是否理解迭代开发和UP：P28 是否理解初始阶段：P39 是否理解细化阶段：P96

第一部分 绪论

第一章 面向对象分析和设计

iterative development: 迭代开发 OOA/D: object-oriented analyse / design，面向对象分析/设计

aglie : 敏捷 UP: unified process，统一过程

OOD的原则和模式

1. responsibility-driven design(职责驱动设计) 九项基本原则(GRASP)

分析与设计

分析(analysis): 强调的是对问题和需求的调查研究，而不是解决方案 设计(desgin): 强调的是满足需求的概念上的解决方案（在软件和硬件方面），而不是其实现

面向对象分析和设计

面向对象分析：强调的是在问题领域内发现和描述对象（或概念） 面向对象设计：强调的是定义软件对象以及它们如何协作以实现需求

简单示例

1. 定义用例
2. 定义领域模型：面向对象分析的结果可以表示为领域模型，在领域模型中展示重要的领域概念或对象
3. 定义交互图
    1. 顺序图
4. 定义设计类图

UML

定义: unified modelling language，是描述、构造和文档化的可视化语言

应用方式：

1. 作为草图：非正式、不完整的图
2. 作为蓝图：详细的设计图
3. 作为编程语言：用UML完成软件系统可执行规格说明

应用UML的三种透视图

1. 概念透视图：用图来描述现实世界或关注领域中的事物
2. 规格说明(软件)透视图：用图（使用与概念透视图中相同的表示法）来描述软件的抽象物或具有规格说明和接口的软件，但并不约定特定实现（例如，非特定为C#或Java中的类）
3. 实现(软件)透视图：用图来描述特定技术（例如Java）中的软件实现

三种透视图对应的类

1. 概念类(conceptional class)：现实世界中的概念或事物。
2. 软件类(software class)
3. 实现类(implementatiojn class): 特定OO语言（如Java）中的类

第2章 迭代、进化和敏捷

软件开发过程(software development process): 描述了构造、部署以及维护软件的方式。

瀑布模型

瀑布模型：在编程之前定义所有或大部分需求，创建出完整的设计，定义计划和时间表。

瀑布、迭代和进化式开发的区别：

1. 后者对部分系统及早地引入了编程和测试，并重复这一循环
2. 后者依赖短时快速的开发步骤、反馈和改写不断明确需求和设计。

迭代开发

迭代: 开发被组织成一系列固定的短期（如三个星期）小项目。

特点：

- 每次迭代都产生经过测试、集成并可执行的局部系统。
- 每次迭代都具有各自的需求分析、设计、实现和测试活动。
- 迭代输出的不是实验性的或将丢弃的原型，迭代开发也不是构造原型，输出的是最终产品的子集。
- 时间定量（timeboxed）（时长固定）：迭代的一个关键思想，下一次迭代的时间必须依照时间表来集成，若没能满足要求，建议删除一些任务，而不是推迟时间。

迭代开发的优点：中文课本P17

迭代计划：风险驱动（以架构为中心）和用户驱动相结合

敏捷开发

敏捷开发（agile development）：通常应用时间定量的迭代和进化式开发、使用自适应计划、提倡增量交付并包含其他提倡敏捷性（快速灵活的响应变更）的价值和实践。

敏捷原则：P21

敏捷建模

- 敏捷方法并不意味着不建模
- 主要目的是为了理解，而非文档
- 不要对大多数软件设计建模或应用UML
- 尽可能使用简单的工具：如白板，摄像机拍摄
- 不要单独建模：因为建模的目的是发现、理解和共享大家的理解
- 使用简单的表示法：方便大家一起理解
- 模型可能是错误的，这只是一次探索
- 开发者应该为自己进行OO设计建模，而不是创建模型后交给其他编程者去实现

统一过程

统一过程(UP): 一种流行的构造面向对象系统的迭代软件开发过程

UP的核心思想：短时间定量迭代、进化和可适应性开发

UP的阶段：

1. 初始(Inception)：大体上的构想、业务案例、范围和模糊评估。
2. 细化(Elaboration)：已精化的构想、核心架构的迭代实现、高风险的解决、确定大多数需求和范围以及进行更为实际的评估。
3. 构造(construction): 对遗留下来的风险较低和比较简单的元素进行迭代实现，准备部署。
4. 移交(Transition): 进行beta测试和部署。

UP科目

科目（discipline）：在一个主题域中的一组相关活动

科目类别：业务建模(bussiness modeling)、需求(requirements)、设计(design)、实现(implementation)、测试(test)、部署(deployment)、配置和变更管理(configuration and change management)、项目管理(project management)、环境(environment)

业务建模：领域模型制品，使应用领域中的重要概念的可视化 需求：用以捕获功能需求和非功能性需求的用例模型及其规格说明制品 设计：设计模型制品，用于对软件对象进行设计 实现：编程和构建系统 环境：建立工具并为项目制定过程

迭代过程中科目的占比变化图

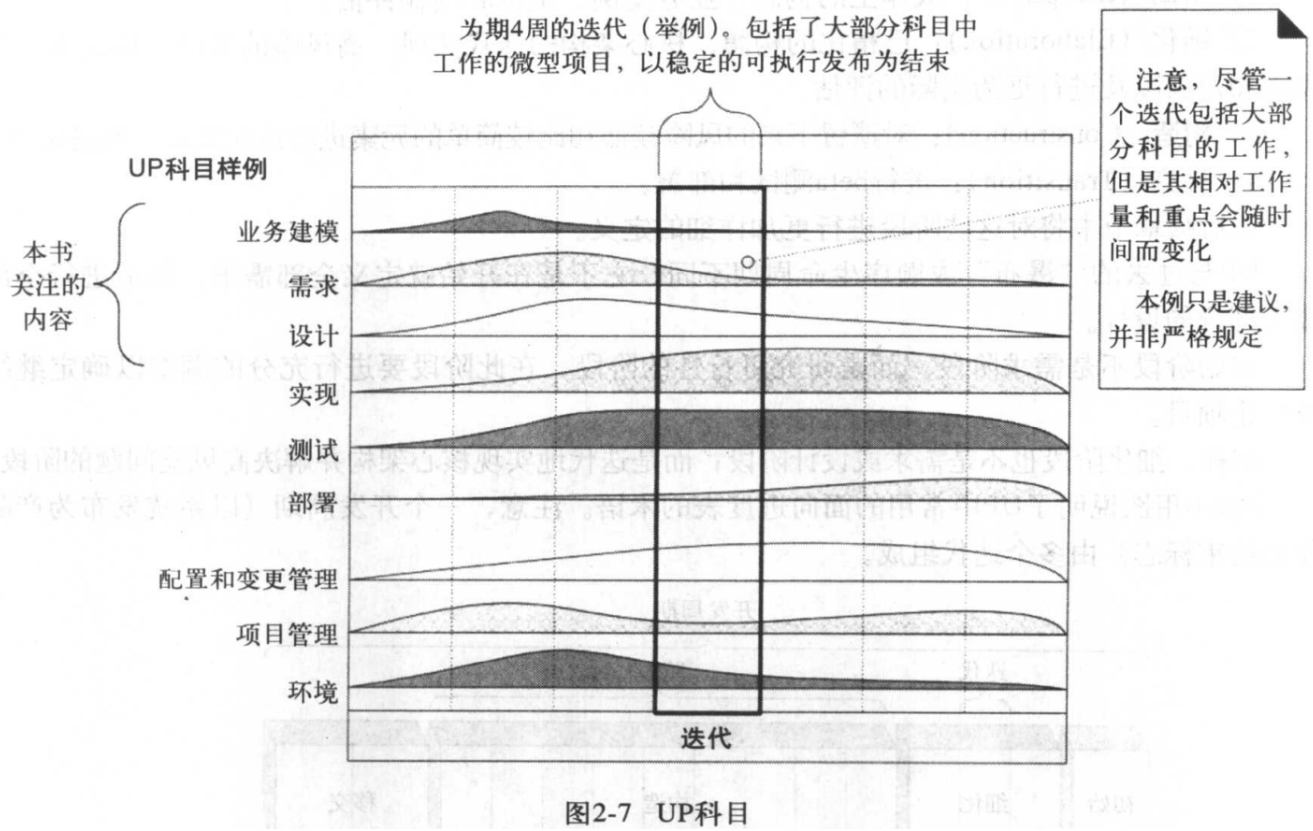
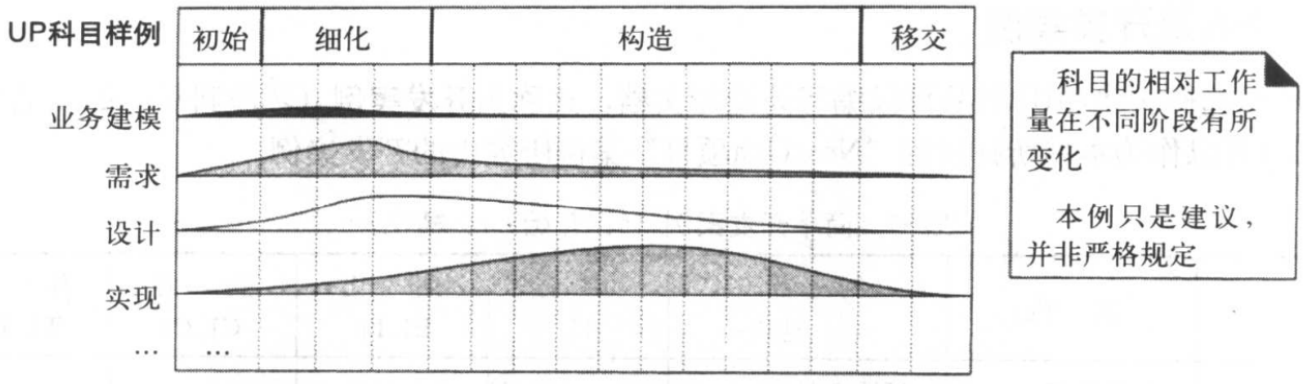


图2-7 UP科目



制品

制品（artifact）：所有工作产品的统称，如代码、Web图形、数据库模型、文本文档、图等。所有制品都是可选的，除了代码

开发案例

开发案例：为项目选择实践和UP可以编写为简短文档。属于环境科目的制品

科目	实践	制品	初始	细化	构造	移交
业务建模	敏捷建模、需求讨论会	领域模型		s		
需求	需求讨论会、设计包装练习、计点投票表决	用例模型、设想、补充性规格说明、词汇表	s	r		
设计	敏捷建模、测试驱动开发	设计模型、软件架构文档、数据模型				
实现	测试驱动开发、结对编程、持续集成、编码标准					
项目管理	敏捷项目管理、Scrum每日例会					
…						

第二部分 初始阶段

第四章 初始阶段不是需求阶段

初始阶段的迭代次数一般只有一两次，时间短 初始阶段的预算和计划是不可靠的，它只是提供了对工作量的粗略估计。 初始阶段的制品：P38 初始阶段的文档可以是初始、概略地文档，这些文档将在细化迭代中精化 初始阶段不是需求阶段，而是研究可行性的阶段（确定项目是否值得深入研究，实质的调查活动发生在细化阶段）。

第五章 进化式需求

需求：系统（更广义的说法是项目）必须提供的能力和必须遵从的条件

进化式需求特点：不断变更，寻找

寻找需求的方法：

1. 与客户一起编写用例
2. 开发者与客户共同参加需求研讨会
3. 请客户代理参加焦点小组
4. 向客户演示每次迭代的成果以求得反馈

需求的分类（按照FURPS+）：P42

1. 功能性(Functional)：
2. 可用性(Usability)：
3. 可靠性(Reliability):
4. 性能(Performance)：
5. 可支持性(Supprtability):
6. +:

- 实现(Implementation)
- 接口(Interface)
- 操作(Operation)
- 包装(Packaging)
- 授权(Legal)

需求的制品:P43

第6章 用例(use case)

用例简介

用例：文本形式的情节描述。通过地讲，就是一组相关地成功和失败场景集合，用以说明某参与者使用系统以实现某些目标。 注：用例不是面向对象的

用例：属于需求科目的制品，在分类上属于FURPS+的F

用例模型（需求科目的制品）：可以包括UML的用例图

用例的表示形式

1. 摘要：通常应用于主成功场景。使用原因：快速
2. 非正式：用几个段落覆盖了不同的场景。使用原因：快速
3. 详述(full dressed)：使用时间：在编写了大量用例后

参与者(actor)：某些具有行为的事物，在所讨论的系统（System under Discussion, SuD）调用其它系统的服务时（还包括自身），可以是人（角色）、计算机系统或组织。

- 主要参与者(primary actor)：具有用户目标，并通过使用SuD的服务完成
- 协助参与者(supporting actor)：为SuD提供服务
- 幕后参与者(offstage actor)：在用例行为中具有影响或利益，但不是主要或协助参与者

场景(scenario)(用例实例)：参与者与系统之间的一些列特定的活动与交互，是使用系统的一个特定的情节或用例的一条执行路径。

Alistair Cookburn （人名）的详细用例模板的介绍

绪言元素

1. 范围
   - 系统用例(system use case)：对一个软件系统的使用
   - 业务用例(bussiness use case)：描述业务人员如何使用业务
2. 级别
   - 用户目标级别（业务流程的基本业务流程）：描述了实现主要参与者目标的场景
   - 子功能级别：将若干常规用例共享重复的步骤分离出来
3. 主要参与者
4. 涉众及其关注点列表 用例应该包含什么？用例应该包含满足所有涉众关注点的事物。
5. 前置条件(precondition)和成功保证
   - 前置条件：给出在用例场景开始之前必须永远为真的条件
   - 成功保证（后置条件）：给出用例成功结束后必须为真的事物。

主成功场景（理想路径，基本流程，典型流程）

准则：将所有条件和分支延迟到扩展部分进行说明

扩展

扩展部分描述了其他所有场景或分支，包括成功和失败路径。通常占据了文本的大部分篇幅

特殊需求

技术和数据变元表

准则

1. 以无用户界面约束的本质风格编写用例
2. 编写简洁的用例
3. 编写黑盒子用例 规定系统必须做什么，而不决定系统如何去做
4. 采用参与者和参与者目标的视点
5. 如何发现用例：详细见课本P63
    1. 选择系统边界：可以通过定义外部参与者来使系统边界变得清晰
    2. 3. 寻找主要参与者和目标 准则：首先集体讨论主要参与者，因为这样可以为将来的研究建立框架
    3. 定义用例：一般来说，为每个用户目标分别定义用例。
6. 什么样的测试有助于发现有用的用例
    1. 老板测试
    2. EBP(Elementary Businesses Procss，基本业务过程)测试
    3. 规模测试

应用UML

准则：

1. 绘制简单的用例图，并与参与者-目标列表关联
2. 制图
3. 不要倚重于制图，保持其简短

各个阶段的用例编写

1. 初始阶段 几个小时编写20个用例名称，以详述形式重新编写其中的10%-20%的用例，这些用例代表复杂的核心功能、需要构建核心架构或者在某些方面极具风险
2. 细化阶段 详述形式编写80%-90%的用例
3. 构造阶段 可能还要编写一些重要的用例，但次数大大小于细化阶段

第7章 其他需求

制品的准则：

1. 初始阶段不应该对制品彻底进行分析
2. 制品应该放在项目Web站点上

设想的准则：

1. 设想文档中尽量避免重复的其他的需求
2. 设想和用例编写顺序没有强制规定

词汇表的准则：

1. 在词汇表中记录组合术语（组合术语是针对原子术语而言的）

第三部分 细化迭代1——基础

第8章 迭代1——基础

细化：构建核心架构，解决高风险元素，定义大部分需求，以及预计总体进度和资源。

细化阶段的内容：

1. 对核心、有风险的软件架构进行编程和测试
2. 发现并稳定需求的主体部分
3. 规避主要风险

特点：

- 细化阶段通常由两个或多个迭代组成，建议每次迭代的时间为2-6周
- 细化阶段不是设计阶段，在该阶段不是要完成所有模型的开发。这一阶段不是要创建可以丢弃的原型，该阶段的代码和设计是具有产品品质的最终系统的一部分。
- 细化阶段是迭代地实现核心架构并解决高风险问题的阶段。
- 进行大多数的需求分析，并且伴以具有产品性质的早期编程和测试。

细化阶段的一些关键思想和最佳实践：P95

第9章 领域模型

领域模型（概念模型，领域对象模型，分析对象模型）：对领域内的概念类或现实世界中对象的可视化表示（不是描述软件类、软件架构领域层或有职责软件对象的一组图）。属于业务建模科目的制品。

领域模型的表示：

1. UML—— 概念透视图
    1. 领域对象或概念类
    2. 概念类之间的关联
    3. 概念类的属性
2. 纯文本形式—— 可视化字典
    - 表示领域的重要抽象、领域词汇和领域的内容信息

区分

1. 领域模型不是软件业务对象 所以它不能表示软件制品、职责或方法
2. 领域模型不是数据模型（通过数据模型的定义来表示储存于某处的持久性数据，如数据库语言）

概念类；是思想、事物或对象

- 符号

- 内涵
- 外延

为什么要创建领域模型：降低OO建模之间的表示差异，即减小我们的思维与软件模型之间的表示差异（领域层软件类的名称要源自于领域模型中的名称）。

如何创建领域模型

1. 寻找概念类
    1. 重用和修改现有的模型 - 首要、最佳最简单的
    2. 使用分区列表
    3. 确定名词短语 - 语言分析(linguistic analysis) 准则：使用这种方法必须小心。不可能存在名词到类的映射机制，并且自然语言中的词语具有二义性。
2. 将其绘制为UML类图中的类
3. 添加相关和属性

准则

1. 敏捷建模 - 绘制类的草图
2. 敏捷建模 - 是否要使用工具维护模型 作者建议使用白板等实体工具绘制草图，用数码摄像机记录。如有需要更新模型，再使用现代软件工具重新绘制草图。
3. 像地图绘制者一样思考，使用领域术语
4. 属性与类的错误：如果我们认为某概念类X不是现实世界中的数字或文本，那么X可能是概念类而不是属性。
5. 何时使用描述类(description class)的使则：
    - 需要有关商品或服务的描述
    - 删除其所描述事务的实例后，导致信息丢失，而这些信息是需要维护
    - 减少冗余或重复信息

关联

UML的关联：两个或多个类元之间的语义联系，涉及这些类元实例之间的连接

准则

1. 何时表示关联 领域模型是从概念出发的，所以是否需要记录的关联，是基于现实世界的需要，而不是软件的需要。
2. 避免大量的关联 连线太多会产生”视觉干扰”，使图变得混乱
3. 领域模型需要的关联
    1. 如果存在需要保持一段时间的关系，将这种语义表示为关联("需要记住"的关联)
    2. 从常见关联列表中派生的关联

关联是否会在软件中实现：关联不是软件的关联，而是针对现实领域从概念角度看的关系，大部分的关联会在软件中加以实现。

应用UML

1. 关联表示法
    - 准则：关联命名为“类名-动词短语-类名”(首字母大写)

2. 角色表示法
    - 定义：关联的每一端称为角色
3. 多重性(multiplicity)表示法
    - 作用：定义了一个类的实例可以与多少另一个类的实例关联
    - 图形表示：P113 - P114
4. 两个类之间的多重关联：P114

属性

属性：属性是对象的逻辑数据值

准则：何时展示属性 - 当需求（例如，用例）建议或暗示需要记住信息时，引入属性

UML属性表示法：`visibility name: type mulitiplicity = default{property-name}`，例如`+ pi: Real = 3.14{readOnly}`，导出属性(derived attribute)在`name`前面加`\`

准则：

1. 领域模型中的属性的类型更应该是简单的数据类型（不一定是编程语言的基本数据类型）
2. 通过关联而不是属性来表示概念类之间的关系
3. 任何属性都不表示外键（企图用属性来表示关系）
4. 对数量和单位进行建模（P122） 一般情况下，可以把数量表示为单独的Quantity类，并且关联到Unit类。通常还可以对Quantity加以规格说明。

迭代和进化式领域建模

准则：每次迭代的领域建模时间不超过几个小时

UP中的领域建模

1. 初始 初始阶段绝不会发起领域建模
2. 细化 领域模型主要在细化阶段的迭代中，这时时最要理解那些重要概念的，并且会通过设计工作将其映射为软件类。

第10章 系统顺序图(System Sequence Diagram，SSD)

系统顺序图：为阐述与所讨论系统相关的输入和输出事件而快速、简单地创建的制品。表示的是对于用例的一个特定场景，外部参与者产生的事件，其顺序和系统之内的事件。

软件系统需要响应的三类事件：

1. 来自参与者（人或计算机）的外部事件，即系统事件
2. 时间事件
3. 错误或异常（通常源于外部）

与其他制品的关系：

1. 它是操作契约和（最重要的）对象设计的输入
2. 用例文本及其所表示的系统事件是创建SSD的输入
3. SSD的操作可以在操作契约中进行分析，在词汇表中被详细描述，并且（最重要的是）作为设计协作对象的起点。

准则：

1. 为每个用例的主成功场景，以及频繁发生的或者复杂的替代场景绘制**SSD**。
2. 对大多数制品来说，一般在词汇表中描述其细节。

迭代和进化式SSD

1. 初始 - 通过不会在该阶段引入SSD
2. 细化 - 大部分的SSD在细化阶段创建，这有利于识别系统事件的细节以便明确系统被设计和处理的操作，有利于编写系统操作契约，并且有利于对估算的支持（例如，通过未调整的功能点和COCOMO2进行宏观估算）

第11章 操作契约

操作契约：操作契约使用前置和后置条件的形式，描述领域模型里对象的详细变化，并作为系统操作的结果。

与其他制品的关系：

1. 操作契约的主要输入是SSD中确定的系统操作、领域模型和领域专家的见解。
2. 契约也可以作为对象设计的输入，因为它们描述的变化很可能是软件对象或数据库所需要的。

操作契约的构成

1. 操作：操作的名称和参数
2. 交叉引用：会发生此操作的用例
3. 前置条件：执行操作之前，对系统或领域模型对象状态的重要假设。
4. 后置条件：最重要的部分。完成操作后，领域模型对象的状态。领域模型的状态变化包括实例的创建与删除、形成或消除关联以及改变属性。

准则：在思考操作契约有新发现时，要对领域模型进行改进。

如何创建和编写契约：P139

UP的操作契约

1. 初始 - 初始阶段不会引入契约
2. 细化 - 如果使用契约的话，大部分契约将在细化阶段进行，这时已经编写了大部分用例，只对最复杂和微妙的系统操作编写契约

第13章 逻辑架构(logical architecture)和UML包图

逻辑架构（logical architecture）：是软件类的宏观组织结构，它将软件类组织为包（或命名为空间）、子系统和层等。

层(layer)：是对类、包或子系统的甚为粗粒度的分组，具有对系统主要方面加以内聚的职责。

- 用户界面(UI)
- 应用逻辑和领域模型(domain)
- 技术服务(technical services)

UML的包图：

- 依赖线是有箭头的虚线，箭头指向被依赖的包
- 具体内容：P148

使用层进行设计

使用层的好处：P149

准则：内聚职责，使关系分离。同一层内的对象在职责上应该具有紧密关联，不同层中对象的职责则不应该混淆。

领域对象：表示问题领域空间的事物，并且与应用或业务逻辑相关。

领域层与领域模型之间的关系：领域层是软件的一部分，领域模型是概念角度分析的一部分，他们是不同的。但是利用来自领域模型的灵感创建领域层，我们可以获得在现实世界和软件设计之间的低表示差异（low represetation gap, LRG）。

准则：不要将外部资源或服务表示为最底层 将外部资源(如某个数据库)表示为“低于”（例如）基础层的层，是对逻辑视图和架构试图部署视图的混淆

模型 - 视图分离原则

1. 不要将非UI对象直接与UI对象连接或耦合
2. 不要在UI对象方法中加入应用逻辑

分离的动机：P154

第14章 迈向对象设计

开发者如何设计对象

1. 编码：边编码边设计
2. 绘图，然后编码（本章的介绍内容，最为流行的方式）
3. 只绘图，不编码

编码前绘制UML需要花费多少时间：时间定量为三周的迭代，花费几个小时或至多一天的时间。

设计对象：

1. 静态建模：UML类图
    - 有助于设计包名、类名、属性和方法特征标记（但不是方法体）的定义
2. 动态建模：UML交互图（顺序图或通信图）
    - 有助于设计逻辑、代码行或方法体
    - 倾向于创建更为有益、困难和重要的图形

类职责协作(CRC, Class-Responsibility-Collaborator)卡：对象设计技术，面向文本的建模技术

第15章 UML交互图

UML使用交互图(Interaction diagram)来描述对象间通过消息的交互。

交互图(interaction diagram)有两种类型：顺序图(sequence diagram)和通信图(communication diagram)

顺序图与通信图的比较：P163

准则：应该花时间使用交互图进行动态设计，而不仅仅是使用类图进行静态对象建模。

UML交互图的表示方法：P164-P179

第16章 UML类图

(详细的表示方法见课本)

UML类图：表示类、接口及其关联

设计类图(Design Class Diagram, DCD)：软件透视图的类图、设计透视图的类图

常用UML类图表示法：P181

领域模型与软件透视图在关联表示上的区别

- 领域模型的类图的关联没有箭头，而软件透视图的类图有

准则：对数据类型对象使用属性文本表示法，对其他对象使用关联线。（两者的语义是等价的，但是在图中展示与另一个类框的关联线能够在视觉线上强调图中对象的类之间的连接）

第17章 GRASP：基于职责设计对象

GRASP，全称为General Responsibility Assignment Software Pattern，即通用职责分配软件模式

职责驱动设计(responsibility-driven design, RDD)

对象的职责分类：

- 行为
  - 自身执行的一些行为，如创建对象或计算
  - 初始化其他对象中的动作
  - 控制和协调其他对象的活动
- 认知
  - 对私有对象的认知
  - 对相关对象的认知
  - 对其能够导出或计算的事物的认知

信息专家(Information Expert)

问题：给对象分配职责的基本原则是什么

解决方案：将职责分配给具有完成该职责所必需信息的类

禁忌：在某些情况下，专家模式建议的方案并不适合，这是由于耦合和内聚的问题产生的。（课本举了数据库存储的问题）

优点：

- 因为对象使用自身信息来完成任务，所以信息的封装性得以维持，支持低耦合
- 行为分布在那些具有所需信息的类之间，因此提倡定义内聚性更强的“轻量级”的类，这样易于理解和维护。（高内聚）

创造者 (Creator)

问题：谁应该负责产生类的实例？

解决方案：如果符合下面的一个或者多个条件，则可将创建类A实例的职责分配给类B：

1. B“包含”或组成聚集了A；
2. B记录A的实例；
3. B频繁使用A；
4. B拥有初始化A的数据并在创建类A的实例时将数据传递给类A；

准则：封装的容器或记录器类是创建其所容纳或记录的事物的良好候选者。

禁忌：当对象的创建具有相当的复杂性（例如为了性能而使用回收的实例），最好的方法是把创建职责委派给具体工厂（Concrete Factory）或抽象工厂（Abstract Factory）

优点：低耦合

低耦合 (Low Coupling)

问题：如何减少因变化产生的影响

解决方案：分配职责以使（不必要的）耦合保持在较低的水平。使用该原则对可选方案进行评估。

禁忌：高耦合对于稳定和普遍使用的元素而言并不是问题。（例如，各种语言自带的官方库）

优点：

1. 不受其他构件变化的影响
2. 易于单独理解
3. 便于复用

控制器 (Controller)

问题：在UI层之上首先接收和协调（”控制“）系统操作的第一个对象是什么

解决方案：把职责分配给能代表下列选择之一的对象：

1. 代表全部“系统”、“根对象”、运行软件设备或主要的子系统（这些是外观控制器(facade controller)的所有变体）
2. 代表发生系统操作的用例场景（用例或会话控制器(session controller)）

本质：委派(delegation)模式

可能出现的问题：控制器分配的职责过多，造成不良（低）内聚。

- 表现：
    - 一个控制器接受所有系统事件，例如外观控制器
    - 控制器有很多属性，并且它维护关于系统或领域的重要信息
- 解决：
    - 增加控制器
    - 设计控制器，使它把完成每个系统操作的职责委派给其他对象

优点：

1. 增加了可复用和接口可插拔的潜力
2. 获得了推测用例状态的机会

例子：JavaFx的listener(监听器)

准则：正常情况下，控制器应当把需要完成的工作委派给其他的对象。控制器只是协调或控制这些活动，本身并不完成大量工作

高内聚 (High Cohesion)

问题：怎样使对象保持内聚、可理解和可管理，同时具有支持低耦合的附加作用

解决方案：通过职责分配保持高内聚，以此来评估备选方案。

使用原因：内聚性较低的类要做许多互不相关的工作，或需要完成大量的工作。

禁忌：在少数情况下，可以接受低内聚（见P230例子）。

优点：

1. 能够更加轻松、清楚地理解设计。
2. 简化了维护和改进工作
3. 通常支持低耦合
4. 由于内聚类的类可以用于某个特定的目的，因此细粒度、相关性强的功能的重用性强

第18章 使用GRASP的对象设计示例

命令 - 查询分离原则(Command-Query Separation Principle, CQS)

原则：

1. 执行动作的命令方法，这种方法通常具有改变对象等副作用，并且返回值是void
2. 向调用者返回数据的查询，这种方法没有副作用，不会永久改变任何对象的状态。

动机：

1. 更容易推出程序的状态
2. 遵循最小意外原则(Least Surprise)原则

例子：

```
public void roll() {
    faceValue = //
}
public int getFaceValue() {
    return faceValue;
}
```

```
public int roll() {
    faceValue = //
    return faceValue;
}
```

代码2经常被使用，但是它违反了CQS原则。

迭代和进化式的对象设计

初始：不进行 细化：为关键用例实现绘制交互图 构造：为当前存在的设计问题构造用例实现

第19章 对可见性进行设计

可见性：一个对象看见其他对象或引用其他对象的能力。 实现对象A到对象B的可见性的方式：

1. 属性可见性(attribute visibility) - B是A的属性
2. 参数可见性(parameter visibility) - B是A方法中的参数

3. 局部可见性(local visibility) - B是A中方法的局部对象（不是参数）
4. 全局可见性(global visibility) - B具有某种方式的全局可见性（首选单例模式）

第20章 将设计映射为代码

将设计映射到面向对象开发的代码需要编写的元素：

1. 类和接口的定义
2. 方法的定义

从交互图创建方法

实现的顺序：类的实现要按照从耦合度最低到最高的顺序来完成

和：极限编程(XP)提倡的优秀时间，测试代码之前完成单元测试

第21章 测试驱动开发和重构

测试驱动(Test-Driven Development, TDD)[测试优先开发(test-first development)]：首先编写测试，然后再编写预计测试的代码

优点：

1. 能够保证编写单元测试
2. 使程序员获得满足感从而更始终如一地坚持编写测试
3. 有助于澄清接口和行为的细节
4. 可证明、可再现、自动的验证
5. 改变事物的信心

重构：重写或重新构建已有代码的结构化和规律性方法，但不会改变已有代码的外在行为，采用一系列少量转换的步骤，并且每一步都结合了重新执行的测试。

第四部分 细化迭代2 - 更多模式

第25 章 GRASP: 更多具有职责的对象

多态(Polymorphism)

问题：如何处理基于类型的选择？如何创建可插拔的软件架构？

解决方案：当相关选择或行为随类型（类）有所不同时，使用多态操作变化的行为类型分配职责。

准则：

1. 不要测试对象的类型，也不要使用条件逻辑来执行基于类型的不同选择
2. 除非在超类中具有默认的行为，否则将超类的多态方法发申明为{abstract}

优点：

1. 易于增加新变化所需的扩展
2. 无需影响客户便能引入新的实现

课本例子：如何支持第三方税金计算器

纯虚构(Pure Fabrication)

问题：当你并不想违背高内聚和低耦合或其他目标，但是基于专家模式所提供的方案又不合适时，哪些对象应该承担这一职责？

解决方案：对人为制作的类分配一组高内聚的职责，该类并不代表领域的概念——虚构的事物，用以支持高内聚，低耦合的复用。

例子：适配器(adapter)、策略(strategy)、命令(command)

优点：

1. 支持高内聚
2. 增加了潜在的复用性

课本例子：如何在数据库中保存对象，P306

间接性(Indirection)

问题：为了避免两个或多个事物之间直接耦合，应该如何分配职责？如何使对象解耦合，以支持低耦合并提高复用性潜力

解决方案：将职责分配给中介对象，使其作为其他构建或服务之间的媒介，以避免它们之间的直接耦合。

例子：适配器(adapter)、外观(facade)和观察者(observer)

优点：实现了构件之间的低耦合

格言：

- 计算机科学中的大多数问题都可以通过增加一层间接性来解决
- 大多数性能问题可以通过去除一层间接性来解决

防止变异(Protected Variations, PV)

问题：如何设计对象、子系统和系统，使其内部的变化或不稳定性不会对其他元素产生不良影响

解决方案：识别不稳定之处，分配职责用以在这些变化之外创建稳定接口。

例子：数据封装、接口、多态、间接性和标准、虚拟机、操作系统（目的都是为了信息隐藏）

优点：

1. 易于增加新变化所需的扩展
2. 可以引入新的实现而无需影响客户
3. 低耦合
4. 能够降低变化的影响成本

第26章 应用GoF(gang-of-four)设计模型

适配器(adapter)

问题：如何解决不相容的接口问题，或者如何为具有不同接口的类似构件提供稳定的接口 解决方案：通过中介适配器对象，将构件的原接口转为其他接口

工厂(factory)(简单工厂, simple factory)(具体工厂, concrete factory)

注：该模式不是GoF的，是GoF的抽象工厂(Abstract Factory)的简化

问题：当有特殊考虑（例如存在复杂创建逻辑，为了改良内聚而分离创建职责等）时，应该由谁来负责创建对象 解决方案：创建被称为工厂的纯虚构（纯虚构这个词表示这个事物在现实生活中找不到相应的概念）对象来处理这些创建职责

```
class ServicesFactory{
    private IAccountAdapter accountingAdaptar;
    private IInventorAdapter inventoryAdapter;
    public IAccountAdapter getAccountAdapter(){}
    public IInventroAdapter getInventorAdapter(){}
}
```

单例模式(singleton)

问题：只有唯一实例的类即为“单实例类”，对象需要全局可见性和单点访问。 解决方案：对类定义静态方法用以返回单实例。

```
class ServicesFactory{
    static ServicesFaactory instance;
    public static synchronized ServicesFactory getInstance(){
        if(instance == null)
            instance = new ServicesFactory();
        return instance;
    }
}
```

策略(strategy)

问题：如何设计变化但相关的算法或政策？如何设计才能使这些算法或政策具有可变更的能力？ 解决方法：在单独的类中分别定义每种算法/政策/策略，并且使其具有共同的接口。 课本例子：打折策略，策略的创建者应该是策略工厂

```
//打折策略
interface ISalePricingStrategy{
    Money getTotal(Sale);
}
//百分比打折
class PersentDicountPricingStrategy:implements  ISalePricingStrategy{
    public Money getTotal(Sale){}
}
//
class AbsoluteDiscountOverThresholdPricingStategy:implements  ISalePricingStrategy{
    public Money getTotal(Sale){}
}
```

组合(composite)

问题：如何能像处理非组合(原子)对象一样，（多态地）处理一组对象或具有组合结构的对象 解决方案：定义组合和原子类对象的类，使它们实现相同的接口 课本例子：多种打折策略和单个打折策略的并存

```
//表示一个组合各种打折的抽象类
abstract class CompositePricingStragegy implements ISalePricingStrategy{
    //添加各种打折的类，可以实现这个抽象类的类
    public void add(ISalePricingStrategy s){}
    public abstract Money getTotal(Sale);
}
class CompositeBestFOrCustomerPricingStrategy extends CompositePricingStragegy{

}
```

外观(facade)

问题：对一组完全不同的接口实现（例如子系统的实现和接口）需要公共、统一的接口。可能会与子系统内部大量事物产生耦合，或者子系统的实现可能改变。怎么办？ 解决方案：对子系统定义唯一的接触点——使用外观封装子系统。该外观对象提供了唯一统一的接口，并负责与子系统构件进行协作。 注意点：外观通常是通过使用单实例类进行访问的 课本例子：判断支付是否有效

观察者(observer)(发布-订阅, publish-subscribe)

问题：不同类型的订阅者对象对关注于发布对象的状态变化或事件，并且想要在发布者产生事件时以自己独特的方式做出反应。此外，发布者想要保持与订阅者的低耦合。如何对此进行设计呢？

解决方案：定义“订阅者”的“监听器”接口，订阅者实现此接口。发布者可以动态注册关注某时间的订阅者，并在事件发生时通知它们。

课本例子：GUI控件上的监听器

第28章 UML活动及其建模

活动图的基本元素： 1. 动作(action) 2. 分区(partition) 3. 分叉点(fork) 4. 连接点(join) 5. 对象节点(object node)

第29章 UML状态机(state machine diagram)图和建模

状态图显示了对象的生命周期：对象经历的事件、对象的转换和对象在这些事件之间的状态。状态图不必描述所有可能的的事件，如果所发生的事件未在图中表示，则说明其不影响该状态机所关注的内容。

基本元素：P353

1. 事件(event)
2. 状态(state)
3. 转换(transition)

状态无关(或非模态)：一个对象对于某件事件的响应总相同 状态无关对象：对于所有事件，对象的响应总是相同 状态依赖对象：对事件的响应根据对象的状态或模式而不同

准则：考虑为具有复杂行为的状态依赖对象而不是状态无关对象建立状态机图。

第30章 用例关联

准则：避免陷入用例关系的陷阱 - 重要的是编写用例

用例之间的关系：

1. 包含关系(include)
    - 用例在其他用例中重复使用
    - 用例非常复杂并冗长，将其分解为子单元便于理解
2. 扩展关系(extend)
3. 泛化关系

具体用例(concrete use case)：由参与者发起，完成了参与者所期望的完整行为。通常是基础业务过程用例 抽象用例(abstract use case)：永远不能被自己实例化，它是其他用例的子功能用例 基础用例(base use case)：包含其他用例，或者是被其他用例扩展或者泛化的用例 附加用例(addition use case)：被其他用例包含的用例，或者扩展、泛化其他用例的用例

第31章 领域模型的精华

超类：表示更为普通的概念 子类：表示更为特殊的概念

泛化(generalization)：是在多个概念中识别共性和定义超类与子类关系的活动

准则：

1. 识别领域中与本次迭代有关的超类和子类，并且在领域模型中阐明
2. 100%规则（定义一致性）：概念超类的定义必须100%适用于子类。子类必须100%与超类一致：属性和关联
3. Is-a：子类集合的所有成员必须是其超类集合的成员

概念类的划分：将一个概念类划分为几个不相交的子类

将概念类划分为子类的原因：P368 何时定义概念超类：P369

抽象概念类：类的每个成员也必须是某个子类的成员

准则：在领域模型中，如果类C可能同时具有多个相同的属性A，则不要将属性A置于C之中 。应该将属性A放在另一个类中，并将其与C关联。

准则：不要在UML中费心地去使用聚合(aggregation)，在适当的时候要使用组合。

组合(composition)(组成聚合，composite aggregation)

1. 在某一时刻，部分的一个实例只能属于一个组成实例
2. 部分必须总是属于组成
3. 组成要负责创建和删除其他部分——即自己或者与其他对象一起协作

何时使用组合：P376 组合的好处：P376

第33章 架构分析

架构分析：架构分析可以被视为需求分析的规格化，其强烈影响“架构”的需求 架构分析的本质是要识别影响架构的因素，理解这些因素的可变性和优先级，并且解决和些问题。

架构分析的重要性：P390

何时开始架构分析：在UP中，甚至早于第一次开发迭代时就应该开始架构分析，因为在早期开发工作中需要识别和解决架构问题。架构分析的失败会导致高风险。

架构分析的常用步骤：

1. 识别和分析对架构有影响的非功能性需求
2. 对于这些在架构方面有重要影响的需求，需求分析可供选择的方法并创建解决这些影响的解决方案（架构决策）。

关键架构需求(architecurally significant requirement)：具有架构意义的因素

架构分析方法：表（例如因素表，factor table）、树

架构设计原则：

1. 低耦合
2. 高内聚
3. 防止变异
4. 关注分离(separation of concern): 在架构级别上考虑低耦合、高内聚目标的大尺度方式
    1. 将有关事物模块话，封装到单独的构件（如子系统）中，并且调用其服务
    2. 使用装饰者
    3. 使用后编译器(post-compiler)和面向方面(aspect-oriented)技术

UP制品的架构信息

1. 初始阶段：不能确定技术上是否可以满足关键的架构需求，开发团队可以实现一个架构概念验证(architectual POC)。
2. 细化阶段：实现核心的风险架构元素，因而，大部分的架构分析都在细化阶段完成的
3. 移交阶段：在理想情况下，架构性因素和决策在移交阶段都被解决，但本阶段结束时人仍可能需要修订SAD以便确保与最终部署的系统一致
4. 后续进化循环：设计新版本之前，通常会重温架构性因素和决策

第35章 包的设计

组织包结构在的准则：

1. 包在水平和垂直划分上的功能性内聚：将参与共同目的、服务、协作、策略和功能的强相关类型组织在一起
2. 由一族接口组成的包：将一组功能上相关的接口放入单独的包，与其实现类分离
3. 用于正式工作的包和用于聚集不稳定类的包
4. 职责越多的包越需要稳定
5. 将不相关的类型分离出去
6. 使用工厂模式减少对具体包的依赖
7. 包之间没有循环依赖

第36章 使用GoF模式完成更多对象设计

本地服务容错(使用本地缓存提高性能)

缓存策略：

1. 惰性初始化(lazy initialization): 当需要的时候才加载
2. 立即初始化(eager initialization)：预加载

故障处理

1. 使用异常
2. 处理错误
    - 集中错误日志(centrallized error logging)
    - 错误会话(error dialog)

GoF - 通过代理(proxy)使用本地服务进行容错

问题：不希望或不可能直接访问真正的主题对象时，应该怎么办 解决方案：通过代理对象增加一层间接性，代理对象实现与主题对象相同的接口，并且负责控制和增强对主题对象的访问。 代理只不过是与被代理对象实现相同接口的对象，它保存指向被代理对象的引用，并且用于控制对被代理对象的访问。

GoF - 抽象工厂

问题：如何创建实现相同接口的一簇相关的类 解决方案：定义一个工厂接口（抽象工厂）。为每一族要创建的事物定义一个具体的工厂类，也可以定义实际的抽象类来实现工厂接口，并且为扩展该抽象类的具体工厂提供公共服务。 代码示例：P436

第37章 使用模式设计持久性框架

存储机制和持久性对象：对象数据库、关系数据库和其他机制

状态(state)模式 问题：对象的行为依赖于它的状态，而它的方法中包含能够反映依赖状态的条件动作的case逻辑。是否存在替代条件逻辑的方法 解决方法：给每个状态创建状态类，并实现一个公共的接口。将语境对象中的依赖于状态的操作委派给其当前的状态对象。确保语境对象总是指向反应其当前状态的状态对象。

命令(comand)模式 问题：如何处理需要诸如排序(优先级)、排队、延迟、记录日志或重做等功能的请求或任务 解决方案：为每个任务创建一个类，并实现共同的接口

使用虚代理(virtual proxy)实现滞后具体化

第39章 架构的文档化：UML和N+1视图模型

软件架构文档(software architecture document, SAD)：描述了有关架构的总体想法，包含架构分析和关键决策

4+1视图模型：逻辑视图、进程视图、部署视图和数据视图；用例视图