Software Engineering Final Exam

VectorPikachu

2024 年 12 月 31 日

1 软件工程概述

1. **软件的定义**：软件一般是指计算机中的**程序**及其**文档**。
2. 软件的发展历史：软硬一体化阶段 → 软件的产品化、产业化阶段（微软和甲骨文），软件的网络化、服务化阶段。
3. 软件的分类：
   (a) 按照软件的**功能**划分：系统软件（编译程序和操作系统）、支撑软件（软件开发环境以及后来的中间件，软件开发环境主要包括环境数据库、各种接口软件和工具组，MySQL，VS，VMWare，Git，Nginx）、应用软件（特定应用领域专用的软件）。
   (b) 按照软件的应用领域划分：
      i. 系统软件：一整套服务于其他程序的程序。
      ii. 应用软件：解决特定业务需求的独立应用程序。
      iii. 工程/科学软件：这类软件带有“数值计算”算法的特征，覆盖了广泛的应用领域。
      iv. 嵌入式软件：存在于某个产品或系统中，可实现和控制面向最终使用者和系统本身的特性和功能。
      v. 产品线软件：产品为多个不同用户的使用提供特定功能。
      vi. Web 应用软件：是一类以网络为中心的软件。
      vii. 人工智能软件：利用非数值算法解决计算和直接分析无法解决的复杂问题。
4. 软件开发的本质和基本手段：
   (a) 软件开发的**含义**：软件开发就是建立问题域到运行平台之间的映射。
   (b) 软件开发的**本质**：不同抽象层术语之间的“映射”，不同抽象层处理逻辑之间的“映射”。
   (c) 分层的基本动机是控制开发的复杂性，一个抽象层是由一组确定的术语定义的。
   (d) 软件工程 = 目标（正确、可用、合算）+ 原则（... 高质量...）+ 活动（需求、设计、实现、确认、支持）。
   (e) 软件工程的目标：生产具有正确性、可用性以及开销合宜的产品。
   (f) 软件工程的活动：生产一个最终满足需求且达到工程目标的软件产品所需要的步骤。

2 软件过程

软件生存周期：软件产品或系统的一系列相关活动的全周期。从形成概念开始，历经开发、交付使用、在使用中不断修订和演化，直到最后被淘汰，让位于新的软件产品。

软件生存周期过程（软件过程）：软件生存周期中的一系列相关过程。**过程**：活动的一个集合，**活动**：任务的一个集合，**任务**：把输入转换成输出的操作。

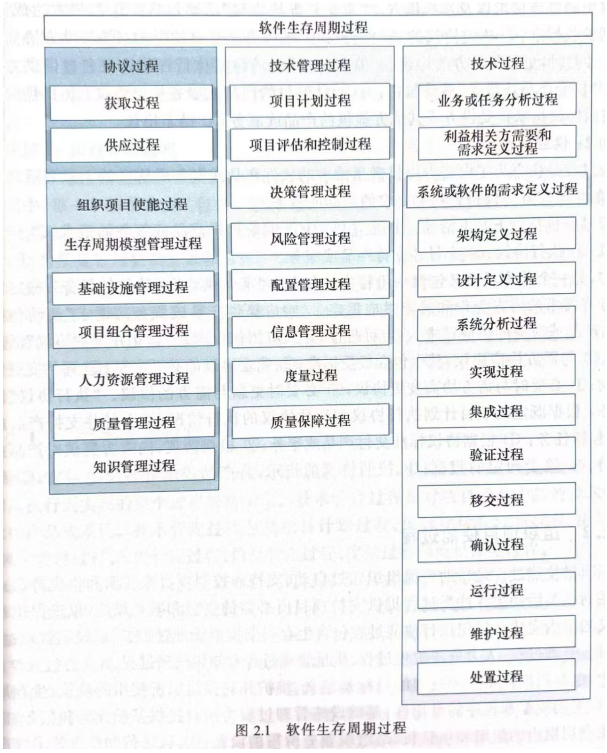
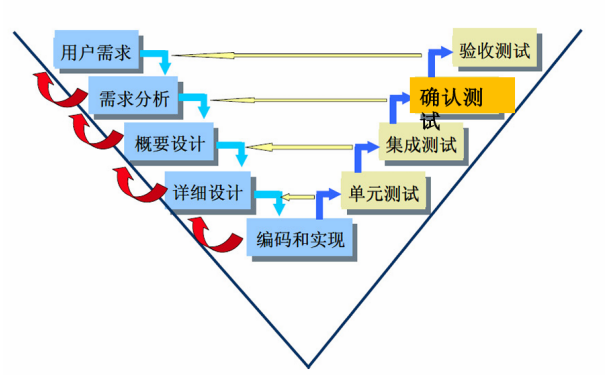


图 2.1 软件生存周期过程

软件生存周期模型：称软件生存周期模型为“软件开发模型”，并把它定义为：软件过程、活动、任务的结构框架。

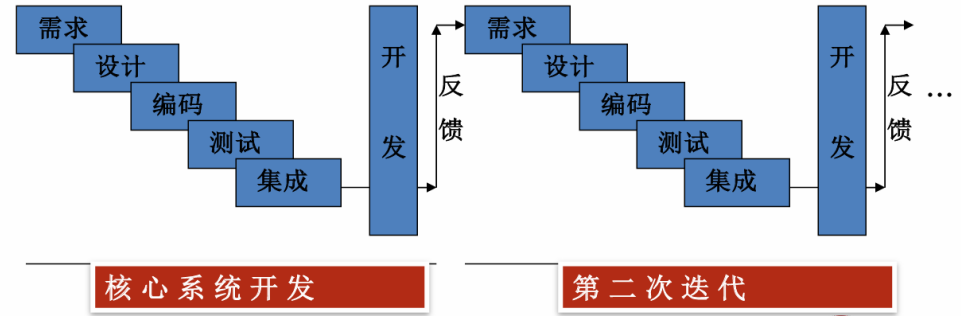
1. 瀑布模型（1970. W.Royce）：按顺序。“反向”步骤流表示对前一个可提交产品的重复变更（又称为“返工”(Rework)）。

2. V 模型（1978 年. Kevin Forsberg & Harold Mooz）：研发人员和测试人员需要同时工作，这样尽可能早地找出程序错误和需求偏离。



3. 增量模型：该模型有一个假设，即需求可以分段，成为一系列增量产品，每一增量可以分别地开发。它是瀑布模型的一个变体。需求的不稳定性。

4. 演化模型（Evolutionary model）：每一步都进行一次完整的开发迭代。实现不能完整的定义需求。



该模型显式地把增量模型扩展到需求阶段。由图可以看出，为了第二个构造增量，使用了第一个构造增量来精化需求。

5. 喷泉模型：软件活动需要多次重复。

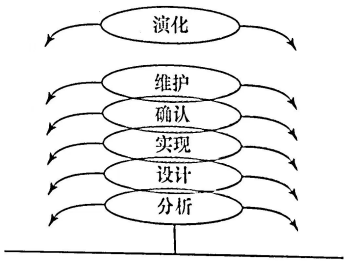
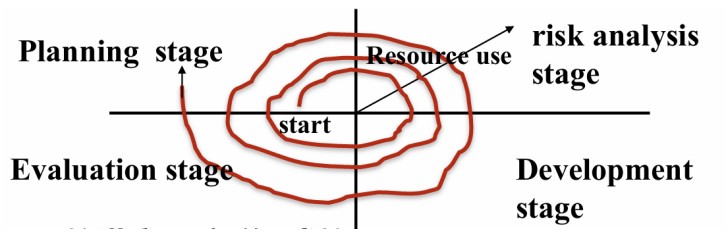
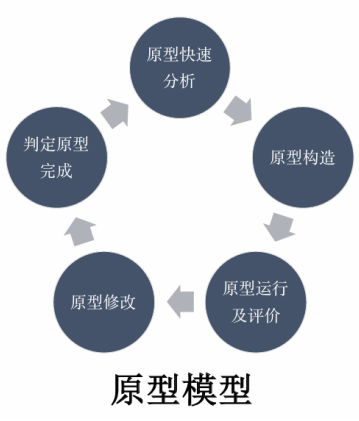


图 2.8 喷泉模型

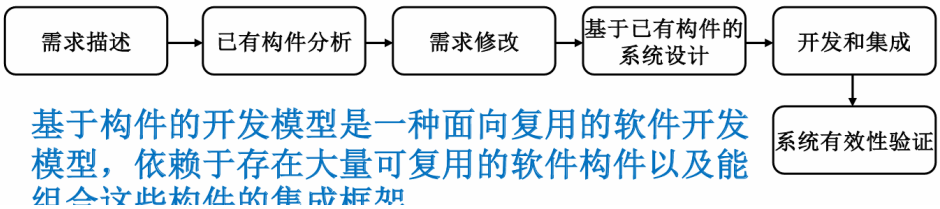
6. 螺旋模型（1988. Dr. Barry Boehm）：该模型将软件生存周期的活动分为四个可重复的阶段：规划、风险分析、开发和评估。



7. 原型模型：原型是快速建立起来的可以在计算机上运行的程序，它所能完成的功能往往是最终产品能完成的功能的一个子集。

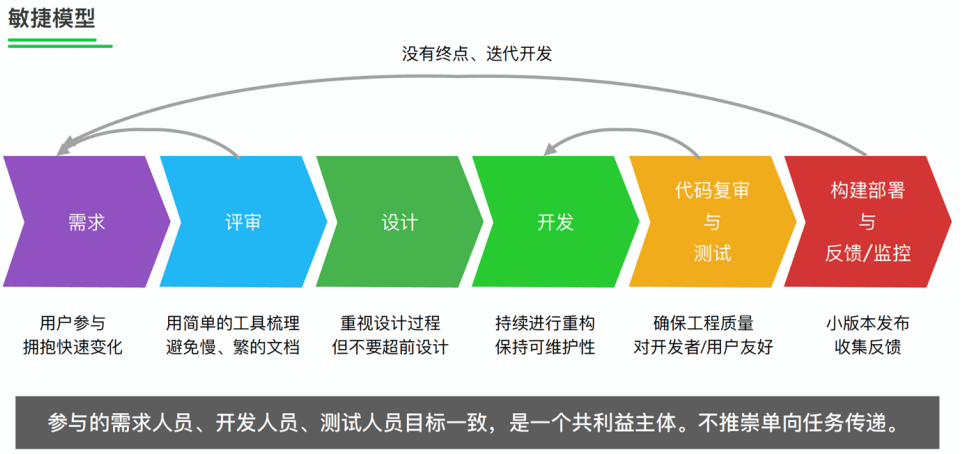


8. 基于构件的开发模型：面向复用的软件开发模型，依赖于存在大量可复用的软件构件以及能组合这些构件的集成框架。



需求妥协。

9. 敏捷模型。



软件过程改进：产品质量的三大要素：人员、技术（设备）、过程。

CMM 的软件过程成熟度框架：初始级、可重复级、已定义级、已管理级、持续优化级。

3 软件需求工程

软件需求：划分软件系统的边界，即确定软件做什么，不做什么。用户解决问题或实现目标所需的软件能力；为满足合同、标准、规范或其他正式强制性文件, 系统或系统组件必须满足或拥有的软件能力。

需求工程是开发软件需求的工程，利用系统化、工程化的方法和技术，指导软件工程师对软件需求进行捕获、分析、记录、验证和管理等工作，以高效开发出准确表达用户需求的软件需求规格说明书。

SRS: Software Requirements Specification.

需求工程的主要任务：完整地定义问题，确定系统的功能和能力。需求获取、需求分析、需求规约（需求文档化）、需求验证、需求管理，最终形成系统的软件需求规格说明书，其中主要成分是系统功能模型。

1. 需求获取的任务是给出软件系统的需求定义，即“软件需求的完整定义”。
2. 需求分析的任务是通过形式化或半形式化手段，建立系统模型，该模型正确地、系统地表达了系统的需求，即“建立完整的需求规约”。
3. 需求验证（需求评审）的任务是保证“需求规约”的正确性、完备性、无二义性、一致性、可验证性、可理解性、可修改性。
4. 需求管理。一般和软件配置管理技术结合。

需求发现技术：自悟、交谈、观察、小组会、提炼（来自技术文档）。

需求的分类：

1. 功能需求：功能需求规约了系统或系统构件必须执行的功能。功能需求是整个需求的主体，即没有功能需求，就没有非功能需求。
2. 性能需求：性能需求 (Performance requirement) 规约了一个系统或系统构件必须具有的性能特性。

   性能需求隐含了一些满足功能需求的设计方案，经常对设计产生一些关键的影响。例如：排序，关于花费时间的规约将确定哪种算法是可行的。
3. 外部接口需求：外部接口需求 (External interface requirement) 规约了系统或系统构件必须与之交互的硬件、软件或数据库元素。它也可能规约其格式、时间或其他因素。

   系统接口、用户接口、硬件接口、软件接口、通信接口、内存约束、操作、地点需求。
4. 设计约束：设计约束限制了系统或系统构件的设计方案。安全和保密属于设计约束。法规政策、并发操作等。

   设计约束需求必须予以满足。
5. 质量属性：质量属性 (Quality attribute) 规约了软件产品必须具有的一个性质是否达到质量方面一个所期望的水平。可靠性、可移植性、可维护性、用户友好型、安全性。

需求的基本性质：必要的、无歧义的、可测的 (Testable)、可跟踪的、可测量的 (Measurable)。

需求规约的作用：技术合同书、其余工作的管理控制点、设计的一个正式的受控的起点、创建产品验收测试计划和用户指南的基础（产生初始测试计划和用户系统操作描述）。

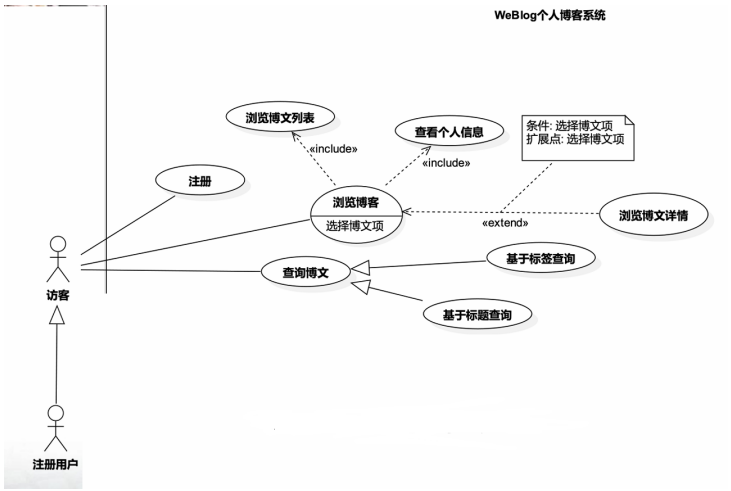
SRS 不能实现：不是设计文档，它是一个“为了”设计的文档；它不是进度或规划文档，不应该包含更适宜包含在工作陈述、软件项目管理计划、软件生存周期管理计划、软件配置管理计划或软件质量保证计划等文档中的信息。

项目的需求 vs 软件的需求：项目的需求是“开发组要做的是什么”，产品需求是“交付给客户的产品是什么”。

软件需求描述：用况图 + 用况故事。

用况图：包括主题、用况、参与者、关联、泛化、包含、扩展。

1. 主题：写在矩形的上方。“Weblog 个人博客系统”。
2. 用况：椭圆形。
3. 参与者：小人。
4. 关联：参与者参与一个用况。关联是参与者和用况之间的唯一关系。
5. 泛化：参与者之间存在泛化关系。用况之间也存在泛化关系。
6. 包含：用况包含其他用况。
7. 扩展：用况扩展其他用况。



用户故事：每个用户故事是一个系统用户可能经历的使用场景。

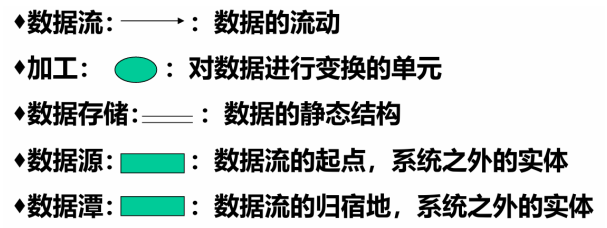
> 名称：注册用户通过WeBlog个人博客系统编辑并发布博文
>
> 用户故事描述：
> 1. 系统在主页提供一个“新建博文”的按钮；
> 2. 注册用户点击按钮；
> 3. 系统转入博文编辑界面，提示用户可以从标题、简介、标签、内容、博文权限五个方面创建新博文；
> 4. 注册用户填写标题、简介，添加标签并设置博文权限；
> 5. 系统以列表形式展现注册用户添加的标签；
> 6. 注册用户通过系统提供的编辑器，实时编写预览博文发布后的效果，或上传并插入图片；
> 7. 如果上述内容均已填写，页面的“提交博文”按钮就转为可点击状态；
> 8. 注册用户点击“提交博文”按钮；
> 9. 系统发布博文，并显示该博文的详情页面。

4 结构化分析方法

这里其实就是需求分析。

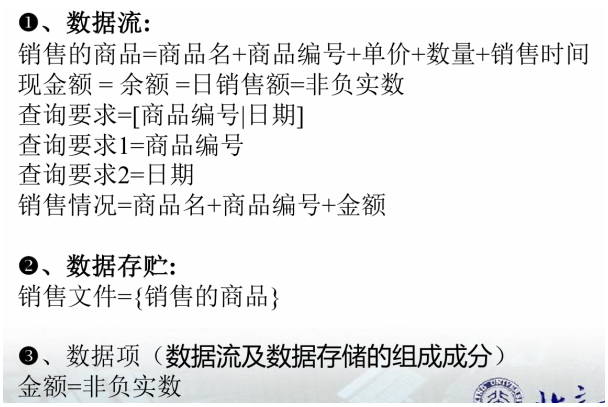
结构化分析的目标：对需求陈述进行分析，解决其中的歧义、不一致等问题，以系统化的形式表达用户的需求，即给出问题的形式化或半形式化的描述（称为系统的概念模型，或系统的需求规约或需求规格说明）。作为开发人员和客户间技术契约的基础，并作为而后开发活动的一个基本输入。

基本术语：数据流（支持数据抽象）、加工（支持过程/功能的抽象）、数据存储（支持数据抽象）、数据源和数据潭（支持系统边界抽象）。



模型表达的工具（后面的建模使用这些工具）：

1. 数据流图：DFD，表达系统功能模型的工具。
2. 数据字典：定义数据流和数据存储。数据流、数据存储、数据项。



3. 判定表或判定树等：定义加工小说明。结构化自然语言、判定表、判定树。

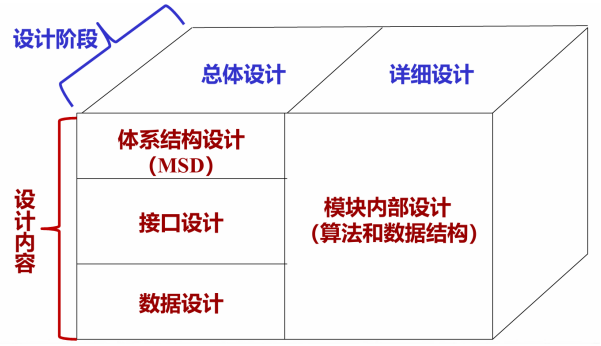
建模过程

1. 建立系统的功能模型。
   (a) 建立系统的环境图（顶层），包含唯一的一个大加工。
   (b) 自顶向下，逐步求精，建立系统的层次数据流图。顶层、0 层、1 层、...
2. 定义数据字典。
3. 描述加工。加工编号、输入流、输出流、加工逻辑。

结构化分析方法得到一个系统需求规格说明书。

5 结构化设计

软件设计目标：依据需求规约，在一个抽象层上建立系统软件模型，包括软件体系结构（数据和程序结构），以及详细的处理算法，产生设计规格说明书。



结构化设计包括 总体设计 和 详细设计 。

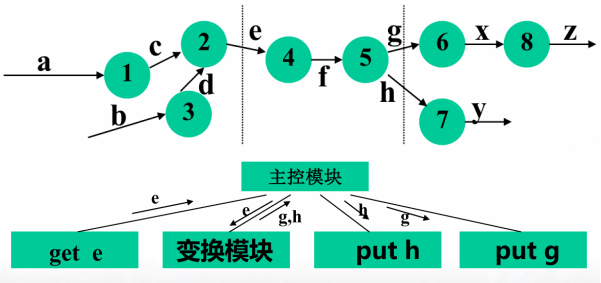
5.1 总体设计

总体设计：确定系统的整体模块结构，即系统实现所需要的软件模块以及这些模块之间的调用关系。其主要任务就是如何将一个系统的 DFD 转化为模块结构图（MSD），或者说把系统的功能需求分配给模块结构图。

这些模块就是一个**黑箱子**。

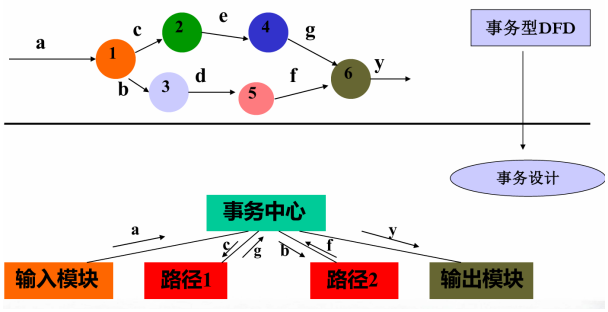
根据 DFD 的不同可以分为：

1. 变换型数据流图：输入部分 + 变换（主加工）+ 输出。（物理输入 vs 逻辑输入，物理输出 vs 逻辑输出）



接着把 输入模块和输出模块分解 。

2. 事务型数据流图：输入到达加工 T，加工 T 选择一条路径走，“集中-发散”。输出一般是一个。



然后把其中的模块继续分解。

模块化：高内聚，低耦合。 如果模块之间存在耦合，尽量使用数据耦合，少用控制耦合，限制公共耦合的范围，坚决避免使用内容耦合。

把上面得到的初始的 MSD 精化。

接下来设计 接口。

1. 内部接口设计。
2. 目标软件与其他软硬件系统之间的接口设计。
3. 用户界面设计。

数据设计：

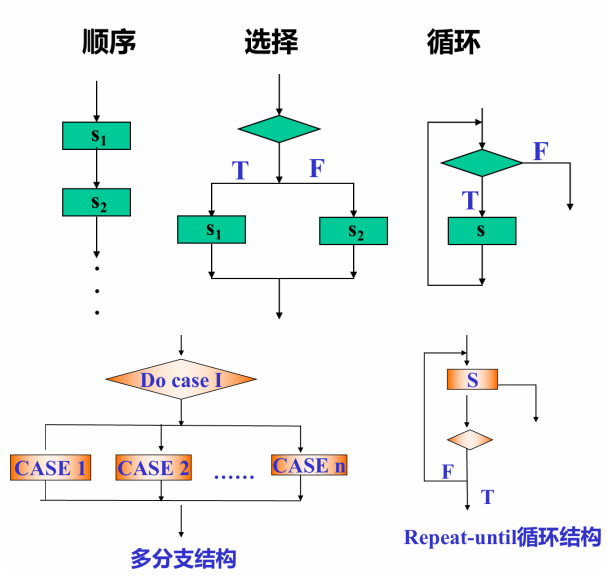
什么时候选择文件存储？非结构化数据、大量数据、大文件、存储数据分散、存储数据共享、存取速度要求高、临时存放数据。

关系型数据库用于管理大量的结构化数据。

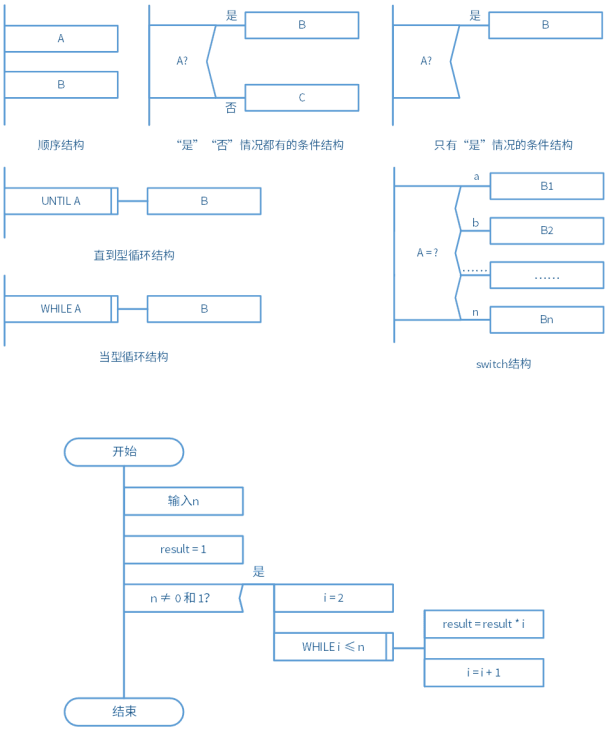
5.2 详细设计

详细设计：其主要任务是定义模块，即给出实现模块功能的实现机制，包括算法和数据结构。

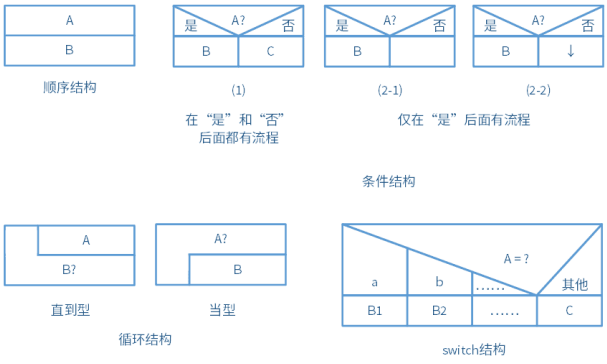
1. 伪码：顺序、选择、循环。不够直观。
2. 程序流程图。不是一种逐步求精的工具．所表达的控制流，可以不受约束随意转移．不易表示数据结构．

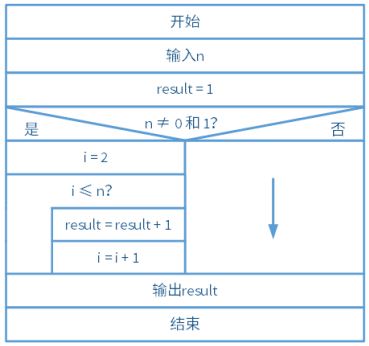


3. PAD 图（问题分析图）。支持自顶向下逐步求精的结构化详细设计。



4. N-S 图（盒图）：支持自顶向下逐步求精的结构化详细设计，并且严格限制了控制从一个处理到另一个处理的转移。





5. 判定表和判定树。

6 面向对象方法

面向对象技术引论

1 第一章 面向对象方法概论

1.1 1.1 基本思想

1. **对象**：客观存在的事物→对象，对象作为基本构成单位。
2. **属性与操作**：属性表示**性质**，操作表示**行为**。
3. **对象的封装**：对象的属性与操作结合为一体，成为一个独立的、不可分的实体，对外屏蔽其内部细节。公开静态的、不变的操作，而把动态的、易变的操作隐藏起来。
4. **分类**：相同属性和相同操作分为**一类**。
5. **聚合**：复杂的对象可以用简单的对象作为其构成部分。
6. **继承**：特殊类继承一般类。
7. **消息**：对象之间通过消息进行通讯，以实现对象之间的动态联系。
8. **关联**：表示对象之间的**静态关系**。

1.2 1.2 什么是面向对象

从**程序设计方法的角度**看，面向对象是一种**程序设计范型**(paradigm)。使用**对象、类、继承、封装、聚合、关联、消息、多态性等**基本概念来进行程序设计。

从**软件方法学的角度**看，面向对象方法是一种运用对象、类、继承、封装、聚合、关联、消息、多态性等概念来构造系统的**软件开发方法**。

程序=<对象, 关系> 对象=算法+数据结构

2 第二章 需求获取与分析

需求工作：起始活动（可行性分析）→ 需求获取活动（得到**软件需求陈述**，通过自然语言、用况图、用户故事等）→ 需求分析活动（一个精确的需求模型）→ 需求规约活动（**软件需求规约**，采用标准的模板） → 需求验证活动 → 需求管理活动

2.1 2.1 需求的定义

一个需求是一个陈述，描述了待开发产品／系统（或项）功能上的能力、性能参数或者其它性质。

2.2 2.2 需求的基本性质

IEEE标准830-1998要求单一需求必须具有5个基本性质:

1. 必要的(Necessary)。缺少了这个需求是不是还可以?
2. 无歧义的(Unambiguous)。只能用一种方式解释吗?
3. 可测试的(Testable)。可以对它进行测试吗?

4. 可跟踪的(Traceable)。可以从一个开发阶段到另一个阶段对它进行跟踪吗？不可能开发过程中需求就消失了。

5. 可测量的(Measurable)。可以对它进行测量吗？应该有一个方法可以衡量需求是否被满足了。

注:确定一个需求是否满足以上五个性质是复杂耗时的过程。

2.3 2.3 需求分类

功能；性能；外部接口；设计约束；质量属性

功能的需求是最重要的，所有别的需求都必须依附在功能这个需求之上。

2.3.1 功能需求

功能需求规约了系统或系统构件必须执行的功能。

1. 对该功能输入的假定，或者为了验证该功能输入，有关检测的假定。（类似于OI竞赛中给定的$n \leq 100$这样）

2. 功能内的任一次序，这一次序是与外部有关的。（应该先干啥）

3. 对异常条件的响应，包括所有内外部所产生的错误。

4. 需求的时序或优先程度。

5. 功能之间的互斥规则。

6. 系统内部状态的假定。

7. 为了该功能的执行，所需要的输入和输出次序。

8. 用于转换或内部计算所需要的公式。

2.3.2 性能需求

系统或系统构件必须具有的性能特性。

注：**性能需求隐含了一些满足功能需求的设计方案**，经常对设计产生一些关键的影响。例如：排序，关于花费时间的规约将确定哪种算法是可行的。

2.3.3 外部接口需求

应当为外部提供什么样的接口，应当使用哪些外部接口，进而必须可以处理这样的格式的数据。

1. 系统接口：与系统的其他应用进行交互。

2. 用户接口：规约对给用户所显示的数据，对用户所要求的数据以及用户如何控制该用户接口。

3. 硬件接口

4. 软件接口：和其他软件产品交互。比如和数据库管理系统交互，和操作系统交互。（问：和操作系统交互的接口到底是软件接口还是系统接口？）

5. 通讯接口：比如必须使用的网络类型。

6. 内存约束：描述易失性存储和永久性存储的特性和限制，特别应描述它们是否被用于与一个系统中其它处理的通讯。（问：为什么内存约束要被使用在外部接口需求这里？）

7. 操作：规约用户如何使系统进入正常和异常的运行以及在系统正常和异常运行下如何与系统进行交互。应该描述在用户组织中的操作模式，包括交互模式和非交互模式；描述每一模式的数据处理支持功能；描述有关系统备份、恢复和升级功能方面的需求。（问：没有理解）

8. 地点需求：如何安装以及调整一个地点，来适应新的系统。

2.3.4 设计约束

设计约束限制了系统或系统构件的设计方案。就约束的本身而言，对其进行权衡或调整是相当困难的，甚至是不可能的。它们必须予以满足。

任取10秒，一个特定应用所消耗的可用计算能力平均不超过50%。（问：为什么不属于性能需求？）

系统必须用C++或其他面向对象语言编写。系统用户接口需要菜单。（问：这里为什么不属于外部接口需求？）

必须在对话窗口的中间显示错误警告，其中使用红色的、14点加粗Arial字体。（问：这里为什么不属于功能需求？）

1. 法规限制
2. 硬件限制
3. 与其他应用的接口
4. 并发操作
5. 审计功能
6. 控制功能
7. 高级语言需求
8. 握手协议
9. 应用的关键程序
10. 安全考虑

2.3.5 质量属性

可靠性、可维护性、易用性（用户友好性）、安全性、可移植性、规模、速度

2.4 2.4 需求发现

2.4.1 1. 自悟（Introspection）

需求人员把自己作为系统的最终用户，审视该系统并提出问题："如果是我使用这一系统，则我需要…"

2.4.2 2. 交谈

在交谈期间需求可能不断增长，或是以前没有认识到的合理需求的一种表现，说是“完美蠕行”(Creeping elegance)病症的体现，以至于很难予以控制，可能导致超出项目成本和进度的限制。

解决方法：项目管理人员和客户管理人员应该定期地对交谈过程的结果进行复审。

2.4.3 3. 观察

尽管了解的这些信息可以通过交谈获取，但“第一手材料”一般总是能够比较好地“符合现实”。

客户可能抵触这一观察。

客户还可能认为开发者在签约之前，就已经熟悉了他们的业务。（问：这句话想表达什么？）

2.4.4 4. 小组会

举行客户和开发人员的联席会议，与客户组织的一些代表共同开发需求。

2.4.5 5. 提炼

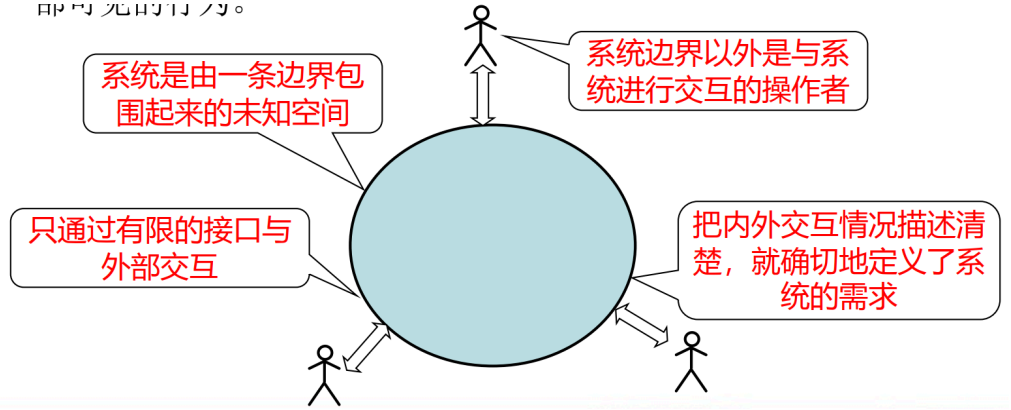
复审技术文档。

提炼方法是针对已经有了部分需求文档的情况。依据产品的本来情况，可能有很多文档需要复审，以确定其中是否包含相关联的信息。在有的情况，也可能只有少数文档需要复审。

2.5 2.5 需求获取技术

2.5.1 用况图（Use case diagram）

把系统看作一个**黑箱**，看它对外部的客观世界发挥什么作用，描述它外部可见的行为。



我们可以有很多参与者在这里面。

优势：易于探讨和理解、易于对需求规范化、有利于进行OOA、有助于发现主动对象、对系统测试来说，产生测试用例、有助于人机界面设计、etc

系统边界、操作者、用况、关联、包含、扩展、泛化

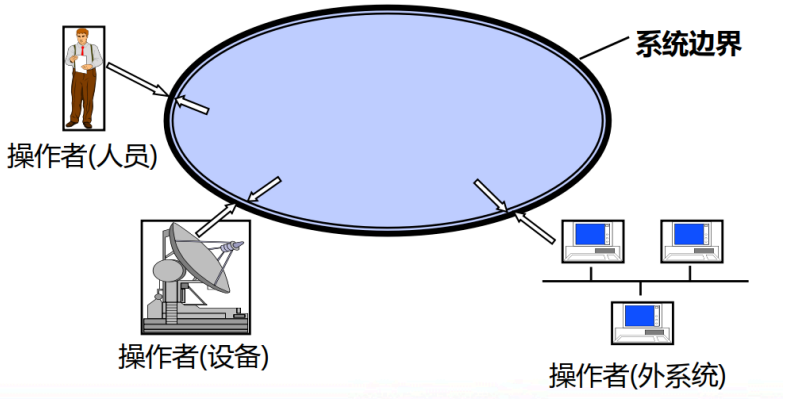
2.5.1.1 系统边界

某些事物可能既有一个对象作为其抽象描述，而本身（作为现实世界中的事物）又是在系统边界以外与系统进行交互的操作者。如超市中的收款员，他本身是现实中的人，作为操作者；在系统边界内，又有一个相应的“收款员”对象来模拟其行为或管理其信息，作为系统成分。

某些事物即使属于问题域，也与系统责任没有什么关系。如超市中的保安员，在现实中与超市有关系，但与所开发的系统超市商品管理系统无关系。这样的事物既不位于系统边界内，也不作为系统的操作者。

2.5.1.2 操作者

尽管在模型中使用操作者，但操作者实际上并不是系统的一部分。它们存在于系统之外。



我们可能的操作者。

2.5.1.3 用况（Use case）

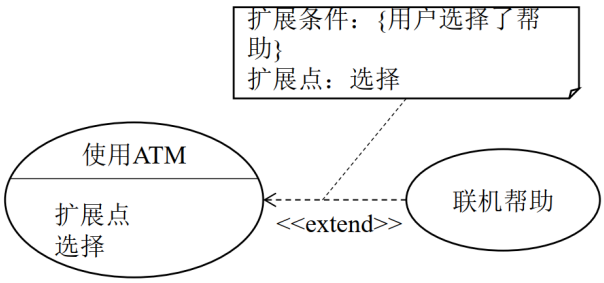
从使用视角看：用况是对操作者使用系统的一项功能时所进行的交互过程的描述。

2.5.1.4 关联

用况和操作者之间的唯一的关系，但是用况之间还有别的其他的关系。

2.5.1.5 扩展（extend）

从基用况到扩展用况的扩展关系表明：按基用况中指定的扩展条件，把扩展用况的行为插入到由基用况中的扩展点定义的位置。

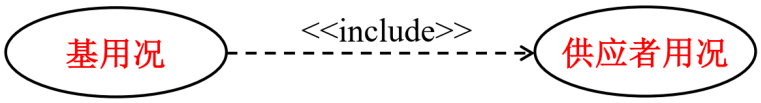


扩展的适用条件：(1) 异常情况；(2) 正常的变形描述时，而且希望采用更多的控制方式时，采用扩展。即在一个变化点上一个附加多个变体的场合下使用；(3) 用扩展关系来区分可实现系统的可配置部分（问：什么意思？）。

2.5.1.6 包含

从基用况到供应者用况的包含关系表明：基用况在它内部说明的某一（些）位置上**显式地**使用供应者用况的行为的结果。

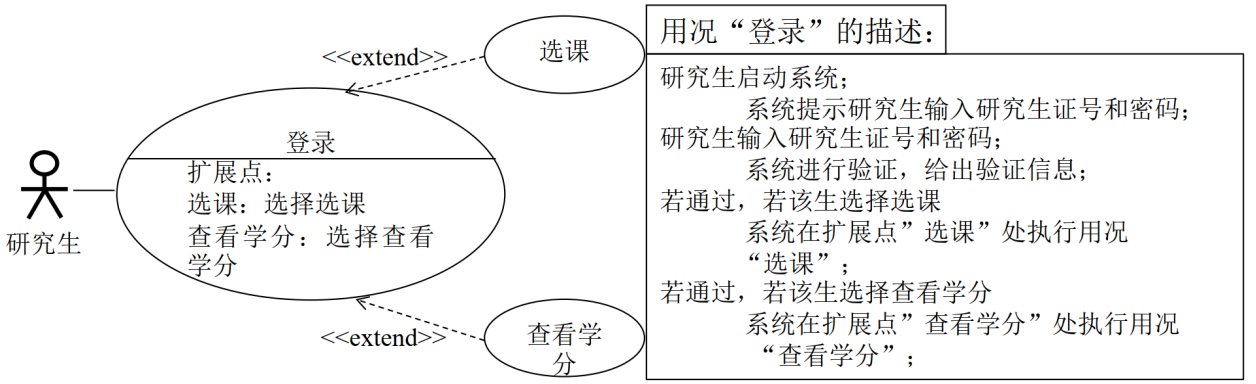
（问：包含和扩展有什么区别呢？我的理解是一个是显式地使用包含用况，一个是隐式地使用扩展用况？）



2.5.1.7 泛化

2.5.2 构建用况图

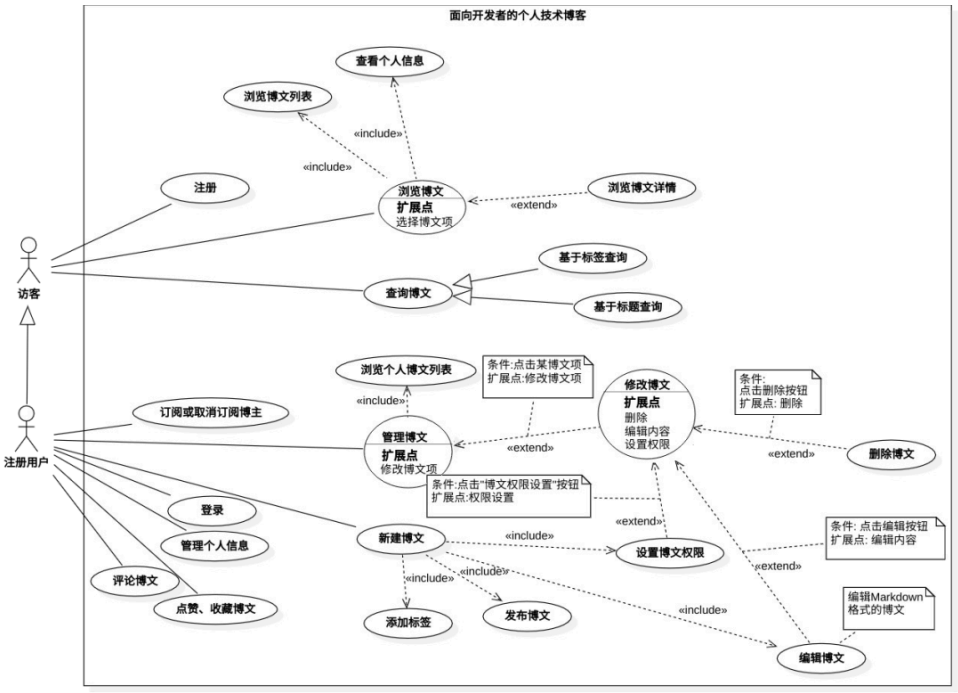
识别操作者、捕获用况（从操作者的角度来看、从系统功能的角度来看、从场景技术的角度来看）、系统的需求建模、审查



2.5.3 用户故事

用户+事件。其中用户表现为对主要用户的描述，事件则是场景或系统/任务价值

用户故事相当于用况图正文的描述说明。UML实际推崇的是用况图，但在敏捷开发中使用面向对象方法时也会将用户故事作为系统的需求描述



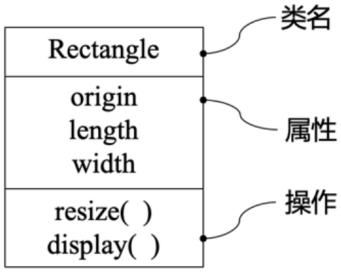
3 第三章 UML

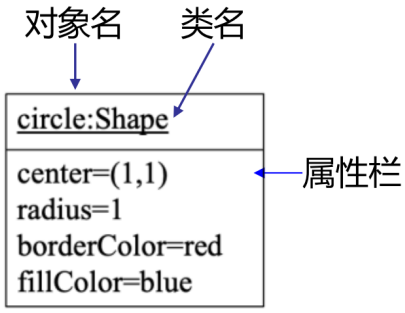
UML是系统分析和设计的工具

3.1 3.1 表达客观事物的术语

3.1.1 类与对象（数据抽象）

参与者、信号、实用程序都是类。



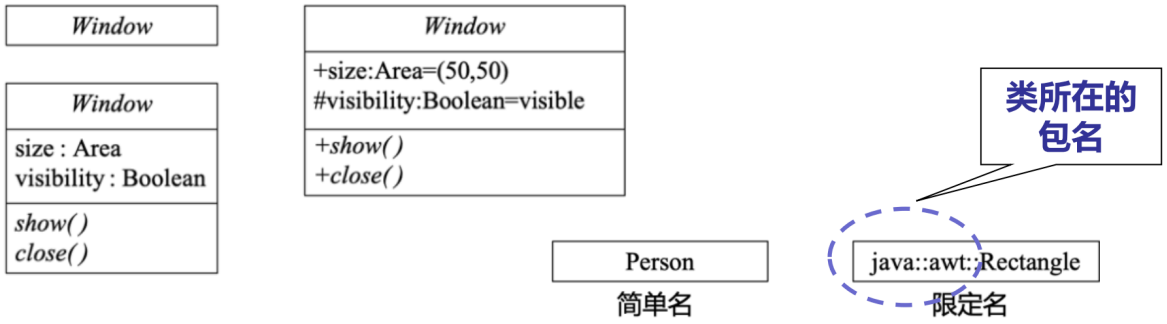


1）表示类的符号　　2）表示对象的符号

为什么对象没有操作栏？这是因为它的操作已经在它的类里面被定义了，而属性栏之所以存在是因为我们在构造这个对象的时候，必须为它填入初始的属性。

抽象类使用斜体字。

类可以有所在的包名。



属性的默认语法：`[可见性]属性名[:类型][多重性][=初始值][{特性串}]`，比如`+name: String[0..1] = "Hello"{readonly}`。

1. 可见性：**为什么引入可见性？**为了支持信息隐蔽这一软件设计原则。信息隐蔽是指在每个模块中所包含的信息（包括表达信息的数据以及表达信息处理的过程）不允许其它不需要这些信息的模块访问。信息隐蔽是实现模块**低耦合**的一种有效途径。

2. 属性名：小写字母开头

3. 类型名

4. 多重性：省略的时候默认为`[1..1]`

5. 初始值

6. 性质串：`a : integer = 10 {frozen}`，这就表示属性是不可改变的

所有的共同的属性，加上下划线，`defaultSize`这样子的。

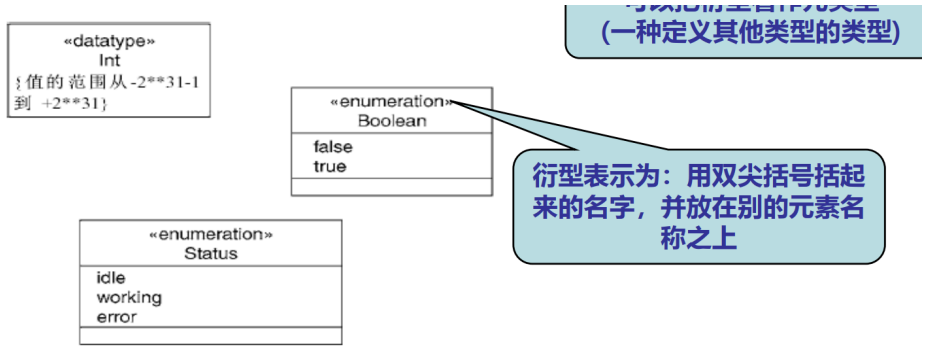
可见性	符号表示	关键字表示	描述
公有的	+	public	所有其他的类/类目都可以使用
受保护的	#	protected	仅其子类/类目才能使用
私有的	-	private	仅本类的操作才能使用
包内的	~	package	仅同一包中声名的类/类目才能使用

操作的默认语法：`[可见性] 操作名 [(参数表)] [: 返回类型] [{性质串}]`，例：`+ set(id : Integer, name : String) : Boolean {...}`。

参数表里的每个参数还可以如下：`[方向] 参数名 : 类型 [= 默认值]`，比如`in name: String = "Hello"`。

`in`输入参数，`inout`输入参数可修改，`out`输出参数。

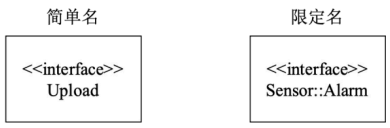
衍型：类型的类型，用双尖括号包裹起来。`<<datatype>>`。

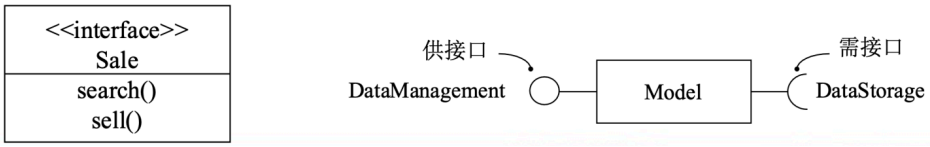


3.1.2 接口（功能抽象）

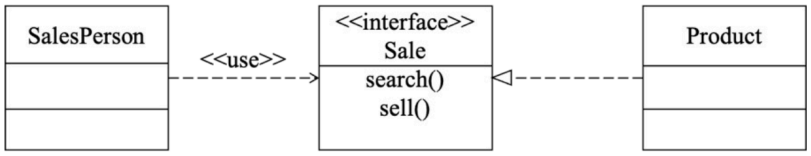
UML表示

1. 用带有分栏和关键字 <<interface>> 的矩形符号来表示接口。其中：在操作分栏中给出接口支持的操作列表；
2. 用小圆圈来表示接口。
   - 左侧圈：供接口，类提供的接口；
   - 右侧半圈：需接口，类需要的接口。
3. 两种表示中均要写出接口名





3、实现接口的类（类目）与该接口之间是一种实现关系，用带有实三角箭头的虚线表示之。使用接口的类（类目）和该接口之间是一种使用关系，用带有《use》标记的虚线箭头表示之。



3.1.3 协作（Collaboration，体现行为结构抽象）

协作是一个交互，涉及交互三要素：交互各方、交互方式以及交互内容 。

使机器人沿着一条路径移动所涉及的类 → 一个协作

3.1.4 用况（功能抽象）

一个用况描述了系统的一个完整的功能需求。

用况是通过协作予以细化的。

3.1.5 主动类（并发行为抽象）

进程和线程都是主动类。

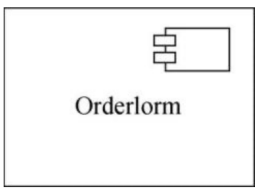
主动类对象的行为通常与其他元素的行为是并发的。

至少具有一个进程或线程的类，能够启动系统的控制活动。

3.1.6 构件

在遵循并提供了一组外部接口的实现的同时，隐藏内部实现。**构件是描述比特世界的软件制品的系统单位。**

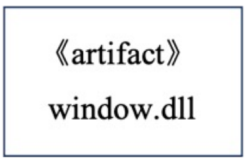
应用、文档、库、页和表都是构件。



3.1.7 制品

系统中包含物理信息的、可替代的物理部件。

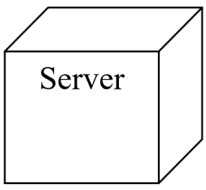
这里说部件是“物理的”，指软件未运行时也存在的部分，主要指文件，如源代码、可执行程序、脚本等；软件系统中可能会存在不同类型的部署制品；制品通常代表对源代码/运行时信息的物理打包。



3.1.8 节点

在运行时存在的物理元素，通常它表示一种具有记忆能力和处理能力的计算机资源（比如服务器、数据库管理系统等）

这里的“物理的”指实际机器设备；一个构件可以驻留在一个节点中，也可以从一个节点移到另一个节点。



3.2 3.2 表达关系的术语

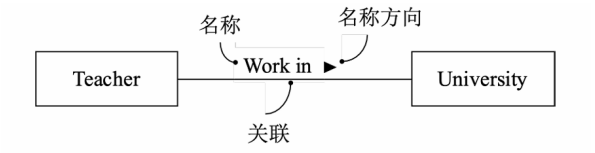
3.2.1 关联（association）

它指明一个类的对象与另一个类的对象间的联系。如果类的对象之间通过属性有连接关系，那么这些类之间的语义关系就是关联。**链是关联的实例**，是对象间的语义连接，是对象引用的元组（列表）。 在最常见的情况下，它是一对对象引用。

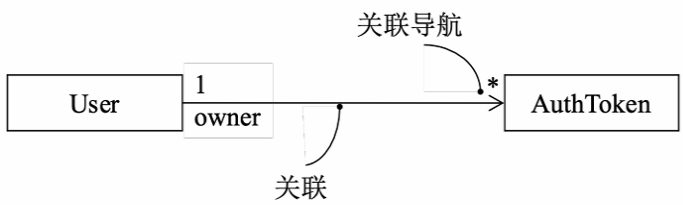
关联表示对象之间的**静态**联系。

教师为学生指导论文

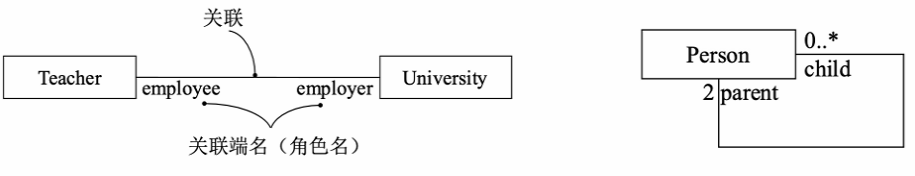
这里的多重性是一个教师可以指导0到多个学生的论文，而一个学生只能被一个教师指导论文。



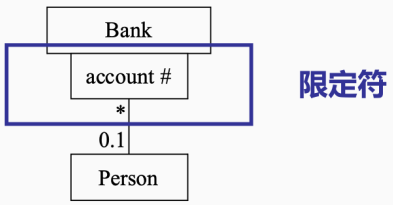
还可以用关联导航来解决。



角色和关联端名：



限定符：

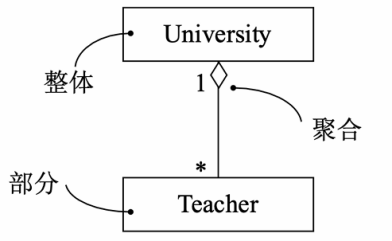


– 上图的限定符有一个属性account#，表明：

给定一个Bank对象，并给定account一个值，可以找到0或1个Person对象。该属性对类“Person”中的对象进行了一个划分。

3.2.1.1 聚合

一种特殊形式的关联，表达一种“整体/部分”关系。即一个类表示了一个大的事物，它是由一些小的事物（部分）组成的。

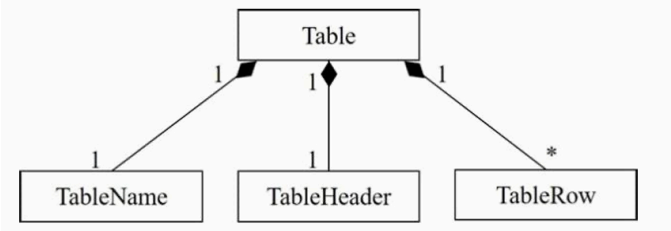


3.2.1.2 组合

组合是一种特殊的聚合。组合中的对象具有相同的生命周期。

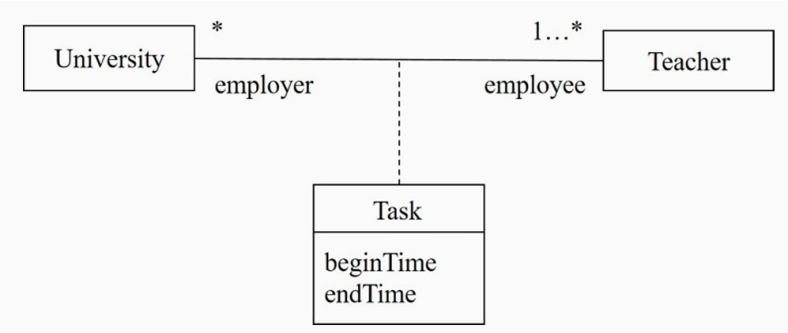
组合的末端的多重性不能超过1。因为一个对象最多组合成另一个对象。

在一个组合中，由一个链所连接的对象而构成的任何元组，必须都属 于同一个整体类的对象（问：没有理解这句话）



3.2.2 关联类

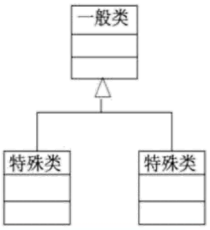
关联类：具有关联和类特性的模型元素，可以被看作是关联，但还有类 的特性；或被看作是一个类，但有关联的特性。



比如我们能认为大学和教师之间的关联是通过任务来获得的。

3.2.3 泛化（generalization）

“is-a-kind-of” 继承关系又称为一 般一特殊关系，在UML中把继承关系称为泛化关系。

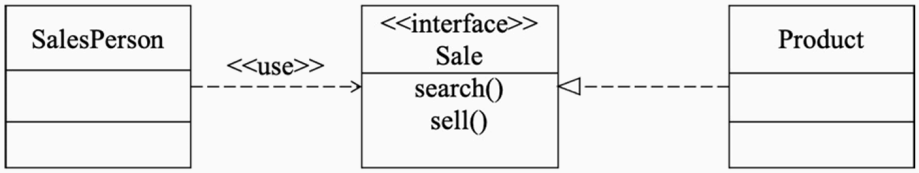


3.2.4 实现（realization）

在两个地方会使用实现的关系：**接口与实现它们的类和构件之间；用况与实现它们的协作之间**。

说明：在以下2个地方会使用实现关系：

1、接口与实现它们的类和构件之间；



2、用况与实现它们的协作之间。

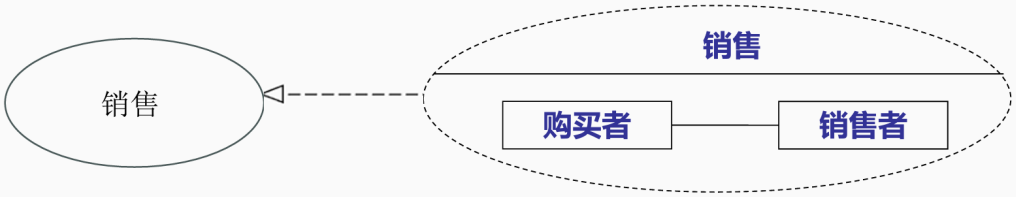
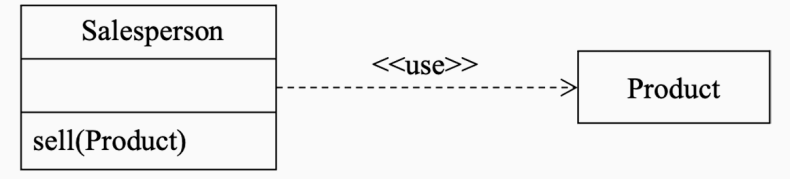


图 一个协作的组合结构图

3.2.5 依赖（dependency）

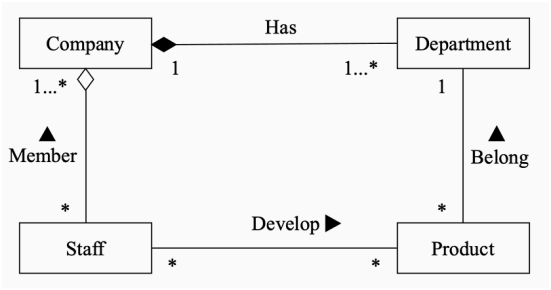
依赖是一种使用关系，用于描述一个类目使用另一类目的信息和服务。



3.2.6 UML关系用法

3.2.6.1 1. 结构关系

使用**“关联”**，模型化系统中存在的大量静态结构和动态结构。



以数据驱动：一个对象需要**导航**到另外一个对象

以行为驱动：一个对象需要与另一个对象**交互**

3.2.6.2 2. 继承关系

使用**“泛化”**，对系统中存在的一般/特殊关系规约。

3.2.6.3 3. 精化关系

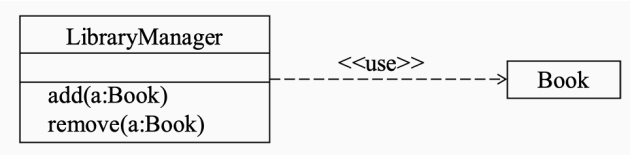
使用**“实现”**，对系统中存在的精化关系规约

该术语用于表达不同抽象层之间的精化，体现“自顶向下，逐步求精”的思想。如系统需求层的用况，可通过协作实现。这两者即可用“实现”规约。

3.2.6.4 4. 依赖关系

使用**“依赖”**，对不是结构、继承、精化的关系规约。

某类只作为另一类的**操作参数**，则可将它们之间的关系抽象为依赖

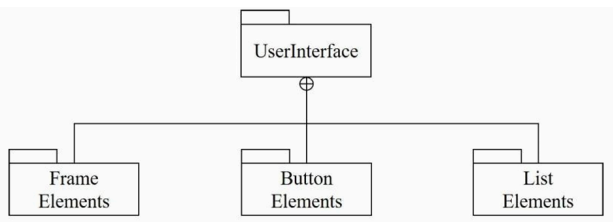


3.3 3.3 信息组织和解释的术语

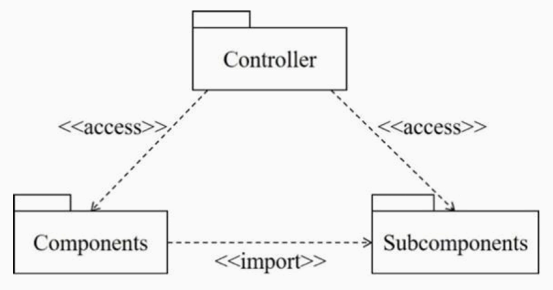
3.3.1 包

可以在小矩形里写包名，大矩形中表示内容。

也可以这样画：



引入依赖和访问依赖。

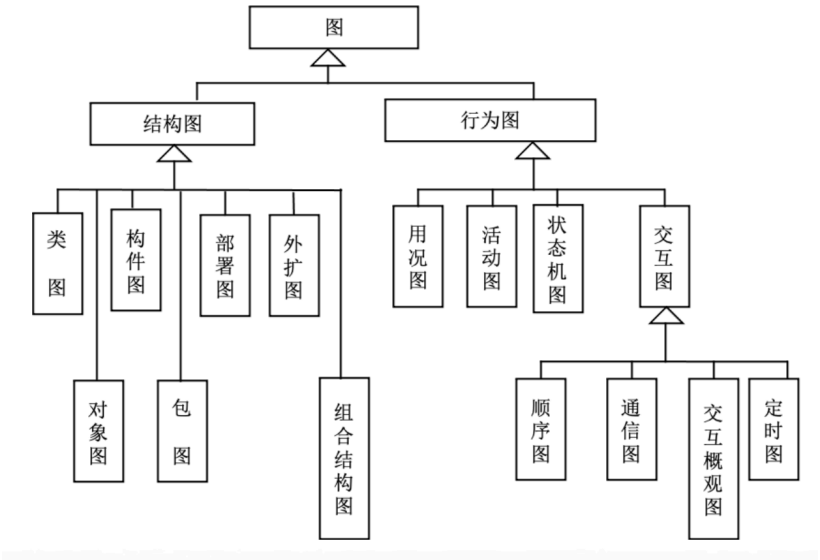


这样表示`Controller`可以访问`Components`和`Subcomponents`。

3.3.2 注解

注解可以是简单文字，也可以内嵌URL或文档链接，用**虚线**连接到所解释说明的模型元素上。

3.4 3.4 十四种图



3.4.1 类图

类图显示了类（及其接口）、类的内部结构以及与其他类的联系。是面向对象分析与设计所得到的**最重要的模型**。

3.4.2 对象图

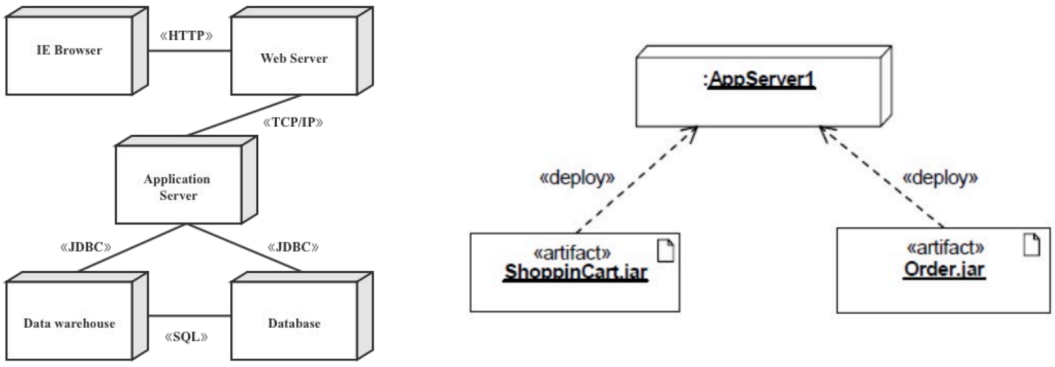
用对象图说明在类图中所发现的事物的实例的数据结构和静态快照。

3.4.3 构件图

就是用来表示构件之间的关系的图。

3.4.4 包图

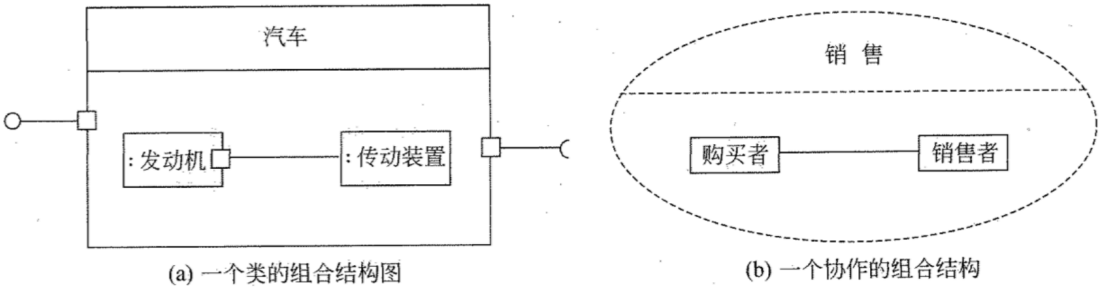
3.4.5 部署图



3.4.6 外扩图

外扩图是在UML2.5的基础上定义新建模元素的图，用以增加新的建模能力

3.4.7 组合结构图



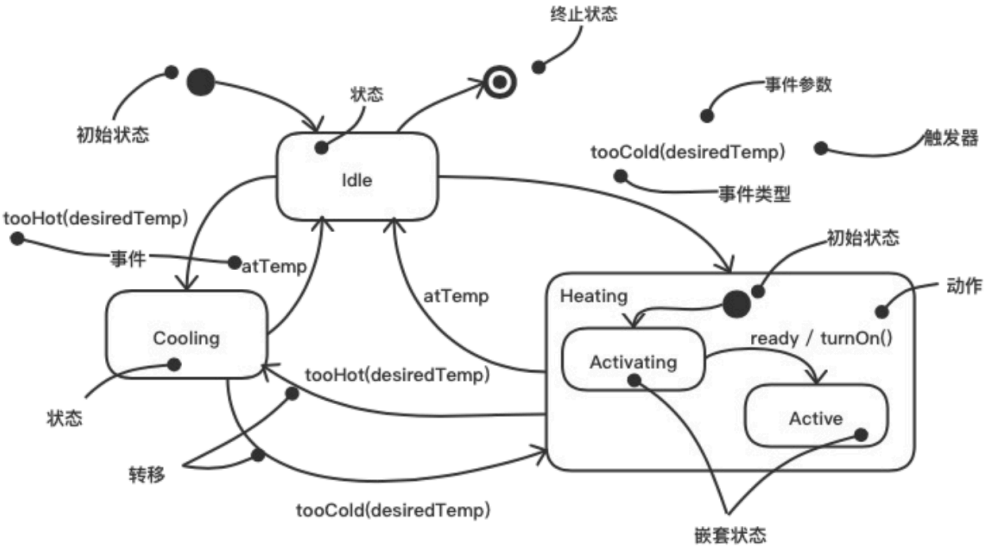
(a) 一个类的组合结构图　　(b) 一个协作的组合结构

所以这个图可以和构件图结合在一起。

3.4.8 用况图

3.4.9 状态机图

通常都是对**反应型对象（reactive object）**的行为进行建模。



状态机图是描述一个对象或其他实体在其生命周期中所经历的各种状态以及状态变迁的图.

一个状态机图，规约了一个对象在其生存期内因**响应事件**所经历的状态序列以及对这些事件所作出的响应。

状态之间的转化是即时的。

延迟事件：延迟事件是指在当前状态下暂不处理，但将推迟到该对象的另一个状态下排队处理的事件。比如我现在想要识别出某一个事件，但是我现在不处理。`activity/defer`

事件：事件就是状态转移的实心线上面标记的东西。`activity(parameter1: Type1, ...)`。

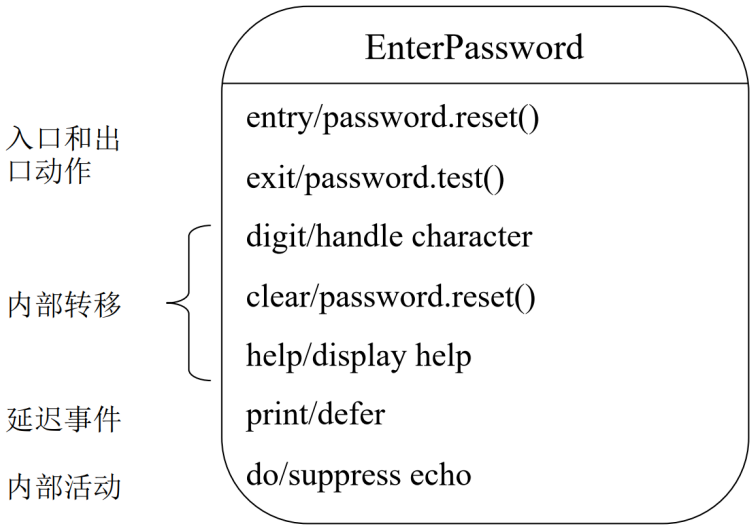
1. 信号（Signal）事件：信号是一种Type，所以它的原类型是`signal`。这里是衍型。
2. 调用（Call）事件：一个调用事件表示对象接收到一个操作调用的请求。
3. 时间事件：时间事件后跟有计算时间量的表达式。`after (2 seconds)`。
4. 变化事件：指定条件变为真了，譬如`at (xxx)/selfTest(), when (altitude < 1000)`。

`activityName[(Parameter list)][Guard][/Action]`。比如：`right-mouse-down(location)[location in window]/object:=pick_object(location)`。

UML内置关键字：

1. `entry`：进入状态时首先执行该动作。它不能有参数或监护条件。
2. `exit`：在退出状态时最后执行该动作。它不能有参数或监护条件。
3. `do`：在状态的进入动作表达式（如果有）执行后，开始执行do活动，并且do活动可与其他的动作或者活动并行。

状态间的转移 vs. 状态内的转移： 如果源状态和目标状态都是自身，那么会先触发该状态的退出动作，再执行该状态的进入动作，和状态内的转移有区别。

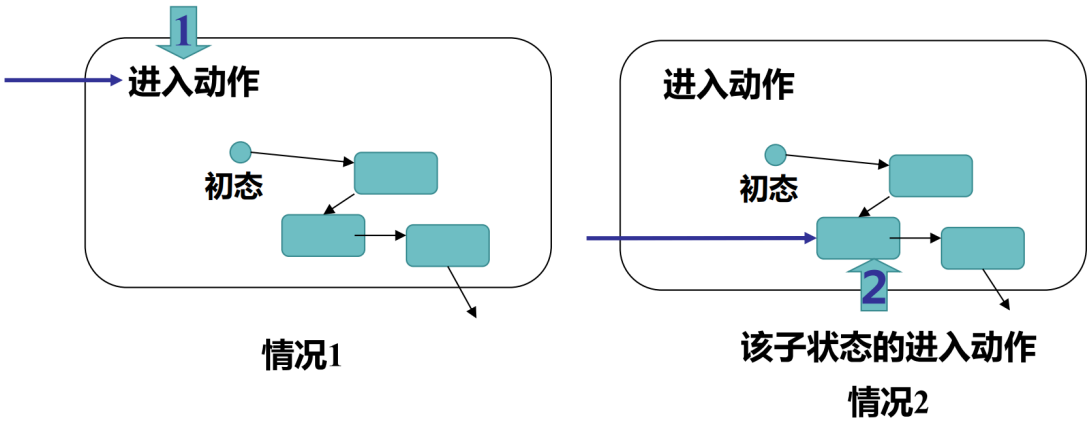


3.4.9.1 组合状态

顺序子状态机（非正交）和并发子状态机（正交）

非正交状态机：最多有一个子初态和一个子终态。

1. 转移到该组合状态：这个被嵌套的子状态机一定有一个初态，以便在进入该组合状态并执行其进入动作后，将控制传送给这一初态。
2. 转移到该组合状态的一个子状态：在执行完**该组合状态的进入动作**（如有的话）和**该子状态的进入动作**后，将控制传送给这一子状态。



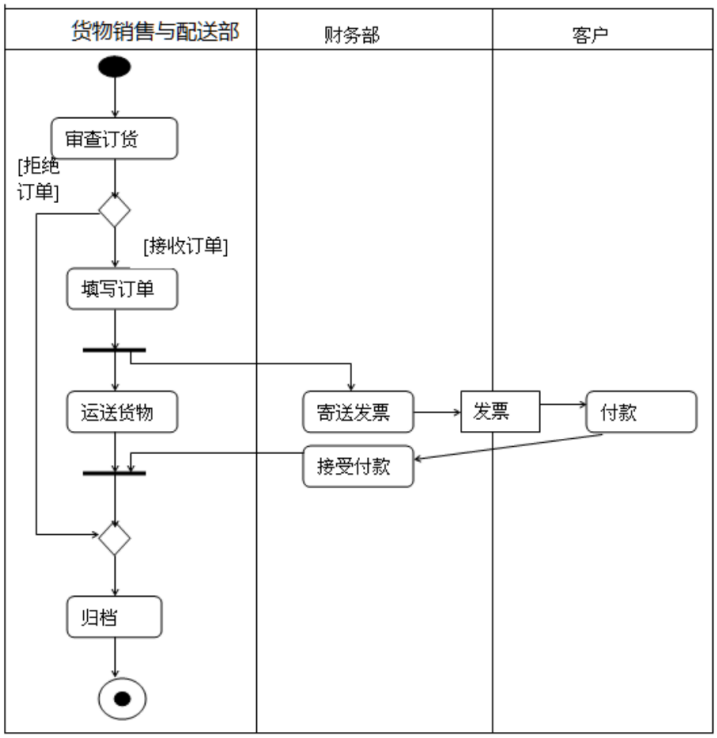
离开的时候：首先离开被嵌套的状态，即执行被嵌套状态的退出动作（如有的话）；然后离开该组合状态，即执行该组合状态的退出动作（如有的话）。

正交状态机：控制流分岔，分成了并发流。

3.4.10 活动图

动作是原子的和即时的。

对象是方的矩形，而动作是圆角的矩形。



3.4.11 顺序图

一种详细表示 对象之间以及对象与参与者实例之间交互的图，它由一组协作的对象（或参与者实例）以及它们之间可发送的消息组成，它强调消息之间的顺序。

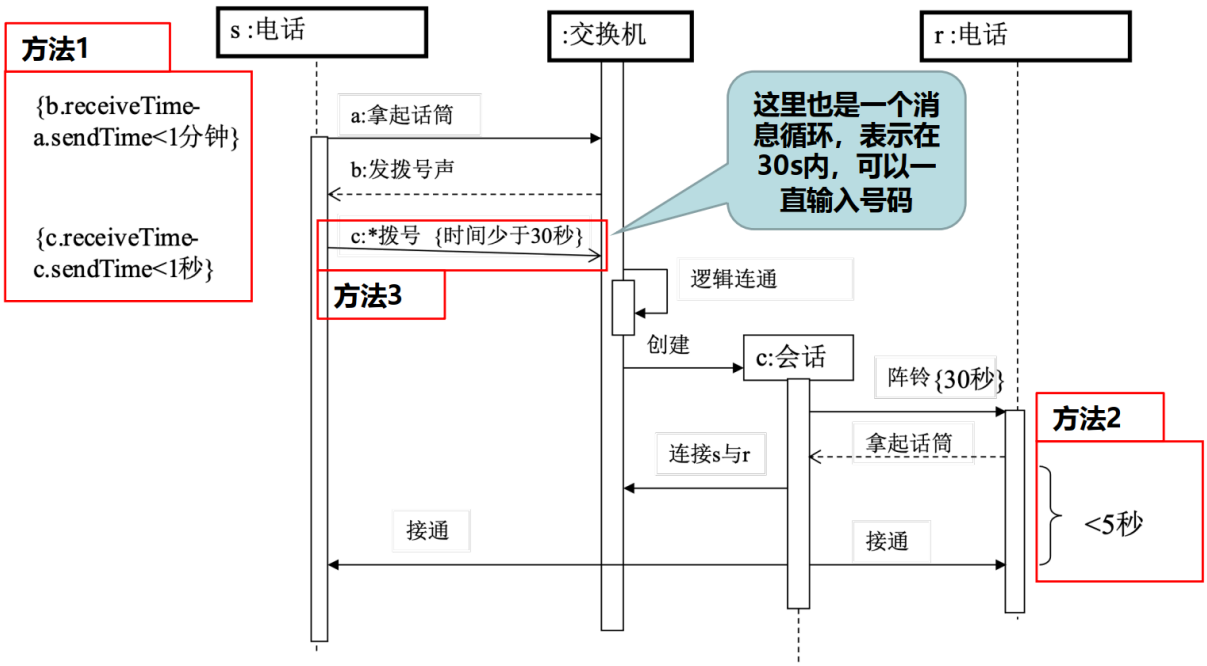
对象：`[ObjectName]: ClassName`。

参与者：一个小人

对象生命线：一个垂直虚线，用X表示被析构。

执行规约：窄长的矩形。

消息：同步消息：实心的箭头，返回虚线箭头 异步消息：实现，空心的箭头



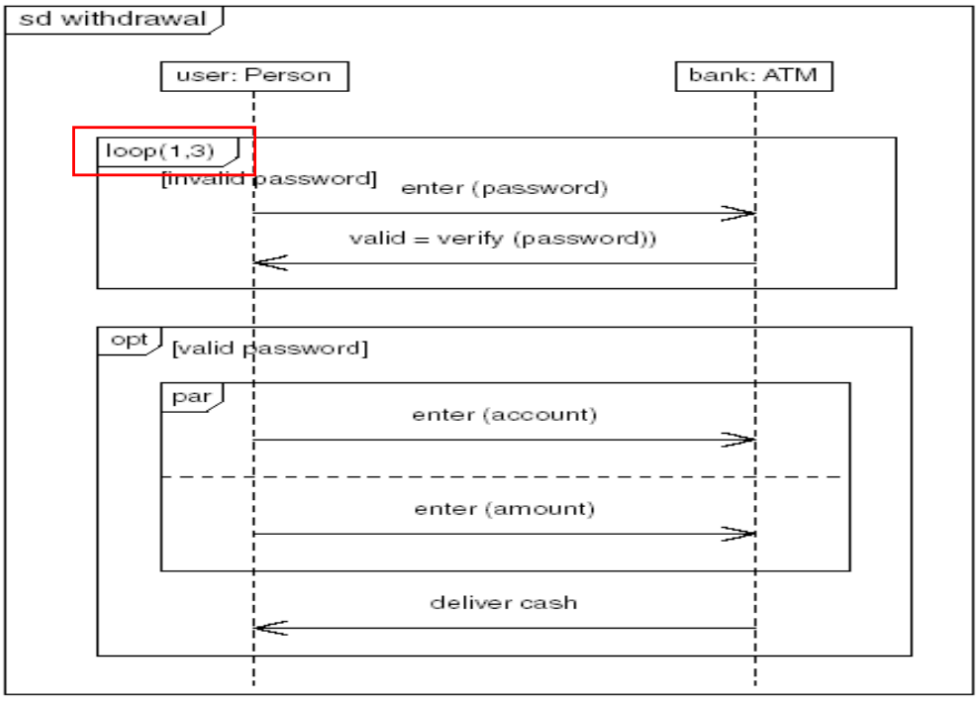
消息分支：把分支画成从一个点出发的多个箭头，每个箭头由监护条件标示

消息循环：`*Message[Guard]`，表示按照给定的表达式一直发送信息。

回调机制：申请对象在服务对象处事先登记所关心的事件，然后继续从事自己的工作；当服务对象监控到这样的事件发生时，再通知申请对象，由申请对象进行处理。这是一个异步的。

3.4.11.1 结构控制

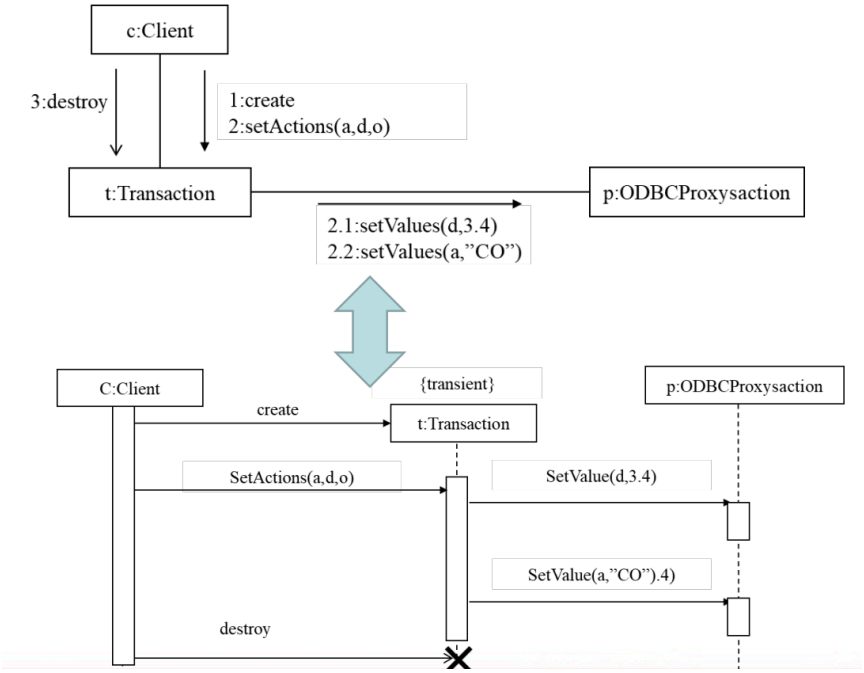
选择（`opt`,为真才执行）、条件（`alt`,有两种选择）、并发（`par`,同时做）、迭代（`loop guard`,循环）



3.4.12 通信图

为表示一个消息的时间顺序，给消息加一个数字前缀（从1号消息开始），在控制流中，每个新的消息的顺序号单调增加（如2，3等等）。为了显示嵌套，可使用带小数点的号码（1表示第一个消息；1.1表示嵌套在消息1中的第一个消息；1.2表示嵌套在消息1中的第二个消息；等等）。

通信图和顺序图两者语义上是等价的。



3.4.13 交互概观图

用于描述系统的宏观行为，是活动图和顺序图的混合物。

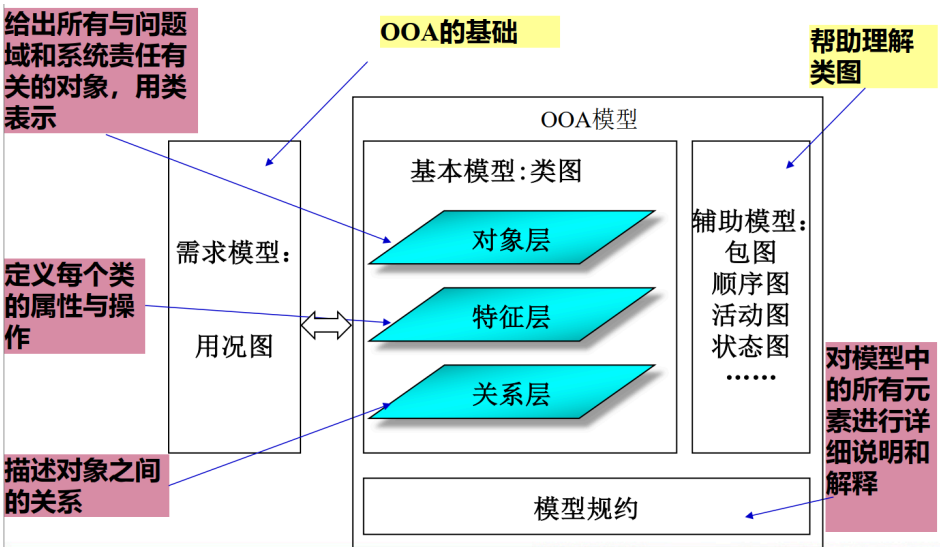
3.4.14 定时图

用于表示交互，它展现了消息跨越不同对象或角色的实际时间，而不仅仅关心消息的相对顺序。

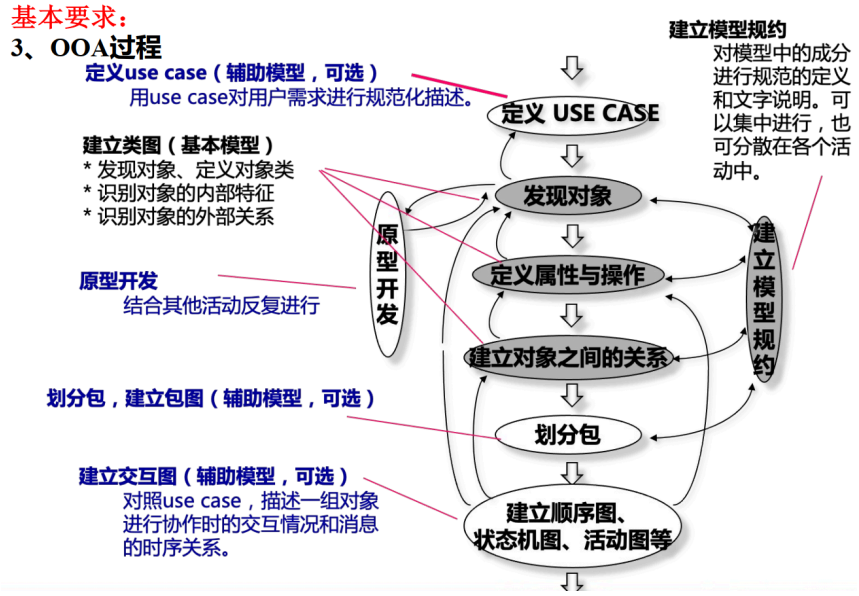
4 第四章 OOA

运用面向对象方法，对问题域（被开发系统的应用领域）和系统责任（所开发系统应具备的职能）进行分析和理解 。

4.1 OOA模型



4.2 OOA过程



1. 发现对象，定义他们的类。

2. 识别对象的内部特征：定义属性、定义操作。

3. 识别对象的外部关系：分类关系（继承）、构成关系、静态联系、使用关系

4. 给出系统的相关顺序图、状态图和活动图等，以建立系统的动态模型

5. 划分包，建立系统的包图

6. 建立系统的详细说明

4.3 面向对象分析方法中的复杂度控制的机制

a） 信息组织的复杂性：

抽象:从许多事物中舍弃个别的、非本质的特征，抽取共同的、本质性 的特征：系统中的对象是对现实世界中事物的抽象； 类是对象的抽象；一般类是对特殊类的抽象； 属性是事物静态特征的抽象； 操作是事物动态特征的抽象。

分类机制:把具有相同属性和操作的对象划分为一类，用类作为这些对象 的抽象描述。

继承:特殊类的对象拥有其一般类的全部属性和服务（一般-特殊结构）；

聚合:把一个复杂的事物看成若干比较简单的事物的组装体，从而简化对 复杂事物的描述。（整体-部分结构）

消息通讯：要求对象之间只能通过消息进行通讯，而不允许在对象之外 直接地存取对象内部的属性。

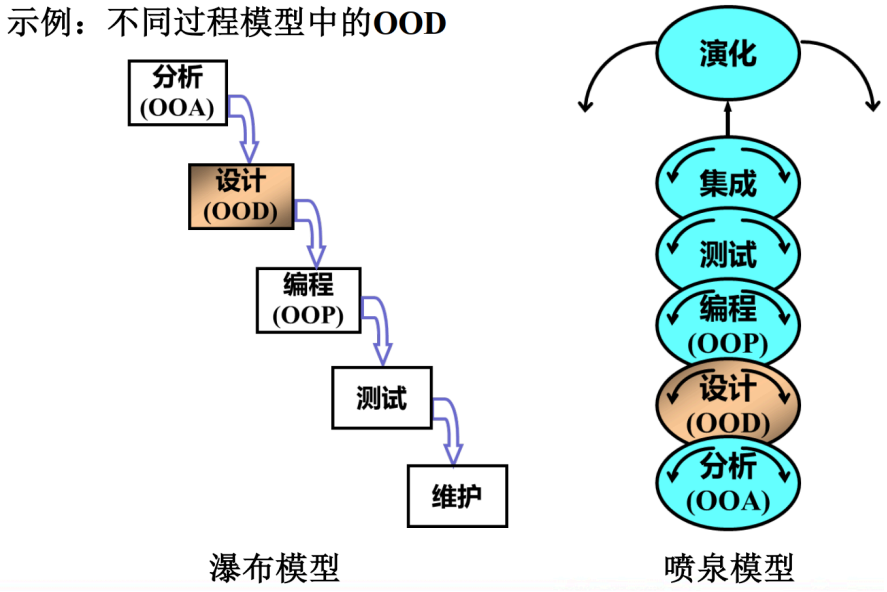
多个视图：从多个角度认识系统

b） 文档组织的复杂性—控制机制

包：使模型具有大小不同的粒度层次，以利于控制复杂性

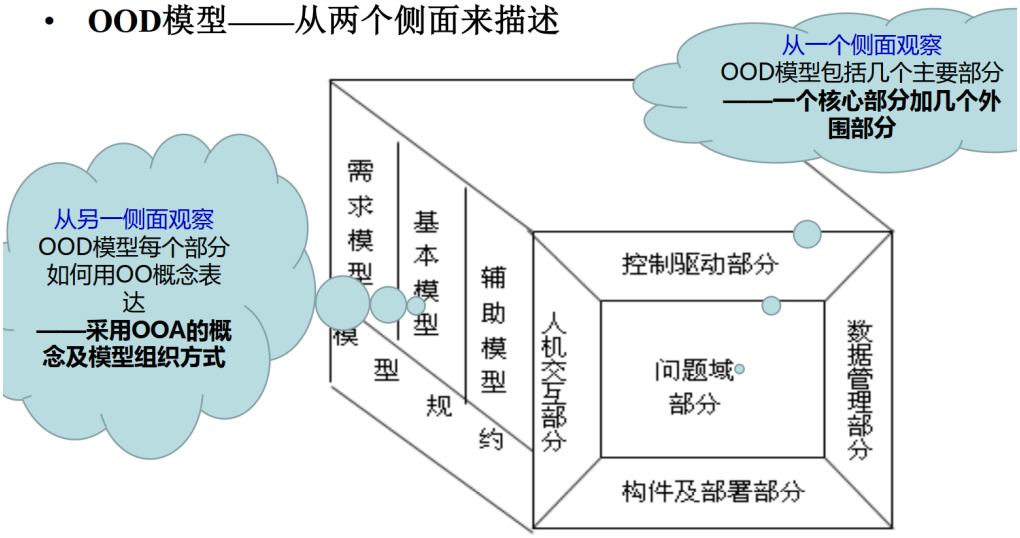
5 第五章 OOD

瀑布模型和喷泉模型



5.1 OOD模型

- **OOD模型——从两个侧面来描述**



5.2 OOD过程

不强调次序，逐个设计每个部分。

5.2.1 问题域部分

它是在OOA模型基础上按**实现条件**进行必要的修改、调整和细节补充而得到的。

实现条件：

1. 编程语言：对**问题域的影响最大**。选定的编程语言不支持某些面向对象的概念与原则；OOA阶段可能将某些与编程语言有关的对象细节推迟到OOD阶段来定义。如对象的创建、删除、复制、转存、初始化等系统行为、属性的数据类型等。

   (a) 为复用设计与编程的类而增加新的结构

   (b) 增加一般类以建立共同协议（比如：提供创建、删除、复制等操作，Java里面的object就可以体现这个问题，它是所有的类的父类，所有的类都要继承它）`object{复用}`

   (c) 按编程语言调整继承和多态：如果编程语言不支持多继承怎么办？（方法1：采用聚合，把多继承转换为单继承，方法2：重新定义对象类，方法3：压平）如果编程语言不支持继承怎么办？（方法1：把继承结构展平，所有需要的信息都放到展平后的类里面去；方法2：再加上聚合，每个类不存在信息冗余）如果编程语言不支持多态怎么办？重新命名函数。

   (d) 提高性能：调整对象分布，合并通讯频繁的类

   (e) 为实现对象永久存储做的修改

   (f) 为编程方便增加底层成分

   (g) 对复杂关联的转化并决定关联的实现方式：多对多关联转化为一对一关联

   (h) 调整与完善属性

   (i) 构造和优化算法

   (j) 决定对象间的可访问性

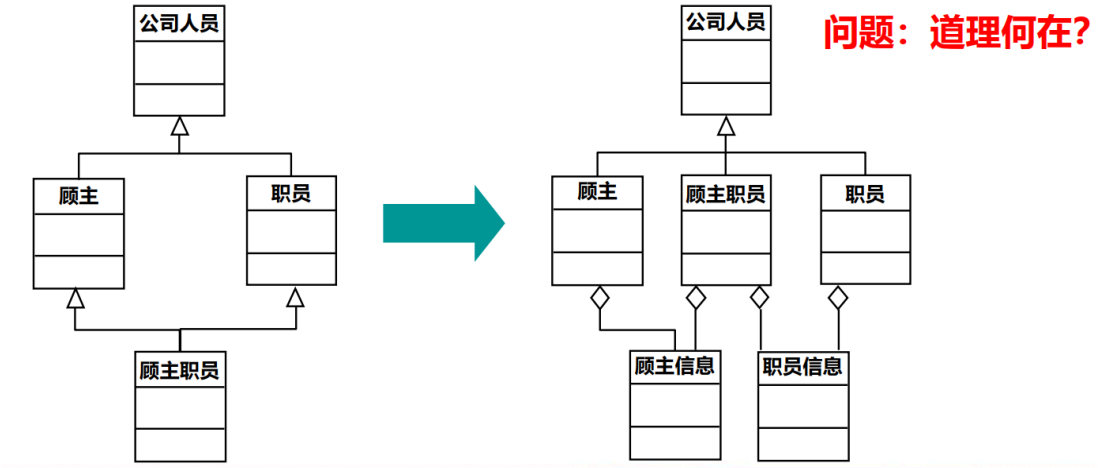
   (k) 定义对象实例：当系统需要通过从外存读取数据来创建一个对象时，先创建该对象，再从外存中读取这个对象数据，把数据赋值给对象的相应属性。

2. 硬件、操作系统及网络设施

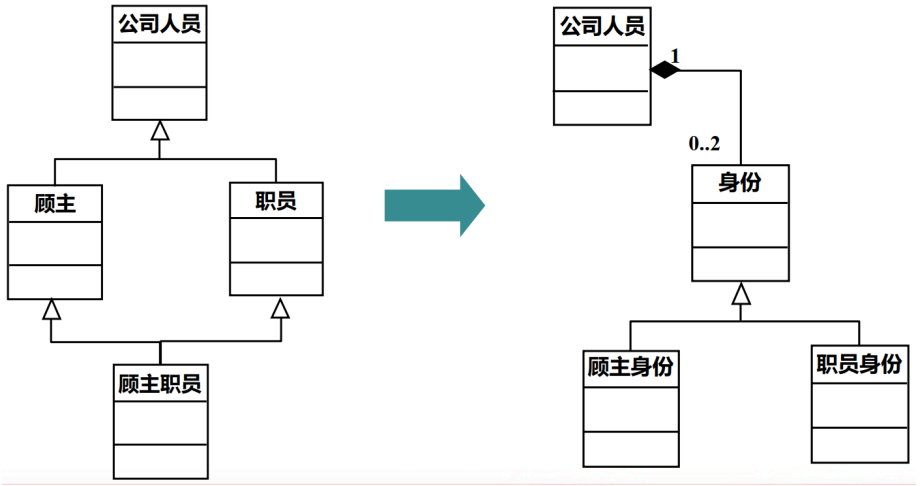
3. 复用支持

4. 数据管理系统

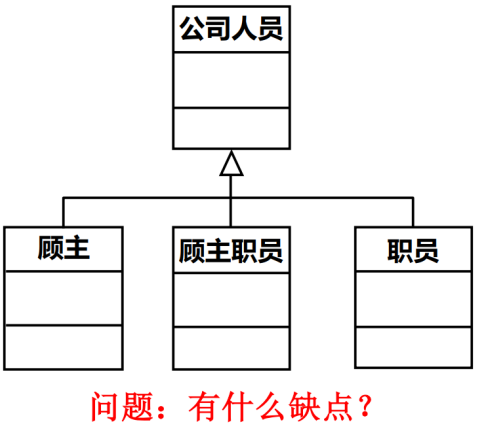
5. 界面支持系统



（道理：尽管继承和聚合反映了现实世界中两种不同的关系，但是从最终效果来看却存在共性—都是使一个类的对象能够拥有另一个（一些）类的属性和操作。）



（重新审视原来用多继承结构表达的实际事物及它们之间的关系。例如，上述例子换一个角度看问题：形成这种分类的原因使什么？从而增加“身份”类，构成单继承。）



（缺点：损失了信息）

为什么要这么做：使反映问题域本质的总体框架和组织结构长期稳定，而细节可变；稳定部分与可变部分分开，使系统从容地适应变化；有利于同一个分析用于不同的设计与实现；支持系统族和相似系统的分析设计及编程结果复用；使一个成功的系统具有超出其生存期的可扩展性

5.2.2 人机交互部分

OOD要设计人机交互的细节，而OOA用人机交互来反映需求

1. 分析与系统交互的人：设计时重点考虑比例最大的人员情况，并适当地兼顾其他人。
2. 从use case分析人机交互

人机交互界面的设计准则：使用简便、一致性、启发性、减少人脑记忆的负担、减少重复的输入、容错性、及时反馈、防止灾难性的错误

5.2.3 控制驱动部分

控制流驱动部分，用于定义和表示并发系统中的每个控制流。用主动对象表示每个控制流(进程、线程），所有的主动类构成控制驱动部分

如何设计控制驱动部分：用包括主动类的类图捕捉控制流的静态结构；用包括主动对象的顺序图或通信图捕捉控制流的动态行为。

5.2.4 数据管理部分

数据管理部分是负责在特定的数据管理系统中存储和检索持久对象的组成部分

其目的是，存储问题域的持久对象、封装这些对象的查找和存储机制，以及为了隔离数据管理方案的影响

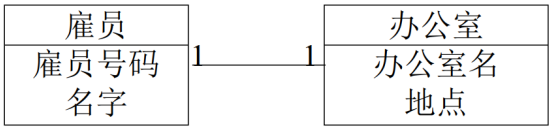
文件系统、R-DBMS（关系数据库系统）、OO-DBMS（面向对象数据库系统）

面向对象	实体-联系	关系数据库
类	实体类型	表
对象	实体实例	行
属性	属性	列
关系	关系	表

4、对关系的存储

– 将关联或聚合映射到表格

- 一对一关联映射到多个表



属性名	是否为空	域
办公室名	N	名字
地点	N	地址

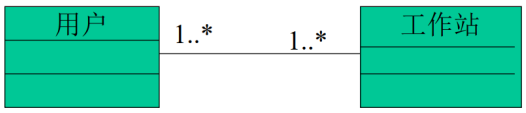
属性名	是否为空	域
雇员号码	N	号码
名字	N	名字

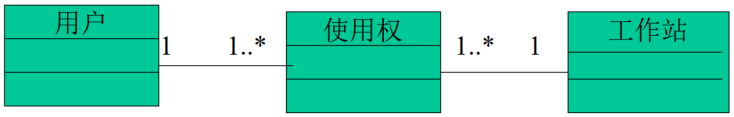
属性名	是否为空	域
雇员号码	N	号码
地点	N	地址

这个关联的表是右下角，应该是一种 `link1 : {EmployeeNumber , Location}` 这样的一行。

如果是1对多，可以为比如公司职员记录公司ID这个属性。或者是公司ID和职员ID建立一个关联表。

如果是多对多呢：





还有对父类和子类的关系的存储可以有下面的三种情况：

1. （下推）：为每一个子类定义一张单独的表。当增加新的父类或者子类的时候对数据库的维护和修改是很大的麻烦。

2. （上拉）去掉继承的网络结构。对实例要存储大量的冗余。

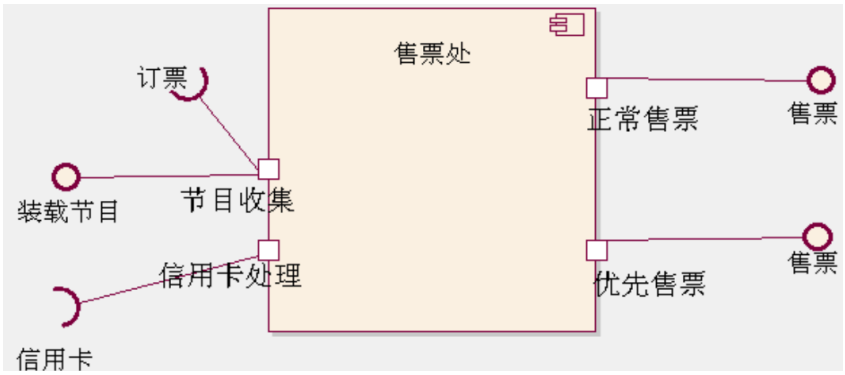
3. （分割表）将父类和子类的状态存储在不同的表中。这种方法很好地反映了继承网格，但它的缺点是访问数据时需要许多跨表连接。

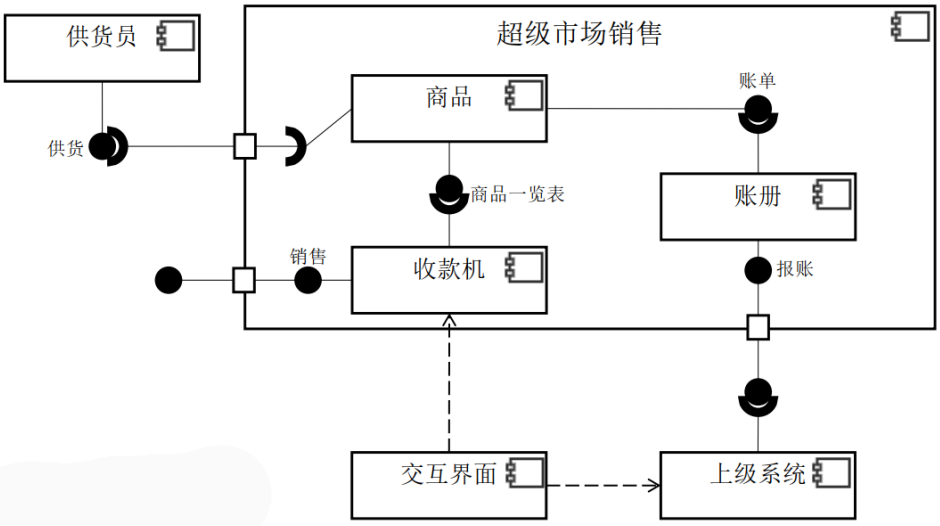
对象存储器。

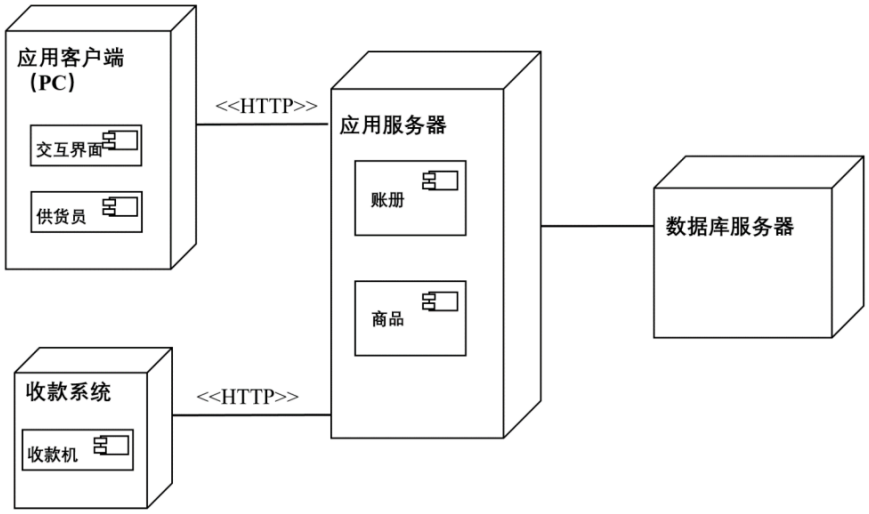
5.2.5 构建及部署部分

构件具有端口和接口。

接口是一个圆圆的或者半圆的，端口是一个方形的那个。







6 第六章OOP&OOPL

6.1 OOPL的基本特性

语言元素可以支持

1. 类的定义
2. 对象的静态声明或者动态创建
3. 属性和操作的定义
4. 继承、聚合、关联和消息的表示

语言机制有:

1. 类机制
2. 封装机制
3. 继承机制

高级特性: 多态、多继承的表示和支持机制

6.2 纯面向对象语言和混合型面向对象语言的各自特点

纯面向对象语言：Smalltalk、Eiffel、Java

较为全面的支持OO，强调严格的封装

混合型：C++、Objective-C、Objective-Pascal、Python

在一种非OO语言的基础上扩充OO成分、对封装采取灵活的态度

术语对照

OOA和OOD	C++
对象 object	对象 object
类 class	类 class
属性 attribute	成员变量 member variable
操作 operation	成员函数 member function
一般/特殊 generalization/specialization	基类/派生类 base class/derived class
整体/部分 whole/part	嵌套对象 nested object 嵌入指针 embedded pointer
消息 message	函数调用 function call
关联 association	对象指针 object pointer

问：为什么说对象指针实现了关联？

OOA和OOD	Java
对象 object	对象 object
类 class	类 class
属性 attribute	属性 attribute
操作 operation	方法 method
一般/特殊 generalization/specialization	基类/派生类 base class/derived class
整体/部分 whole/part	嵌套引用 nested reference
消息 message	方法调用 method call
关联 association	对象引用 object reference

OOA和OOD	Python
对象 object	对象 object
类 class	类 class
属性 attribute	属性 attribute
操作 operation	方法 method
一般/特殊 generalization/specialization	基类/派生类 base class/derived class
整体/部分 whole/part	嵌套引用 nested reference
消息 message	方法调用 method call
关联 association	对象引用 object reference

7 第七章 设计模式

7.1 外观模式

在高层定义接口，隐藏复杂性。可以理解为UI界面。

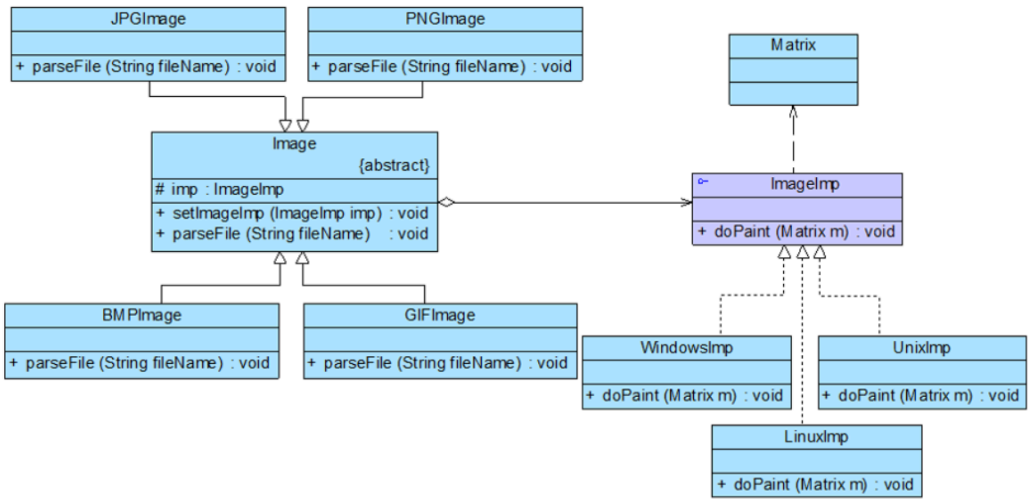
或者类似于一个APP的页面。

1. 隐藏复杂性
2. 可以用来包装遗产系统
3. 用来检测某项的访问量

7.2 桥接模式

将抽象部分与它的实现部分分离，使它们都可以独立地变化

识别出一个类所具有的两个独立变化的维度，将它们设计为两个继承等级结构，并建立抽象和实现的聚合关系



跨平台图像浏览系统结构图

7.3 单例模式

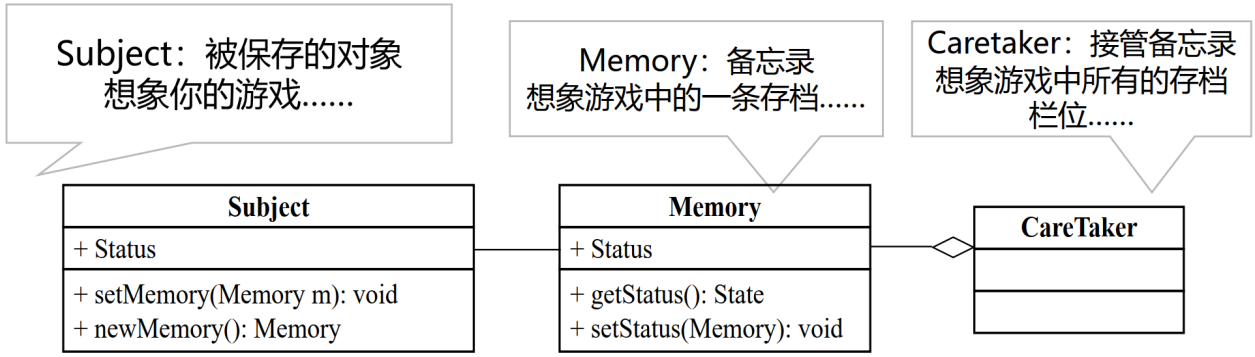
一个类只有一个实例，并提供全局的访问点。

全局变量无法阻止多次的实例化，而单例把实例化的方法私有了，这样就只能在一开始创建的时候实例化一次了。

先判断单例是否已经实例化，没有实例化的话再对实例化函数加锁。

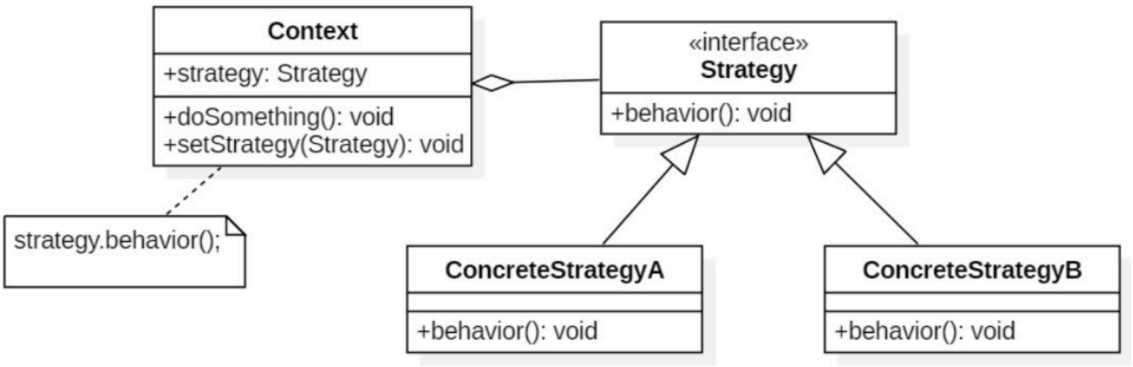
7.4 备忘录模式

类似于一个存档。

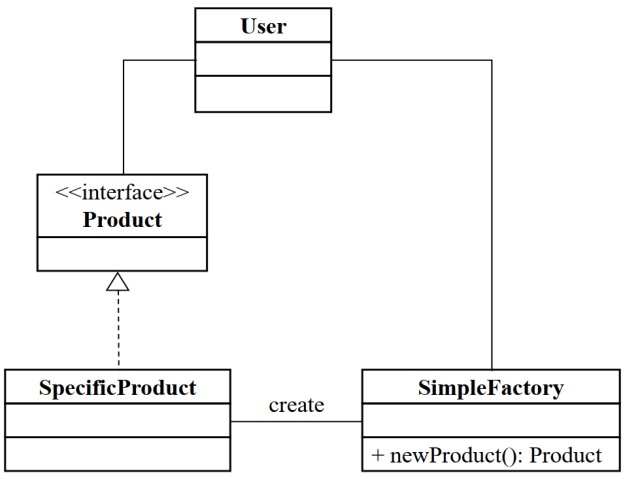


7.5 策略模式

优势：低耦合！策略可以自由切换，系统扩展性、灵活性更高



7.6 工厂方法模式

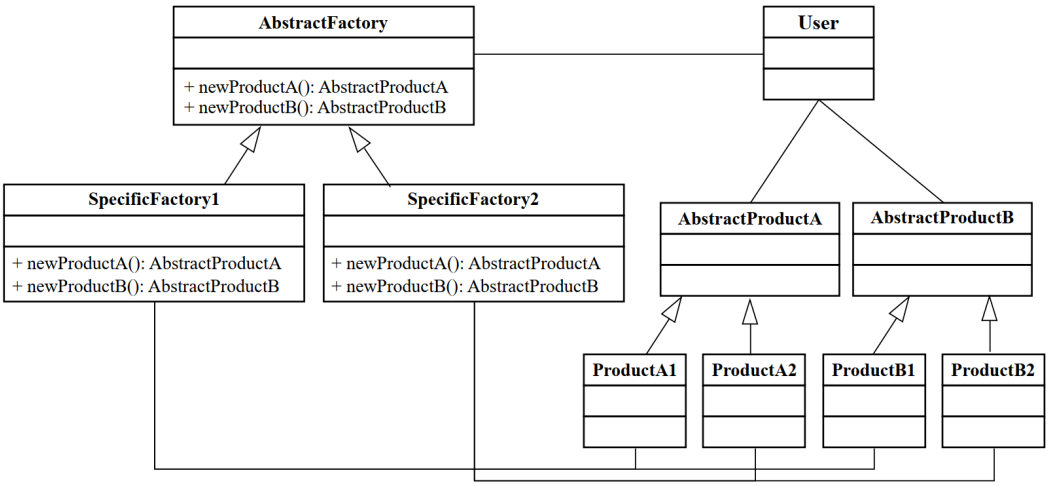


让所有的实例化都通过工厂来完成。

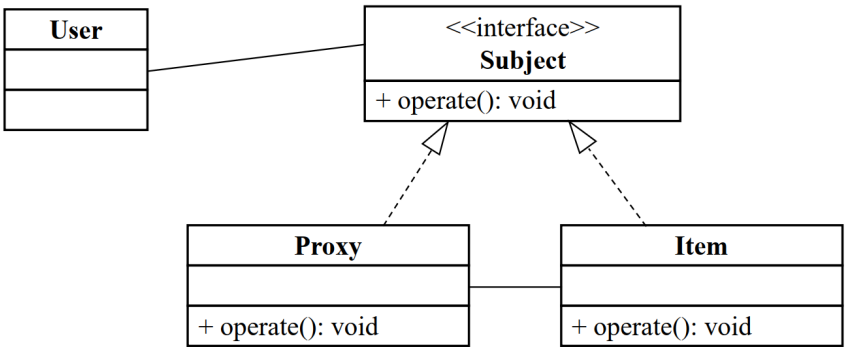
可以减少构造对象的时候的复杂性。

7.7 抽象工厂模式

用来创建对象族的。



7.8 代理模式



代理还可以再加入一点自己的操作。

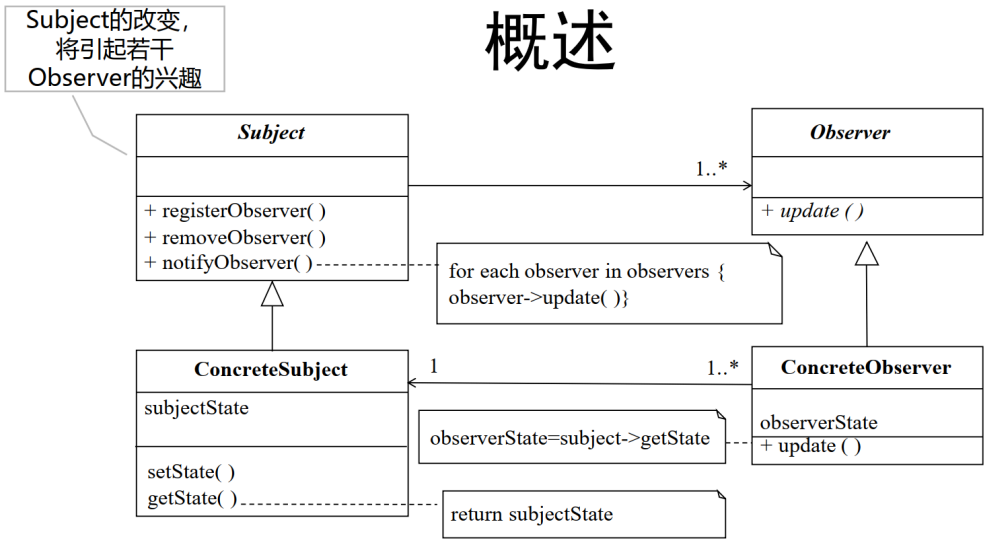
7.9 迭代器模式

提供访问数据结构的方法

7.10 访问者模式

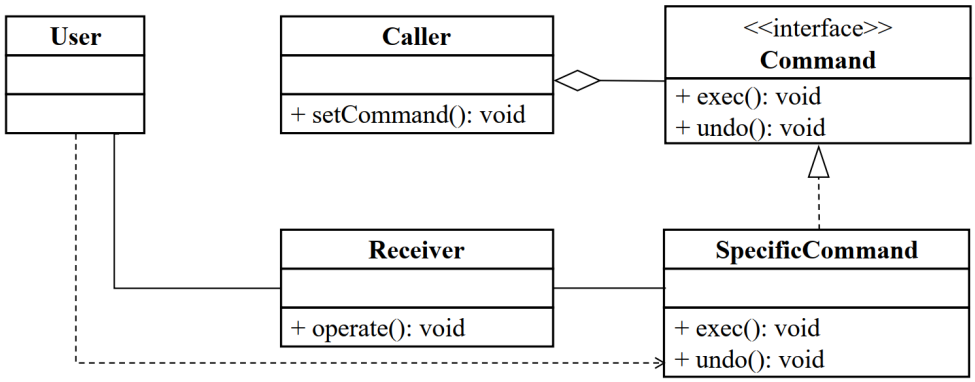
访问复杂的数据结构

7.11 观察者模式（发布-订阅模式）



7.12 命令模式

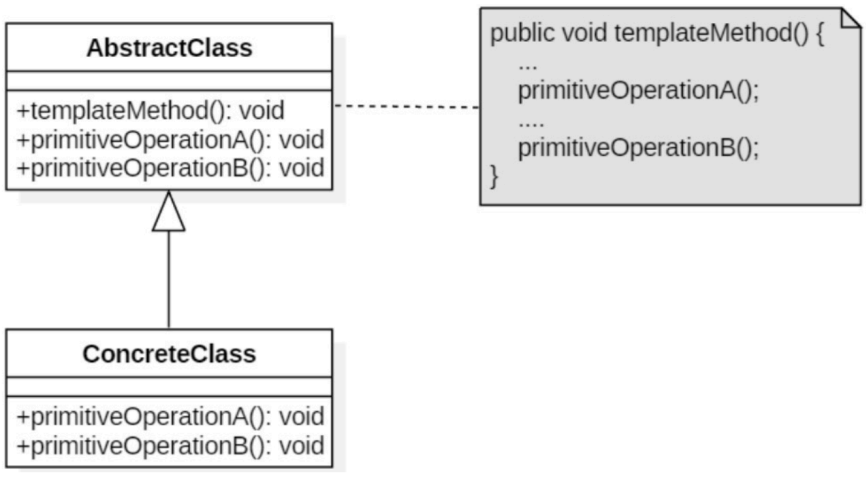
将一个请求封装成一个对象，使得发出请求的责任和执行请求的责任分开。



命令模式降低了耦合，命令的使用者不再需要了解系统底层的逻辑，只需要使用包装好的命令。

7.13 模板模式

定义一个算法的骨架，把一些操作延迟到子类中进行。



7.14 空对象

一个方法返回 NULL，意味着方法的调用端需要去检查返回值是否是 NULL，这么做会导致非常多的冗余的检查代码。并且如果某一个调用端忘记了做这个检查返回值，而直接使用返回的对象，那么就有可能抛出空指针异常。

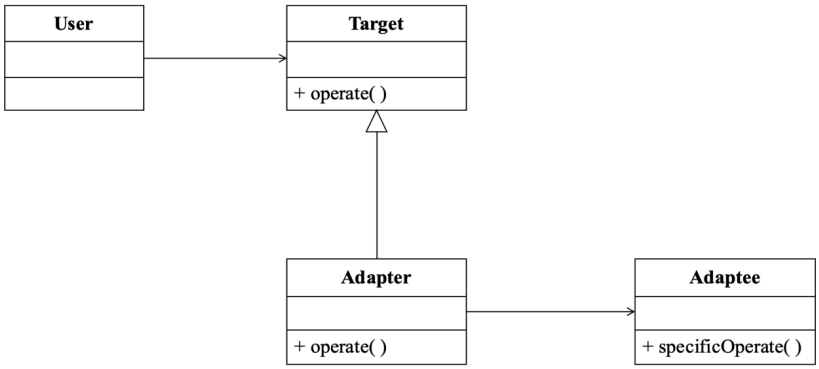
7.15 Mixin模式

还不太理解

7.16 适配器模式

定义一个转换器，将一个类的接口转换成用户需要的另一个接口

比如要修改数据格式。



7.17 依赖注入

对象声明自己的依赖，而该依赖由外部注入的形式为其提供

可以降低耦合度（如果自己里面再声明一个类，耦合度太高了）

8 第八章 代码风格和编码规范

防御式编程：数组越界、空指针、函数参数、返回值、外部接口。

考虑到输入的不确定性，在程序代码的主要逻辑之外增加了大量的检查，包括对所有外来数据的检查、对所有输入参数的检查等，并最终决定如何处理不符合预期的输入。

理解构成良好代码风格的主要因素：

1. 命名—符合语义（是什么、做什么、易于识别）
2. 注释—合理使用序言式注释与功能性注释
3. 视觉组织—清晰的缩进、空格与换行
4. 语句构造—避免太长的子程序、嵌套、语句，避免一行多语句
5. 输入输出—检查输入输出、简化输入输出步骤、合理的输入输出提示、为输出添加注释
6. 避免“魔法”—何为魔法串/数字

9 第九章 面向对象测试

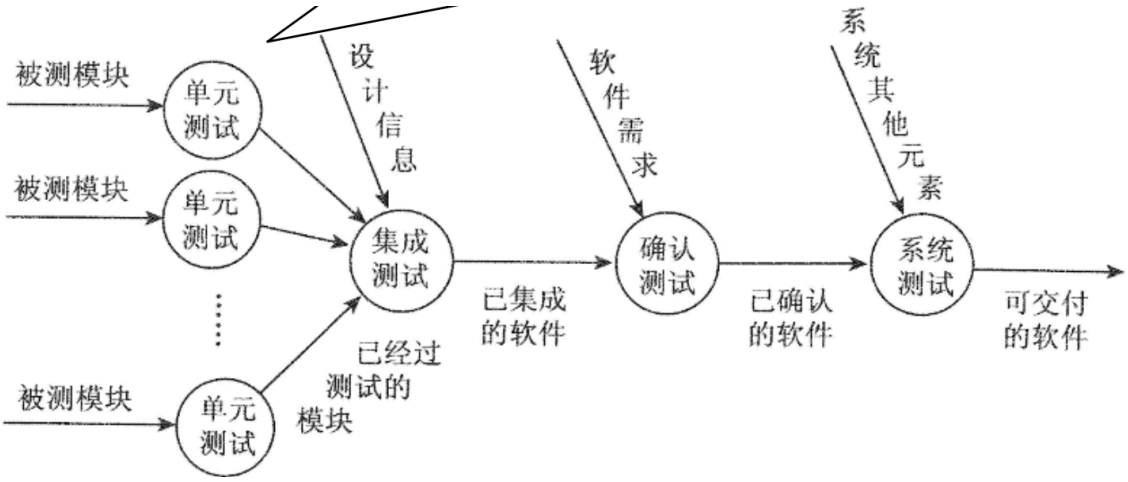
9.1 软件测试的步骤

1. 单元测试：集中于每个独立的模块。该测试以详细设计文档为指导，测试模块内的重要控制路径。（往往采用白盒测试技术）

2. 集成测试：集中于模块的组装。其目标是发现与接口有关的错误，将经过单元测试的模块构成一个满足设计要求的软件结构。

3. 确认测试（有效性测试）：目标是发现软件实现的功能与需求规格说明书不一致的错误。（通常采用黑盒测试技术）

4. 系统测试：集中检验系统所有元素（包括硬件、软件）之间协作是否合适，整个系统的性能、功能是否达到。



9.1.1 单元测试

1. 主要依据详细设计说明书和源代码清单

2. 主要测试模块的I/O条件和模块的逻辑结构

3. 主要采用白盒测试方法设计测试用例，辅以黑盒测试的测试用例，使之对任何合理的和不合理的输入都要鉴别和响应

4. 要对所有的局部和全局数据结构、外部接口和程序代码的关键部分进行代码审查

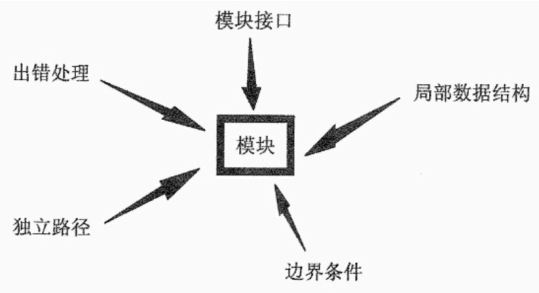


图2 单元测试的内容

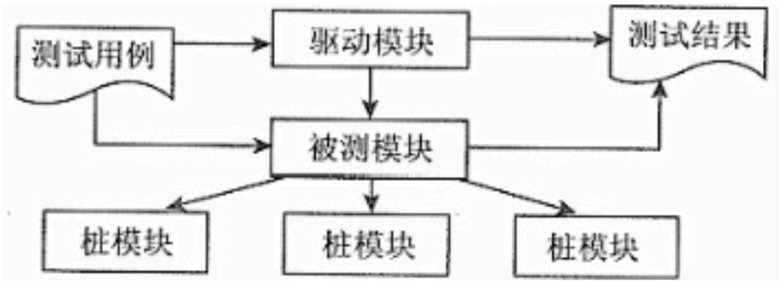


图3 单元测试的测试环境

9.1.2 集成测试

1. 把各个模块连接起来的时候，穿越模块接口的数据是否丢失

2. 一个模块的功能是否对另一个模块的功能产生不利的影响

3. 各个子功能组合起来，能否达到预期要求的父功能

4. 全局数据结构是否有问题

5. 单个模块的误差累加起来，是否达到不可接受的程度

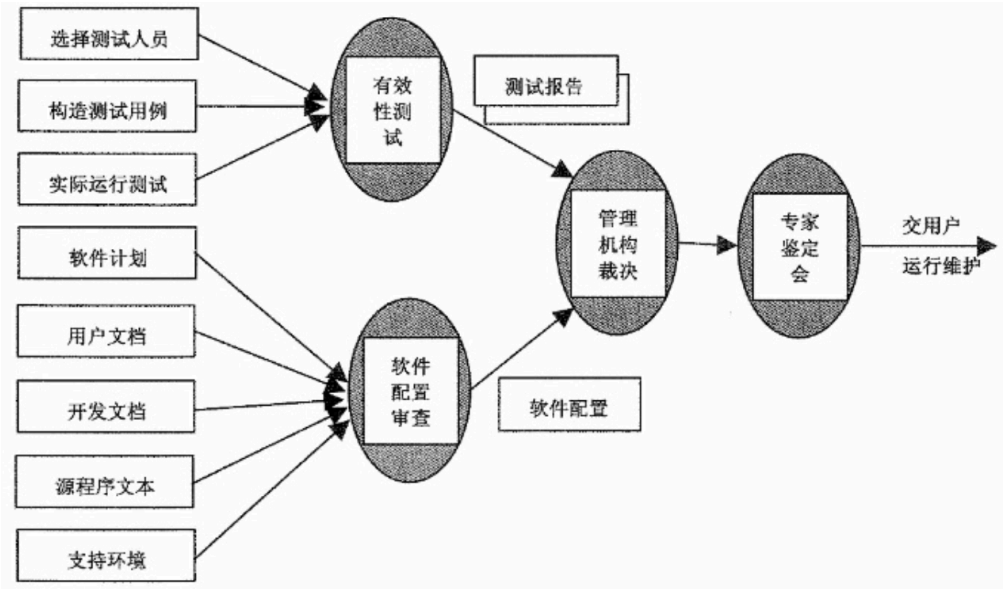
集成测试的方法：一次性组装方法、增量式组装方法

9.1.3 确认测试

有效性测试，即验证软件的功能和性能及其他特性是否与用户的要求一致

有效性测试是在模拟的环境（可能是开发环境）下，运用黑盒测试的方法，验证被测软件是否满足需求规格说明书列出的需求。

软件配置审查是保证软件配置的所有成分齐全，各方面的质量符合要求，具有维护阶段所必须的细节，并且已编排好分类的目录。



9.1.4 系统测试

集中检验系统所有元素（包括硬件、信息等）之间协作是否合适，整个系统的性能、功能是否达到。

系统测试实际上是一系列不同的测试，以下是用于系统测试的几种典型软件系统测试：

1. **功能测试**：在规定的时间内运行软件系统的所有功能，以验证这个软件系统有无严重错误。
2. **恢复测试**：是一种系统测试，它指采取各种人工干预方式强制性地使软件出错，使其不能正常工作，进而检验系统的恢复能力
3. **安全性测试**：就是试图去验证建立在系统内的预防机制，以防止来自非正常的侵入
4. **强度测试**：是在非正常数量、频率或容量资源方式下运行一个系统 （已经不正常了）
5. 压力测试：系统正常运行的最低限度
6. **性能测试**：测试软件在被组装进系统的环境下运行时的性能 （还是正常的）
7. **可用性测试**：从使用的合理性、方便性等角度对软件系统进行检验，以发现人为因素或使用上的问题
8. **部署测试（配置测试）**：软件必须在多种平台及操作系统环境中运行。有时将部署测试称为配置测试，是在软件将要在其中运行的每一种环境中测试软件。另外，部署测试检查客户将要使用的所有安装程序及专业安装软件，并检查用于向最终用户介绍软件的所有文档。

9.2 面向对象的测试策略

9.2.1 面向对象单元测试

也就是**类测试**。面向对象软件的类测试相当于传统软件中的单元测试，**类包含的操作是最小的可测试单元**

功能性测试：以类的规格说明为基础，主要检查类是否符合规格说明的要求。功能性测试包括两个层次：**类的规格说明和方法的规格说明**

结构性测试：从程序出发，对类中的方法进行测试，需要考虑其中的代码是否正确。测试分为两层 ：第一层考虑类中各独立的方法，即方法要做单独测试；第二层考虑方法之间的相互作用，即方法需要进行综合测试

基于状态的测试：基于状态的测试是通过检查对象的状态在执行某个方法后是否会转移到预期状态的一种测试技术；跟踪监视对象数据成员的值的变化。

9.2.2 面向对象集成测试

基于线程的测试（thread-based testing）：对响应系统的一个输入或一组类进行集成，每个线程单独地集成和测试事件所需的，应用回归测试以确保没有产生副作用

基于使用的测试(use-based testing)：通过测试很少使用服务类的那些类（称之为独立类）开始构造系统，独立类测试完后，利用独立类测试下一层次的类（称之为依赖类）。继续依赖类的测试直到完成整个系统

簇测试（cluster testing）是面向对象软件集成测试中的一步 ：利用试图发现协作中的错误的测试用例来测试协作的类簇

9.2.3 面向对象系统测试

功能测试、强度测试、性能测试、安全测试、回复测试、可用性测试、安装/卸载测试

9.3 面向对象软件的测试模型

<table>
<tr><td colspan="3">OO System Test
（面向对象系统测试）</td></tr>
<tr><td></td><td colspan="2">OO Integrate Test
（面向对象集成测试）</td></tr>
<tr><td></td><td></td><td>OO Unit Test
（面向对象单元测试）</td></tr>
<tr><td>OOA Test</td><td>OOD Test</td><td>OOP Test</td></tr>
<tr><td>OOA</td><td>OOD</td><td>OOP</td></tr>
</table>

9.3.1 OOA Test

对认定的**对象**的测试:

1. 测试认定的对象是否全面，是否问题空间中所有涉及的实例都反映在认定的抽象对象中
2. 测试认定的对象是否具有多个属性，**只有一个属性的对象通常应看成其他对象的属性，而不是抽象为独立的对象**
3. 测试被认定为同一对象的实例是否具有**区别于**其他实例的共同属性
4. 测试被认定为同一对象的实例是否提供或者需要相同的服务。**如果服务随着不同的实例而变化，那么认定的对象就需要进行分解或者继承来分类表示**
5. 认定的对象的名称应该尽量准确、适用

结构分为两类：分类结构（一般-特殊）和组装结构（整体-部分）

对认定的**分类结构**的测试。

对认定的**组装结构**的测试。

对定义的属性和实例关联的测试

对定义的服务和消息关联的测试

9.3.2 OOD Test

对认定的类的测试

对构造的类层次结构的测试

对类库支持的测试

9.3.3 OOP Test

数据成员是否满足数据封装的要求

类是否实现了要求的功能

7 软件测试

软件测试：使用人工或自动手段，运行或测定某个系统的过程，其目的是检验它是否满足规定的需求，或是清楚了解预期结果与实际结果之间的差异。

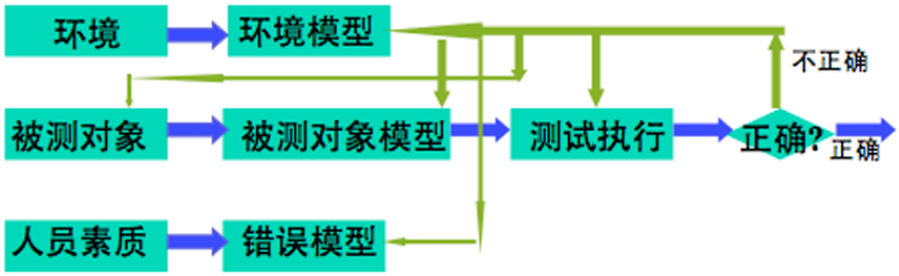
软件测试可分为静态分析和动态测试。

软件调试：发现所编写软件中的错误，确定错误的位置并加以排除，使之能由计算机或相关软件正确理解与执行的方法与过程。

软件测试和调试的主要区别

1. 测试从一个侧面证明程序员的“失败”，而调试是为了证明程序员的正确。
2. 测试以已知条件开始，使用预先定义的程序，且有预知的结果，不可预见的仅是程序是否通过测试。调试一般是以不可知的内部条件开始，除统计性调试外，结果是不可预见的。
3. 测试是有计划的，并要进行测试设计；而调试是不受时间约束的。
4. 测试是一个发现错误、改正错误、重新测试的过程；而调试是一个推理过程。
5. 测试的执行是有规律的，而调试的执行往往要求程序员进行必要推理以至知觉的“飞跃”。
6. 测试经常是由独立的测试组在不了解软件设计的条件下完成的；而调试必须由了解详细设计的程序员完成。
7. 大多数测试的执行和设计可由工具支持，而调试时，程序员能利用的工具主要是调试器。

测试过程模型：



错误（error）是指“与所期望的设计之间的偏差，该偏差可能产生不期望的系统行为或失效”。

故障（fault）是指“导致错误或失效的不正常的条件”。故障可以是偶然性的或是系统性的。

失效（failure）是指“与所规约的系统执行之间的偏差”。失效是系统故障或错误的后果。

error → fault → failure。

白盒测试又称结构测试或逻辑驱动的测试，该测试技术中，程序的内部实现逻辑对于测试人员是透明的，更关注程序的逻辑和结构。

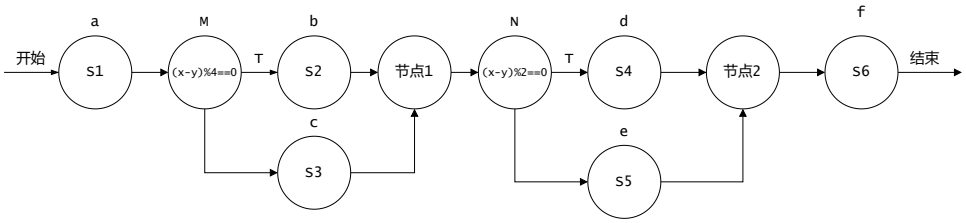
黑盒测试也称功能测试或数据驱动测试，它是在已知产品所应具有的功能，通过测试来检测每个功能是否都能正常使用。

白盒测试的主要方法有语句覆盖、分支覆盖、条件覆盖、分支-条件覆盖、条件组合覆盖、路径覆盖等，一般用于单元测试和集成测试中。

7.1 白盒测试技术

依据程序的逻辑结构的测试

控制流程图（又称控制流图）：一种表示程序运行逻辑的有向图，用有标记的圆表示一个或多个语句、决策条件以及程序过程等，边表示程序控制流。其基本元素是节点、判定、过程块。



1. 路径测试（PX）：执行所有可能的穿过程序的控制流程路径。
2. 语句测试（P1）：至少执行程序中所有语句一次。如果遵循这一规定，则我们说达到了 100% 语句覆盖率（用 C1 表达）。语句覆盖是最弱的逻辑覆盖准则。
3. 分支测试（P2）：至少执行程序中每一分支一次。如果遵循这一规定，则我们说达到了 100% 分支覆盖率（用 C2 表示）。
4. 条件组合测试，就是设计足够的测试用例，使每个判定中的所有可能的条件取值组合至少执行一次。如果遵循这一规定，则我们说就实现了条件组合覆盖。只要满足了条件组合覆盖，就一定满足分支覆盖。

语句覆盖是最弱的逻辑覆盖准则，发现不了判断中逻辑运算符出现的错误；分支覆盖比语句覆盖稍强，但它未必能发现每个条件的错误；条件组合覆盖比分支覆盖更强，只要满足了条件组合覆盖就一定满足分支覆盖；路径覆盖是最强的，但一般是不可实现的。

7.2 黑盒测试（功能测试）技术

依据软件行为的描述的测试

1. 等价类：输入域的一个子集，在该子集中，各个输入数据对于揭示程序中的错误都是等效的。即：以等价类中的某代表值进行的测试，等价于对该类中其他取值的测试。
2. 有效等价类：指那些对于软件的规格说明书而言，是合理的、有意义的输入数据所构成的集合。
3. 无效等价类：指那些对于软件的规格说明书而言，是不合理的、无意义的输入数据所构成的集合。

设计无效等价类的时候仅包括一个未被覆盖的无效等价类。

因为某些程序中对某一输入错误的检查往往会屏蔽对其它输入错误的检查。因此设计无效等价类的测试用例时应该仅包括一个未被覆盖的无效等价类。

边界值分析是一种最常用的黑盒测试技术。

7.3 非功能性测试

1. 压力测试是指测试软件系统短时间内超过系统预期负载时系统的运行情况。
2. 安全测试：通过软件的攻防技术，即模拟可能存在的软件攻击，以此发现软件的漏洞，进而修补漏洞，从而保证软件的安全。
3. 灾难恢复测试主要测试系统遭遇不可抗力系统崩溃后，系统恢复的能力。
4. 兼容性测试主要验证软件与其所处环境的兼容情况，其中软件与部署环境之间的不兼容性问题尤为频发。

7.4 软件测试步骤

1. 单元测试（往往采用白盒测试技术）：集中于每个独立的模块。该测试以详细设计文档为指导，测试模块内的重要控制路径。
2. 集成测试：集中于模块的组装。其目标是发现与接口有关的错误，将经过单元测试的模块构成一个满足设计要求的软件结构。
3. 有效性测试（确认测试）：目标是发现软件实现的功能与需求规格说明书不一致的错误。（通常采用黑盒测试技术）
4. 系统测试：集中检验系统所有元素（包括硬件、软件）之间协作是否合适，整个系统的性能、功能是否达到。
   (a) 功能测试：是最基本的系统测试。可以通过采用自动化测试框架或编写功能测试脚本来完成。当前的功能测试已经进入自动化阶段。
   (b) 恢复测试：采取各种人工干预方式强制性地使软件出错，使其不能正常工作，进而检验系统的恢复能力。
   (c) 安全性测试：试图去验证建立在系统内的预防机制，以防止来自非正常的侵入。
   (d) 强度测试：在非正常数量、频率或容量资源方式下运行一个系统。目的是检查在系统运行环境不正常乃至发生故障的情况下，系统可以运行到何种程度的测试。
   (e) 性能测试：测试软件在被组装进系统的环境下运行时的性能。
   (f) 可用性测试：从使用的合理性、方便性等角度对软件系统进行检验，以发现人为因素或使用上的问题。
   (g) 部署测试（配置测试）：软件必须在多种平台及操作系统环境中运行。有时将部署测试称为配置测试，是在软件运行的每一种环境中测试软件。

8 软件项目管理

软件项目管理是为了使软件项目能够顺利完成，而对成本、人员、进度、质量、风险等进行分析和管理的活动。

软件项目管理的四要素（4P）：人员（People）、产品（Product）、过程（Process）和项目（Project）。

项目管理的四大核心知识领域是指范围、时间、成本和质量。这四个方面会形成具体项目的项目目标。

五大项目管理辅助知识领域包括人力资源管理、风险管理、沟通管理和采购管理、干系人管理。之所以称其为辅助知识领域，是因为项目目标是通过他们来实现的。

项目整体管理包括在项目生命周期中协调所有其他项目管理知识领域所涉及的过程。它确保项目所有的组成要素在正确的时间结合在一起，以成功地完成项目。

项目整体管理所包括的几个主要过程有：项目计划制定、项目计划执行、整体变更控制。

9 软件集成、交付与部署

软件集成是将各个软件子系统整合成一个大系统的过程。

软件交付是在软件集成的基础上，处理生产部署所需的后续阶段，交付不一定需要最终部署。文档交付、源代码交付、可执行程序交付。

软件部署是使软件系统启动运行的过程，包括如下阶段：发布、安装和激活、停用、卸载、更新、内置更新、版本追踪。

持续集成 (CI, Continuous Integration)

持续交付 (CD, Continuous Delivery)

持续部署 (CD, Continuous Deployment)

自动化 + 非常频繁。

10 软件维护、演化与再工程，软件质量

软件维护：软件产品在交付之后，为改正错误、改进性能或其他属性，或者为了适应变化了的环境而对软件产品所进行的修改活动就是软件维护。

软件维护可分为四种类型：改正性维护、适应性维护、完善性维护和预防性维护。

在维护工作流程正式开展前，需要由申请维护的用户填写软件维护申请报告；维护工作流程结束后维护组织需要纪录维护档案与进行维护评价。维护工作流程：确认维护要求 → 评价错误严重性（改正性维护）或确定维护申请优先次序（其它维护）→ 维护实施（包括修改软件需求说明、修改软件设计、设计评审、修改源程序、测试）→ 情况评审

软件演化是软件产品交付给客户之后所发生的一系列功能增强和结构改进活动，以满足变化的软件需求。

软件维护 vs 软件演化：

软件维护：工作是根据用户反馈修复软件的某些缺陷或者添加某些细小的功能特性等，通常不会发布一个新的版本。

软件演化：通常发生在软件演化范畴的软件修改活动都是较大粒度的功能更新和软件结构变更。

软件再工程过程中的活动：库存目录分析、文档重构、逆向工程、代码重构、数据重构、正向工程

正向工程 vs 预防性维护

正向工程：重新构建现有的软件系统，属于“再工程”的一环

预防性维护：对需要维护的软件或软件中的某一部分（重新）进行设计、编制和测试，属于“维护”范畴

软件质量的最一般定义：有效的软件过程、有用的产品、同时为软件生产者和使用者增值。

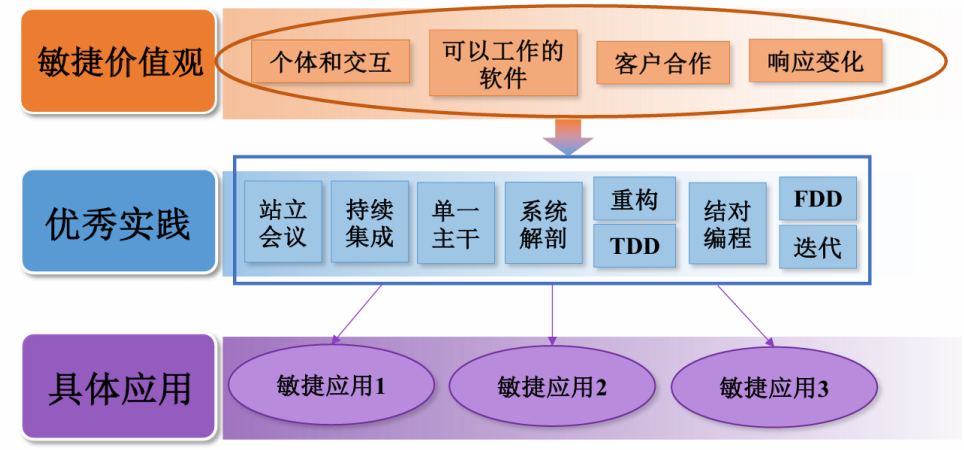
评价软件质量的国际标准，由 6 个关键的软件质量属性组成：功能性、可靠性、易用性、效率、维护性、可移植性。

提高软件质量：软件工程方法、项目管理技术、质量控制、质量保证。

11 敏捷开发方法

敏捷方法是一种专注快速开发的增量式开发，频繁地发布软件、降低过程开销、生产高质量的代码。用户支持参与到开发过程中。

个体和交互胜过过程和工具可以工作的软件胜过面面俱到的文档客户合作胜过合同谈判响应变化胜过遵循计划



极限编程是一种著名的敏捷方法，它集成了一系列的好的编程经验，如频繁地发布软件、连续地改善软件和让客户参与到软件开发团队中。

Scrum 是一种敏捷软件过程模型，它的核心是一组冲刺循环，开发一个系统增量有固定的时间周期。规划是基于积压的工作的优先权安排的，选择高优先权的任务开始冲刺循环。极限编程的一个特别长处是在创建程序特征之前开发自动测试，在增量集成进系统的时候所有的测试必须成功执行。

12 代码风格与编程规范

12.1 防御式编程

1. 检查程序的所有外部输入数据的值是否在允许的范围内。
2. 检查数组访问是否存在越界问题。
3. 检查迭代器是否合法。
4. 使用指针、引用前，检查他们是否为空。
5. 检查函数的参数类型是否准确、参数值是否在允许的范围内。
6. 检查函数的返回值是否在允许的范围内。

12.2 代码风格

1. 命名—符合语义（是什么、做什么、易于识别）。
2. 注释—合理使用序言式注释与功能性注释。
3. 视觉组织—清晰的缩进、空格与换行。
4. 语句构造—避免太长的子程序、嵌套、语句，避免一行多语句。
5. 输入输出—检查输入输出、简化输入输出步骤、合理的输入输出提示、为输出添加注释。
6. 避免“魔法”—何为魔法串/数字。

13 24-25 秋季学期期末考试

十个选择题，十个判断题，一个简答题，一个分析题，一个考软件测试的，一个建模题。
考了 CMM 的已定义级。
考了哪种软件开发模型的特点是需求的不稳定性。
要求写出接口的定义并举例说明。
要求写出依赖的定义并举例说明。
要求写出 OOD 的五个部分并说明分别是什么。
考了 pull request 是属于单元测试的吗。
考了软件维护有哪几种类型的。
是否需要在需求分析完之后就完成有效性测试的编写计划。
要求建立被测对象的模型，写出条件覆盖的测试用例。
建模题给出了一个简化的外卖预订系统，有顾客、卖家、交易中心和外卖小哥四种角色。
要求建立用况图、顶层 DFD、类图、选择一个交互建立顺序图、钱款的交互的活动图。