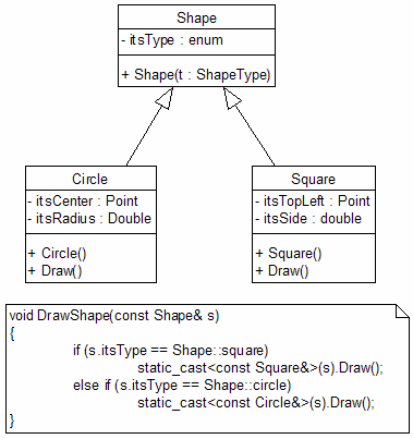
179

子类型必须能够替换掉其基类型

- 问题的根源是关于行为的：
  - 基类中有的行为在子类中不存在或不适当。
  - IS A的本质是指行为的一致，而不是生活中的语言。
  - 违反了LSP原则的本质是派生类的行为与基类中的不一致。
- 如果违反了LSP原则，常会导致在运行时的类型判断(RTTI)违反OCP原则。
  - 例如：函数A的参数是基类型，调用时传递的对象是子类型，正常情况下，增加子类型都不会影响到函数A的，如果违反了LSP，则函数A必须小心的判断传进来的具体类型，否则就会出错，这就已经违反了OCP原则。

180

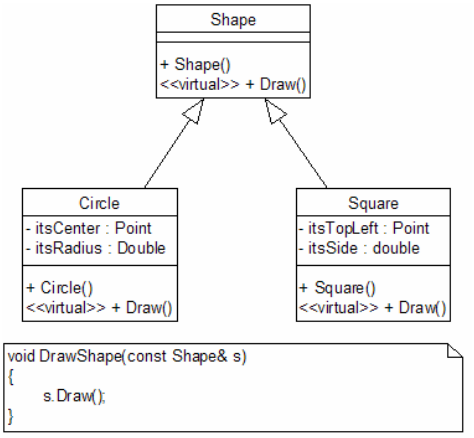
违反LSP导致违反OCP的简单例子




```
void DrawShape(const Shape& s)
{
        if (s.itsType == Shape::square)
                static_cast<const Square&>(s).Draw();
        else if (s.itsType == Shape::circle)
                static_cast<const Circle&>(s).Draw();
}
```

181

改善



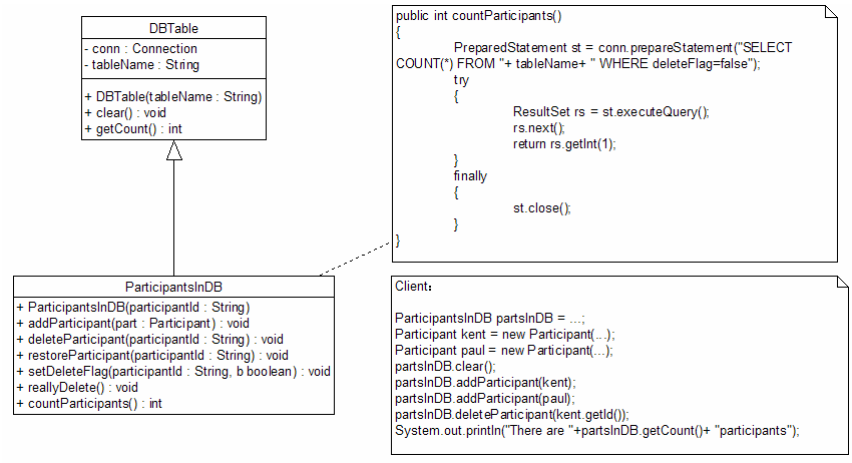

```
void DrawShape(const Shape& s)
{
    s.Draw();
}
```

182

例：会议管理系统

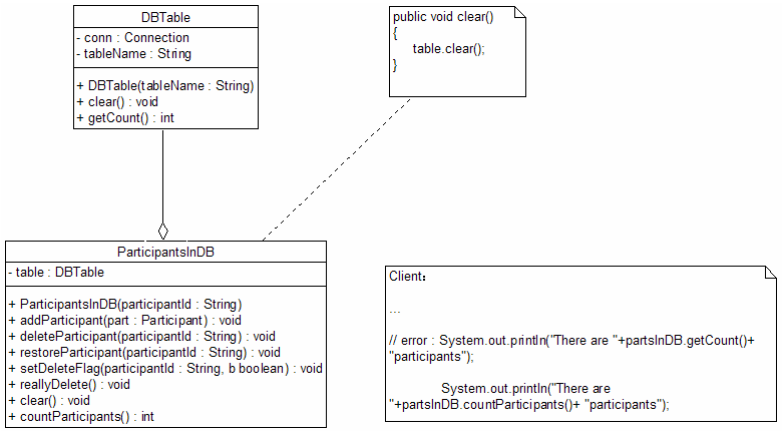
- 问题描述
- 用来管理各种各样的会议参与者信息。数据库里面有个表**Participants**，里面的每条记录表示一个参会者。
- 因为经常会发生用户误删掉某个参会者的信息。所以用户删除时，并不会真的删除那参会者的信息，而只是将该记录的删除标记设为**true**。**24**小时以后，系统会自动将这条记录删除。但是在这**24**小时以内，如果用户改变主意了，系统还可以将这条记录还原，将删除标记设置为**false**。

183



184

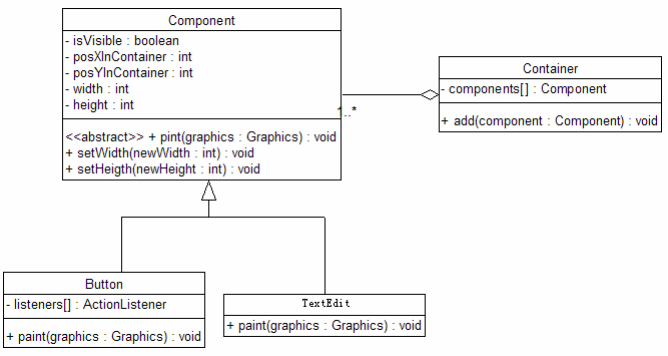
改善



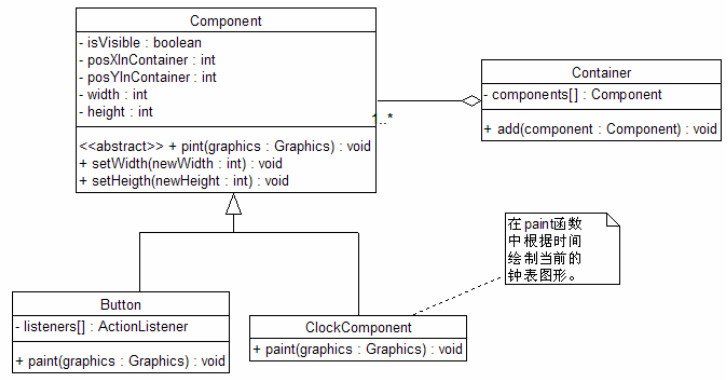
185

例：GUI对象

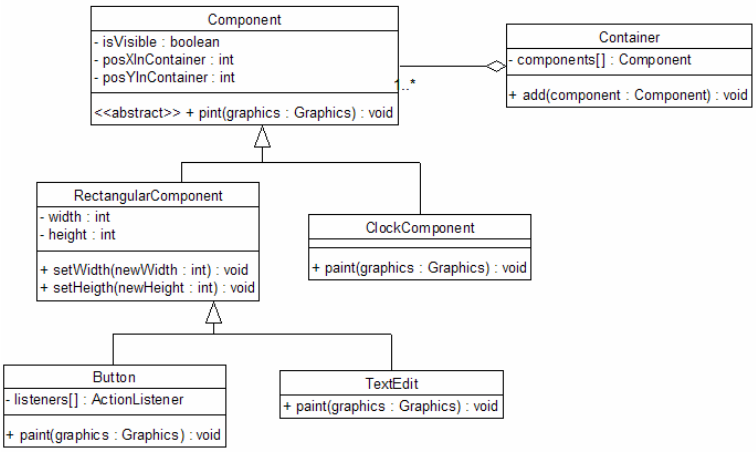
- **假定一个Component代表一个GUI对象，如按钮或者文本框等。**



186

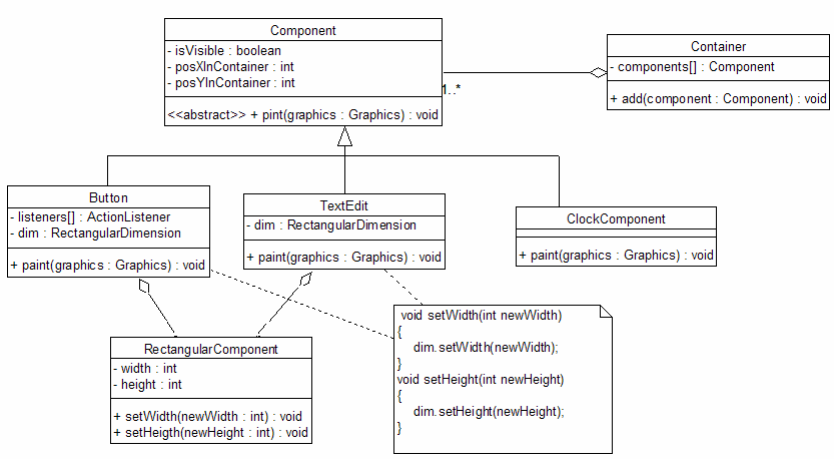

Component
- isVisible : boolean
- posXInContainer : int
- posYInContainer : int
- width : int
- height : int
<<abstract>> + pint(graphics : Graphics) : void
+ setWidth(newWidth : int) : void
+ setHeigth(newHeight : int) : void
Container
- components[] : Component
+ add(component : Component) : void
1..*
在 paint函数
中根据时间
绘制当前的
钟表图形。
Button
- listeners[] : ActionListener
+ paint(graphics : Graphics) : void
ClockComponent
+ paint(graphics : Graphics) : void

187


Component
- isVisible : boolean
- posXInContainer : int
- posYInContainer : int
<<abstract>> + pint(graphics : Graphics) : void
Container
- components[] : Component
+ add(component : Component) : void
1..*
RectangularComponent
- width : int
- height : int
+ setWidth(newWidth : int) : void
+ setHeigth(newHeight : int) : void
ClockComponent
+ paint(graphics : Graphics) : void
Button
- listeners[] : ActionListener
+ paint(graphics : Graphics) : void
TextEdit
+ paint(graphics : Graphics) : void

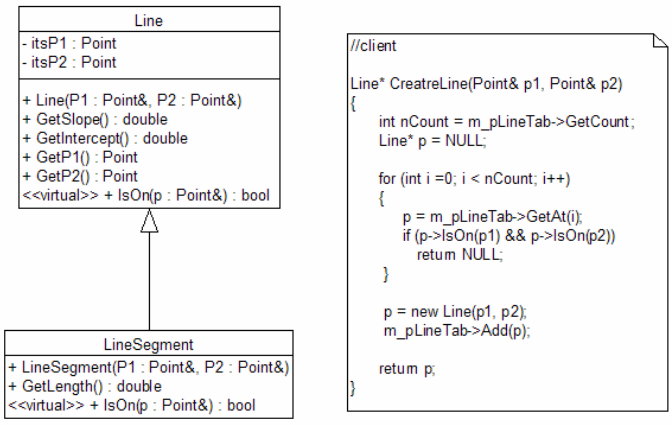
188

改善2


Component
- isVisible : boolean
- posXInContainer : int
- posYInContainer : int
<<abstract>> + pint(graphics : Graphics) : void
1..*
Container
- components[] : Component
+ add(component : Component) : void
Button
- listeners[] : ActionListener
- dim : RectangularDimension
+ paint(graphics : Graphics) : void
TextEdit
- dim : RectangularDimension
+ paint(graphics : Graphics) : void
ClockComponent
+ paint(graphics : Graphics) : void
RectangularComponent
- width : int
- height : int
+ setWidth(newWidth : int) : void
+ setHeigth(newHeight : int) : void
void setWidth(int newWidth)
{
dim.setWidth(newWidth);
}
void setHeight(int newHeight)
{
dim.setHeight(newHeight);
}

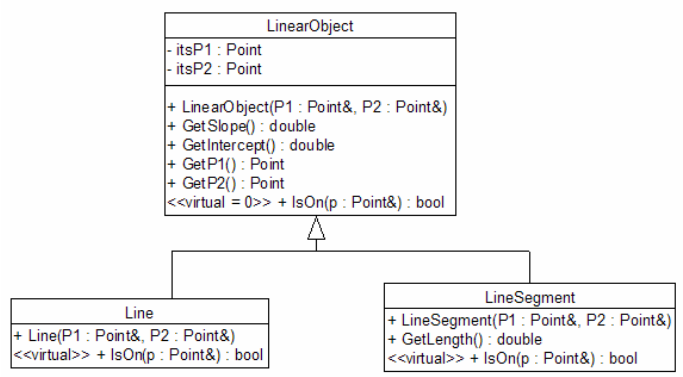
189

例


Line
- itsP1 : Point
- itsP2 : Point
+ Line(P1 : Point&, P2 : Point&)
+ GetSlope() : double
+ GetIntercept() : double
+ GetP1() : Point
+ GetP2() : Point
<<virtual>> + IsOn(p : Point&) : bool
LineSegment
+ LineSegment(P1 : Point&, P2 : Point&)
+ GetLength() : double
<<virtual>> + IsOn(p : Point&) : bool
//client
Line* CreatreLine(Point& p1, Point& p2)
{
int nCount = m_pLineTab->GetCount;
Line* p = NULL;
for (int i =0; i < nCount; i++)
{
p = m_pLineTab->GetAt(i);
if (p->IsOn(p1) && p->IsOn(p2))
return NULL;
}
p = new Line(p1, p2);
m_pLineTab->Add(p);
return p;
}

190

改善：提取公共部分替代继承



191

接口隔离原则（ISP）

康凯

192

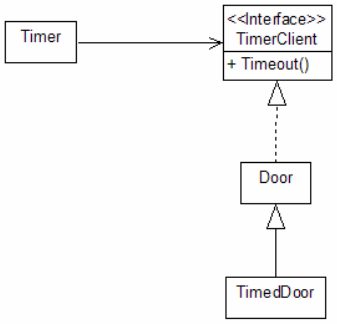
例

- 安全系统，有一些Door对象可以被加锁和解锁，并且Door对象知道自己是开着还是关着的。

- 需求变化：
- 实现一个TimedDoor，如果门开着的时间过长，它就会发出警报声。为了做到这一点，TimeDoor对象需要和一个Timer的对象交互。如何将TimerClient与TimedDoor类联系起来？

193

一种方案




```
class Door
{
public:
        virtual void Lock() = 0;
        virtual void Unlock()  = 0;
        virtual bool IsDoorOpen() = 0;
};

class Timer
{
public:
        void Register(int timerout, TimerClient*
client);
};

class TimerClient
{
public:
        virtual void TimeOut() = 0;
};
```

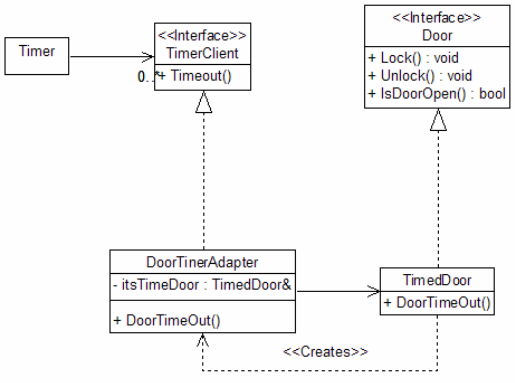
194

问题

- Door依赖于TimerClient
  - 并不是所有种类的Door都需要定时器。
  - 违反LSP：不需时钟的派生类中需要提供TimeOut的退化方法。
  - 不必要的复杂和重复性：使用这些派生类的客户程序被迫包含TimerClient的定义。
- Door接口被TimeOut方法污染了：
  - 持续的加入方法会使Door接口不断变胖。
  - 每次Door中加入一个方法，其它派生类中都需要对自己不需要的方法提供退化处理。
- 客户对TimerClient接口的改变会影响接口使用者。
  - 例如需要在Timer中注册多个超时通知（在TimeOut方法中加入超时ID），则即使不需要定时器的类也会受影响。

195

使用委托分离接口



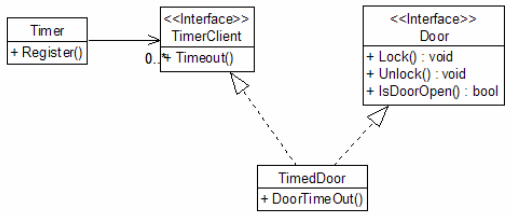

```
class Timer
{
public:
        void Register(int timerout, TimerClient* client);
};

class TimedDoor : public Door
{
public:
        virtual void DoorTimeOut();
};

class DoorTimeAdapter : public TimerClient
{
public:
        DoorTimerAdapter(TimeDoor& theDoor) :
itsTimeDoor(TheDoor)
        {
        }
        virtual void TimeOut(int timeOutID)
        { itsTimedDoor.DoorTimeOut(timeOutID);
        }
};
```

196

使用多重继承分离接口



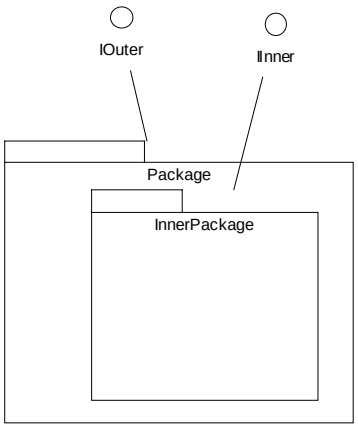

```
class TimedDoor : public Door, public TimerClient
{
public:
        virtual void DoorTimeOut(int timeOutID) { //do something;}
};

//client
TimerClient* tmClient = NULL;
bool bRet = CreateDoorInstance(&tmClient);

int timeID = tmClient->GetTmID();
timer->Register(timerout, timeID, tmClient);
```

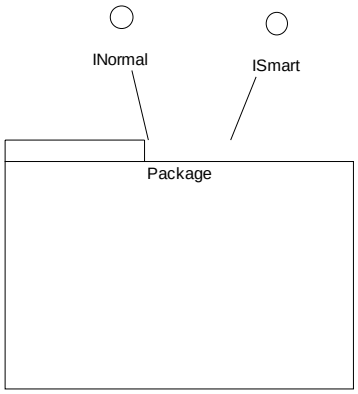
197

内接口与外接口



198

普通接口与智能接口



199

软件系统坏死的症状

200

“Copy”程序

- 一个从键盘读入字符并输出到打印机的程序。

```
void Copy()
{
    int c;
    while ((c = RdKbd()) != EOF)
        WrtPtr(c);
}
```

201

需求在变化

- 用户希望Copy程序能从纸带读入机中读入信息。
  - 现实中的约束——不能改变接口
- Copy程序的第一次修改结果

```
bool ptFlag = false;
//remember to reset this flag
void Copy()
{
    int c;
    while ((c = (ptFlag ? Rdpt() : Rdkbd())) != EOF)
        WrtPtr(c);
}
```

202

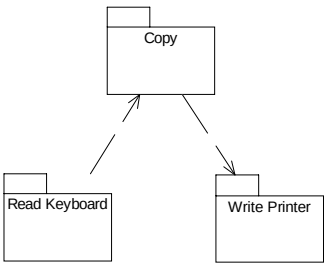
需求在变化2

- 客户希望Copy程序有时可以输出到纸带穿孔机上。

```
//Copy程序的第二次修改结果
bool ptFlag = false;
bool punchFlag = false;
//remember to reset these flag
void Copy()
{
    int c;
    while ((c = (ptFlag ? Rdpt() : Rdkbd())) != EOF)
        punchFlag ? WrtPunch(c) : WrtPtr(c);
}
```

203

初始的设计的问题



- 依赖关系的不灵活性：
  - 高层模块直接依赖于底层细节。
  - Copy模块直接依赖于KeyBoardReader和PrinterWriter
  - 依赖倒置：可以用Strategy模式实现依赖的倒置。
- 针对实现编程，而没有针对接口编程。

204

依赖倒置原则（DIP）

康凯

205

相关概念

- 关于解耦
  - 依赖倒置（DIP）
  - 控制反转（IoC）
  - 依赖注入（DI）
  - 服务定位器（SL）

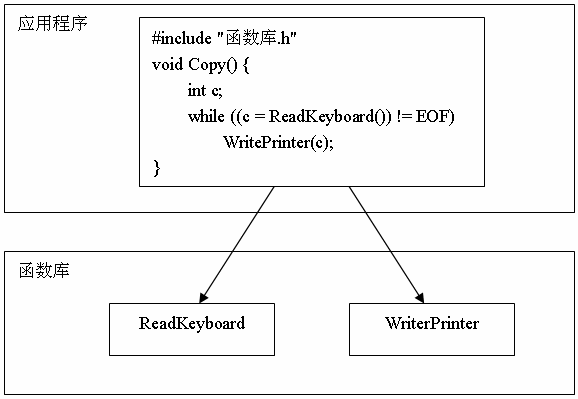
  - 有些是手段，有些是原则，不过其间的差异并不太重要，更重要的是其共同点：其根本目标都是解除依赖，将组件的配置与使用分离开。

- 其它名词
  - 服务
  - 组件
  - 框架
  - 类库
  - 应用程序

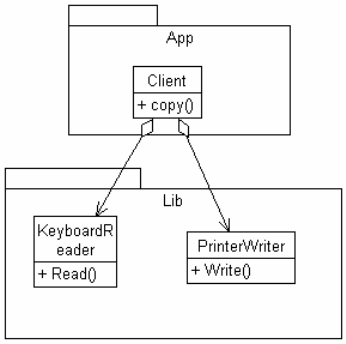
206

接口和实现分离

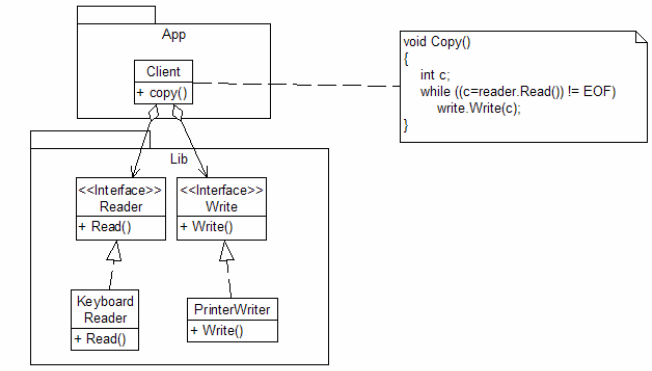
- 面向过程的接口与实现分离



207

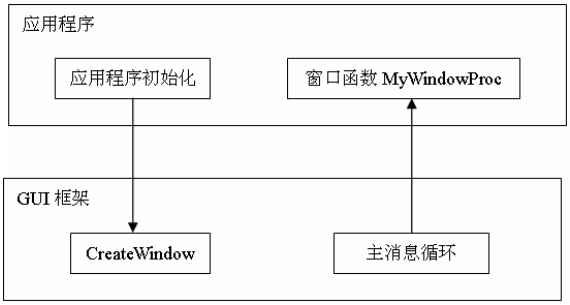


208



209

依赖于特定函数的框架



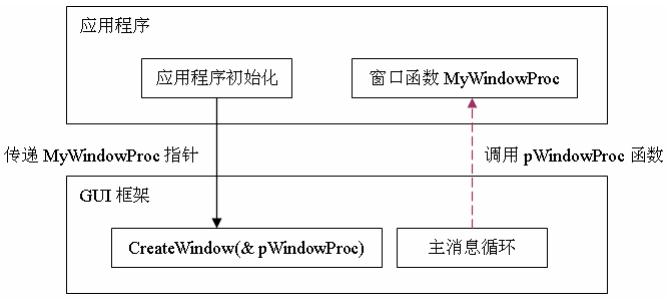
210

控制反转（IoC）

- 问题：双向依赖：
  - 应用程序依赖GUI框架：应用程序调用GUI框架中的CreateWindow()函数来创建窗口
  - GUI框架依赖应用程序：GUI框架不了解该窗口接收到窗口消息后应该如何处理，这一点只有应用程序最为清楚。因此，当GUI框架需要发送窗口消息时，又必须调用应用程序定义的某个特定的窗口函数。
- 双向依赖关系的缺陷：由于GUI框架调用了应用程序中的某个特定函数，GUI框架无法独立存在，换一个新的应用程序，GUI框架就要做相应的修改。
- **如何消解框架系统对应用程序的依赖关系是实现框架系统的关键。**

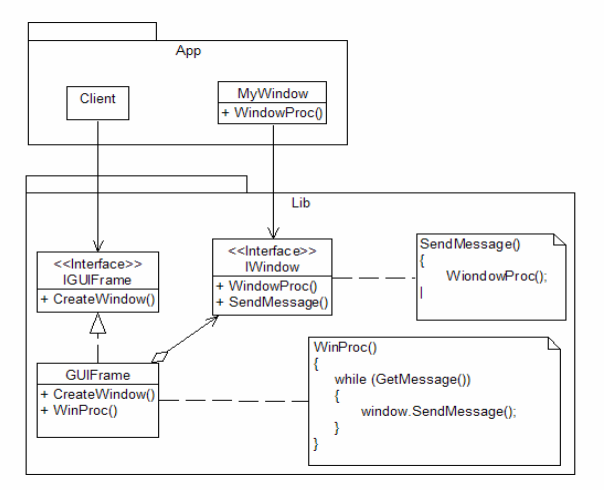
211

- 通过回调函数消解框架到运用程序的依赖：

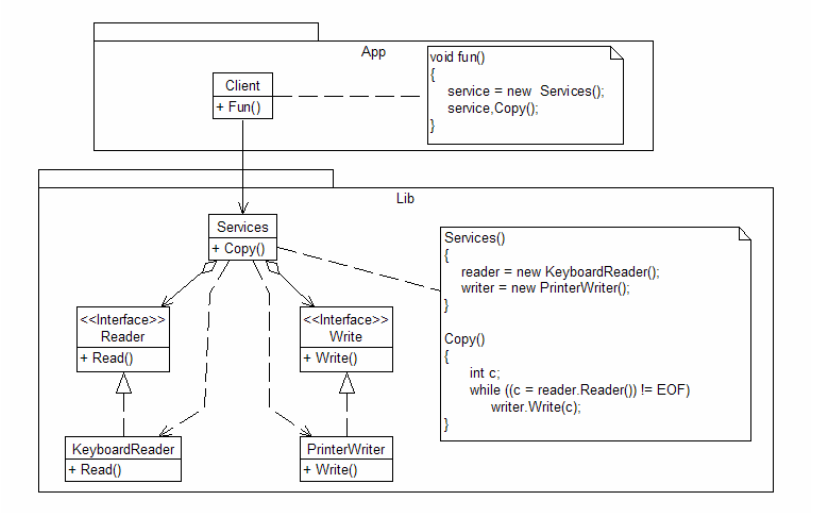


212

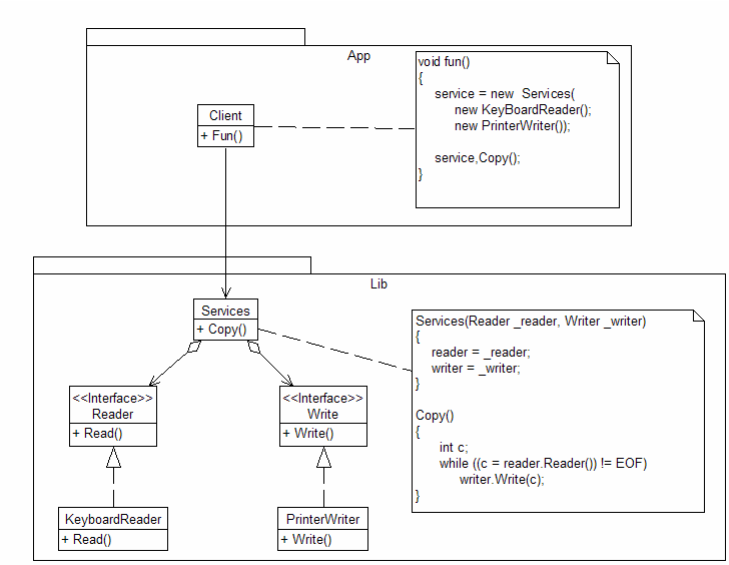
- 通过模板方式消解GUI框架到应用程序的依赖



213



214



215

电影清单的例子

- 一个组件，用于提供一份电影清单，清单上列出的影片都是由一位特定的导演执导的。

```
class MovieLister {
    ...
      public Movie[] moviesDirectedBy(String arg) {
          List allMovies = finder.findAll();
          for (Iterator it = allMovies.iterator(); it.hasNext();) {
              Movie movie = (Movie) it.next();
              if (!movie.getDirector().equals(arg)) it.remove();
          }
        return (Movie[]) allMovies.toArray(new Movie[allMovies.size()]);
      }
}
```

216

- 其中真正想要考察的是如何将MovieLister对象与特定的finder对象连接起来。
- 要求：我们希望moviesDirectedBy方法完全不依赖于影片的实际存储方式。
  - 这个方法只能引用一个finder对象，
  - 而finder对象必须知道如何实现findAll的细节。

- 给finder定义一个接口：

```
public interface MovieFinder
{
    List findAll();
}
```

- 当要实际寻找影片时，就必须涉及到MovieFinder 的某个具体子类。在这里，我把“涉及具体子类”的代码放在MovieLister 类的构造函数中。

217

对抗变化

- 从一个逗号分隔的文本文件中获得影片列表。
- class MovieLister...

```
private MovieFinder finder;
public MovieLister() {
    finder = new ColonDelimitedMovieFinder("movies1.txt");
}
```

- 对抗变化：
  - 文件清单的名字更改：
    - 可以从一个配置文件获得文件名。
  - 如果用SQL 数据库、XML 文件、web service，或者另一种格式的文本文件来存储影片清单：
    - 用另一个具体的类来获取数据。该类从MovieFinder接口派生即可。
  - 创建合适的MovieFinder派生类的实例：
    - 不能对抗此变化。

218

客户代码

```
public void testWithPico()
{
    MutablePicoContainer pico = configureContainer();
    MovieLister lister = (MovieLister)
            pico.getComponentInstance(MovieLister.class);

    Movie[] movies = lister.moviesDirectedBy("Sergio  Leone");

    assertEquals("Once Upon a Time in the West",  movies[0].getTitle());
}
```

219

Spring 注入

- 为了让MovieLister 类接受注入，需要为它定义一个设值方法，它接受类型为MovieFinder的参数：

```
class MovieLister...
  private MovieFinder finder;
  public void setFinder(MovieFinder finder) {
        this.finder = finder;
  }
```

- 类似地在MovieFinder的实现类中，定义一个设值方法，接受类型为String 的参数：

```
class ColonMovieFinder...
  public void setFilename(String filename) {
      this.filename = filename;
  }
```

220

配置文件

- 设定配置文件：XML 文件是比较理想的配置方式。

```
<beans>
<bean id="MovieLister" class="spring.MovieLister">
<property name="finder">
<ref local="MovieFinder"/>
</property>
</bean>

<bean id="MovieFinder" class="spring.ColonMovieFinder">
<property name="filename">
<value>movies1.txt</value>
</property>
</bean>
</beans>
```

221

第三单元： UML与软件设计思想

222

设计模式

什么是设计模式

- Christopher Alexander：
  - “每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次一次地使用该方案而不必重复劳动”。
- 设计模式：
  - 模式是一种问题的解决思路，它已经适用与一个实践环境，并且可以适应于其它的环境。

设计模式在实际开发中

- 复用现有的、高质量的、针对常见的重复出现问题的解决方案。
- 建立通用的术语以改善团队内部的沟通。
- 将思考转移到更高的视角。
- 判断是否拥有正确的设计，而不仅仅使一个可以运行的设计。
- 改善代码的可修改性。
- 发现“庞大的继承体系”的替代方案。

225

模式的分类

- 设计模式分为三个类型：创建型、结构型和行为设计模式。
- 创建型设计模式描述了实例化对象的相关技术，解决了与创建对象有关的问题。
- 结构型设计模式描述了在软件系统中组织类和对象的常用方法，避免了一个类被赋予过多职责而破坏类的封装性和信息的隐藏，和类之间功能重叠的问题。
- 行为设计模式负责分配对象的职责，为对象间协作建模提供了有效的策略。

226

GoF中的模式分类

		目 的		
		创 建 型	结 构 型	行 为 型
范围	类	Factory Method(3.3)	Adapter(类)(4.1)	Interpreter(5.3) Template Method(5.10)
	对象	Abstract Factory(3.1) Builder(3.2) Prototype(3.4) Singleton(3.5)	Adapter(对象)(4.1) Bridge(4.2) Composite(4.3) Decorator(4.4) Facade(4.5) Flyweight(4.6) Proxy(4.7)	Chain of Responsibility(5.1) Command(5.2) Iterator(5.4) Mediator(5.5) Memento(5.6) Observer(5.7) State(5.8) Strategy(5.9) Visitor(5.10)

227

设计模式的特点

- 设计模式最根本的意图是适应需求变化
  - 隔离变化的部分与不变的部分，将之封装起来。
- 针对接口编程，而不要针对实现编程
- 达成高内聚合低耦合，提高复用
- 提倡优先使用聚合，而不是继承

228

无穷尽变化的需求

- 软件设计最大的敌人：应付需求无穷尽的变化。
  - 项目开发就在反复的修改、更新中无限期地延迟交付的日期。
  - 没有 “银弹”：要彻底将变化扼杀在摇篮之中，是不现实的。
  - 那么，积极地面对“变化”，方才是可取的态度。
- “拥抱变化”
  - 从软件工程方法的角度。
  - 从软件设计方法的角度：封装变化。

229

例

- 一个日志记录工具。目前需要提供一个日志API，提供客户方便地调用。
- 该日志要求被记录到指定的文本文件中，记录的内容属于字符串类型，其值由客户提供。

230

实现

- 可以容易地定义一个日志对象。

```
public class Log
{
    public void Write(string target, string log)
    {
        //实现内容;
    }
}
```

当客户需要调用日志的功能时，可以创建日志对象，完成日志的记录：

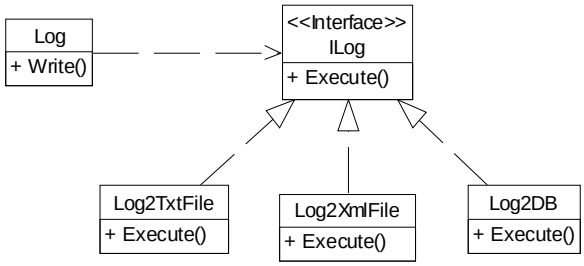
```
Log log = new Log();
log.Write("error.log", "log");
```

231

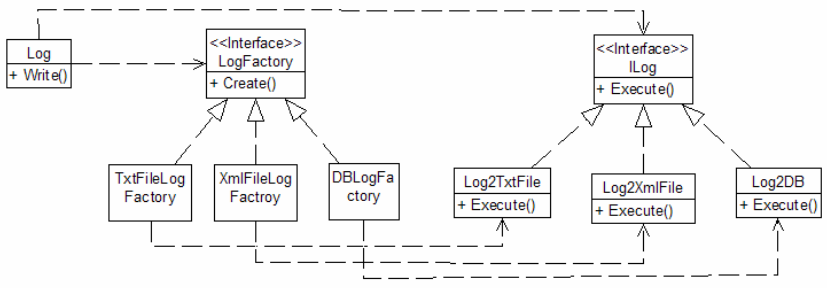
问题

- 需求变化：
- 随着日志记录的频繁使用，有关日志的文件越来越多，日志的查询与管理也变得越不方便。此时，客户提出，需要改变日志的记录方式，将日志内容写入到指定的数据表中。
- 显然如果仍然按照前面的设计，具有较大的局限性。

232


Log
+ Write()
<<Interface>>
ILog
+ Execute()
Log2TxtFile
+ Execute()
Log2XmlFile
+ Execute()
Log2DB
+ Execute()

233


Log
+ Write()
<<Interface>>
LogFactory
+ Create()
<<Interface>>
ILog
+ Execute()
TxtFileLog
Factory
XmlFileLog
Factroy
DBLogFa
ctory
Log2TxtFile
+ Execute()
Log2XmlFile
+ Execute()
Log2DB
+ Execute()

234

例

- 我们需要设计一个数据库组件，它能够访问Sql Server数据库。如果用ADO.Net，需要使用如下的对象：
- SqlConnection, SqlCommand, SqlDataAdapter等。

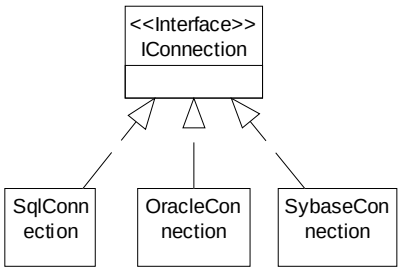
235

- 不用模式的做法：可以直接创建这些对象：
- SqlConnection connection = new

  SqlConnection(strConnection);
- SqlCommand command = new SqlCommand(connection);
- SqlDataAdapter adapter = new SqlDataAdapter();

236

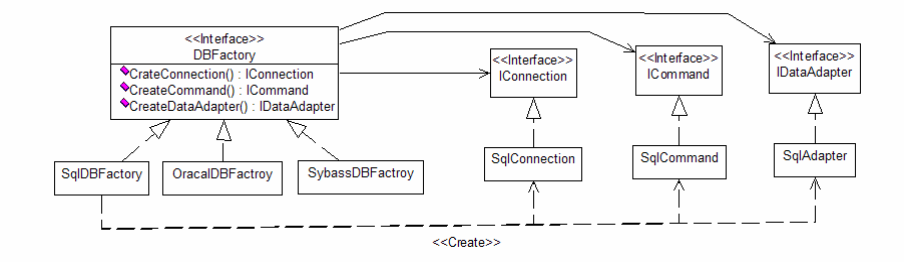
- **Connection对象：**



237

- 统一的IConnection接口，支持各种数据库的Connection对象都实现了IConnection接口。
- 现在，可以利用多态的原理创建对象：
  IConnection connection = new SqlConnection(strConnection);
- 从这个结构可以看到，根据访问的数据库的不同，对象的创建可能会发生变化。也就是说，我们需要设计的数据库组件，以现在的结构来看，仍然存在无法应对对象创建发生变化的问题。

238



239

- 这样的方式提高了数据库组件的可扩展性。
- 我们将可能发生变化的部分封装起来，放到程序固定的部分：
  - 初始化部分。
  - 全局变量（单件模式）。
  - 配置文件中
- 通过读取配置文件的值，创建相对应的对象。不需要修改代码，也不需要重新编译，仅仅是修改xml文件，就能实现数据库类型的改变。

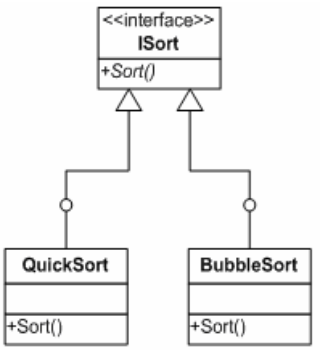
240

例

- 需要为自己的框架提供一个负责排序的组件。
- 目前需要实现的是冒泡排序算法和快速排序算法，根据“面向接口编程”的思想，我们可以为这些排序算法提供一个统一的接口ISort，在这个接口中有一个方法Sort()，它能接受一个object数组参数。对数组进行排序后，返回该数组。
- 接口的定义：

```
public interface ISort
{
        void Sort(ref object[] beSorted);
}
```

241

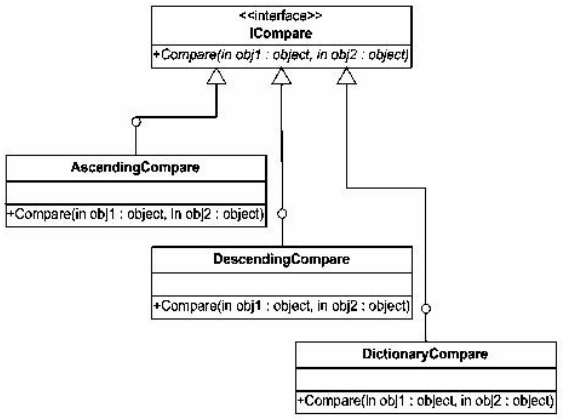


242

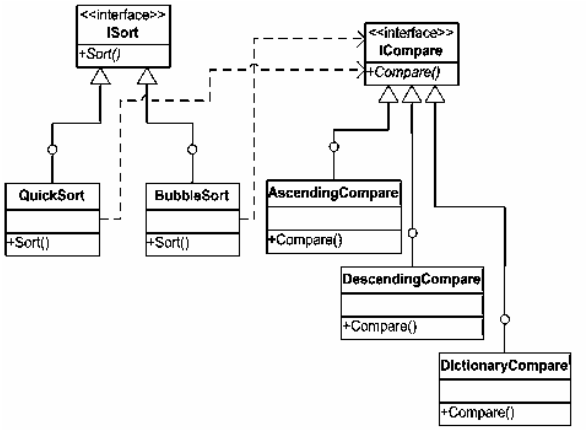
- 然而一般对于排序而言，排列是有顺序之分的，例如升序，或者降序，返回的结果也不相同。最简单的方法可利用if语句来实现这一目的，例如在QuickSort类中：

```
public class QuickSort:ISort
{
    private string m_SortType;
    public QuickSort(string sortType)
    {
        m_SortType = sortType;
    }
    public void Sort(ref object[] beSorted)
    {
        if (m_SortType.ToUpper().Trim() == “ASCENDING”)
        {
            //执行升序的快速排序;
        }
        else
        {
            //执行降序的快速排序;
        }
    }
}
```

243



244



245

万变不离其宗

- 学习设计模式的重点是学习每种模式的思想，而不应拘泥于它的某种具体结构图和实现。因为模式是灵活的，其实现可以是千变万化的，只是所谓万变不离其宗。

246

常用的软件架构风格及适用情况分析

康凯

247

几种典型的架构模式

- 系统软件
  - 分层(Layer)
  - 管道和过滤器(Pipes and Filters)
  - 黑板(Blackboard)
- 开发分布式软件
  - 经纪人(Broker)
  - 客户/服务器（Client/Server）
  - 点对点（Peer to Peer）
- 交互软件
  - 模型-视图-控制器（Model-View-Controller）
  - 显示-抽象-控制（Presentation-Abstraction-COntrol）

248

其它

- 面向对象风格(ADT)
- 基于消息广播且面向图形用户界面的Chiron2风格
- 基于事件的隐式调用风格(Event-based, Implicit Invocation)
- …

249

分层(Layer)

- 从不同的层次来观察系统，处理不同层次问题的对象被封装到不同的层中。
- 软件为什么要分层？
  为了实现“高内聚、低耦合”。把问题划分开来各个解决，易于控制，易于延展，易于分配资源…
- 面向对象的、基于模块化的组件设计需要能够方便地修改应用程序的各个部分。完成这一目标的一种好方法就是在层上工作，将一个应用程序的主要功能分离到不同的层或者级中。

250

管道和过滤器（Pipes and Filters）

- 管道和过滤器架构模式是为处理数据流的系统提供的一种模式。它是由过滤器和管道组成的.每个处理步骤都被封装在一个过滤器组件中，数据通过相邻过滤器之间的管道进行传输。每个过滤器可以单独修改，功能单一，并且它们之间的顺序可以进行配置。

251

黑板(Blackboard)

- 又称看板模式：在这种架构中，有两种不同的构件：一种是表示当前状态中心数据结构；另一种是一种相互独立的构件，这些构件对中心数据进行操作。这种架构主要用于数据库和人工智能系统的开发。
- 模式识别、数据挖掘。

252

经纪人(Broker)

- 客户和服务器通过一个经纪人部件进行通信，经纪人负责协调客户和服务器之间的操作，并且为客户和服务器发送请求和结果信息。

253

客户/服务器（Client/Server）

- 系统分为客户和服务器，服务器一直处于侦听的状态，客户主动连接服务器，每个服务器可以为多个客户服务。

254

点对点（Peer to Peer）

- 系统中的节点都处于平等的地位，每个节点都可以连接其他节点。在这种架构中，一般需要由一个中心服务器完成发现和管理节点的操作。ICQ以及Web Service技术的大多数应用，都是典型的点对点结构。

255

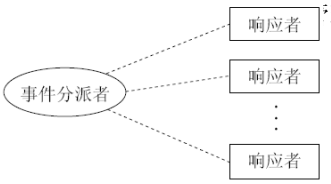
模型-视图-控制器（MVC）

- 当应用程序的用户界面非常复杂，且关于用户界面的需求很容易变化时，我们可以把交互类型的软件抽象成模型、视图和控制器这三类组件单元，这种抽象可以很好地分离用户界面和业务逻辑，适应变化的需求。大多数现代交互软件都在一定程度上符合这一架构模型的特点。
- **MVC模式最吸引人之处在于它迫使用户必须抽象自己的代码，把项目分为表示、逻辑和控制三部分,每部分间的关联较小。**
- 以MVC模式构造软件,可以使得软件结构灵活、重用性好、扩展性佳。

256

Event-based风格

- 优点：
  - 支持复用，构件通过登记它所感兴趣的事件被引入系统；
  - 便于系统演化，构件可以容易地升级或更换。
- 缺点：
  - 系统行为难以控制，发出事件的构件放弃了对系统的控制，因此不能确定系统中有无或有多少其它构件对该事件感兴趣，系统的行为不能依赖于特定的处理顺序。
  - 同事件关联的会有少量的数据，但有些情况下需要通过共享区传递数据，这时[illegible]理问题。



257

SOA 及分层架构设计

258

SOA的架构的特点

- 服务（Service）
  - 定义良好的，自包含的，不依赖于上下文和其它服务的一组功能
- SOA（Service-Oriented Architecture）
  - 本质上是一组服务的集合
  - 服务之间相互沟通
    - 可以是简单的数据传输，或者是由两个或多个服务共同参与的一些活动，SOA也包括Service之间的连通技术。

259

- OO vs. SOA
  - –OO的扩展遇到了挑战
    - 随着时间的推移，接口继承的复杂度在累积
    - 随着系统间距离的延伸，调用成本在上升，类型系统的不同步
    - 扩展组件的功能成本高，不可确定未来需求，不可堆叠的扩展方式
    - 重用与标准化，重用是OO的第一原则，难以维持和维护复杂的重用标准和机制

260

- –OO vs. SOA
  - – OO仍然适用于服务的开发
    - 明显的性能优势
    - 成熟的设计与开发方法
  - – SOA适用于系统的互联
    - 互操作性的要求强于性能的要求

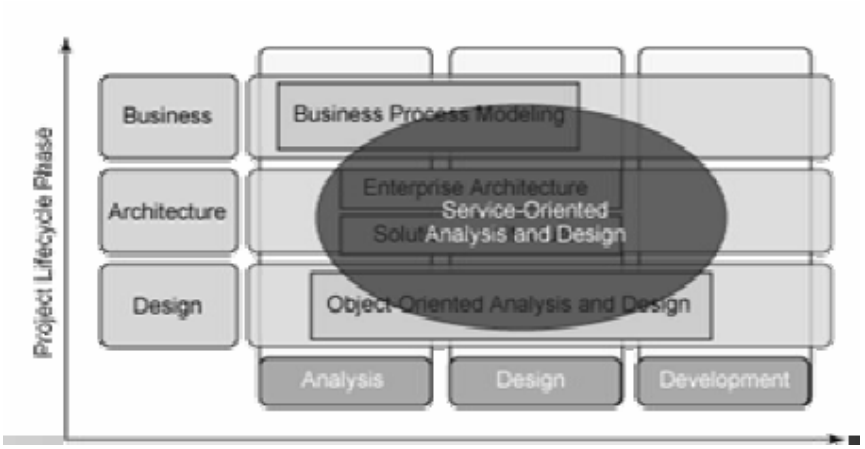
261

- SOA 既不是一种语言，也不是一种具体的技术，它是一种新的软件系统架构模型。 SOA 最主要的应用场合在于解决在Internet环境下的不同商业应用之间的业务集成问题。
- SOA 架构的出现为企业系统架构提供了更加灵活的构建方式，如果企业架构设计师基于 SOA 来构建系统架构，就可以从底层架构的级别来保证整个系统的松耦合性以及灵活性，这都为未来企业业务逻辑的扩展打好了基础。

262

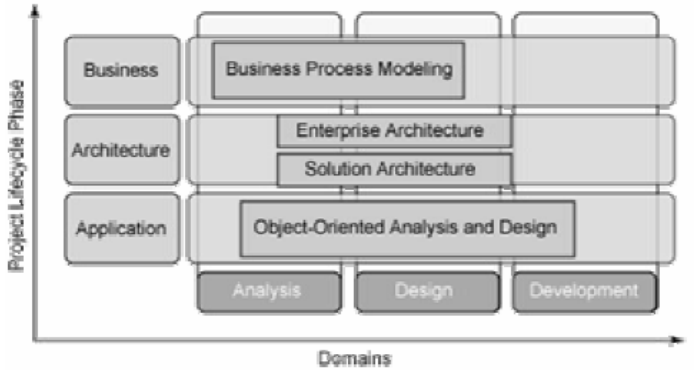
- 特性：
  - 松耦合性
    - 松耦合性要求 SOA 架构中的不同服务之间应该保持一种松耦合的关系，也就是应该保持一种相对独立无依赖的关系；
  - 位置透明性
    - 位置透明性要求 SOA 系统中的所有服务对于他们的调用者来说都是位置透明的，也就是说每个服务的调用者只需要知道他们调用的是哪一个服务，但并不需要知道所调用服务的物理位置在哪里；
  - 协议无关性。
    - 协议无关性要求每一个服务都可以通过不同的协议来调用。

263



264

- BPM、EA 和OOAD



265

OOAD

- OO 支持应用程序分析、设计和开发的完整生命周期
- SOAD 应该尽可能多地利用OO 分析技术
  - 将OO 分析成功地应用于SOA 项目
    - 必须一次分析多个系统，用例模型必须继续扮演重要的角色
    - SOAD 主要是流程，而不是用户驱动的。SOAD 需要BPM 和用例建模活动之间的强链接
- OO设计的目标是能够进行快速而有效的设计、开发以及执行灵活且可扩展的应用程序
  - 从OO 的角度看，每件事情都是对象。

266

- 与SO 有关的OO 设计的主要问题
  - OOAD
    - 粒度级别集中在类级
    - 对于业务服务建模来说，这样的抽象级别过低
    - 诸如继承这样的强关联产生了相关方之间一定程度的紧耦合（因而具有依赖性）
  - SOA
    - 试图通过松耦合来促进灵活性和敏捷性
    - 在SOA 中还没有服务实例的跨平台继承支持和表示法来避免需要处理服务生命周期维护管理问题（如远程垃圾收集）

267

SOAD 服务定义层次

- SOAD服务
  - 可能包括许多协作或编排服务
  - 并没有排除RUP 采用的OO 观点，而是在其上实现了另一个抽象层。
    - 其作用是封装作为正式的跨层接口结构中的软件服务的组件

268

- SOAD的基本要求：
  - 必须正式（至少半正式）地定义流程和表示法
    - 通过选择和组合OOAD、BPM 和EA
  - 必须有结构化的方法来概念化服务：
    - OOAD提供了应用程序层上的类和对象，而BPM 具有事件驱动的流程模型
      - SOAD 需要将它们结合在一起
    - 方法不再是面向用例的，而是由业务事件和流程驱动的
      - – 用例建模是在更低的层次上作为第二步进行的
    - 方法包括语法、语义和策略
      - 需要特别的组合、语义代理和运行时发现

269

- 质量因素的考虑
  - 构思良好的服务给业务带来了灵活性和敏捷性
  - 通过松散耦合、封装和信息隐藏使重构更加容易
  - 设计良好的服务是有意义的，并且不只适用于企业应用程序
  - 服务之间的依赖性减到最少，并且是显式声明的
  - 服务抽象是内聚、完整和一致的
    - 例如，在设计服务及其操作签名时应该考虑其CRUD
  - 通常假定服务是无状态的
  - 领域专家无需深奥的专业知识就可以理解服务命名
  - 在SOA 中，所有的服务都遵循相同的设计体系（通过模式和模板连接的）和交互模式；底层架构模式容易识别
  - 服务和服务使用者的开发除了领域知识之外只需要基本的编程语言技能；中间件专业知识只有少数的专业人员才需要

270

第五单元：架构设计实践

271

- 面向对象设计的步骤：
  - 一、静态设计。
  - 二、模块间的通信及耦合设计。
  - 三、动态设计。
  - 四、模块调整。

272

一、静态设计

- 1、按层+高内聚低耦合的原则进行模块划分
  - 1）高内聚低耦合原则(GRASP：高内聚低耦合)
  - 2）按功能分解
  - 3）按业务进行分解
  - 4）以数据转换为中心分解
  - 5）实际运用中的折中

273

一、静态设计（续）

- 2、划分层次 （架构风格）
  - 1）将模块划入对应的层
  - 2）分层与分区
  - 3）逻辑模块与实体组件的对应关系

274

一、静态设计（续）

- 3、为模块进行职责分配
  - 1）信息专家+控制者
  - 2）隔离关注面（GRASP：保护变量、间接模式）
  - 3）低耦合原则
  - 4）适当采用设计模式

275

一、静态设计（续）

- 4、用设计模式优化核心结构
- 1）经典模式运用：
  - 用策略/桥接模式作为中心骨架（多态模式）
  - 用工厂/抽象工厂模式进行组装。（创建者模式）
  - 用命令模式处理事务

276

一、静态设计（续）

- 5、模块结构的常用形式
- 1) 容器模块 + 控制者 + 功能模块 + 临时构建的小类。（纯虚构模式）
  - a) 单例模式
  - b) 命令模式
- 2) 核心模块的接口设计
  - a) 外观模式
  - b) 适配器模式
  - c) 代理模式
  - d) 调停者模式
- 3) 变换型模块结构
- 4) 事务型模块结构

277

二、模块间的通信及耦合设计

- 1、组件式程序设计
  - 通常一个领域的专用资产要应用到不相关的领域是比较困难的，组件式开发的首要工作是领域工程，在这个领域内提取可被复用的系统对象，创建可复用资产，开发复用组件。
  - 面向对象技术的特点是通过对象之间的职责分工和高度协作来完成任务。这样的好　处是代码量较少,系统布局合理,重用程度高,但是当对象的个数大量增加的时　候,对象之间的高度耦合的关系将会使得系统变得复杂,难以理解.

278

- 基于插件模型的系统，具有以下特点：
  - 所有业务功能均是“砖块”，更新及维护简便。
  - 子系统的概念变得模糊，子系统是多个模块的组合。
  - 有效避免功能的重复开发。
  - 可以根据用户需求定制系统，真正的随需而变。
  - 当功能插件达到一定规模时，应用系统的开发将变成二次开发。
  - 实现化整为零的组织结构，每个模块由微小的团队负责，甚至一个人负责。
  - 开发能力要求降低，类似于报表、查询等易变、简单的模块，维护人员即可完成。

279

二、模块间的通信及耦合设计（续）

- 2、通讯机制
  - 1）观察者模式
  - 2）本地SDK
  - 3）轮询

280

二、模块间的通信及耦合设计（续）

- 3、解耦
  - 1）针对接口编程；（面向对象原则：接口单一原则）
  - 2）增加间接模块；（间接模式）
  - 3）依赖注入

281

二、模块间的通信及耦合设计（续）

- 4、设计数据层
  - 1）数据结构选用的设计
  - 2）数据访问层的设计

282

三、动态设计

- 1、抽象与统一不同的因素
  - 1）根据业务寻找关键因素
  - 2）向复杂的情况靠齐
- 2、常用的流程抽象手段
  - 1）依赖注入 / 控制反转 / 依赖倒置（面向对象原则）
  - 2）表格法
  - 3）配置文件

283

三、动态设计（续）

- 3、逻辑控制：
  - 1）控制者模式
  - 2）信息专家模式
- 4、消息通知机制
  - 1）MVC模式
  - 2）观察者模式
  - 3）责任链模式
  - 4）中介者模式

284

四、模块调整

- 1、调整模块等级
  - 1) 适当封装；
  - 2) 把属性提升为类
  - 3) 将类降为属性
  - 4) 将类提升为组件

285

四、模块调整（续）

- 2、类细化
  - 1) 李氏替换原则
  - 2) 避免脆弱基类等
- 3、用设计模式优化设计
  - 1) 在主体的框架上进行调整
    - a)访问者模式
    - b)装饰模式

286

四、模块调整（续）

- 4、编码时构建适当的动态临时类
  - 1) 命令模式
  - 2) 事务处理类型
  - 3) 纯虚构

287

四、模块调整（续）

- 5、效率的优化
  - 1) 效率与结构的折中
  - 2) 优化效率的3步骤

288