

软件工程



孙吉鹏 谷一滕

杜泽林 张晓敏

林子童 徐卫霞

袁郭苑 鲍 伟

特别鸣谢

史清华 老师

排版：谢颖

版本号：V1.1

修订时间：2018.12



山软智库-让知识回归平凡

写在前面

当我们开始整理这门跟我们专业名字一模一样的课程时，我们的心情真是很复杂：一方面不知不觉我们已经学到了我们专业的本命课程，这个大一时我们看课程计划觉得无比重要的课，倍感岁月不饶人；

另一方面，当我们学完这学期再审视这门课，又觉得它像Apple的一句广告：

“说简单，也高深”

说它简单，觉得它在一开始复习时恍惚有种背政治的感觉，比起烧脑的程序设计和巧妙计算来说真的很轻松；

说它高深，是因为如果让所有就业的软件工程毕业生投票自己最后悔没好好学的课，它一定是champion。

当你抱着一颗佛系的心看这些知识时，你会觉得它那么教条；但当你真的准备干点大事带着挚友们开发个软件时，你会觉得它说的是那么针针见血：

如果你是负责人，你如何知道你需要什么招什么样的人来组成一个开发团队？你就是有了合适的团队你靠什么告诉客户你会花多长时间多少钱去完成这个项目才让他们放心把项目交给你？你就是拿到了项目你怎么让客户客观理性准确无误地告诉你他们的需求是什么？你就是得到了明确无疑的需求你怎么确定为了实现这个你该怎么设计这些功能？你就是得到了一份巧夺天工的设计你怎么保证你的程序员们能高效合作地把这些设计准确实现出来？你就是有了一批勤恳踏实的程序员写完了这些代码你怎么保证它们功能上没有错误？你就是所有功能都完美实现了你怎么保证它们合在一起还能干活？你就是把功能们合在一起天衣无缝你怎么交付给客户使用这个系统？你就是让客户心甘情愿地收下了这个系统你怎么保证用户用起来出了意外怎么维护？你就是维护好了这个系统你怎么迭代增量开发下一个系统让团队不会沦为临时工？.....

我们团队在开发智库时切实感受到了这些问题，而这些问题在这门课中都有答案，于是我们想带大家带着需求重新审视一下这门课程，希望通过知识让亲爱的朋友，你，不会在这里留下惘然的遗憾；当然如果你追求分数，也能保证你通过这些知识可以心想事成。

OK，让我们开始吧！

目录

第一章 软件工程概述.....	6
一、章节概述：.....	6
二、章节框架：.....	6
三、知识点解析：.....	7
1.1 什么是软件工程.....	7
1.2 软件工程取得了哪些进展.....	8
1.3 什么是好的软件.....	9
1.4 软件工程涉及的人员.....	9
1.5 系统的开发方法.....	9
1.6 工程化的方法.....	10
1.7 开发团队的成员.....	11
1.8 软件工程发生了多大的变化.....	11
1.9 章末案例思考.....	13
四、典型例题：.....	13
第二章 过程和生命周期的建模.....	14
一、章节概述.....	14
二、章节框架.....	14
三、知识点解析.....	14
2.1 过程与生命周期.....	14
2.2 过程模型.....	15
2.3 过程建模工具和技术.....	23
第三章 计划和管理项目.....	24
一、章节概述.....	24
二、章节框架.....	24
三、知识点解析.....	25
3.1 跟踪项目进展.....	25
3.2 项目人事组织.....	27
3.3 工作量估算.....	28
3.4 风险管理.....	29
3.5 项目计划.....	29
四、典型例题.....	30
第四章 获取需求.....	31
一、章节概述：.....	31
二、章节框架：.....	31
三、知识点解析：.....	32
4.1 需求的过程.....	32
4.2 需求的引出.....	32
4.3 需求的类型.....	33
4.4 需求的特征.....	34
4.5 需求的表示.....	35

3.6 需求文档化.....	42
3.7 需求确认.....	42
3.8 测量需求.....	42
四、典型例题:	43
第五章 设计体系结构.....	44
一、章节概述.....	44
二、章节框架.....	44
三、知识点解析:	45
5.1 设计过程.....	45
5.2 设计过程中的若干问题.....	46
5.3 模块独立性.....	48
5.4 体系结构的评估和改进.....	50
5.5 软件产品线.....	51
第六章 考虑对象.....	52
一、章节概述.....	52
二、章节框架.....	52
三、知识点解析.....	53
6.1 什么是面向对象 (OO)	53
6.2 *OO 开发过程.....	53
6.3 用例模型.....	55
6.4 OO 的表示---使用 UML 的一个样板.....	56
6.5 OO 系统设计.....	58
6.6 OO 设计原则.....	61
6.7 其他的 UML 图.....	61
6.8 OO 程序设计中的其他问题.....	63
四、典型例题:	63
第七章 编写程序.....	64
一、章节概述.....	64
二、章节框架.....	64
三、知识点解析.....	65
7.1.1 编程标准对自身的作用.....	65
7.1.2 编程标准对他人的作用.....	65
7.1.3 设计与编程实现相匹配.....	65
四、典型例题.....	67
第八章 测试程序.....	68
一、章节概述.....	68
二、章节框架.....	68
三、知识点解析.....	68
8.1 故障和失效.....	68
8.2 测试之前应该明确的几个问题.....	70
8.3 单元测试.....	71
8.4 集成测试——得到一个正常运作的系统。.....	73
8.5 测试面向对象的系统.....	74
8.6 测试计划.....	75

8.7 补充.....	75
四、例题.....	75
第九章 测试系统.....	77
一、章节概述.....	77
二、章节框架.....	77
三、知识点解析.....	78
9.1 系统测试综述.....	78
9.2 系统配置.....	78
9.3 回归测试.....	80
9.4 功能测试.....	81
9.5 因果图.....	81
9.6 性能测试.....	81
9.7 验收测试.....	82
9.8 安装测试.....	83

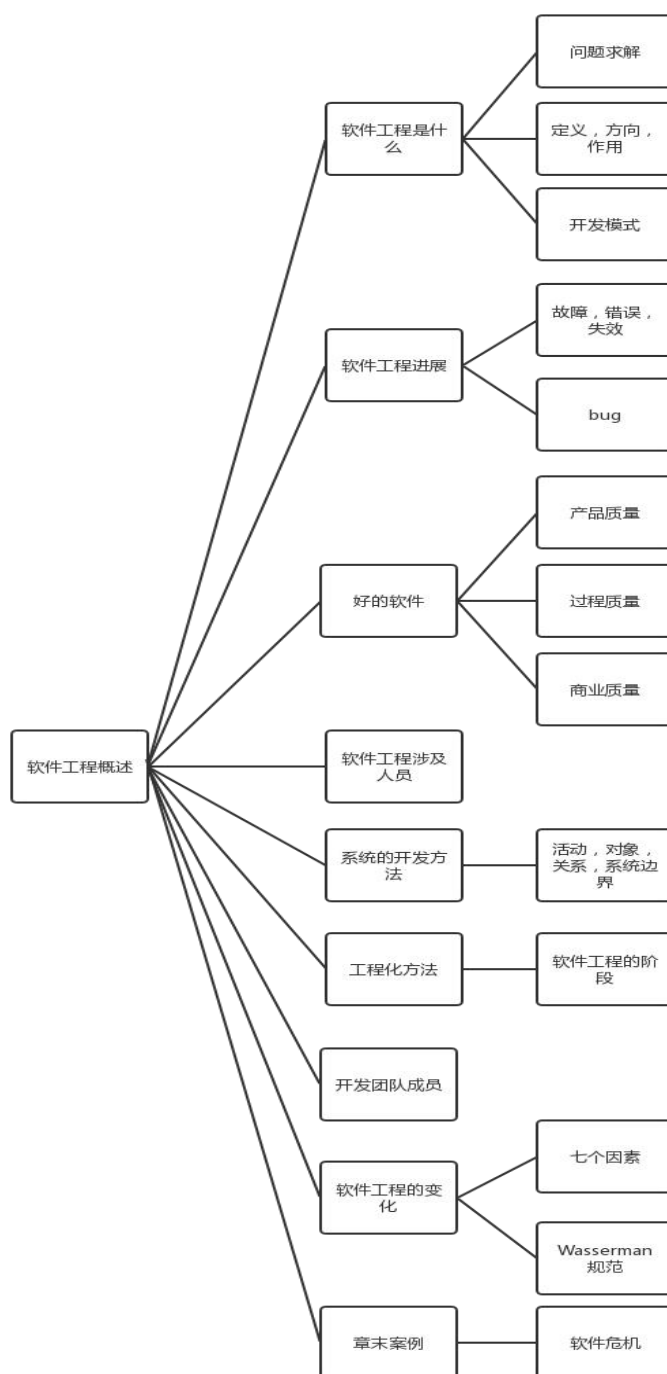
第一章 软件工程概述

作者 谷一滕

一、章节概述：

这一章介绍软件工程的发展历程，其所使用的技术及工具；如何分析问题以及寻求解决方案；软件开发人员们取得的进展以及需要努力的方向；软件开发的涉及人员及 Wasserman 规范将实践融为一体的八个概念。

二、章节框架：



三、知识点解析：

1.1 什么是软件工程



软件工程涉及的就是“软件开发的规范”，在规范里面体现出内在的思想与软件工程之规律。

1.1.1 问题求解

分析(analysis)：将问题分解成可以理解并能够处理的若干小部分，确定问题的本质含义。

合成(synthesis)：将每个小问题的解决方案组合成一个大的结构，合成解决方案。

工具(tool)：用更好的方式完成某事情的设备或自动化系统。

过程(produce)：把工具和技术结合起来，共同生产特定产品。

范型(paradigm)：构造软件的特定方法、途径或哲学（如面向对象开发的模式、结构化开发的模式、基于过程开发的模式、某种订制开发的模式）。

1.1.2 软件工程师的角色是什么

以计算机科学理论和计算机功能为基础，通过对要解决问题的本质的了解，采用相应的工具和技术，实现设计方案，推出高质量的软件产品。（将计算机作为问题求解的工具）

1.1.3 软件工程（SE）的定义、方法、作用：

SE：在将有关软件开发与应用的概念科学体系化的基础上，研究如何有计划、有效率、经济地开发和利用能在计算机上正确运行的软件理论和技术工程的方法学，以及一些开发和维护软件的方法、过程、原则等。它是一个系统工程，既有对技术问题的综合分析，也有对开发过程和参与者的管理。

SE 的方法：面向对象模式，结构化模式，基于过程的模式等。

SE 的作用：付出较低的开发成本，达到要求的软件功能，取得较好的软件性能，开发的软件易于移植，需要较低的维护费用，能按时完成开发工作，及时交付使用。

1.1.4 开发模式

软件开发的全部过程，活动和任务的结构框架，它能直观的表达软件开发全过程，明确要完成的主要活动，任务和开发策略。

1.2. 软件工程取得了哪些进展

1.2.1 故障，错误和失效各自的含义（含举例）及它们之间的联系：

错误(error)：是在软件开发过程中人为产生的错误（需求说明中的错误，代码中的错误）。

故障(fault)：软件功能实现过程中产生的问题，是错误导致的结果，是软件中一个错误的表现（一个错误可能产生多个故障，静态存在）。

失效(failure)：系统违背了它应有的行为（在系统交付前或交付后被发现，动态存在）。

联系：人为原因导致程序错误；该错误编译到系统中导致系统故障；用户使用该系统时，因故障导致失效。故障是系统内部视图，从开发者的角度看待问题；失效是系统外部视图，从用户角度看到的问题。而且并不是所有的故障会导致失效，只要不执行故障代码，或者不进入某个特定状态，那么故障就不会使代码失效。

1.2.2 零缺陷软件

由于市场压力促使软件开发人员快速交付产品，零故障无法实现。

1.2.3 关于 bug

改正(fixing)有时比重写(rewriting)整个系统要困难。

错误修正的越晚，付出的代价越大。

复审(review)十分重要，是正式团队的行为规范；自己检查只能找出开发阶段故障的 1/5，同行评审能够揭示其余 4/5 的故障。

1.3 什么是好的软件

从三个方面考虑软件的质量：产品的质量、生产该产品的过程的质量以及在产品将使用的商业环境背景下的质量。

1.3.1 产品(product)的质量

用户：从失效的数目和类型等外部特性进行评价，如果软件具有足够的功能，并且易于学习和使用；或者虽然难以学习和使用，但是由于功能值得这些付出，用户就断定软件是高质量的。

开发者：从故障的数目和类型等内部特征来作为产品质量的依据。

1.3.2 过程(process)的质量

有很多过程都会影响到最终的产品质量，只要有活动出了差错，产品的质量就会受到影响；开发和维护过程的质量与产品的质量是同等重要的。

几个量化模型：CMM、ISO 9000、SPICE（了解）

1.3.3 商业(business)环境背景下的质量

(1) 技术价值与商业价值的联系与区别：

技术价值：技术指标（速度，正确的运行时间，维护成本等）。

商业价值：机构对软件是否与其战略利益相吻合的一种价值评估。

误区：技术质量不会自动转化为商业价值。

(2) 目标

将技术价值和商业价值统一起来，改进过程所带来的商业价值。

1.4 软件工程涉及的人员

客户(customer): 为将要开发的软件系统支付费用的公司、组织或个人。

开发者(developer): 为客户构建软件系统的公司、组织或个人。

用户(user): 实际使用系统的人。

有时, 他们可能是同一个人或同一组人。

1.5 系统的开发方法

要开发一个项目, 必须知道系统包含哪些对象或活动。

系统(system): 实体、活动、关系、系统边界的集合。

1.5.1 软件系统的系统要素(组成) (对象(实体) + 活动 + 关系 + 系统边界)

(1) 活动和对象

活动(activity): 活动是发生在系统中的某些事情, 通常描述为由某个触发器引发的事件, 活动通过改变某一特性把一个事物转变成另一个事物。

对象(object)或实体(entity): 活动中涉及的元素称为对象或实体(如记录数据的对象)。



(2) 关系和系统边界

关系(relationship): 对实体和活动间数据项及动作相互关系的描述。

系统边界(system boundary): 用于描述系统中包含什么, 不包含什么。

1.5.2 相互联系的系统

内容概述: 几乎不存在与其他系统没有关联的系统, 因此刻画系统边界十分重要, 很容易了解什么在系统内部、什么不在以及什么超出了边界。系统可能存在于另一个系统中。边界定义的详细正确, 那么根据较小的部分构建较大的系统是相对容易的。开发可以由内而外, 但设计最好得由大到小, 这就带来了难度。

◆增量式开发方法

包含一系列阶段, 其中每一个阶段都使前面的系统不受当前系统约束的限制。系统逐渐地从旧的软件和硬件中脱离开, 直到它体现出新系统的设计。

1.6 工程化的方法

大型软件开发过程中的工程化途径与方法

1.6.1 构建系统

现代软件工程大致包含的几个阶段及各个阶段文档:

(1) 需求分析: 包括问题定义、可行性研究、需求分析【《SRS》即《软件需求规格说明书》】与复审(所有人)。

(2) 系统设计：包括用户界面的设计【《SAD》即《软件系统结构图》：如何制作软件】与复审（开发者与客户）。

(3) 程序设计：包括模块功能算法与数据描述设计【相关文档】与复审（开发者）。

(4) 程序实现：包括编程与 debug【源代码和注释】与复审（开发者、码农）。

(5) 单元测试：模块功能测试与性能测试【测试报告】与复审（测试团队）。

(6) 集成测试：按照结构图进行测试【测试报告】与复审（测试团队）。

(7) 系统测试：按《SRS》对系统总体功能进行测试与复审（开发者与客户）。

(8) 系统提交：交付产品【用户手册和操作手册】与复审。

(9) 系统维修：修改软件的过程，为改错或满足新需求【维修报告】与复审（维修团队）。

注：圆括号中的为测试人员，方括号为生成的文档。

额外说明（了解）：以上阶段只是大致的划分，软件团队实际执行时比这要复杂，还有若干辅助性过程和阶段，而有些阶段实际是相互重叠、相互影响的。

1.7 开发团队的成员

需求分析人员、设计人员、程序员、测试人员、培训人员、维护人员、资料管理人员、配置管理人员。

软件工程各阶段各自的工作：（了解）

需求设计（分析员、客户）：将客户想要的分解为离散需求。

系统设计（分析员、设计员）：生成系统层描述（系统要做什么）。

程序设计（设计员、程序员）：实现指定需求的代码。

程序实现（程序员）：编代码。

单元测试（程序员、测试员）：发现各种错误。

集成测试（测试团队）：检查系统功能。

系统测试（测试员、客户、培训员）：根据《SRS》检查要求。

系统交付（培训员）：培训用户。

系统维修（维修团队）：寻找故障，根据客户需求变化，对系统作出修改。

资料管理（资料管理员）：维持一个软件的不同版本之间各种文档的对应关系，包括需求规格说明、设计描述、程序文档、培训手册、测试数据进度等。

配置管理（配置管理员）：维护需求、设计、实现和测试之间的对应关系。

软件架构师：属于高级程序员，侧重开发过程和模式的选择和论证（在国内和分析员差不多，其工作重点与分析员有所不同，但就开发来说，其工作似乎更重要些，而分析员的工作更偏重于市场与用户需求）。

1.8 软件工程发生了多大的变化

1.8.1 变化的本质

早期程序：线性输入，输出为字母数字，系统设计方式分为两种：转换（transformation）：将输入转换为输出；事务(transaction)：由输入决定哪个功能将被执行。因此瀑布模型开发方式可行。

当今程序：多系统运行，跨平台运行，基础功能：网络控制，安全性，用户界面表示和

处理，以及数据或对象管理等。相较于早期变化巨大，瀑布模型不适用。

◆ 使现代软件工程实践发生变化的七个关键因素 (by Wasserman)

- (1) 商用产品投入市场时间的紧迫性
- (2) 计算技术在经济中的转变：更低的硬件成本，更高的开发、维护成本
- (3) 功能强大的桌面计算的可用性
- (4) 广泛的局域网和广域网
- (5) 面向对象技术的采用及其有效性
- (6) 使用窗口、图标、菜单和指示器的图形用户界面
- (7) 软件开发瀑布模型的不可预测性

说明（了解）：瀑布模型沿袭了传统系统工程的大规模批发制造的理念，假定生产活动为线性，这与现代软件的生产方式相矛盾。不再是有足够的灵活性和适应性来满足并行开发或并行运行这样的商业软件需求，因此不可预测。

结论（了解）：对一个系统进行划分，以便并行地开发其子系统，需要一个与瀑布模型有很大不同的开发模型。

1.8.2 软件工程的 Wasserman 规范（或基本概念）

(1) 抽象(abstraction)：基于某种层次归纳水平的问题描述。它使我们将注意力集中在问题的关键方面而非细节。

(2) 分析、设计方法和符号描述系统：

使用标准表示来对程序进行描述。利于交流，利于建模并检查其完整性和一致性，利于对需求和设计部件进行重用。

(3) 用户界面原型化(prototyping)：

建立系统的小型版，通常具有有限的关键功能，以利于用户评价和选择，证明设计或方法的可行性。

(4) 软件体系结构：定义一组体系结构单元及其相互关系集来描述软件系统。

单元分解的方法

（以下了解）

- (1) 基于功能的模块化分解：基于指派到模块的功能。
- (2) 基于数据的分解：基于外部数据结构。
- (3) 面向事件的分解：基于系统必须处理的事件。
- (4) 由外到内的分解：基于系统用户的输入。
- (5) 面向对象的设计：基于标识的对象的类以及它们之间的相互关系。

(5) 软件过程：软件开发活动中的各种组织及规范方法。

（以下了解）

因应用类型和组织文化之间的巨大差异，故难以对软件过程本身进行预先指定，也就是说：使过程本身规范化是不可能的。软件过程不可能以抽象和模块化的方式作为软件工程的基础。

(6) 重用或复用(reuse): 重复采用以前开发的软件系统中具有共性的部件, 用到新的开发项目中去 (注: 这里的重用绝不仅仅是源代码的重用)。

(7) 测度或度量(measurement): 通用的评价方法和体系, 有助于使过程和产品的特定特性更加可见, 包括量化描述系统、量化审核系统。

(8) 工具和集成环境: 通过框架比较软件工程环境提供的服务, 以决定其好坏。工具: 由于厂商很少针对整个开发生命周期, 因此对于工具的比较集中于小的活动集, 例如测试或设计。

(以下了解)

工具集成中必须处理的五个问题: (by Wasserman)

平台集成、表示继承、过程集成、数据集成、控制集成。

总结: 以上八个概念将软件工程作为一门科学学科, 也是本书的八个线索。

1.9 章末案例思考

什么是软件危机? 它有哪些典型表现? 为什么会出现软件危机?

软件危机: 落后的软件生产方式无法满足迅速增长的计算机软件需求, 从而导致软件开发与维护过程中出现一系列严重问题的现象。

典型表现:

- (1) 对软件开发成本和进度的估计常常很不准确。
- (2) 用户对“已完成”软件系统不满意的现象经常发生。
- (3) 软件产品的质量往往靠不住。
- (4) 软件常常是不可维护的。
- (5) 软件通常没有适当的文档资料。
- (6) 软件成本在计算机系统总成本中所占的比例逐年上升。
- (7) 软件开发生产率提高的速度, 远跟不上计算机应用迅速普及深入的趋势。

出现的原因: 一方面与软件本身的特点有关, 另一方面也和软件开发与维护的方法不正确有关。

- (1) 软件缺乏“可见性”, 管理和控制软件开发过程相当困难。
- (2) 软件规模庞大, 而且程序复杂性将随着程序规模的增加而呈指数上升。
- (3) 开发时期引入错误, 导致软件维护通常意味着改正或修改原来的设计, 客观上使得软件较难维护。
- (4) 软件专业人员对软件开发和维护中或多或少地采用了错误的方法和技术。

四、典型例题：

1. 软件工程的出现是由于（ ）。
- A. 软件危机的出现 B. 计算机硬件技术的发展
- C. 软件社会化的需求 D. 计算机软件技术的发展

答案：A

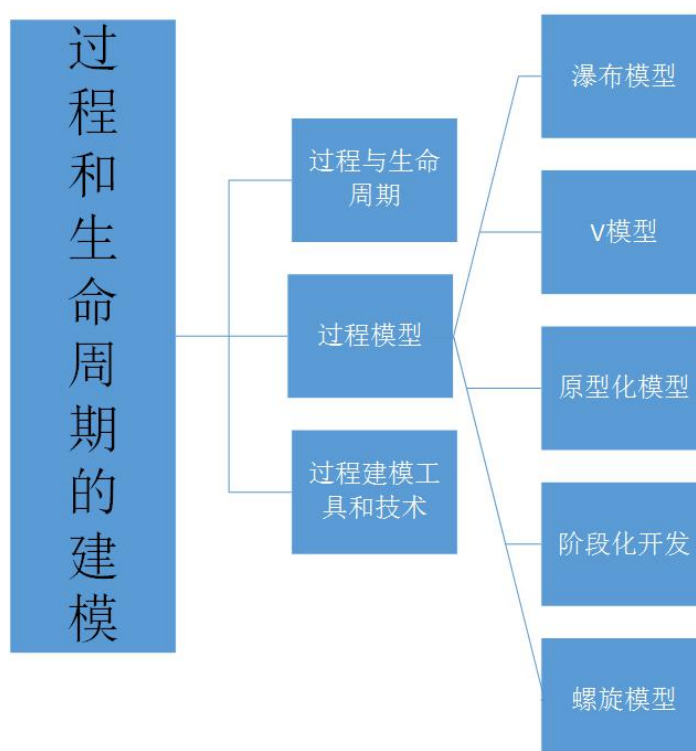
第二章 过程和生命周期的建模

作者 杜泽林

一、章节概述

这一章我们要详细分析软件工程中的各种步骤的组织方式，以便我们协调各种活动。面对的主要问题有如何理解过程模型，如何应对软件开发过程中的种种状况，为此我们首先要解释过程模型的意义，然后为大家介绍各种模型的思想，为了响应变化和规避风险，会涉及到原型化和迭代开发的思想。关键在于思想的理解，模型是固化的，思想运用是灵活的。本章重点在过程与生命周期和过程模型。

二、章节框架



三、知识点解析

2.1 过程与生命周期

2.1.1 过程的定义


一组有序的任务，它涉及活动、约束和资源使用的一系列步骤，用于产生某种想要的输出。

过程不仅仅是步骤，过程是步骤的集合，它将步骤组织起来使人们能够生产满足一系列目标和标准的产品。

2.1.2 软件生命周期

软件开发过程描述了软件产品从概念到实现、交付、使用和维护的整个过程，因此，有时把软件开发过程称为软件生命周期。

2.1.3 过程的重要意义

- (1) 它强制活动具有一致性和一定的结构。
- (2) 过程结构允许我们分析、理解、控制和改进组成过程的活动，并以此来指导我们的活动。 
- (3) 它使我们获取经验并把经验传授给他人。

2.2 过程模型

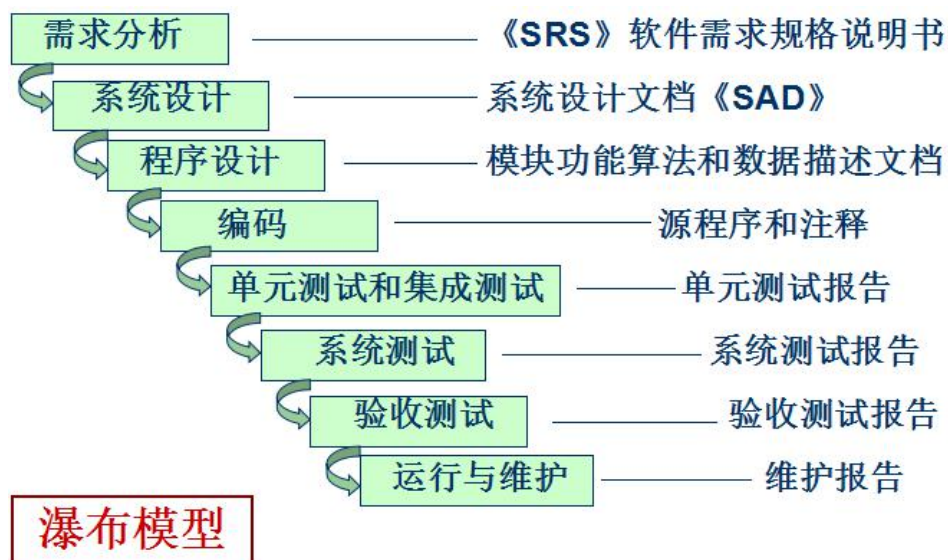
2.2.1 为何需要为过程建立模型：

- (1) 达成共识：开发团队在记录开发过程的描述时，自然的对软件所涉及到的活动，资源，约束等达成共识，这共识就是“模型”。
- (2) 发现缺陷：发现过程层面的缺陷(过程实施时的不一致性、多余部分、缺省部分、不完善部分)，从而让过程更有效。
- (3) 评价与优化：模型应该反映开发的诸多目标，并评价候选活动的有效性和正确性，以构建高质量软件。



2.2.2 瀑布模型

线性的安排每一个阶段，将开发阶段描述为从一个阶段瀑布般地转换到另一个阶段。一个开发阶段必须在另一个开发阶段开始之前完成。



2.2.1 瀑布模型的优点：

- (1) 它的简单性使得开发人员很容易向不熟悉软件开发的客户作出解释。
- (2) 每一个过程活动都有与其相关联的里程碑和可交付产品，以便于项目经理评估项目进度。
- (3) 瀑布模型是最基础的模型，很多其他更复杂的模型实际上是在瀑布模型的基础上的润色，如加入反馈循环以及额外的活动。

2.2.2 瀑布模型的缺点：

- (1) 除了一些理解非常充分的问题之外，实际上软件是通过大量的迭代进行开发的。软件是一个创造的过程，不是一个制造的过程。软件变动时，该模型无法处理实际过程中的重复开发问题。
- (2) 文档转换有困难。它说明了每一个活动的产品（例如，需求、设计或代码），但没有揭示一个活动如何把一种制品转化为另外一种制品（例如，从需求文档转化为设计文档）。

2.2.3 瀑布模型的优化——原型化的瀑布模型

我们注意到瀑布模型将过程抽象为线性发展的，这种抽象给它带来了简单、明确和直观的优点，但同时也招致了许多问题。可是实际的开发中每个步骤并不是理想中的线性发展的状态，比如在设计阶段了解到客户更新了需求，或者在设计阶段的疏漏直到测试时才被发现，以上的情况都需要返回之前的步骤进行修正才能解决。我们需要对瀑布模型进行优化，引入原型化以有助于控制活动之间的往返。

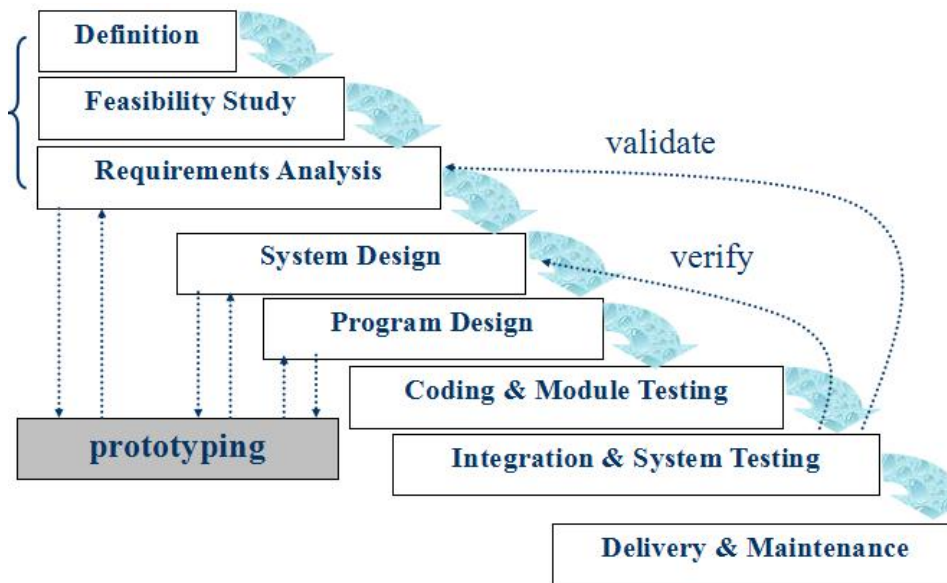
(1) 原型的概念

一种部分开发的产品，用来让用户和开发者共同研究，提出意见，为最终产品定型。

原型可以理解成小样，在某一阶段产品定型前先做一些小样，通过对各种样品的评价和分析，并最终为产品定型。

原型化的瀑布模型如下图所示，在实现代码前的需求和设计阶段引进原型化的概念，在需求分析阶段，通过设计和分析原型以确保需求是一致、可行和符合实际的，避免在测试阶段付出巨大的代价进行修正；在设计阶段，原型化有助于开发人员评价可选的设计策略以及决定哪一种策略是最适合的。为了确保产品符合需要，在测试阶段要进行确认/核准

(validate) 和验证 (verify)。



(2)区分确认和验证：

确认：确保系统实现了所有需求。

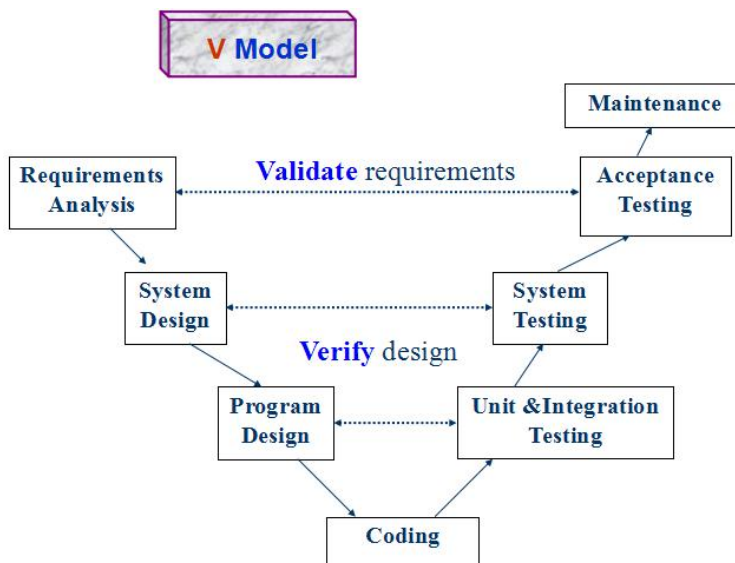
验证：确保每一项功能都是正确的。

确认保证开发人员构造的是正确的产品，而验证检查实现的质量。

2.2.3 V 模型

V 模型是瀑布模型的变种，它说明测试活动是如何与分析 and 设计相联系的。

如下图所示，编码位于 V 型的顶点，分析和设计在左边，测试和维护在右边。测试的每个步骤都与分析和设计相对应，如果在验证和确认期间发现了问题，可以重新执行响应的步骤加以修正。验收测试对应需求分析，系统测试对应系统设计，单元测试和集成测试对应程序设计。

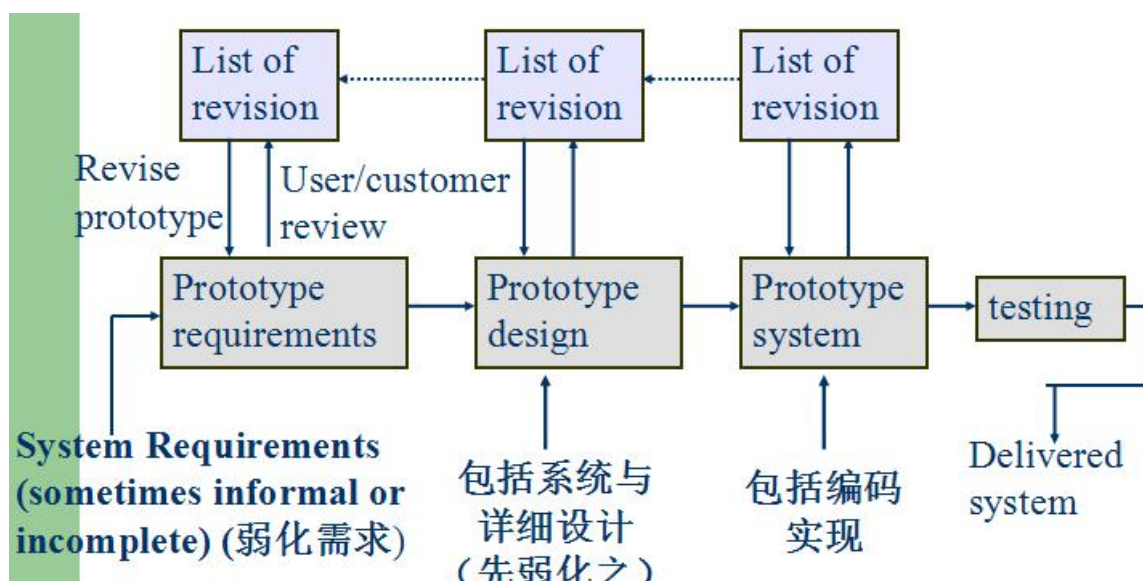


V 模型与瀑布模型的区别：

- (1) V 模型使得隐藏在瀑布模型中的迭代和重做活动更加明确。
- (2) 瀑布模型关注文档和制品，V 模型关注活动和正确性。

2.2.4 原型化模型——“在工作中学会工作”

之前在瀑布模型的优化中介绍了原型化的思想，原型化并不依附于瀑布模型，原型化模型本身是有效的过程模型的基础。因为它允许用户以独立的工程模型的方式，每一阶段都基于原型的建立，以快速构造系统，逐步完成各阶段任务。



如上图所示，原型化模型并不依赖于明确的需求或设计，在情况不明朗的情况下使用原型化模型，先根据简单的需求和设计构造系统的简单样品以理解或澄清问题，以确保开发人员、用户和客户对产品达成共识。也就是说，要根据对每一阶段样品的反响明确需求和设计的具体内容。原型化设计有助于开发人员和客户达成共识，减少了开发中的风险和不确定性。但是为达成共识可能会需要反复进行原型设计。

2.2.5 阶段化开发模型

系统被设计成部分提交，每次用户只能得到部分功能，而其他部分处于开发过程中。

循环周期：从软件开发时整理需求文档到系统交付经过的时间。

理解为何要阶段化开发，我们需要了解现在的商业环境不允许产品长时间拖延，所以我们要缩短循环周期，使用阶段化开发模型使系统能一部分一部分的交付，从而在系统其余部分正在开发的同时，用户已经获得了一部分功能。

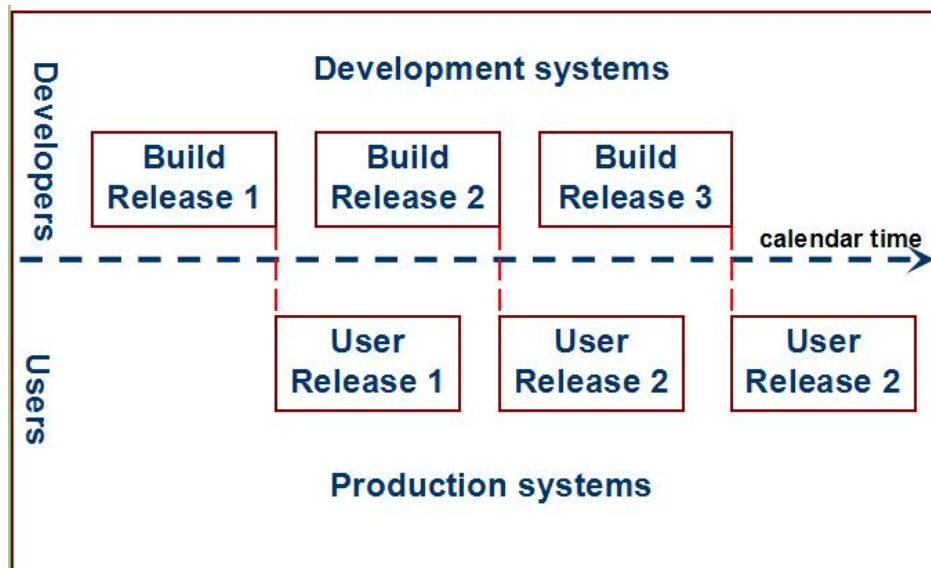
(1) 产品系统和开发系统

因为一边开发一边交付，所以有两个系统在并行运行。

运行系统/产品系统：当前正在被客户和用户使用的系统。

开发系统：准备代替现行产品系统的下一个版本。

两者关系如下图所示，开发人员总是在开发 $n+1$ ，而与此同时 n 正在运行。



(2) 增量开发和迭代开发

因为有不发布开发系统和已经运行的产品系统，我们需要有一种组织两者的方式。增量开发和迭代开发是两种最常用的方式。

①增量开发：系统需求按照功能分成若干子系统，开始建造的版本是规模小的、部分功能的系统，后续版本添加包含新功能的子系统，最后版本是包含全部功能的子系统集。

②迭代开发：系统开始就提供了整体功能框架，后续版本陆续增强各个子系统，最后版本使各个子系统的功能达到最强。

(3) 将增量开发和迭代开发相结合

一个新发布的版本可能包含新功能，并对已有功能做了改进。

两种开发方式结合的原因：

- ①观察用户反馈。
- ②为新功能开拓市场。
- ③及时修复问题。
- ④针对不同版本设置不同专业领域技术的优化。

(4) 进化式迭代开发

统一过程 (UP/RUP)：用例驱动的、以基本架构为中心的、迭代式和增量性的软件开发过程框架。它使用对象管理组织 (OMG) 的 UML 并与对象管理组织 (OMG) 的软件过程工程原模型 (SPEM) 等相兼容。

①统一过程的特点：

(a)统一过程"将重复一系列生命期，这些生命期构成了一个系统的寿命。每个生命期都以向客户推出一个产品版本而结束。

(b)每个周期包括四个阶段：开始阶段、确立阶段、构建阶段和移交阶段。每个阶段可以进一步划分为多次迭代。

②三个支持工序和六个核心工序：

支持工序：

(a)配置变更管理工序，用来管理系统和需求变更的配置。

(b) 项目管理工序，用来管理项目。

(c) 环境配置工序，用来配置项目的环境，包括所涉及到的过程和工具。

核心工序：

(a) 业务模型工序，通过业务模型获取相关知识以理解需要系统自动完成的业务。

(b) 需求工序，通过用例模型获取相关知识以理解自动完成业务的系统需求。

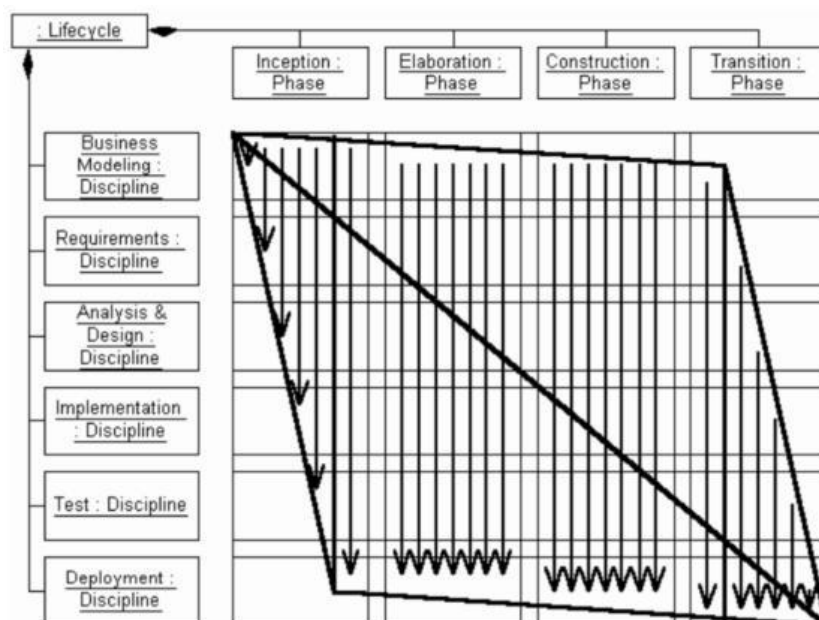
(c) 分析设计工序，通过分析/设计模型以分析需求，设计系统结构。

(d) 实现工序，基于实现模型实现系统。

(e) 测试工序，通过测试模型进行针对需求的系统测试。

(f) 部署工序，通过部署模型部署系统。

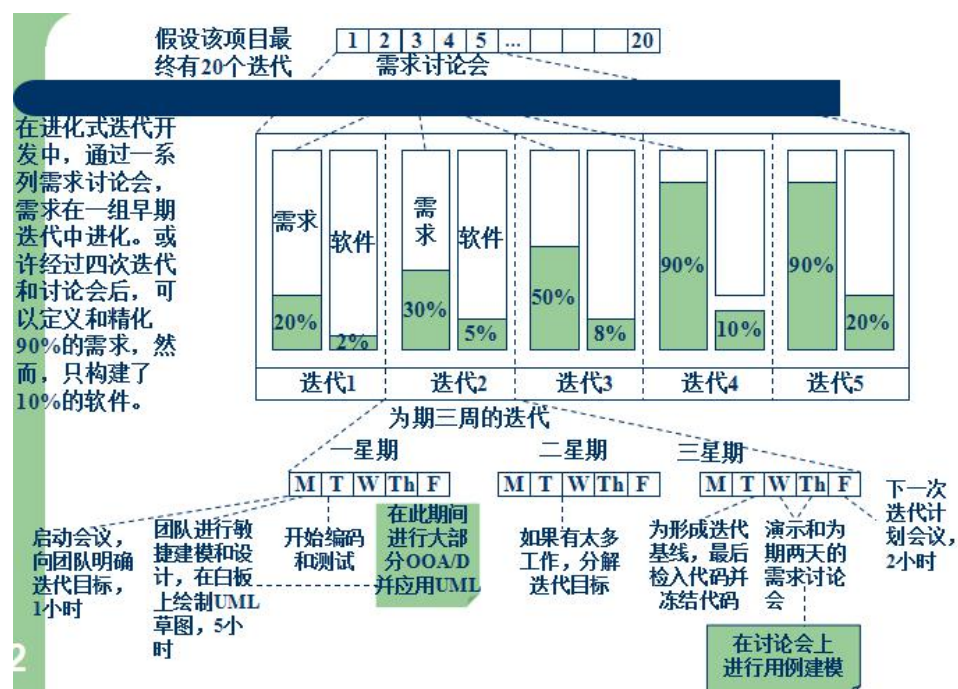
下图展示了统一过程的四个阶段和六个核心工序之间的关系，上面一行表示四个阶段，左边一列表示六个核心工序，二者都包含在软件项目的生命期中。每个阶段都包含六个工序，但是重点不同。开始阶段最关注业务模型，几乎不涉及测试和部署；确立阶段最关心需求和分析；构建阶段最关心实现；移交阶段最关心测试和部署，几乎不涉及业务模型和需求。图中箭头表示六个工序执行顺序，菱形覆盖的面积表示关注程度。



(5) 什么是进化式迭代开发

- ① 进化式迭代开发是统一开发过程的关键实践。
- ② 开发被组织成一系列固定的短期小项目。
- ③ 每次迭代都产生经过测试、集成并可执行的局部系统。
- ④ 每次迭代都具有各自的需求分析、设计、实现和测试。
- ⑤ 随着时间和一次次迭代，系统增量式完善。

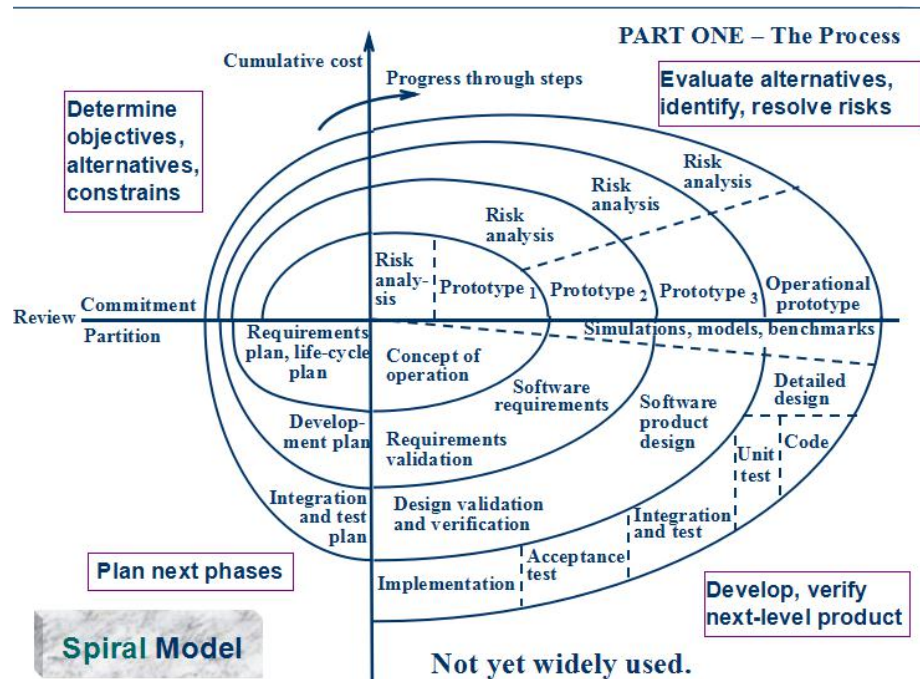
结合下图理解进化式迭代开发。



2.2.6 螺旋模型

此法将开发活动与风险管理结合起来，以降低和控制风险。有些类似于迭代开发模型，结合了迭代的思想，同时也结合了原型化的思想。该模型的适用范围于较大型软件工程项目。

如下图所示，螺旋模型每次迭代有四个任务，依次是计划、目标/可选方案、风险评估、开发与测试。螺旋模型共有四次迭代，依次是操作概念、软件需求、软件设计、开发与测试。每一次迭代都根据需求和约束进行风险分析，以权衡不同选择，并且在确定选择之前，通过原型化验证可行性和期望度。



2.2.7 敏捷方法

之前介绍的几种模型逐渐引入了原型化和迭代开发的思想，使得过程模型变得庞大，而敏捷方法打破了这种局面。

敏捷方法强调灵活性在快速有效的软件生产中所发挥的作用，是重量级方法的叛逆者。

(1) 敏捷方法的四条原则：

- ①个体和交互的价值胜过过程和工具。
- ②可以工作的软件胜过面面俱到的文档。
- ③客户合作胜过合同谈判。
- ④响应变化胜过遵循计划。

这四条原则反映了敏捷方法的软件过程倾向性。它强调人与人之间的交互是复杂的，并且其效果从来都是难以预期的，但却是工作中最重要的方面。

敏捷开发的总体目标：尽可能早的，持续的对有价值的软件的交付活动，以客户满意。

(2) 敏捷开发过程的几种方法：

①极限编程(XP)：激发人员创造性，使管理负担最小的一组技术，是敏捷方法中最主要的流派。(稍后有详细介绍)

②Crystal (水晶法)：每一个不同的项目都需要一套不同的策略、约定和方法论。

③SCRUM (并列争球法)：使用迭代的方法，其中把每 30 天一次的迭代称为一个“冲刺”，并按需求的优先级别来实现产品。

④Adaptive Software Development(ASD) (自适应软件开发)

⑤Feature Driven Development(FDD) (特征驱动软件开发)

(3) 极限编程 (XP)

(a) 四个变量：成本、时间、质量和范围，通过研究变量之间的相互作用，将项目开发分析的更加透彻，成功讲述一个项目成功的原则。

不同的任务对这四个变量有不同的要求，分析哪一个变量是项目进展的制约，集中精力解决关键问题。

(b) 四个准则：

- ①沟通：客户与开发者之间持续的交流意见。
- ②简单性：鼓励开发者选择最简单的设计或实现来应对客户的需求。
- ③反馈：指在软件开发过程中的各个活动中，包含各种反馈循环工作。
- ④勇气：指尽早的和经常性的交付软件功能的承诺。

(c) 十二条原则：计划游戏、小版本、隐喻、简单设计、测试、重构、结队编程、代码集体所有、持续集成、每周工作 40 小时、现场客户、编码标准

①小版本：系统设计要支持尽可能早的交付。(测试要简单有效。)

②简单设计：只处理当前需求，使设计保持简单。(因为假设需求是变化的)

③编码标准：编码支持其他实践，例如测试和重构等。

其余每条的介绍详见课本。

例题

关于小版本(小型发布)的说明:敏捷开发方法中,对计划的发布版本应该(B)。

A: 按产品特性交付:需要交付的特性都必须交付,必要时推迟发布时间

B: 按日期交付:按照预定发布时间进行发布,必要时裁剪部分功能特性。

C: 临时决定:我们会平衡一下,临时根据市场要求和开发进展来确定,可能会同时调整交付时间和特性。

D: 在迭代模式下,没有必要计划版本。每个迭代都应该完成可发布的版本,按照市场需要发布迭代版本即可。

解析:敏捷方法强调灵活性,“尽可能早的,持续的对有价值的软件的交付活动”是其总体目标。

2.3 过程建模工具和技术

建模工具与技术是在过程模型之内的具体运用。

2.3.1 两种主要种类的模型

(1) 静态建模——Lai 表示法

描述一个过程如何由输入转换为输出。

综合的过程符号描述系统,允许人们在任何详细的层次上对任何过程建模,该模型范式中可由人员完成角色,由资源完成活动,最后导致软件工件/制品的产生。过程模型可以用角色、活动、加工项(工件)来显示彼此之间的关系,用状态表显示每个加工项(工件)在特定时间的完成情况。

过程的元素:

- ①活动:过程中要发生的事件。各种前后关系、触发条件、规则、团队成员等等。也可以理解为子过程。
- ②序列:活动顺序等等。
- ③过程模型:小型工程可以认为是开发方式等描述。
- ④资源:活动所需的各种资源标注。
- ⑤控制:针对活动的外部影响等。
- ⑥策略:各种指导原则,包括约束等。
- ⑦组织:各种层次化结构等描述。包括物理的和软件逻辑的结构。

(2) 动态建模

推演一个过程,用户和开发人员可以看到中间产品和最终产品如何随着时间的推移进行转换。

系统动力学:展示资源流(非一般性输入)如何通过活动成为输出。

(3) 在所有的软件开发过程模型中,你认为哪些过程给予你最大的灵活性以应对需求的变更?

- ①设计对于分析模型应该是可跟踪的:软件的模块可能被映射到多个需求上。
- ②设计结构应当尽可能的模拟实际问题。
- ③设计应当表现出一致性。
- ④不要把设计当成编写代码。
- ⑤在创建设计时就应该能够评估质量。

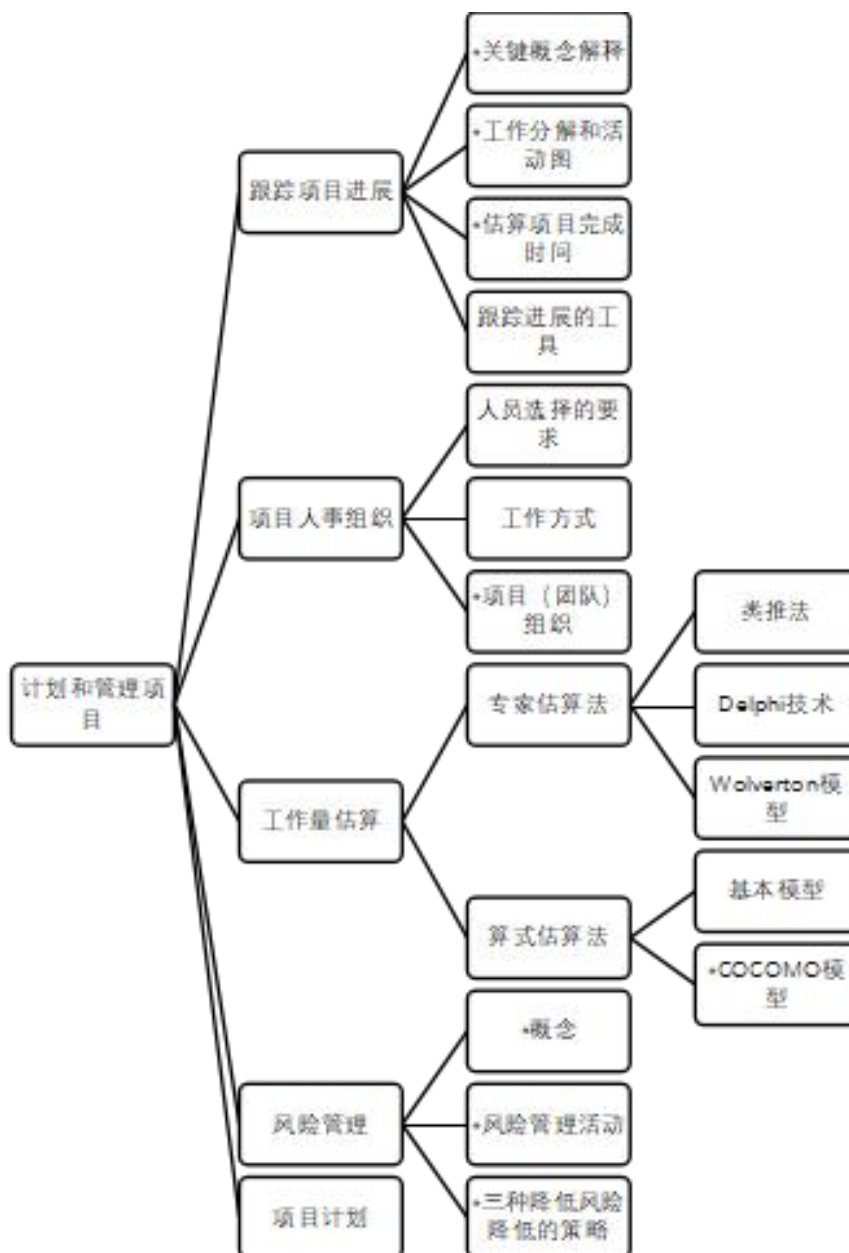
第三章 计划和管理项目

作者：孙吉鹏

一、章节概述

本章在软件工程中处于前期总览全局的位置，它关系着整个项目的进度计划，人事结构和成本分析。这一部分是客户考量项目的关键，回答了客户关心的多长时间做完和花费多少预算做完的 how long and how much 问题，直接关系着预算与经费，是一个项目负责人应当格外重视的部分。这一章的工作体现在制作出项目计划文档中，通过文档内容与客户进行交流展示。从课程角度，这一部分的活动图分析和概念解析是重点，需要用心掌握。

二、章节框架



三、知识点解析

如果要开始一个项目，我们需要跟客户讲述一下我们要做哪些工作来实现这个项目，做多长时间，用多少花费。回答不上这些问题估计客户是不敢找我们干的，回答这些问题就需要我们：

- (1) 首先有明确的实现项目的各个步骤活动的具体时间计划
- (2) 干的时候十分清楚自己干到了哪一步
- (3) 预估整体的预算是多少，凭什么值这个价。

基于这些需求，我们有了这一部分的内容，即，我们怎么跟踪项目进展到哪里了？怎么组织人去完成项目？怎么预估工作量？怎么通过风险管理节约成本？这些都是站在一个项目负责人的角度应当考虑的问题。

3.1 跟踪项目进展

如何确定该怎么做，做到哪里了？

3.1.1 关键概念介绍

(1) 项目进度 (Project Schedule)

项目进度是对特定项目的软件开发周期的刻画。包括对项目阶段、步骤、活动的分解，对各个离散活动的交互关系的描述，以及对各个活动完成时间及整个项目完成时间的初步估算。

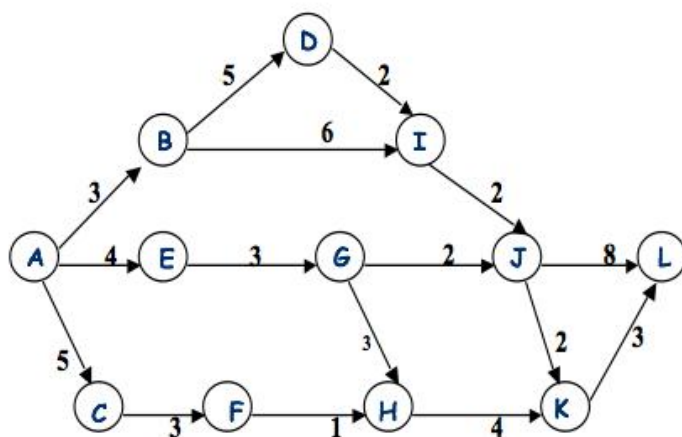
(2) 项目活动 (Project Activity)

项目的一部分，一般占用项目进度计划中的一段时间

(3) 里程碑 (Milestone)

指特定的时间点，标志着活动的结束，通常伴随着提交物。（如一般性文档，功能模块的说明，子系统的说明和展示，精确度的说明和展示，可靠性，安全性，性能说明或展示文档）

3.1.2 工作分解和活动图



活动图 (Activity Graph)

描述了活动和活动间依赖关系的图，其中节点表示项目的里程碑（活动结束），线表示活动。如图例中 A-→B 的部分表示 A-→B 的这条线代表的这个活动从 A 开始，需要做 3 天，才能结束，到达 B 里程碑（标志着 A-→B 这条线代表的活动的结束）。一定要十分注意这一点，图中的点并不代表活动，并不能说活动 A 用 3 天到达活动 B，这是不准确的，如到达 I 里程碑的边有两条，D-→I，B-→I，意思是两个活动，完成后到达里程碑 I，并不能说 I 是个活动，如果这么理解会在计算最晚开始时间时出现错误。

3.1.3 估算项目完成时间

(1) 关键路径 (Critical Paths)

从起点到终点总花费时间最长的路径，即这个项目的最短完成时间，因为如果这条路径无法完成那么整个项目都不能算完成。所以这条路径上的任务耽误一点都会影响最后项目完成时间。

如上图，其关键路径为 A-→B-→D-→I-→J-→L = 20，其他路径都比它短。

(2) 冗余时间 (Slack Time)

在不耽误总体进度的前提下最晚的开始时间和最早开始时间的差值，表示这个任务的机动开始时间，从最早开始时间开始，最晚可以拖的天数，再晚就会影响整个项目的完成时间。

(3) 最早最晚开始时间的计算

首先关键路径上的任务由于不能耽误，所以它们的最早最晚开始时间是相等的，假设从 1 时刻开始，关键路径上的任务的最早，最晚时间都是前一个任务的完成时间加上本任务的完成时间，得到：〈最早开始时间，最晚开始时间，冗余时间〉

$L_{ab} = \langle 1, 1, 0 \rangle$; $L_{bd} = \langle 4, 4, 0 \rangle$; $L_{di} = \langle 9, 9, 0 \rangle$; $L_{ij} = \langle 11, 11, 0 \rangle$

$L_{jl} = \langle 13, 13, 0 \rangle$

之后从后向前，根据关键路径上的里程碑的最晚结束时间减去对应任务的持续时间，计算出除不汇入关键路径上的里程碑的拥有多个出边的点（该图中不存在，如果 E-→F 有边则 E 为这类点）的各任务的最晚开始时间，最后的结束里程碑是 $1+20 = 21$ ，得到：

$L_{kl} = \langle ?, 21-3, ? \rangle$; $L_{jk} = \langle ?, 18-2, ? \rangle$; $L_{hk} = \langle ?, 18-4, ? \rangle$; $L_{gh} = \langle ?, 14-3, ? \rangle$

$L_{eg} = \langle ?, 11-3, ? \rangle$; $L_{ae} = \langle 1, 8-4, 3 \rangle$; $L_{fh} = \langle ?, 14-1, ? \rangle$; $L_{cf} = \langle ?, 13-3, ? \rangle$

$L_{ac} = \langle 1, 10-5, 4 \rangle$; $L_{bi} = \langle ?, 11-6, ? \rangle$

对于除不汇入关键路径上的里程碑的拥有多个出边的点，它们的含义就是可以有多个最晚开始时间，但这时要选择那个较小的最晚开始时间，因为如果选择较大的最晚结束时间而不选择将来可能汇入关键路径的边的较小的最晚结束时间，很可能会耽误关键路径的完成时间，导致最后整体的拖延。

再从前向后，算出与关键路径上的里程碑相连的点所相关的任务的最早开始时间，即里程碑任务的最早结束时间加上对应的任务持续时间，对于不跟里程碑任务相连的多任务的汇点的最早开始时间，先不予计算，如里程碑 H 对应的 H-→K 任务的开始时间：

$L_{cf} = \langle 1+5, 10, 4 \rangle$; $L_{fh} = \langle 6+3, 13, 4 \rangle$; $L_{gh} = \langle 5+3, 11, 3 \rangle$;

$L_{jk} = \langle 13, 16, 3 \rangle$; $L_{bi} = \langle 1+3, 5, 1 \rangle$

由于非关键路径上的汇点的存在导致无法确定那个任务的最早开始时间晚，由于下一个任务的最早开始时间应当是上个任务的最早开始时间中的较晚的那个（只有两个任务都完成，下一个任务才能开始）

Lgh 的最早完成时间是 $8+3 = 11$, Lfh 的最早完成时间是 $9+1 = 10$, $11 > 10$, 所以 Lhk = <11, 14, 3>, 同理, Ljk 的最早结束时间是 $11+2 = 13$, Lhk 的最早结束时间是 $11+4 = 15$, $15 > 13$, 所以

Lkl = <15, 18, 3>

自此算出了所有的活动的最早, 最晚开始时间。

3.1.4 跟踪进展的工具

甘特图, 资源直方图, 开销对比图

3.2 项目人事组织

3.2.1 人员选择的要求 (软件人员应具备的能力)

(1) 完成工作的能力 (2) 对工作的兴趣 (3) 开发类似应用的经验 (4) 使用类似工具或语言的经验 (5) 使用类似开发环境的经验 (6) 使用类似技术的经验 (7) 培训 (8) 与其他人交流的能力 (9) 与其他人共同承担责任的能力 (10) 管理技能

3.2.2 工作方式

外向, 内向; 感性, 理性, 不同性格的人搭配会产生不一样的效果。课本上只是粗略的一个分析。 (如果喜欢研究可以查看荣格 16 型人格, 欢迎交流哦~)

3.2.3 项目 (团队) 组织 (Project Organization)

(1) 主程序员负责制 (Chief Programmer Team)

由一个主程序员负责系统设计和开发, 其他的成员向其汇报, 主程序员对每一个决定有绝对决策权。

优势:

使交流最小化

迅速做出决定

缺点:

创造性低

对主程序员要求高, 个人主观性强

(2) 忘我方法 (Egoless Approach)

每个成员平等的承担责任, 而且过程与个人是分开的; 批评是针对产品和结果的, 不针对个人的。

(3) 项目组织的结构化和创造性

结构化较强的团队:

按时完成任务, 单工作比较循规蹈矩, 项目普通但是功能完备。适合人员较多, 项目稳定性和一致性高, 使用较正规的结构。

结构化较弱的团队：

不能按时完成任务但是创造性强，涉及大量的不确定性因素时采用较为民主的方法和相关的团队结构

3.3 工作量估算

3.3.1 专家估算法

很多工作量估算方法依赖于专家的判断。使用专家的知识 and 经验，对软件项目的工作量进行评估，预测的精确性基于估算者的能力、经验、客观性和洞察力。是对构建整个系统或其子系统所需的工作量做出经验性的猜测。

类推法：

通过做出三种成本估计（悲观估计：x；乐观估计：y；最可能的预测：z），通过公式 $(x+4y+z)/6$ 计算 beta 概率分布均值

Delphi 技术：

几组专家预测平均值的反复修正

Wolverton 模型：

通过计算 O（老问题），N（新问题）和 E（容易的），M（适中的），H（困难的）的 2*3 种组合和不同编程部分的搭配系数矩阵来确定成本。

总体来说，专家估计法差异性，主观性，当前数据依赖的影响。现在大多采用算式估算法。

3.3.2 算式估算法

研究人员已经创建出表示工作量和影响工作量的因素之间关系的模型。这些模型通常用方程式描述，其中工作量是因变量，而其他因素是自变量。大部分模型认为项目规模是方程式中影响最大的因素，表示工作量的方程式是：

$$E = (a + bSc) m(X)$$

E 为工作量，a, b, c 都为常数，S 是估算的系统规模，X 是一个 $x_1 \dots x_n$ 维度成本因素的向量，m 是基于该因素的调整因子。

3.2.1 COCOMO 模型的三个阶段的基本工作原理和含义

COCOMO 模型的关键在于针对项目开发的不同阶段来设置工作量的衡量标准，逐步细化，逐渐准确。： $E = bSc m(X)$

在阶段一，项目通常构建原型以解决包含用户界面、软件和系统交互、性能和技术成熟性等方面在内的高风险问题。这时，人们对正在创建的最终产品可能的规模知之甚少，因此 COCOMO II 用应用点来估算规模。

在阶段二，即早期设计阶段，已经决定将项目开发向前推进，但是设计人员必须研究几种可选的体系结构和操作的概念。同样，仍然没有足够的信息支持准确的工作量和工期估算，但是远比第一阶段知道的信息要多。在阶段二，COCOMO II 使用功能点对规模进行测量。

在阶段三，即后体系结构阶段，开发已经开始，而且已经知道了更多的信息。在这个阶段，可以根据功能点或代码行来进行规模估算，而且可以较为轻松地估算很多成本因素。

3.4 风险管理

3.4.1 什么是风险

概念：软件生产过程中不希望看到的，有负面结果的事件

方面：风险损失，风险概率（相乘为风险暴露（Risk Exposure），即数学期望）

3.4.2 风险管理活动

风险评价：风险识别，风险分析，风险优先级分配

风险控制：风险降低，风险管理计划，风险化解

3.4.3 风险控制

(1) 三种降低风险的策略

避免风险 (Avoiding the risk)：改变功能和性能需求，使风险没机会发生。比如用 C 语言的程序有内存泄漏的风险改用 Java，避免风险。

转移风险 (Transferring the risk)：通过把风险分配到其他系统中，或者购买保险以便在风险成为事实时弥补经济上的损失。

假设风险 (Assuming the risk)：用项目资源，接受并控制风险。比如在开发时主动有意识地进行测试。

(2) 主要的风险管理活动

产品过大：从一个小的产品内核开始，在以后的开发循环中再添加各种功能。

过难或是复杂的功能：在工程开始时化简这些功能，再考虑它们的代替品。

系统支持问题：建立一个早期原型或者小产品版本，以确定你了解支持系统是如何工作的。（通过对核心功能的测试，可以确定其他系统对本软件的系统支持程度）

测试时间：按照 TSPi 进行工作，使用规范的 PSP 方法。

产品控制：这就是在工程开始时进行配置管理的原因。

协同工作问题：工作人员合理搭配问题

3.5 项目计划 (Project Plan)

与客户就风险分析和管理，项目成本估算，进度和组织结构进行交流使用的文档。这相当于这一部分内容的最后呈现方式。

四、典型例题

a. 名词解释:

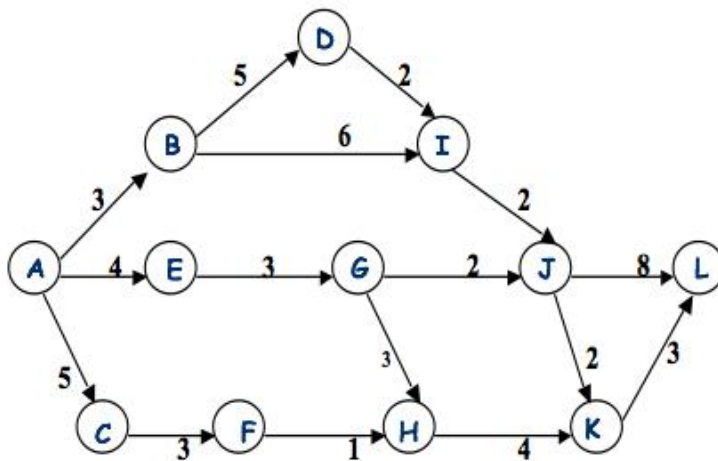
项目进度 (Project Schedule)

b. 简述题:

COCOM02 模型的工作原理

主程序员负责制的优缺点

c. 综合应用题:



找出关键路径，最早，最晚开始时间（解析见重点内容解析部分）

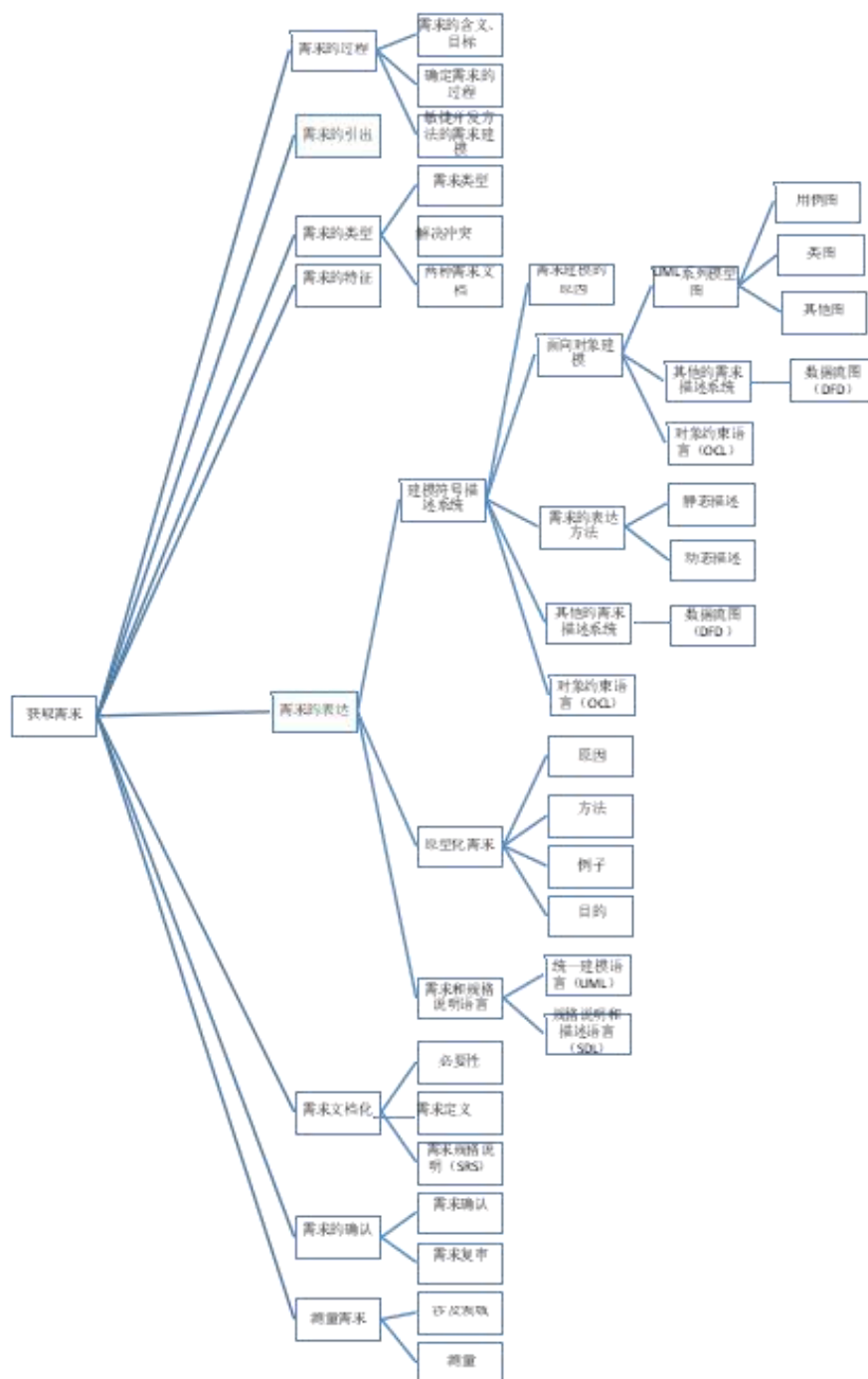
第四章 获取需求

作者 徐卫霞

一、章节概述：

需求对于系统开发来讲是第一步，也是“决定生死”的一步，所以只有确定正确的需求才能保证后期的工作方向是正确的，否则只会做无用功，劳民伤财。确定需求需要进行诸多方面的操作，本章针对需求的确定、需求建模、复审需求、文档化需求等方面对需求进行研究。

二、章节框架：



三、知识点解析：

4.1 需求的过程

4.1.1 需求

定义：对来自用户的关于软件系统的期望行为的综合描述，涉及系统的对象、状态、约束、功能等。

任务：理解客户的问题和需求，针对的是客户和问题，不是问题和实现

4.1.2 确定需求的过程

- ①原始需求获取：客户给出的需求
- ②问题分析：理解需求并通过建模或模型化方式进行描述
- ③规格说明草稿：利用符号描述系统将定义规范化表示
- ④需求核准：开发人员与客户进行核准
- ⑤软件规格说明（SRS）

4.1.3 补充材料：

问题：需求分析时，若小团队且需求不确定，可采用敏捷开发方法；若大团队进行需求确定的开发时可采用“重量级”过程。

①敏捷开发方法的需求建模

适用范围：小团队，不确定的需求

方法：增量式开发（或迭代式开发）

②“重量级”过程

适用范围：大团队，确定的需求

特点：开发人员将编码推迟到已经对需求进行了建模和分析，详细的设计已完成，其中每一步都需要模型，模型间是相关的、相互配合的，以便于设计完全实现需求。

4.2 需求的引出

需求的引出是极为重要的一部分，我们必须使用各种技术来确定客户和用户到底想要什么。

4.2.1 风险承担者

风险承担者包括：委托人，客户，用户，领域专家，市场研究人员，炉石或审计人员，软件工程师或其他技术专家

4.2.2 需求引出的具体手段

- ①与风险承担者进行会谈
- ②评审相关文档
- ③观察当前系统
- ④做用户的学徒，当用户进行任务时更详细的进行学习
- ⑤以小组形式与用户和风险承担者交谈

- ⑥使用特定领域的策略
- ⑦就如何改进产品，与当前的和潜在用户进行集体讨论

4.3 需求的类型

4.3.1 需求的类型

- ①功能需求：描述系统内部功能或系统与外部功能的交互作用，涉及系统输入应对、实体状态变化、输出结果、设计约束、过程约束等。
- ②设计约束：已经做出的设计决策或限制问题解决方案集的设计决策。涵盖物理环境、接口、用户等方面。
- ③过程约束：对用于构建系统的技术和资源的限制，涵盖资源、文档等方面。
- ④非功能需求：描述软件方案必须具备的某些质量特征，例如系统性能、安全性、响应时间等。

4.3.2 解决冲突

(1)需求的优先级划分

当进行需求的引出时，可能会碰到大家对“需求是什么”存在分歧，此时采用对需求进行优先级划分的方法是有效的

- ①必须要被满足的需求
- ②非常值得做但是不是必须的需求
- ③可选的需求（可做可不做）

(2)使需求变得可测试

方法：

- ①针对需求确定一种量化的描述方法，避免模糊的表达方式
- ②将各种指代用词替代为实体的正式名字
- ③每个名词或事项应在需求文档中给出唯一定义

4.3.3 两种需求文档（需求文档化）

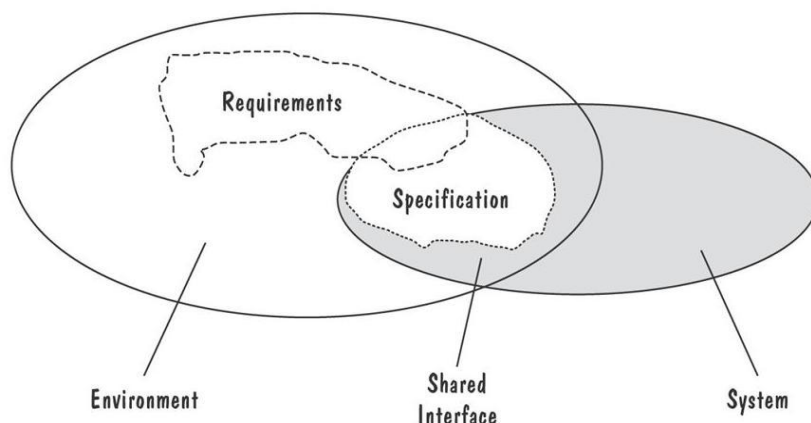
(1) 需求定义

完整罗列了客户期望的需求

(2) 需求规格说明（SRS）

将需求重述为关于要构建的系统将如何运转的规格说明。

需求定义和需求规格说明的关系如下图所示：



(3) 补充材料:

配置管理：使软件过程各阶段文档保持一致的系列过程。

软件配置管理：

定义：一种标识、组织和控制修改的技术，目的是最有效的提高生产率，能够协调软件开发，使混乱减少到最小。

任务：制定软件配置管理计划；确定配置表示规则；实施变更控制；报告配置状态；进行配置审核；进行版本管理和发行管理。

与软件开发过程的区别：变更的评估和批准都需要由软件配置管理人员去做，开发过程应纳入配置管理过程的控制之下。

忽视软件配置管理可能导致的混乱现象：发错了版本，安装后不工作，异地不能正常工作，已经解决的缺陷过后又出现错误，开发人员把产品拿出去出售赢利，找不到最新修改了的源程序，找不到编程序的人

4.4 需求的特征

需求的特征：

正确性；一致性；无二义性；完备性；可行性；相关性；可测试性；可跟踪性

补充：

正确：我们和客户都应该评审需求文档，确保它们符合我们对需求的理解

一致：一般来讲，如果不可能同时满足两个需求，那么这两个需求就是不一致的。

无二义：如果需求的多个读者能够一致、有效地解释需求，那么需求就是无二义性的。

完备：如果需求指定了所有约束下的、所有状态下的、所有可能的输入的输出以及必需的行为，那么这组需求就是完备的。

可行：当用户要求两个或更多的质量需求时，常常会出现可行性问题

相关：有时，某个需求会不必要地限制开发人员，或者会包含与客户需要没有直接关系的功能。

可测试：如果需求能够提示验收测试（明确证明最终系统是否满足需求），需求就是可测试的。

可跟踪：对需求进行精心组织并唯一标记，已达到易于引用的目的；求定义中的每一条都在需求规格说明中有对应，反之亦然。

4.5 需求的表示

4.5.1 建模符号描述系统

(1) 需求建模的原因：

软件工程原理的两个显著特点：可重复过程；建模的标准表示法

需求建模的意义：通过用与客户原始需求完全不同的方式来重新描述需求，可以促使客户为了确认需求的准确性，仔细的检查我们的模型。

(2) 需求的表达方式

利用自然语言进行描述的缺点：所有使用者必须用同样的方式解释含义，很难做到；不易识别系统的各种元素

使用符号描述系统的优点：以严格可控的方式定义需求，使其易追踪和管理分类：

a. 静态描述

仅定义实体（对象）、属性和关系，不描述关系如何随时间变化。

方式：①间接引用②递归关系③公式化定义④形式化语言表达式⑤数据抽象：类似类图

b. 动态描述

将系统状态等随时间的变化进行描述。

方式：①判定表②功能性描述和变迁表

(3) 面向对象建模

面向对象思想比较自然的模拟了人类认识客观世界的方式和法则，面向对象的分析和设计应该从建模开始，构造模型通常出于以下几个目的：在着手解决复杂问题之前，对解决方案进行检测；OO 建模方便客户与其他相关人员交流；加强视觉效果；对复杂问题进行量化和简化。

a. UML：

目标：运用面向对象概念来构造系统模型；建立起从概念模型直至可执行体之间明显的对应关系；着眼于有重大影响的问题；创造一种对人和机器都适用的建模语言

UML 系列模型：

(1) 用例图

用例图着重于从系统外部执行者角度描述系统需要提供哪些功能，并且指明了这些功能的执行者是谁，用例图在 UML 中占有很重要的地位，甚至称为 UML 是一种用例驱动的开发方法。

①组成介绍:

执行者

可能使用这些用例的人或外部程序。


用例

对系统提供的功能（或称系统的用途）的一种描述。


特点：本质上讲，一个用例是用户（角色）与计算机之间为达到某个目的的一次典型交互作用；描述了用户提出的一些可见需求；用例可大可小；用例对应一个具体的用户目标；用例也必须描述用户没有直接提出的一些需求。


②图符:



 系统：用于界定系统功能范围(边界)，描述该系统功能的用例都置于其中，而描述外部实体的执行者都置于其外。

—— 关联：连接执行者和用例，表示执行者所代表的系统外部实体与该用例所描述的系统需求有关。

A  B 使用：由用例A连向用例B, 表示用例A中使用了用例B中的行为或功能。

A  B 扩展：由用例A连向用例B, 表示用例B描述了一项基本需求，而用例A则描述了该基本需求的特殊情况。

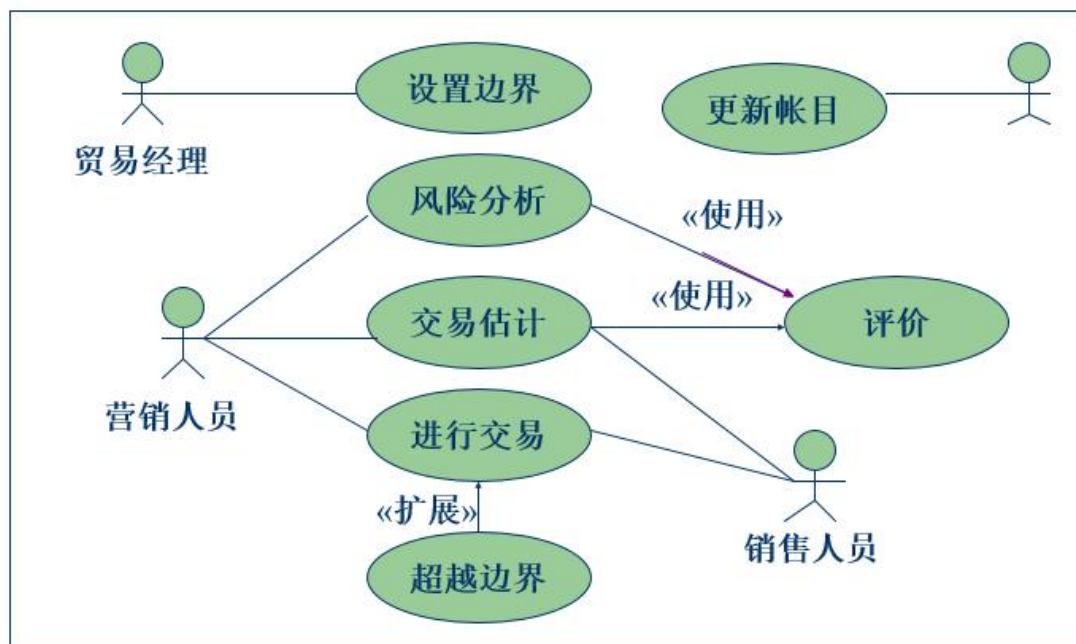


注释体：对UML实体进行文字描述



注释连接：将注释体与要描述的实体连接，说明该注释体是针对该实体所进行的描述。

③实例：



④用例模型的获取：

获取执行者：谁使用系统的主要功能（主要使用者）；谁需要系统支持他们的日常工作；谁来维护、管理系统使其能正常工作（辅助使用者）；系统需要控制哪些硬件；系统需要与其他哪些系统交互；对系统产生的结果感兴趣的是哪些人

获取用例：执行者要求系统提供哪些功能；执行者需要读取、产生、删除、修改或存储系统中的信息有哪些类型；必须提醒执行者的系统事件有哪些；执行者必须提醒系统事件有哪些；怎样把这些事件表示成用例中的功能？

PS：用例只跟参与者打交道，不能把功能分解成大量用例；用例不能是内部实现，也不能没有结果。

(2)类图

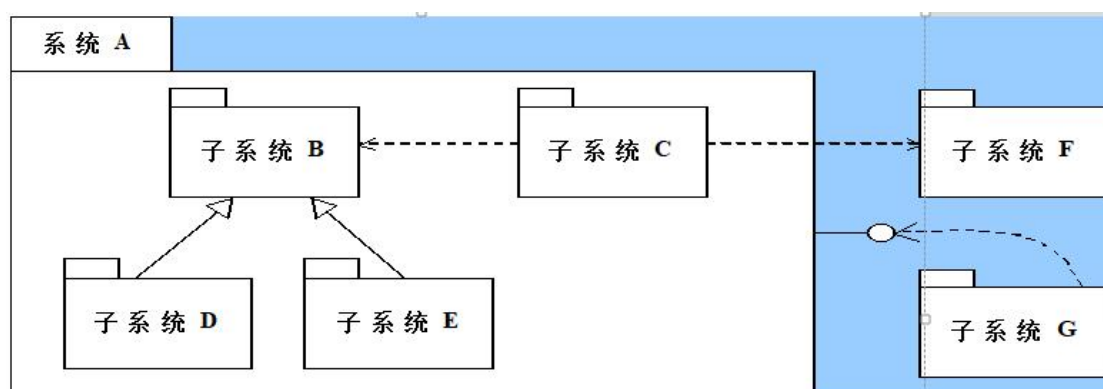
定义：描述了系统中的类及其相互之间的各种关系，其本质反映了系统中包含的各种对象的类型及对象间的静态关系（关联、子类型等关系）

(3).其他图

①包图：

介绍：包图也存在类图里面的继承、引用等依赖关系，也包含接口，接口与包之间用带小圆圈的实线相连。

示例：

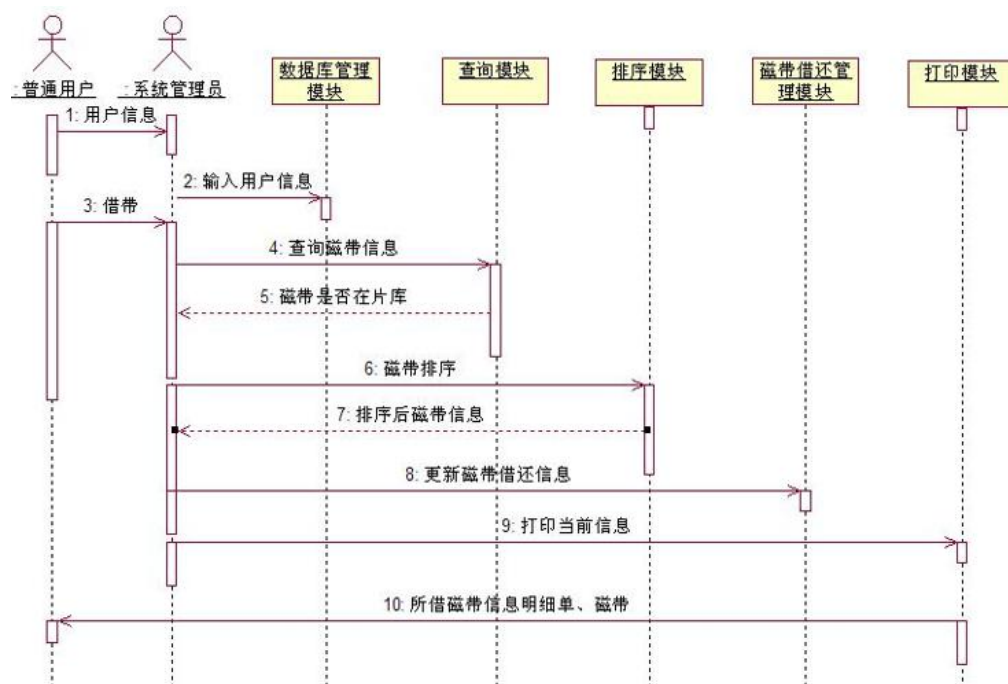


示例介绍： B、C、D、E 在 A 中，C 依赖于 B，C 也依赖于外部的 F，而 D、E 分别继承了 B，而 G 使用了软件系统 A 的接口

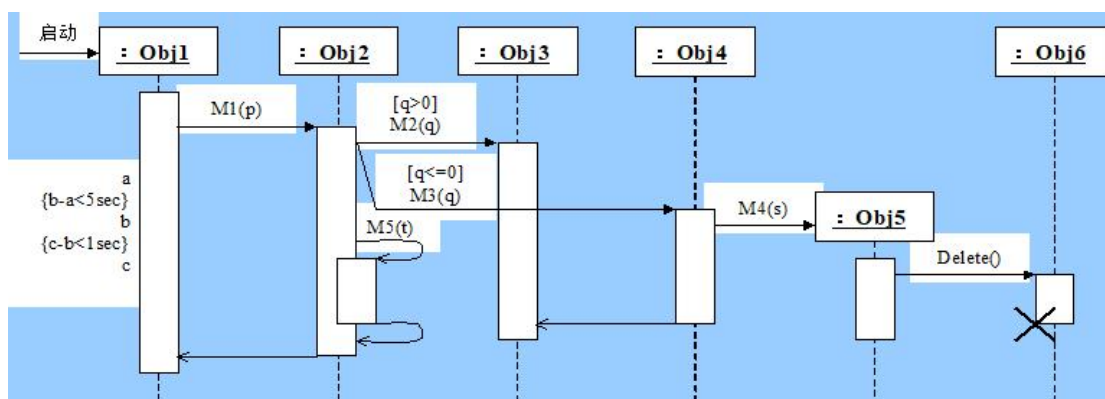
②序列图：

介绍：序列图中的对象可以是并发执行的，每一个对象有自己运行的线程控制着。这时，需要通过激活、异步消息、同步控制和活动对象来表示。序列图有两种，一种是描述特定对象之间生存期中消息通信的所有情节，称作一般序列图；一种是描述消息通信的个别情节的实例序列图，如果需要描述所有的情节，则需要多个实例序列图。

简单画法示例：



复杂画法示例：



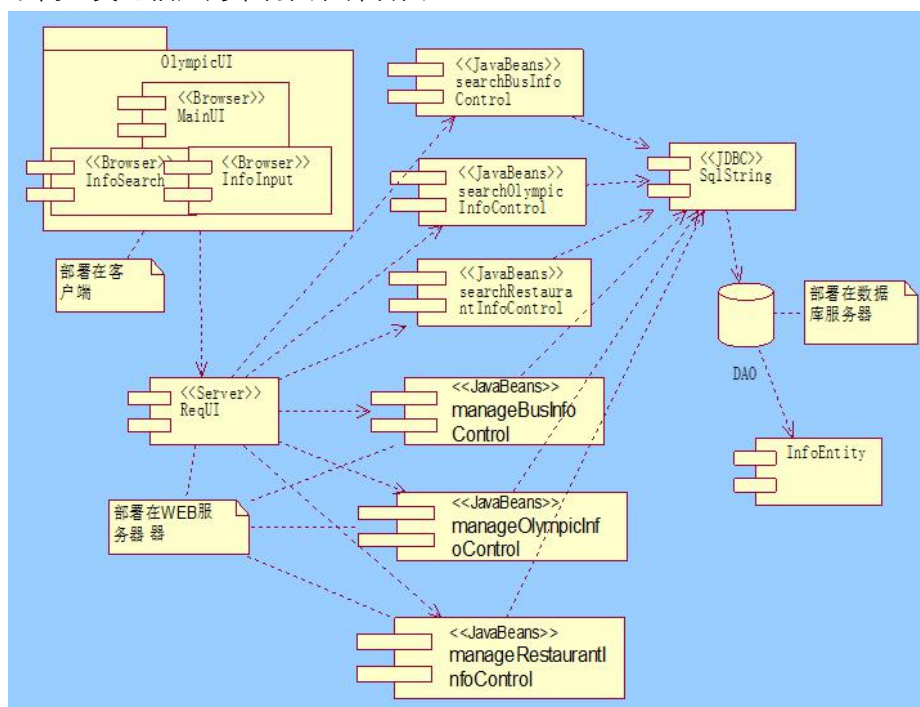
复杂画法解析：M2 和 M3 是有条件的互斥消息，M4 消息创建了对象 Obj5，它的 delete 消息则撤销了对象 Obj6，M5 消息进行了递归调用，Obj1 左面的标签和约束指明了 a、b、c 三点之间的时间约束要求。

③部署图

介绍：部署图描述了系统在运行时的物理结构、配置和关系，涉及处理器、设备、通讯等硬件单元和软件部件。部署图的描述是基于代表硬件单元的节点之上的。

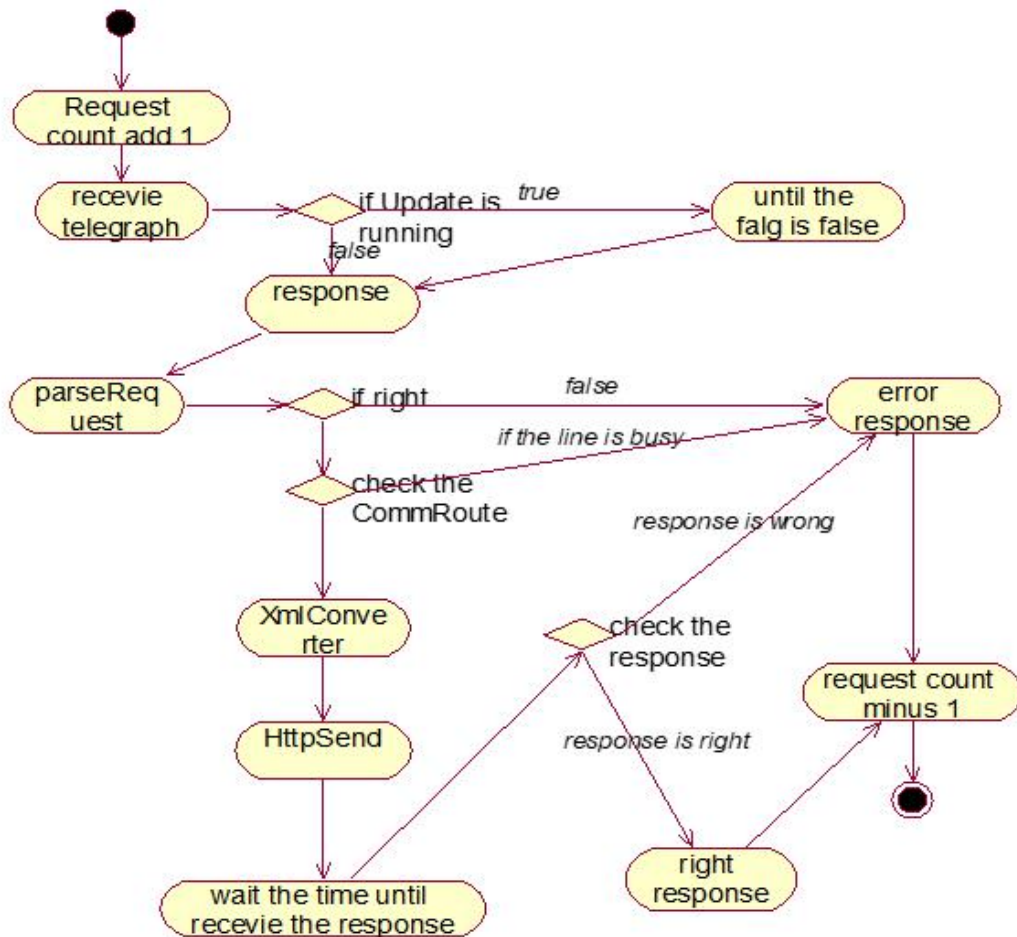
节点：系统中计算资源的代表，每个节点是一个计算机设备及受其管理支配的、处理器、打印机、通讯设备等，以及在其上运行的类、对象等部件。通过这些部件的识别，可以跟踪确认系统的组成以及组成间的关系和在硬件上的配置关系。节点按照硬件资源标识。节点的描述包括三方面：位置（标识）、构成（包含的硬件设备和软件部件）、能力（计算能力、计算功能、等）。可以把节点作为类或实例加以描述。节点类描述了具有相同特性的等待实例化的节点，实例代表了实际出现的节点。（有个别时候把某个客户端描述成一个综合节点类）在把部件分配给节点时，需要考虑许多因素。例如：资源和能力的利用、地理位置对功能实现和系统性能的影响、资源配置与效率的发挥、保密和安全性的实现、系统综合性能的影响、可扩展和可移植等特性的影响。

示例：奥运福娃的架构图和部署图：



④活动图：

示例：



示例介绍：圆形矩形表示活动，实心圆表示起点，内部包含实心圆的圆表示终点。

b. 其他的需求描述系统

数据流图（DFD）：

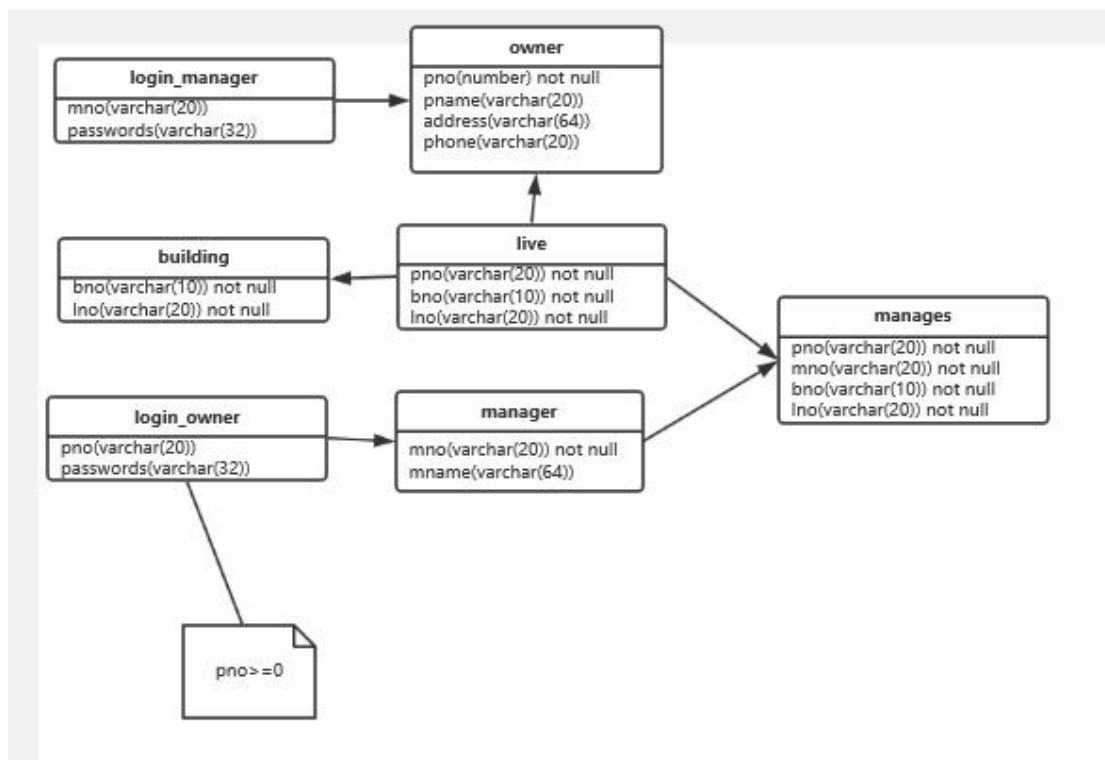
目标：描述数据进入、转换、离开系统，重点在于数据流，而不是控制流
图符：



c. 对象约束语言（OCL）

定义：表述对象模型（例如，ER 图）上的约束

示例：此处 OCL 要求 $pno \geq 0$



3.5.2 需求和规格说明语言

分类：

- ①统一建模语言（UML）
- ②规格说明和描述语言（SDL）

3.5.3 原型化需求

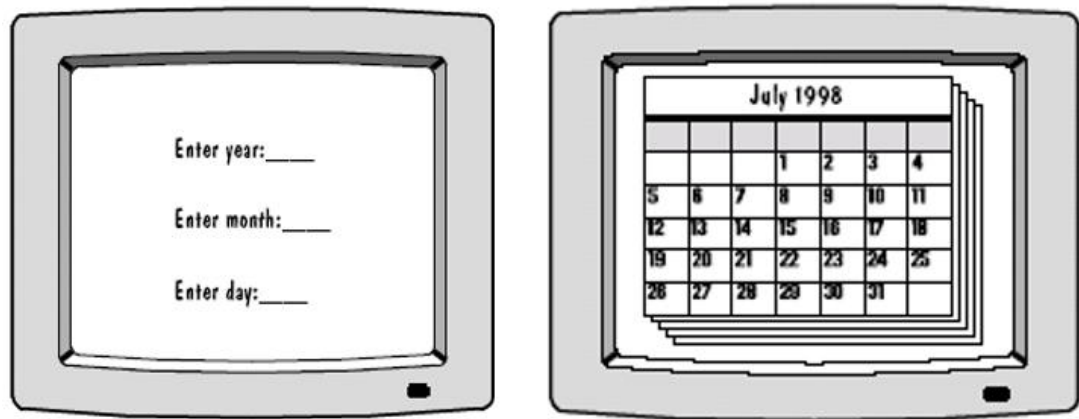
(1) 原因：用户有时不能确定自己的详细/完整需求；开发者无法确定自己的方案是否真实可行；帮助获取需求的更多细节。

(2) 方法：

- ①抛弃型原型：仅用于了解问题，探索可行性，用完扔掉
- ②进化型原型：用于了解问题，并作为将来提交的系统的一部分。

PS：上述两种方法合称为快速原型化，在设计之前进行原型化，可以帮助理解决定最终设计的需求

(3) 示例：



(4) 目的：有些需求难以用文字、符号描述，原型可以帮助我们找到“好的视觉和感觉”；可以评价非功能需求的性能和效率

3.6 需求文档化

3.6.1 必要性：进行配置管理；每个阶段都需要对文档进行参考

3.6.2 需求定义：包含一般用途；系统的背景和目标；描述客户建议的方法；详细特征；运营环境等

3.6.3 需求规格说明（SRS）：包含文件系统接口；对功能、性能要求进行正式化的重述等

3.7 需求确认

3.7.1 需求确认：检查需求规格文档与需求定义的对应性

3.7.2 需求复审

3.8 测量需求

3.8.1 测量集中的领域

产品，过程，资源

3.8.2 测量措施：

- ①产品方面：需求的数目；评估需求文档
- ②过程方面：需求变化的数目；引起需求变化的原因

四、典型例题：

名词解释：

1. 抛弃式原型 (Throw-away prototype)：在进行需求原型化时，仅用于了解问题，探索可行性，用完扔掉
2. 演化型原型 (Evolutionary Prototype)：在进行需求原型化时，用于了解问题，并作为将来提交的系统的一部分。
3. 功能性需求 (functional requirement)：描述系统内部功能或系统与外部功能的交互作用，涉及系统输入应对、实体状态变化、输出结果、设计约束、过程约束等。
4. 获取需求的步骤：
 - ①原始需求获取：客户给出的需求
 - ②问题分析：理解需求并通过建模或模型化方式进行描述
 - ③规格说明草稿：利用符号描述系统将定义规范化表示
 - ④需求核准：开发人员与客户进行核准
 - ⑤软件规格说明 (SRS)

选择题：

1. 需求分析阶段的研究对象是 ()

- A. 系统分析员要求 B. 用户要求
C. 软硬件要求 D. 系统要求

解析：需求分析主要是理解需求并通过建模或模型化方式进行描述，所以此阶段主要研究的对象是用户要求

2. 采用 UML 分析用户需求时，用例 UC1 可以出现在用例 UC2 出现的任何位置，那么 UC1 和 UC2 的关系是 ()

- A: include B: extends C: generalize D: call

解析：UC1 可出现在 UC2 出现的任何位置说明 UC1 具有 UC2 全部的特征，所以 UC1 继承自 UC2。

简述题：

1. 列出类图中各个类间的基本关系

泛化：继承关系，用带三角的实线表示

实现：类和接口的关系，用带三角虚线表示

关联：拥有的关系，一个类知道另一个类的属性和方法，如老师和学生，用带普通箭头的实线表示，双向拥有则为直线，例如：老师———学生———▶课程

聚合：整体和部分的关系，部分可以离开整体独立存在，用带空心菱形的实线表示。

组合：整体和部分的关系，部分不能离开整体独立存在，用带实心菱形的实线表示

第五章 设计体系结构

作者 林子童

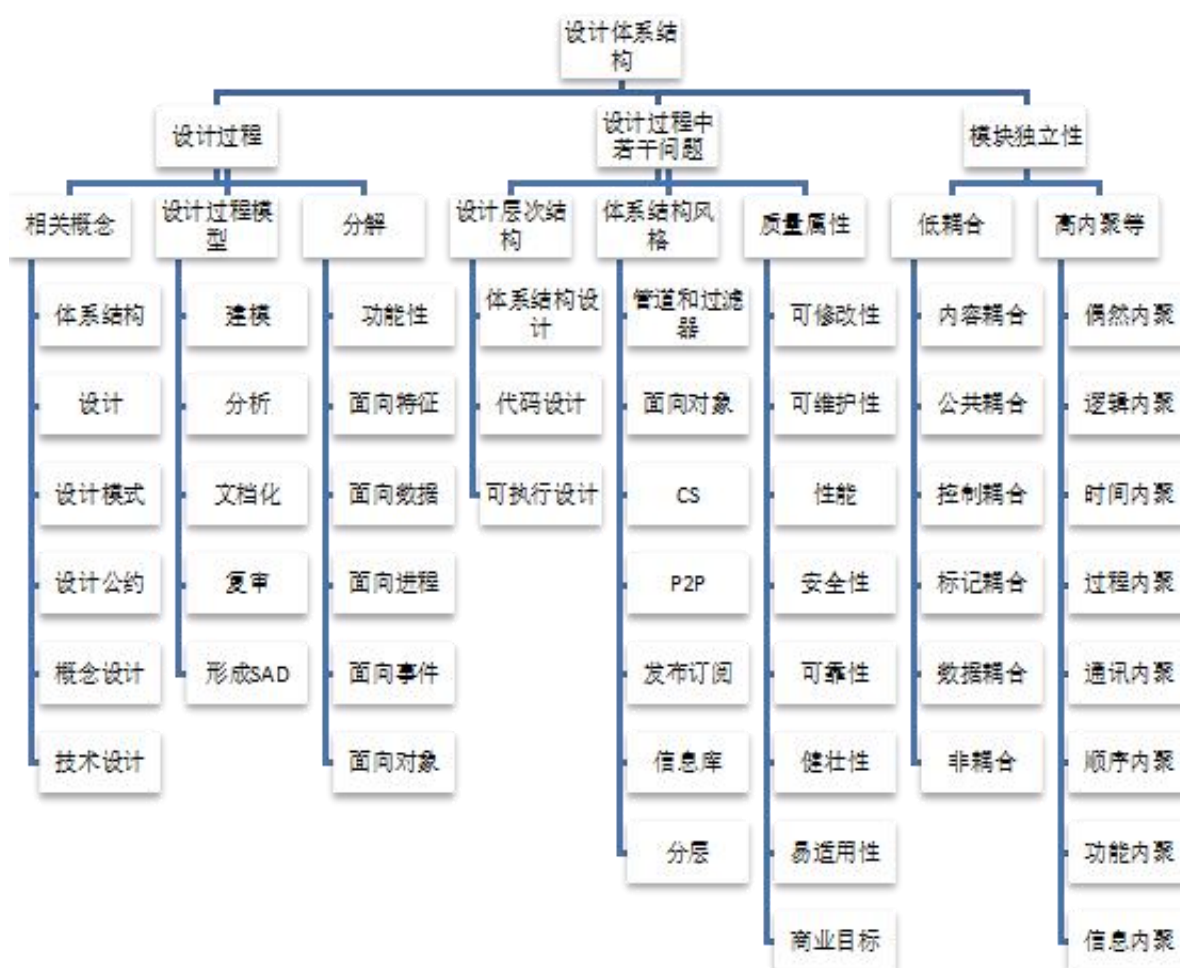
一、章节概述

需求定义和需求分析之后的步骤是对系统进行设计，说明软件系统是如何构造的。对于较小规模系统需求以后就可以简单进入到数据结构和算法设计，进而实现该软件系统，但是构建较大规模系统，就需要将系统分解为规模可管理的子系统或模块，进而进行详细设计。

为了便于理解，我们可以这样说：软件工程的过程分为需求、设计、编码和测试，五六两章讲的是设计方面的内容，对于较大的系统，设计分为概要设计(第五章体系结构设计)和详细设计(第六章模块设计)，概要设计的内容为将一个复杂系统进行模块划分、建立模块的层次结构及调用关系、确定模块间的接口及人机界面，并设计应用系统的总体数据结构和数据库结构等，而详细设计则是根据概要设计赋予的局部任务和对外接口，设计模块算法、流程、状态转换等更为详细的内容。

二、章节框架

以下仅为**部分**重点概念图，本章所有重点内容加粗表示。



三、知识点解析：

5.1 设计过程

5.1.1 相关概念

(1) **体系结构 Architecture**: 一种软件解决方案，用于解释如何将系统分解为单元，以及单元如何相互关联，还包括这些单元的所有外部特性。

(2) **设计 Design**: 将需求中的问题描述转变成软件解决方案的创造性过程

(3) **例程设计 routine design**: 通过对与相似问题的解决方案进行复用和调整来解决某个问题

克隆 cloning: 借鉴现有的整个设计设置包括它的代码，对它做少许的调整来解决特定问题

参考模型 reference model: 用于特定应用领域中标准的、一般性的体系结构，指导我们把系统分解成主要构件以及构件之间如何交互

体系结构风格 architectural style: 是已建立的、大规模的系统结构模式，有一系列定义好了的规则、元素和技术。体系结构不是完整的、细节化的解决方案，而是用于提供各种构建组合起来的方法模板。关注的是构件间各种不同的通信、同步或共享数据的方式。

设计模式 design pattern: 一种针对单个软件模块或少量模块而给出的一般性解决方案，它提供较低层次的设计决策。它是一个共同的设计结构的关键方面，包括对象和实例，角色和协作，责任分配，第六章详细讨论

设计公约 Design Convention: 一系列设计决策和建议的集合，用于提高系统某方面的设计质量。当一种设计公约发展成熟时，将会被封装成设计模式或体系结构风格，最后可能被内嵌为一种程序语言结构

(4) **创新设计 innovation design**: 与例程设计相反，用以创造全新解决方案

设计原则 design principle: 是把系统功能和行为分解成模块的指导方针，描述一些良好设计的特征

(5) **概念设计和技术设计**是两个迭代的过程

概念设计 conceptual design 相当于系统设计，确切地告诉客户系统要做什么，即软件架构和功能

技术设计 technical design 相当于程序设计，一旦客户认可概念设计，系统构建人员就将概念设计转换为更为详细的文档，即技术设计，技术设计确切的告诉开发人员系统如何将如何运转，包括：主要的硬件组件及其功能；软件组件的层次和功能；数据结构和数据流

概念设计强调的是系统功能，而技术设计描述的是系统将要采取的方式。

(6) **构件 component**: 一个可标识的运行时的元素，是有着定义明确接口的、自包含的软件部分，是可以单独开发、购买和销售的实体。

5.1.2 设计过程模型

(1) **Modeling 建模**: 尝试可能的分解，根据需求描述的系统的特性等确定软件体系结构风格

(2) **Analysis 分析**: 分析初步的体系结构，主要关注软件系统的质量属性性能、安全性、可靠性等、各种约束等等。关注系统级别决策

(3) **Documentation 文档化**: 确定各个不同的模型视图。

(4) **Review 复审**: 检查文档是否满足了所有需求。

(5) final output: SAD:Software Architecture Document 软件体系结构文档, 用来和开发团队中其他人员交流系统级别设计决策的有力工具。

5.1.3 分解

分解是把大的系统分解成为更小的部分使问题变得更易于处理, 是一种自顶向下的方法; 另一种自底向上的方法是将笑的模块以及小的构件打包成一个更大的整体, 被认为其设计出来的系统更加易于维护。

六种设计方法

(1) 功能性分解 Functional decomposition:按照功能和需求进行分解

(2) 面向特征的分解 Feature-oriented design:功能性分解的一种, 为各个模块指定了各自的特征

(3) 面向数据的分解 Data-oriented decomposition :关注如何将数据分解成模块

(4) 面向进程的分解 Process-oriented decomposition:将系统分解成为并发进程

(5) 面向事件的分解 Event-oriented decomposition:将事件分给不同的模块

(6) 面向对象的设计 Object-oriented design:将对象分配给模块

模块化 Modular:只有当系统的每个活动都仅由对应的软件单元实现, 并且每个软件单元的输入和输出都已经明确的被定义时, 这个设计才是模块化的。

5.2 设计过程中的若干问题

5.2.1 三种设计层次及其关系

(1) 体系结构设计, 相当于系统设计, 将 SAS 中确定的系统能力和实现这些能力的系统构件关联起来

(2) 代码设计 code: 各个构件/模块的算法和数据结构设计

(3) 可执行设计 executable: 最底层的设计, 包括内存分配, 数据格式、位模式等

这是一种自顶向下的设计, 首先设计体系结构, 然后进行代码设计, 最后是可执行设计, 具有可重复性, 可以多次修改

软件需求包含系统功能性需求以及性能、可靠性和易用性等质量需求。体系结构风格 architecture style 提供了粗粒度的解决方案, 关注的是构件间各种不同的通信、同步或共享数据的方式。策略 tactic 是更精细的设计决策, 能够帮助改进设计, 达到指定的质量目标。这两部分分 5.2.2 和 5.2.3 两小节进行论述。当然, 技术方面问题以外还有很多社会学上的问题也就是协作设计需要考虑。这一部分就不再赘述。

5.2.2 几种体系结构风格

我们关注的是每种体系结构风格的组成元素、元素间交互特征和组中系统性质好坏等

(1) 管道和过滤器: 一种组件式设计, 常用于语义分析、字典分析等设计。

管道 pipe: 将数据从一个过滤器传输到下一个过滤器的连接器, 不对数据做任何改变

过滤器 filter: 数据转换构件, 对输入数据进行转换

(2) 面向对象设计: 用于一般软件设计

(3) 客户-服务器 CS: 服务器提供服务, 客户通过请求/应答协议访问服务

(4) 对等网络 P2P, peer-to-peer: 体系结构中, 每个构件都只执行它自己的进程并且

对于其他同级构建，每个构件本身既是客户端又是服务器。如文件共享网络

(5) 发布-订阅：用于分组交换网络软件设计等等。事件驱动，基于构件之间对时间广播和反应实现交互。一个构件订阅感兴趣的事件，一旦该事件发生，另一个构件进行发布来通知订阅者。隐含调用 **implicit invocation** 就是一种常见的发布订阅体系结构。

(6) 信息库 **repository**：常用于信号处理、知识发现、模式识别系统等设计

由中心数据存储以及与其相关联的访问构件组成。其重要特性就是对于系统关键数据的中心式管理，如黑板类型信息库系统：包括黑板本身，知识源和控制。当黑板的状态发生变化时，知识源将作出响应。也就是访问数据的构件是被动的。

(7) 分层系统就是将系统的软件单元按层次划分，每一层为它的上层提供服务，同时又作为下层的客户。例如计网中的开放式系统互联 **OSI**。优点是高度的抽象以及相对容易增加和修改层，缺点是结构化系统的层次的难度以及系统性能会受到层之间的额外协调影响。

5.2.3 质量属性的几个概念（策略）

(1) 可修改性 **modifiability** 指更改软件系统的难易程度，是绝大部分体系结构风格的基础。对于某个特定改变，为了减少受直接影响的单元，要预测预期改变然后封装在各自的软件单元内、增加内举行、增加软件单元的通用性；为了减少受间接影响的单元，要降低耦合性增加接口设计的多重性。另外，可以设计自我管理软件，检测环境改变进而做出适当反应。

(2) 可维护性 **Maintenability** 是指理解、改正、改动、改进软件的难易程度

(3) 性能 **performance** 描述了系统速度和容量上的特点，主要包括响应时间（请求反应时间）、吞吐量（处理请求速度）和负载（并发用户数）几项内容

提高性能的策略：

①提高资源利用率，如增加软件并行程度；复制数据以及分布式共享数据

②有效的管理资源分配，使用一些调度策略以达到响应时间最小化，吞吐量、资源利用率、公平最大化，高级或紧急事件优先处理等，如 **FIFO**

③降低对资源的需求，如设计适应性能要求的拓扑结构或允许有限变更等

(4) 安全性 **security** 是实现软件各种安全保密措施的有效性度量。

免疫力：系统阻挡攻击企图的能力，可通过更改体系结构保证系统安全特征、最小化系统安全漏洞提高免疫力

弹性：系统快速容易地从成功的攻击中恢复的能力。可通过功能分段使攻击影响最小化、增加快速恢复能力来提升。

P2P 网络：它的复制和荣誉冗余是它的优势，但是因为它的设计本身就是鼓励数据共享的，所以存在非常大的安全问题

(5) 可靠性 **reliability** 是软件产品在假设的环境下，按照规定的条件和规定的时间区间完成规定功能的能力。

主动故障检测：可以使用周期性检查或预测系统故障，如无法提供服务，提供错误服务，数据损坏，死锁等等；可以使用某种形式的冗余如 **n** 版本编程

故障恢复：撤销事务，回退，备份，服务降级，实时修正，实时报告等等

(6) 健壮性 **robust** 是系统在不正确的输入或意外的环境条件下依然保持正确工作的能力。可靠性与软件本身内容是否有错误有关，而健壮性与软件容忍错误或外部环境异常时的表现有关。要防御的设计系统，必须遵循互相怀疑策略，每个软件单元都假设其他软件单元含有故障；在检测故障时，与可靠性策略中的故障恢复基本类似。

(7) 易使用性 **usability** 是用户能够操作软件系统的容易程度。将用户界面放置

于自己的软件单元方便于为不同类型用户定制;系统需要一个进程来监听一些用户发起的命令;系统需要维护一个环境模型来支持系统发起的活动要求。

(8) 商业目标

5.2.4 模块化与抽象层次

(1) 模块化设计 **modularity** 也称作关注点分离,是一种把系统中各不相关的部分进行分离的原则。在模块化的设计中,构件清晰地定义了输入和输出,设计目标明确,功能独立,可以做独立测试。

(2) 抽象 **abstraction**: 对细节的隐藏称为抽象,是基于某种归纳水平的问题描述,是我们集中于问题的关系。当探讨或分析两个模块共享某数据时,模块各自的私有细节应隐藏

(3) 模块都以某种不同层次结构的抽象的形式出现,越上层、越早期的模块层次或框架是越抽象的设计。将模块化部件和抽象层次结合,顶层模块通过隐藏细节可以给出解决方案,其他层次模块显示主要职能和实施细节,就可以用不同的方式设计不同构件的能力。

5.2.5 设计用户界面

用户界面可能是棘手的事情设计,因为不同的人有不同的风格的感知,理解和工作。

(1) 设计界面要注意解决的要素:

- ①隐喻: 可识别和学习的基本术语、图像和概念等
- ②思维模型: 数据、功能、任务的组织与表示
- ③模型的导航规则: 怎样在数据、功能、活动和角色中移动及切换
- ④外观: 系统向用户传输信息的外观特征
- ⑤感觉: 向用户提供有吸引力的体验的交互技术

(2) 文化问题: 需要考虑使用系统的用户的信仰、价值观、道德规范、传统、风俗和传说。两种解决方法: ①使用国际设计/无偏见设计,排除特定的文化参考或偏见②采用定制界面,使不同用户看到额界面

(3) 用户偏爱: 为具有不同偏好的人选择备选界面

优秀的软件必须易于理解、实施、测试和修改(尤为重要),并且正确设计了需求。为了实现以上所述要求,需要有以下几种特征,现在分小节进行讨论

- ①模块独立,低耦合高内聚
- ②能正确进行异常处理和识别,可以通过故障树进行分析
- ③能够进行错误预防,有较好的容错设计
- ④进行安全性分析

5.3 模块独立性

模块独立取决于两部分: 内聚和耦合要记住耦合和内聚的概念、层次划分以及每一种基本分类的含义和实例

5.3.1 耦合 coupling

耦合度是指两个软件之间的相互关联程度,耦合程度取决于模块之间的依赖关系的多

少，可以划分为紧密耦合、松散耦合和非耦合。模块之间的依赖关系有：一个模块引用另一个模块、模块间传递数据量、某个模块控制其他模块的数量。为了使模块可以独立设计和修改，应尽可能减少耦合度。模块耦合有六个级别，从高到底依次为：

(1) 内容耦合 **content**：一个模块实际上修改了另一个模块，被修改的模块完全依赖于修改他的模块。可能的情况有：一个模块修改另一个模块内部数据项或代码，或分支转移到另一个模块。如 **goto** 语句

(2) 公共耦合 **common**：不同模块可以从公共数据存储区来访问和修改数据。

(3) 控制耦合 **control**：一个模块通过传递参数或返回代码来控制另一个模块的活动

(4) 标记/特征耦合 **stamp**：使用一个复杂的数据结构进行模块间传递消息，并且传递的是该数据结构本身。比如将一个数组传递给另一个模块，数组仅用于计算而非控制

(5) 数据耦合 **data**：模块间传递的是数据值，这是最受欢迎的一种耦合。如一个数值被当做参数传递给另一个模块，这个数值在另一个模块中只会参与计算而非控制。

(6) 非耦合 **uncoupled**：模块相互之间没有信息传递，如两个毫无关系的方法，但是一般完全没有耦合是不现实的。

面向对象的设计中模块间耦合度较低，面向对象设计目标之一就是实现松散耦合

5.3.2 内聚 cohesion

内聚度是指模块内部各组成成分（如数据、功能、内部模块）的关联程度，内聚度越高，模块各成分间相互联系越密切，与总目标越相关。内聚分为低内聚和高内聚。下面内聚度从低到高进行介绍

(1) 偶然内聚 **coincidental**：模块各部分不相关，只为方便或偶然性原因放入同一模块。比如强行放入一个类中没有任何关系的方法

(2) 逻辑内聚 **logical**：模块中各部分只通过代码的逻辑结构相关联，会共享程序状态和代码结构，但相对于数据、功能和目标的内聚比较弱。比如因为有相同的某一个计算步骤而放在一起的两个没有关系的计算。

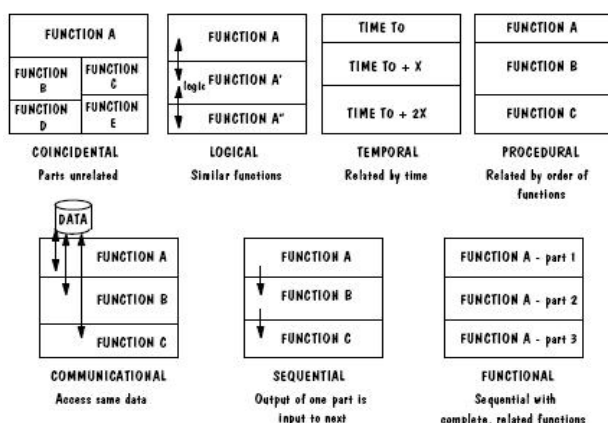


图 5.1 几种内聚的形式

(3) 时间内聚 **temporal**：部件各部分要求在同一时间完成或被同一任务使用而形成联系。比如初始化模块中需要完成变量赋值、打开某文件等工作。

(4) 过程内聚 **procedurally**：要求必须按照某个确定的顺序执行一系列功能，模块内功能组合在一起只是为了确保这个顺序。其与时间性内聚相比优点在于其功能总是涉及相关活动和针对相关目标，如写数据->检查数据->操作数据这一过程

(5) 通讯内聚 **communicational**：各部分访问和操作同一数据集，如将来自于同一传感器的所有不相干数据取出这一模块

(6) 顺序内聚 **sequential**：各部分有输入输出关系，操作统一数据集，并且操作有顺序

(7) 功能内聚 **functional**：理想情况，各部分组成单一功能，且每个处理元素对功能都是必须的，每个元素执行且只执行设计功能，如一个简单的输出程序

(8) 信息内聚 **information**：功能内聚的基础上调整为数据抽象化和基于对象的设计
面向对象设计的目的是高内聚

5.4 体系结构的评估和改进

5.4.1 故障树分析

故障树：通过分解设计寻找可能导致实现的情形。根节点表示想分析的故障/失效，其他节点表示事件或者表示导致根节点失效所发生的故障。其中父节点用逻辑门操作表示，与门指二者必须同时发生才会使父节点发生；或门则是发生一个子节点就足以引发父节点。

通过故障树分析，我们可以获取一个割集树，割集指割集树中子节点的集合，表示引起父节点失效所需要的事件的最小集合通过图 5.2 可以看到引发 G1 的割集是图中四个叶子节点。

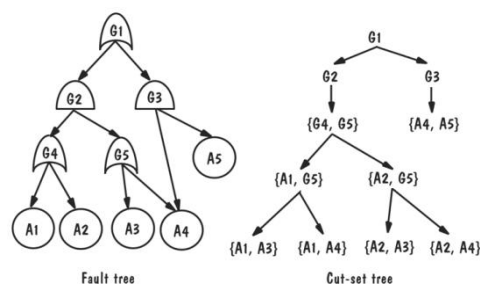


图 5.2 故障树和割集树

5.4.2 错误预防和容错

- (1) 正确区分错误、故障和失效并知道三者的关系（第一章）
- (2) 主动进行故障检测：定期检查故障或预测故障，为此应给一个功能预留多个实现途径
- (3) 被动故障检测：强化故障恢复
- (4) 进行容错设计：当软件失败发生时，采取措施减少损失并将损害隔离开来，在用户接受的条件下使系统继续运行。

5.4.3 设计复审（review）的两种方法：验证与确认

复审定义：检查文档是否满足所有功能及质量需求。

两种设计检验的方法

- (1) 验证 verification：确保设计遵循良好的设计原则，设计文档满足阅读者的需要。验证检查某样东西是否符合之前已定好的标准，就是要用数据证明我们是不是在正确的制造产品。更注重过程正确性，强调做得正确
- (2) 确认 validation：确认设计能够满足用户需求。确认检查软件在最终的运行环境上是否达到预期的目标，就是要用数据证明我们是不是制造了正确的产品。更注重结果正确性，强调做的东西正确。
- (3) 验证更多是从开发商角度来做评审、测试来验证产品需求、架构设计等方面是否和用户要求一致，确认更多是从用户的角度或者可以是模拟用户角度来验证产品是否和自己想要的一致。

5.4.4 设计复审的重要性

- (1) 复审中批评和讨论是“忘我”的，能将开发人员更好地团结在一起，提倡并增强了成员之间的交流
- (2) 在评审过程中故障的改正还比较容易，成本还不高，在这时候发现故障和问题会使每一个人受益。

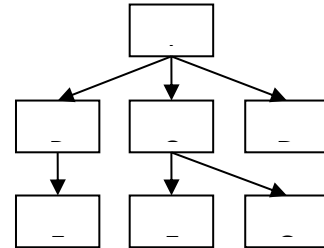
5.5 软件产品线

(1) 软件产品线：是指具有一组可管理的公共特性的软件密集性系统的合集，这些系统满足特定的市场需求或任务需求，并且按预定义的方式从一个公共的核心资产集开发得到

(2) 显著特征：把衍生产品视为产品系列，在开始的时候就计划好同时开发他们。一系列的共性会被描述成可复用的资源集合（包括需求、设计、代码和测试用例），存储在核心资产库 **core asset base** 中。开发过程类似于一个装配线，许多构件不是从零开始而是对核心资产库中的构件进行修改以适应新的需求，然后装配在一起。

5.6 使用图

表示软件单元之间的使用关系如果软件单元 A 需要一个正确的 B 才能完成 A 的任务，那么 A 使用 B。使用图中节点表示软件单元，有向边从使用其他单元的软件单元出发指向被使用的单元。我们使用术语扇入 **fan-in** 表示某个软件单元被使用的数量，扇出 **fan-out** 表示某个软件单元使用其他单元的数量。高扇出往往表明该软件单元做得太多，或许可以分解。



第六章 考虑对象

作者 袁郭苑

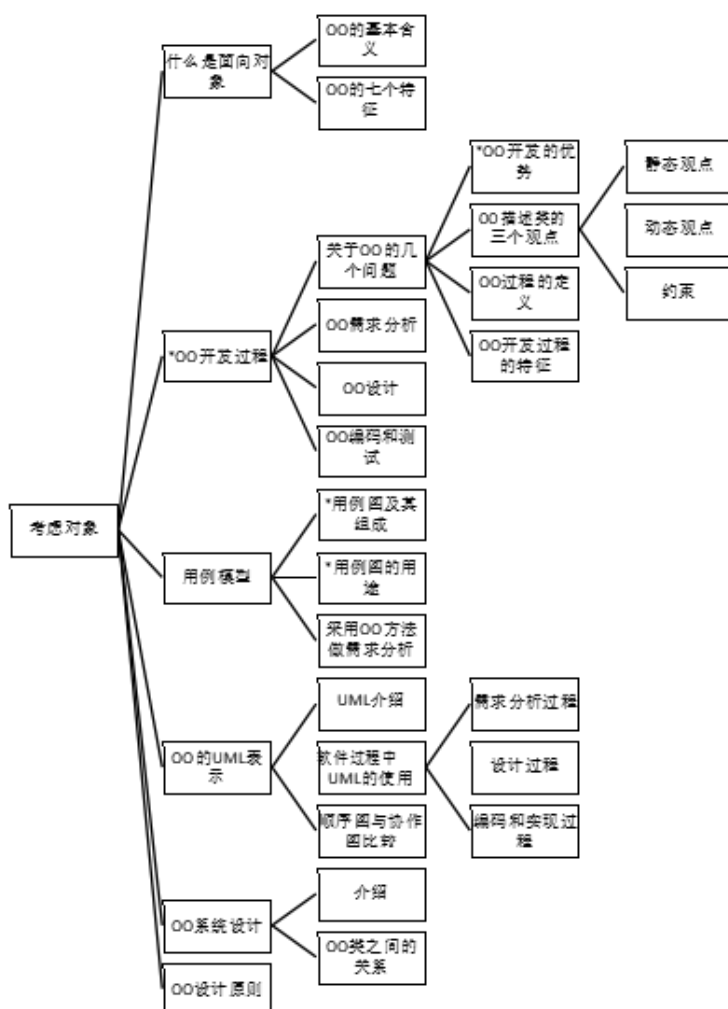
一、章节概述

在进入这一章节之前，让我们先思考几个问题，为什么面向对象的软件开发在当今的软件开发中引起了越来越多的关注呢？到底什么是面向对象？比之于传统面向过程的软件开发，面向对象的软件开发有哪些独特的特征呢？

我们进入大学以来学习的第一门语言就是 Java 语言，一门**面向对象**的高级程序开发语言，我们认识到“一切事物皆对象”，通过面向对象的思维方式，将现实世界的事物抽象成对象，我们通过将现实世界中的关系抽象成类和继承等，帮助人们实现对现实世界的抽象。面向对象既是指一种程序设计范型，同时也是一种程序开发的方法，所以它有自己独特的特征。我们会在接下来详细讨论面向对象的开发过程，主要包括 OO 需求分析、OO 设计、OO 编码与测试。此外，我们还会进一步理解 UML (Unified Modeling Language) 在模型化和软件开发系统中的强大功能，同时复习面向对象类之间的关系，面向对象的设计原则等知识。

经过接下来内容的学习，相信你会对面向对象及其在软件开发中的作用产生更深更到位的理解，并在以后的编程设计中从中受益。

二、章节框架



三、知识点解析

6.1 什么是面向对象 (OO)

6.1.1 面向对象的基本含义

是一种软件开发方法, 它将问题及其解决方法组织成一系列独立的对象, 数据结构和动作都被包括在其中。

6.1.2 面向对象的七个特征

①标识

确定对象的身份。对象名称将一个对象和另一个对象区分开来, 或区分对象自身的状态, 使对象可辨别。

②抽象

层次化的角度描述对象

③分类

将在属性和行为上有共同点的对象分成一个类 (一个对象是某特定类的一个实例)

④封装

封装一个对象的属性和行为, 隐藏其实现细节

⑤继承

根据对象间相同点和不同点分层次地组织对象

⑥多态

指允许不同类的对象对同一函数调用做出响应。即同一函数调用可以根据调用对象的不同而采用多种不同的行为方式。

多态存在的三个必要条件: 继承、重写、父类引用指向子类对象

⑦持久性

持久对象不随着创建它的进程结束而消亡 (因为在外存中存贮)

6.2 *OO 开发过程

6.2.1 关于 OO 的几个问题

①*OO 开发的优势 (相对于传统过程式开发)

语言的一致性: 采用相同的语义结构 (类、对象、接口、属性、行为) 描述问题和解决方案

全开发过程的一致性: 从需求分析和定义、高层设计、底层设计到编码和测试等, 所有的过程都采用相同的语义结构

②OO 描述类的三个观点

静态观点: 包括描述对象、属性、行为和关系

动态观点: 包括描述通信、时间控制、状态和状态转变

约束: 描述软件结构和动态行为的约束

③OO 过程的定义

OO 需求分析和定义+OO 高层设计+OO 底层设计+OOP+OO 测试

④OO 开发过程的特征

全开发过程的一致性 是 OO 开发过程和传统过程式开发的关键区别

- 00 方法在软件制作全流程上尚缺乏理论支持，比如测试理论等
- 00 可以被使用于许多不同的软件生命周期
- 00 开发过程更多地考虑了类和对象复用的可能性

6.2.2 00 需求分析

定义+对象建模（用例图<use case diagram>来确定系统的基本功能和边界等，为系统设计做一定准备）+素描式类图<class hierarchy diagram>+数据流图<data flow diagram>+场景（用自然语言描述过程逻辑，所有的条件或约束）

6.2.3 00 设计

①系统设计

步骤：整个系统的构成-->确定类-->确定对象和类之间的交互和关系-->确定其他的一些图

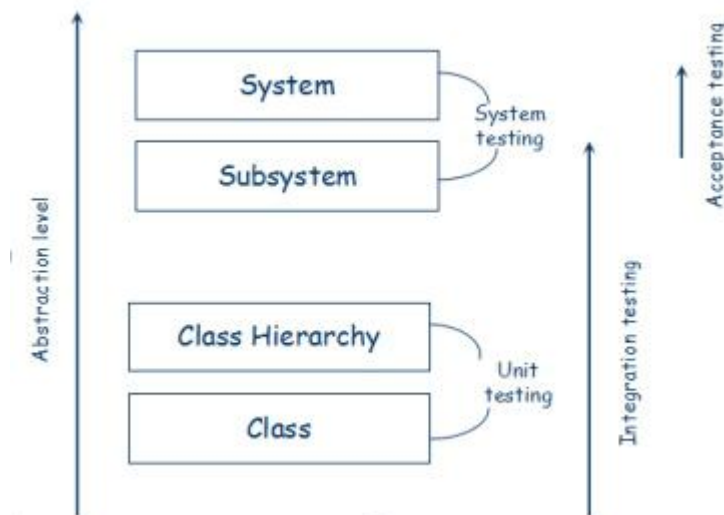
②程序设计

步骤：设计算法-->确定类库细节（在系统规模较大的时候，插入类库的细节时还要有一定的次序，即较复杂的类图结构细节化时的次序问题）-->考虑非功能性需求（根据非功能性需求进行追加式的设计以确保软件设计质量）

6.2.4 00 编码与测试

00 编码即把模型转换成 00 编程语言

00 测试是一系列的测试活动，包括单元测试、集成测试、系统测试和验收测试



从上图我们可以看出，单元测试包括与功能相关的类的测试，也包括类层次结构本身的测试

6.3 用例模型

UML 用例图是 OO 开发过程中获得需求定义的重要方法。

6.3.1 用例图的组成

用例图：表示一个用户、外部系统或其他实体和在开发系统的关系

①用例

描述系统提供的特定功能，用椭圆表示：



②执行者

和系统交互的实体（用户、设备或其他），用小人表示：



③包含

对已定义用例的复用，用以提取公共行为，用带箭头的实线表示：



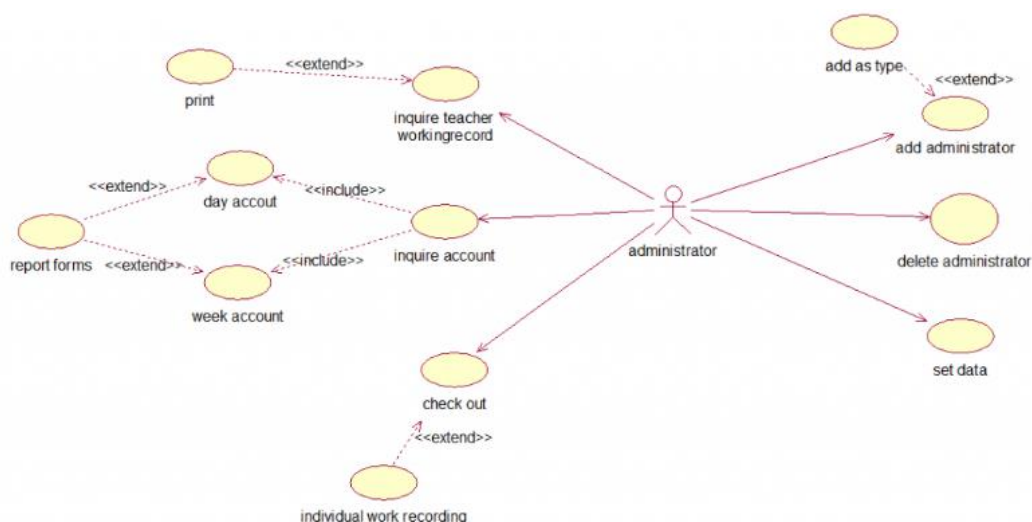
④扩展

对一个用例的扩展使用，用以下图标表示：



6.3.2 用例图实例

下图是机房收费系统中超级管理员行使的功能：



6.3.3 用例图的用途

- ①阐明需求（需求本身是比较难以描述的）
- ②便于找到需求中存在的问题并完善系统的功能需求
- ③便于客户、设计人员、测试人员的沟通交流

④在系统分析中是更正式建模的基础

6.3.4 采用 OO 方法做需求分析

①确定类（粗略的定义）

概念层类图<conceptual class diagram>：描述应用领域中的概念，这些概念和类有很自然的联系，但两者并没有直接的映射关系

领域模型：这不是对软件对象的描述，而是对现实世界中的概念的表示

素描<sketch>：描述大致的轮廓而不去确定细节

②类或领域中的动态行为

有时使用活动图来描述行为，也可以使用状态图

6.4 OO 的表示---使用 UML 的一个样板

6.4.1 UML 介绍

统一建模语言（UML）是一种符号表示方法

三个角度：①动态：用例图、活动图、顺序图、协作图、状态图

②静态：类图、对象图、包图、部署图、组件图

③约束和形式化：对象约束语言（OCL）

6.4.2 软件过程中 UML 的使用

①需求分析过程

工作流图（DFD 或活动图）、概念类图、用例图、包含在前述各图中的描述语言

②设计过程

step1：类图（改进概念类图）、对象图（解释每个对象）

step2：活动图、状态图

step3：顺序图、协作图

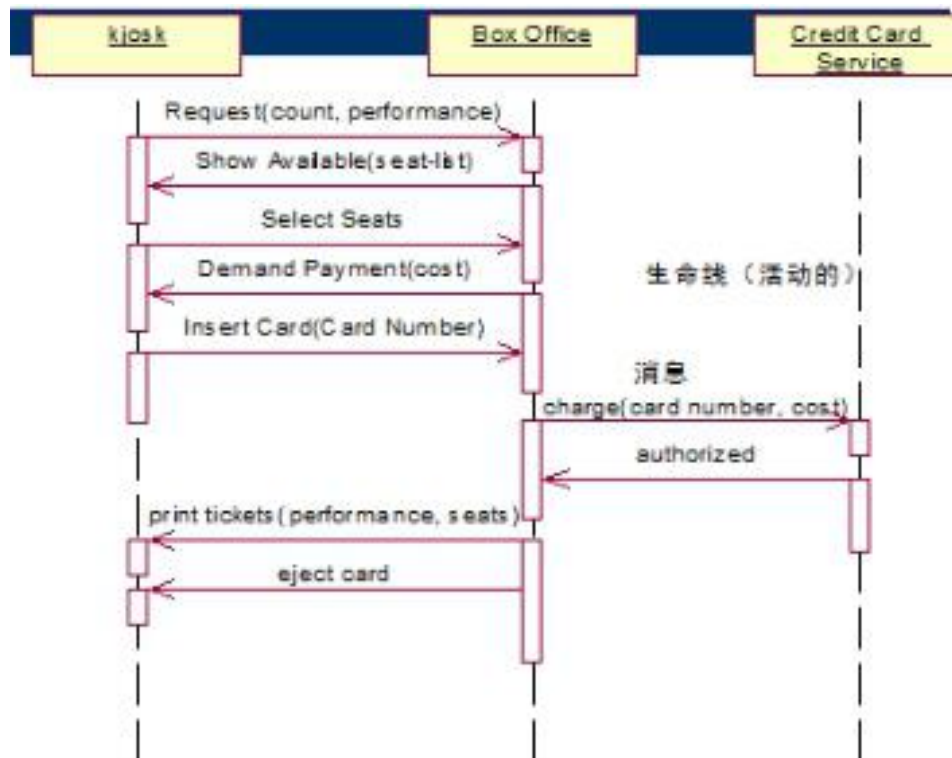
③编码和实现过程

包图、组件图、部署图

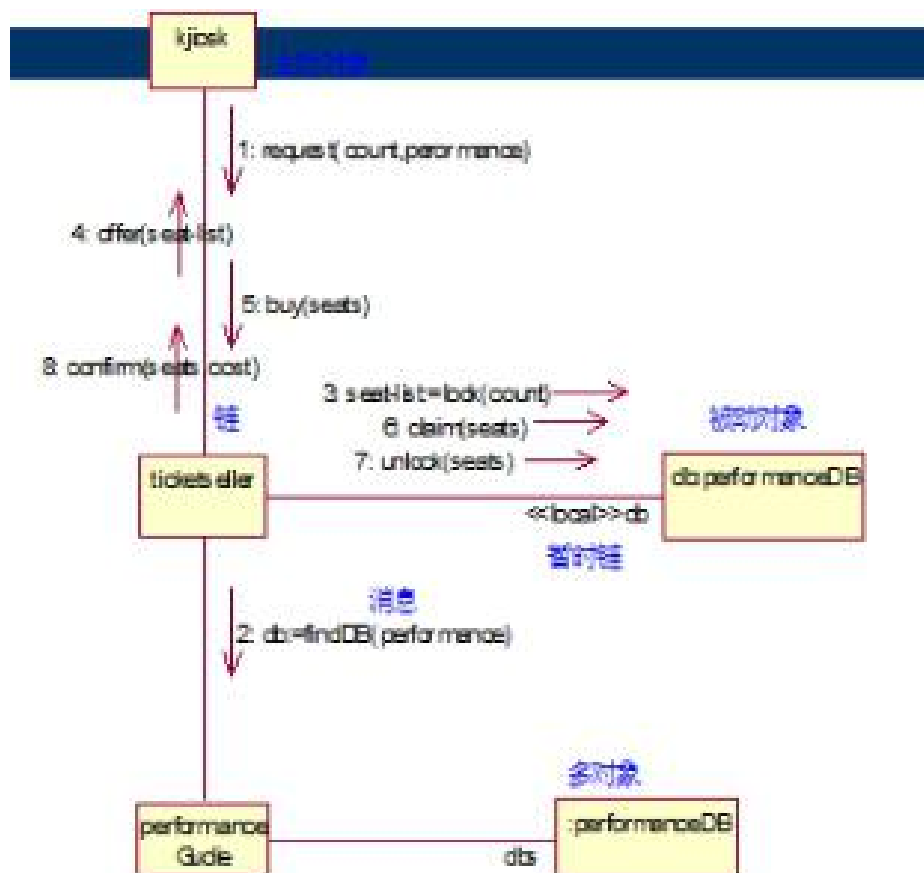
6.4.3 顺序图和协作图的比较

共同点：两者都可以表示各对象间的交互关系

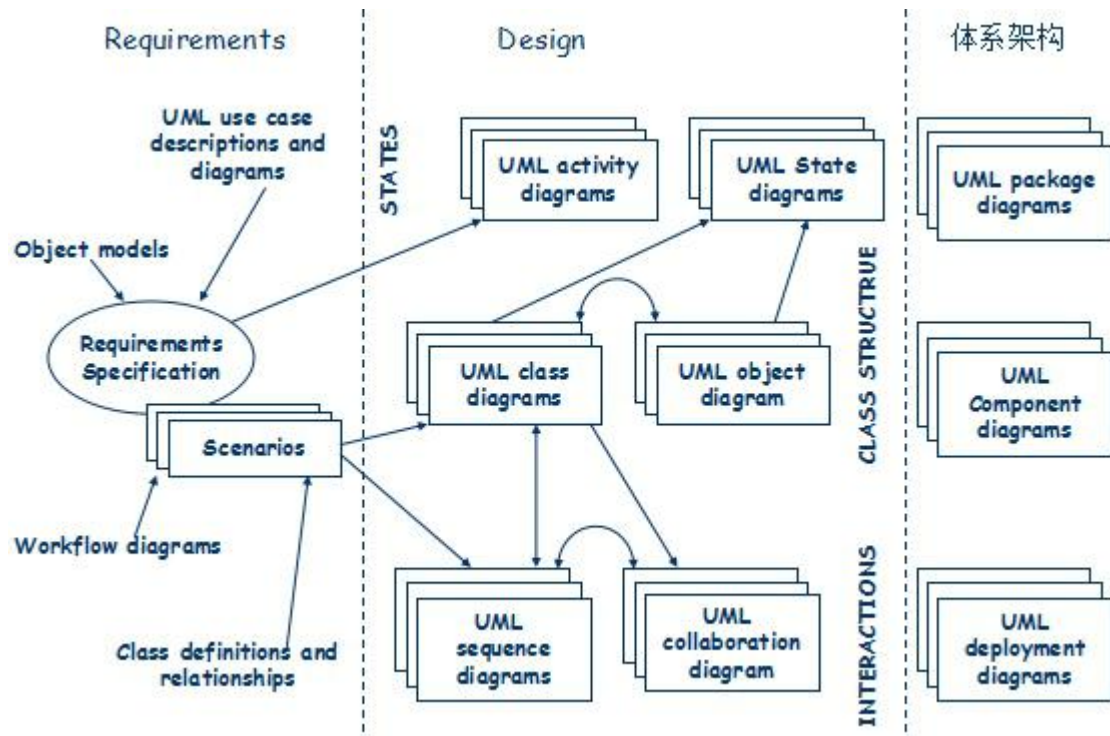
不同点：两者侧重点不同。顺序图用消息的几何排列关系来表达消息的时间顺序，各角色（最上方实体序列可以使用各种角色）之间的相关关系是隐含的



协作图用各个角色的几何排列图形来表示角色之间的关系，并用消息来说明这些关系



6.4.4 UML 支持 OO 开发过程的图示



6.5 OO 系统设计

6.5.1 介绍

OO 设计是从类图（描述对象类型和它们的静态关系）开始的

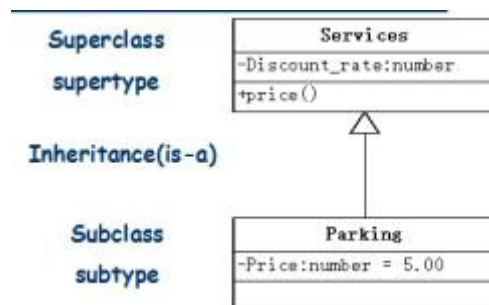
6.5.2 OO 类之间的关系

① 继承 (inheritance)

"is-a"关系, java 使用"extends", C++/C#使用":", 用以下图标表示:

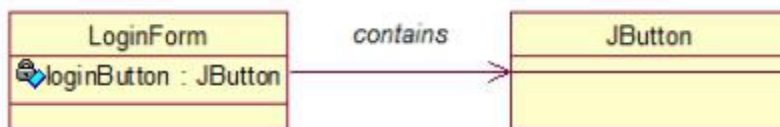


示例:



②关联

使用 java、C#或 C++实现关联（association）时，通常将一个类的对象作为另一个类的成员变量，示例图如下：

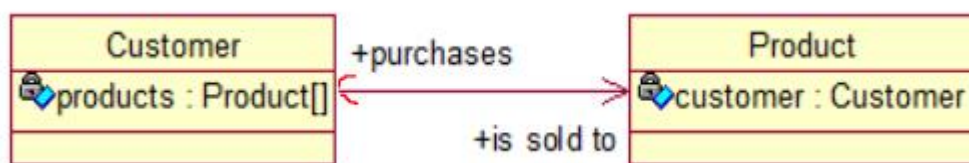


录界面类 `LoginForm` 中包含一个 `Jbutton` 类型的注册按钮 `loginButton`，它们之间可以表示为关联关系

关联包括：

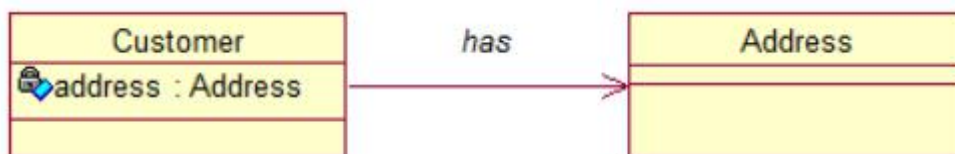
1) 双向关联：

默认情况下，关联式双向的，示意图如下：



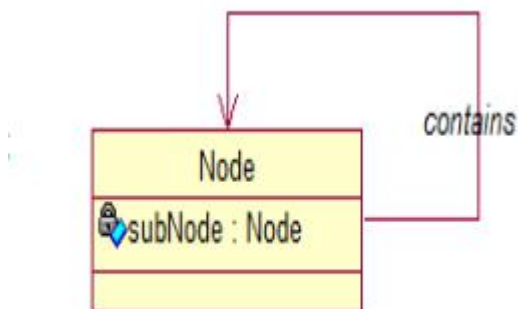
2) 单向关联：

类的关联关系可以是单向的，示意图如下：



3) 自关联：

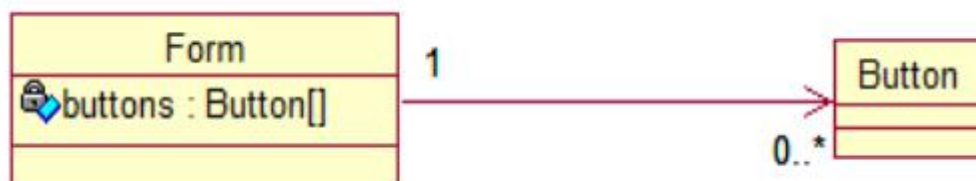
在系统中可能会存在一些类的属性对象类型为该类本身，示意图如下：



上图代表一种**实体类型**的关系

4) 多重性关联

又称为重数性关联关系，表示两个关联对象在数量上的对应关系，示意图如下：

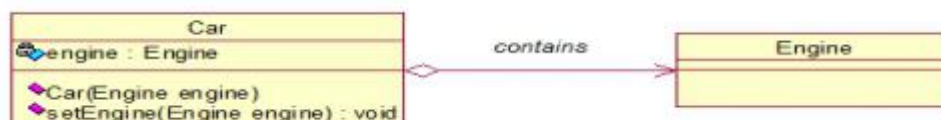


常见的多重性表示方式如下表所示：

表示方式	多重性说明
1..1	表示另一个类的一个对象只与该类的一个对象有关系
0..*	表示另一个类的一个对象只与该类的零个或多个对象有关系
1..*	表示另一个类的一个对象只与该类的一个或多个对象有关系
0..1	表示另一个类的一个对象没有或只与该类的一个对象有关系
m..n	表示另一个类的一个对象与该类最少m个，最多n个对象有关系 (m<n)

③聚合 (UML2.0 去掉了聚合)

聚合(Aggregation)关系中的成员对象是整体对象的一部分，但是成员对象可以脱离整体对象存在，示例图如下：



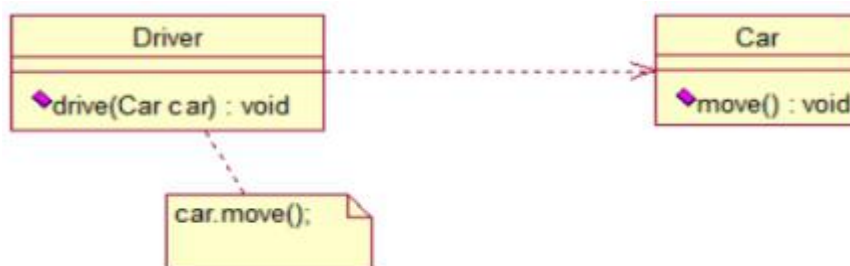
④组合

组合(Composition)关系也表示部分与整体的关系。但是组合关系中整体对象可以控制成员对象的生命周期，一旦整体对象不存在，成员对象也将不存在，两者是共生关系。示例图如下：



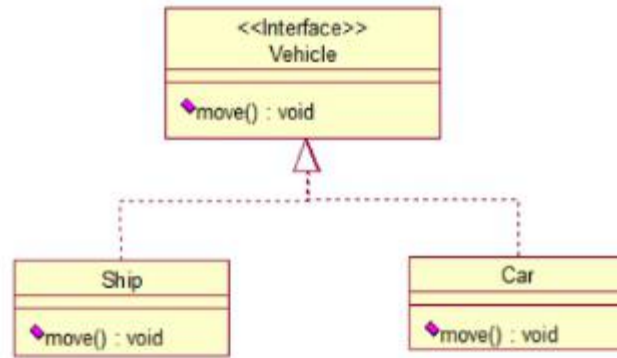
⑤依赖

依赖(Dependency)关系是一种使用关系，大多数情况下依赖关系体现在某个类的方法使用另一个类的对象作为参数。示例图如下：



⑥接口与实现

在 java 和 C#中都引入了接口，接口中通常没有属性，而且所有的操作都是抽象的。接口之间也有继承和依赖关系，但是接口设计很重要的一种关系是实现关系，即类实现了接口。示例图如下：



6.6 OO 设计原则

面向对象设计原则为支持**可维护性复用**而诞生，这些原则蕴含在很多设计模式中，设计原则主要有以下七个：

- ①单一职责原则
- ②重用原则
- ③开闭原则
- ④替换原则
- ⑤依赖倒置原则
- ⑥接口隔离原则
- ⑦迪米特法则

6.7 其他的 UML 图

6.7.1 类描述模板

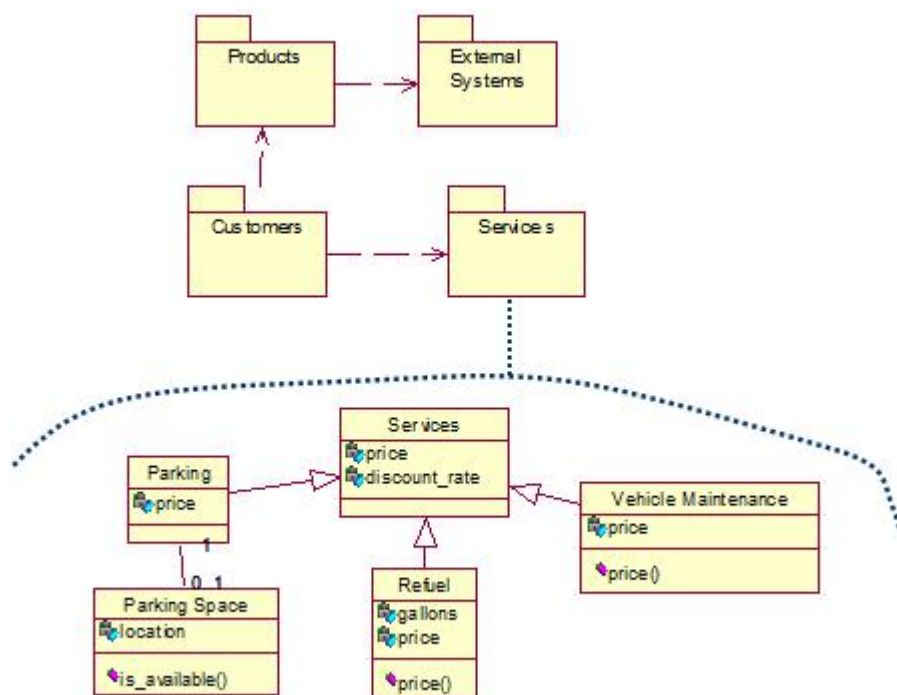
描述了程序设计的基础（类的层次、操作描述、各种状态、与外部的接口等等）

6.7.2 包图

类的集合形成一个包

用途：展示包和类之间的依赖；在测试时发挥重要用途

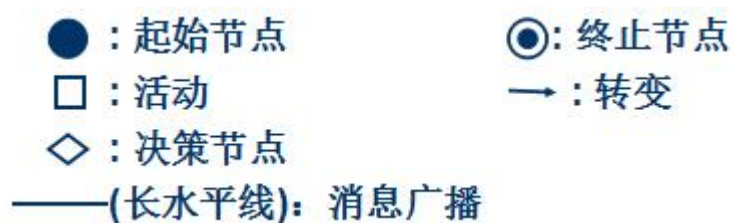
示例：皇家服务站包图如下：



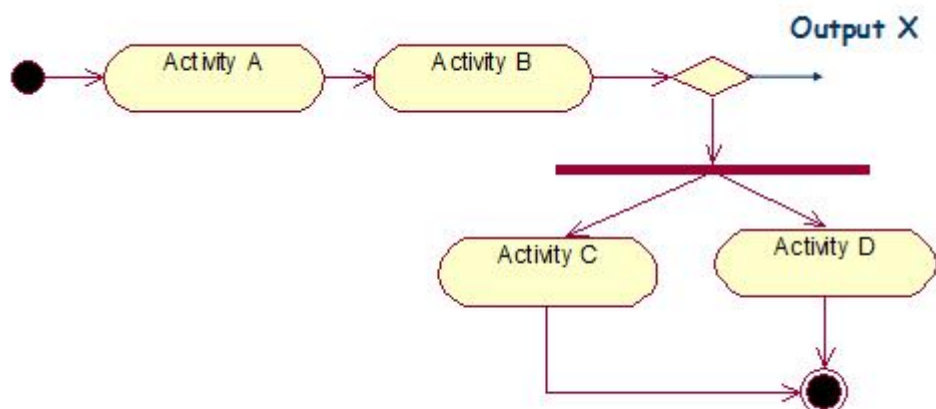
6.7.3 活动图

描述活动和过程流

组成：



示例图如下：



6.8 OO 程序设计中的其他问题

6.8.1 由类和对象开始的程序设计

接口的三种含义：①程序设计阶段，需要详细说明对象之间起交互作用的接口的特征（操作签名）②多态意义上的接口，这允许我们利用多态的优点（我们可以只针对基类写出一段程序，但它可以适应于这个类的家族，因为编译器会自动找出合适的对象来执行操作）实现多态性的手段又叫做动态绑定③多继承意义上的接口（**java** 通过实现多接口的形式实现多继承）

6.8.2 OO 设计模式

设计模式的定义：软件设计过程中，涉及到的常用问题，以及解决这些问题的方案和核心内容

这一部分详细内容请大家参照面向对象课程的智库文件

四、典型例题：

Q: 请说明 OO 的基本设计方法和技巧

A: OO 设计是从类图开始的，在需求分析完成之后，我们通过正确分析各个类之间的关系设计类图（可以通过改进之前制作的概念类图），要遵循 OO 设计原则，采用适当的 OO 设计模式<答题时可在此基础上进一步扩展>

第七章 编写程序

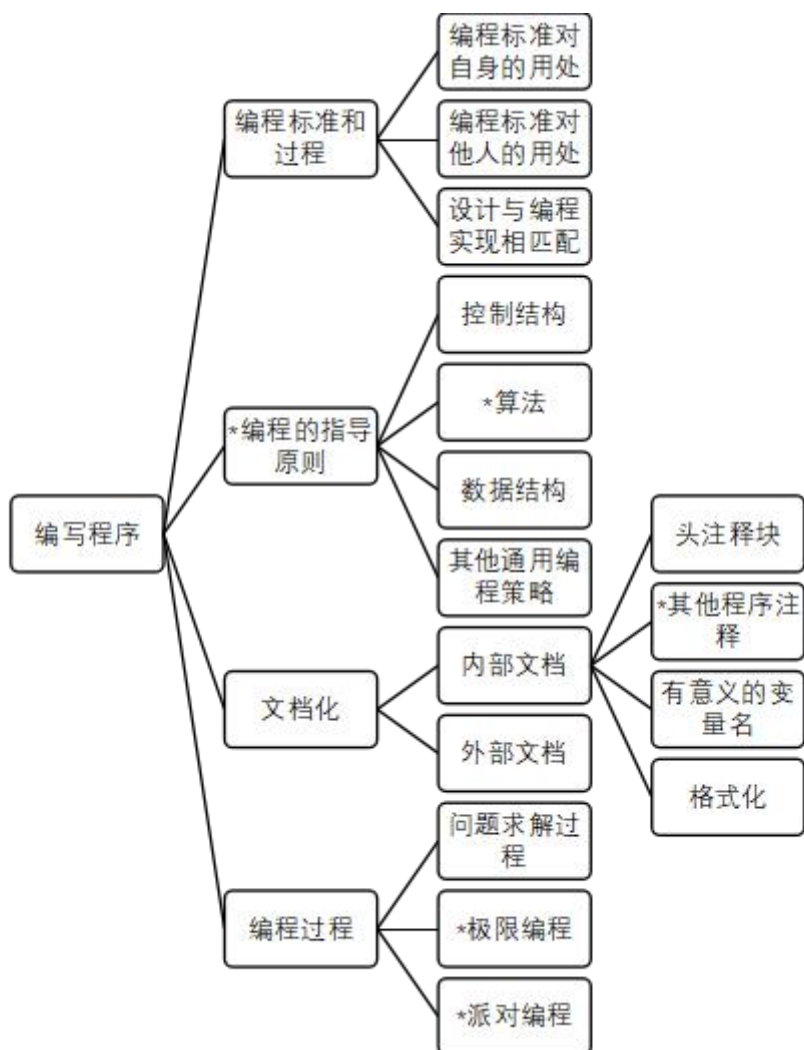
作者 孙吉鹏

一、章节概述

这一章节在整体软件工程中属于代码实现这一过程的方法论，即给程序员的指导手册。不同于学生时代的单打独斗，进入公司后程序员需要合作一起完成一个项目的编程工作，这就需要有一定的标准和规范来让程序员编写的代码易于理解以提高沟通效率，方便合作；需要一定的编程指导原则来实现设计人员的设计；需要形成内外部文档便于团队内的长久沟通；需要了解不同的编程过程来提高生产效率。我们看到，这就是程序实现人员的工作职责和项目负责人对他们的殷切期望：在团队合作的背景下，高效地实现设计文档的内容。

从课程的角度，由于软件工程这门课侧重于整体的工程流程，而这章大多关注于编程实现，在其他课程中有所涉及，所以这一章的考察点主要在于概念理解上。

二、章节框架



三、 知识点解析

为什么说编程工作是复杂甚至令人气馁 (daunting) 的任务?

(1) 设计人员可能没有处理平台和编程环境的所有特性, 易于用图表示的结构和关系并不是总能直截了当的编写代码。

- (2) 编写易于理解的代码
- (3) 编写的代码要易于重用
- (4) 需要比照设计进行检查

7.1 编程标准和步骤 (Programming Standards and Procedures)

不同于学习阶段的独立开发, 公司中开发项目需要多人协同合作, 编程标准和步骤就是公司们要求公司的代码符合某种风格, 格式和内容标准。这样代码和相关的文档对每一个读者就会非常清晰。

7.1.1 编程标准对自身的作用

- (1) 帮助自己组织想法, 避免错误。
- (2) 一些过程包括编写代码文档的方法, 使得它更清晰且易于遵循
- (3) 有助于将设计转化成代码 (维护设计构件和代码构件的一致性) 。

7.1.2 编程标准对他人的作用

- (1) 易于维护 (2) 易于测试 (3) 易于重用

7.1.3 设计与编程实现相匹配

- (1) 在程序设计构件和程序代码构件间建立起直接的对应关系是最关键的标准。
- (2) 设计诸如低耦合, 高内聚, 定义明确等特性的接口

7.2 编程的指导原则 (Programming guidelines)

一般性的编程原则应当从哪三个方面考虑?

控制结构, 算法, 数据结构

7.2.1 控制结构

当设计转变成代码时, 我们希望保留组件的控制结构, 在隐含调用的面向对象设计中, 控制是基于系统状态和变量而变化的。

代码重组, 模块化代码, 注意构件通用性, 注释里体现代码耦合, 提高代码内聚度

7.2.2 算法

设计时通常会指定一类算法, 编程时遵循。

选择算法时要在执行时间, 设计质量, 标准和客户需求之间平衡考虑。

编写某种算法时所涉及的问题

- (1) 编写更快代码的代价。可能会是代码更加复杂, 从而要花费更多的时间编写代码
- (2) 测试代码的时间代价。代码的复杂度要求有更多的测试用例或测试数据
- (3) 用户理解代码的时间代价。
- (4) 需要修改代码时, 修改代码的时间代价。

7.2.3 数据结构

编写程序时，应该安排数据的格式并进行存储，这样的数据管理和操作才能简明易懂。

7.2.4 通用编程策略

- (1) 局部化输入和输出（单独设计 IO），更加易于维护。
- (2) 设计阶段包含伪代码
- (3) 改动时从需求改动，重新设计，重新编码，不要打补丁
- (4) 重用

7.3 文档化 (Documentation)

7.3.1 内部文档

内部文档包含的信息是面向阅读程序源码的那些人的，因此它提供概要信息以识别程序，描述数据结构，算法和控制流。包含：

- (1) 头注释块 (header comment block, HCB)

将一组注释信息放在每个构件的开始部分，包含构件名，作者，配置在整个系统设计的哪个部分上，何时编写和修改的，为什么要有该构件，构件是如何使用数据结构，算法和控制的。

- (2) 其他程序注释

包含：

- a. 可以对程序正在做什么提供逐行的解释。
- b. 将代码分解成表示主要活动的段，每个活动再分解成更小的步骤。
- c. 随着时间进行修改的记录。

- (3) 有意义的变量名和语句标记

命名时尽量用有意义的变量名进行命名

- (4) 安排格式以增强理解

注意缩进和间隔来反映基本的控制结构。

7.3.2 外部文档

外部文档的内容面向不看实际代码的人，如设计人员考虑修改或改进时；它在系统层面回答问题。

7.4 编程过程

7.4.1 编程作为求解过程

分为四步：理解问题，制定计划，执行计划，回顾

7.4.2 极限编程 (Extreme Programming, XP) (第二章具体介绍)

什么是极限编程

极限编程是敏捷过程的一种具体形式，提供敏捷方法最一般原则的指导方针。XP 的支持者强调敏捷方法的 4 个特性：交流、简单性、勇气以及反馈。

交流是指客户与开发人员之间持续地交换看法；简单性激励开发人员选择最简单的设计或实现来处理客户的需要；勇气体现在尽早地和经常交付功能的承诺；在软件开发过程中的各种活动中，都包含反馈循环。例如，程序员一起工作，针对实现设计的最佳方式，相互提供反馈；客户和程序员一起工作时，以完成计划的任务。

极限编程的两类参与者：

客户：

- (1) 定义程序员将要实现的系统特征，使用故事的形式描述系统工作的方式
- (2) 描述测试计划，验证是否实现了所描述的故事
- (3) 为故事及其测试分配优先级

程序员：

将客户需求予以编程实现

7.4.3 派(结)对编程 (Pair Programming)

结对编程属于主要的敏捷开发方法，开发方式是两个程序员共同开发程序，且角色分工明确：一个负责编写程序，另一个负责复审和测试，两个人定期交换角色。

优点：提高生产率和质量，但证据不充分，模棱两可

缺点：会抑制问题求解的基本步骤，扰乱对问题的关注

四、典型例题

1. 简述编程应考虑三方面原则

(1) 控制结构：当设计转变成代码时，我们希望保留组件的控制结构，在隐含调用的面向对象设计中，控制是基于系统状态和变量而变化的。

(2) 算法：在编写代码时，程序设计通常会制定一类算法，用于编写组件。

(3) 数据结构：编写程序时，应该安排数据的格式并进行存储，这样的数据管理和操作才能简明易懂。

2. 名词解释：

派对编程

极限编程

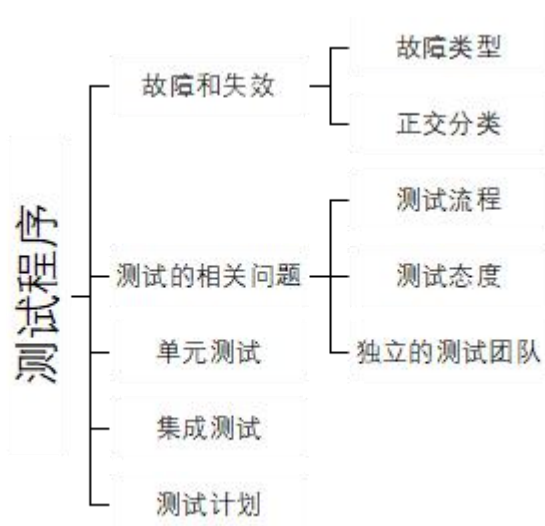
第八章 测试程序

作者 鲍伟

一 章节概述

对于任何一个系统，测试的过程都是必不可少的。测试并不想我们想象的一样单纯，它也有自己的一套体系流程。本章介绍的就是有关测试的前半部分——单元测试和集成测试。将重新深入地认识故障和失效，了解测试的目的、流程和方法。如何从单元测试到将单元集成一个系统也是本章学习的内容。

二 章节框架



三 知识点解析

8.1 故障和失效

8.1.1 故障

故障的定义：由错误（error）引起的系统内在问题。

故障出现的原因：

- (1) 软件本身，系统处理大量的状态，复杂的公式，活动，算法等；
- (2) 客户不清晰的需求；

(3)其他原因，如项目的规模，众多的参与者导致的复杂性。

8.1.2 失效

失效的定义：软件的动作与需求描述的不相符。

造成失效的原因：

- (1)错误的规格说明，或者遗漏了一些需求；
- (2)对于指定的硬件和软件，说明中存在一些无法完成的实现；
- (3)错误的系统设计；
- (4)错误的程序设计，错误的实现。

故障的识别(fault identification)：是确定由哪一个或者哪些故障引起失效的过程。

故障改正(fault correction)或故障去除(fault removal)：修改系统使得故障得以全部去除的过程。

8.1.3 故障的分类

分类的原因：知道正在处理的是什么类别的故障对于我们具体的测试，以及之后的故障改正都是有很大帮助的。

具体分类

(1)**算法故障(algorithmic fault)**：由于处理步骤中的某些错误，使得对于给定的输入，构件的算法或逻辑没有产生适当的输出。

(2)**计算故障(computation fault)**或**精读故障(precision fault)**：一个公式的实现是错误的，或者计算结果没有达到要求的精度。

(3)**文档故障(documentation fault)**：文档与程序实际做的事情不一致。

(4)**压力故障(stress fault)**或**过载故障(overload fault)**：对队列长度、缓冲区大小、表的维度等的使用超出了规定的的能力。

(5)**能力故障(capacity fault)**或**边界故障(boundary fault)**：系统活动到达指定的极限时，系统性能会变得不可接受。

(6)**计时故障(timing fault)**或**协调故障(coordination fault)**：几个同时执行或仔细定义顺序执行的进程之间细条不适当。

(7)**吞吐量故障(throughput fault)**或**性能故障(performance fault)**：系统不能以需求规定的速度执行。

(8)**恢复故障(recovery fault)**：当系统失效时，不能表现得像设计人员希望的或客户要求的那样。

(9)**硬件和系统软件故障(hardware and system software fault)**：当提供的硬件或者系统软件实际上并没有按照文档中的操作条件或步骤运作时。

(10)**标准和过程故障(standards and procesure fault)**：代码没有遵循组织机构的标准和过程。

8.1.4 正交缺陷分类

定义：被分类的任何一项故障都只属于一个类别，则分类方案是正交的。如果一个故障属于不止一个类，则失去了度量的意义。

8.2 测试之前应该明确的几个问题

8.2.1 测试的组织

1 测试的阶段：

(1)模块测试(module testing)、构件测试(component testing)或单元测试(unit testing)：将每个程序构件与系统中的其他构件隔离，对其本身进行测试。

(2)集成测试(integration testing)：验证系统构件是否能够按照系统和程序设计规格说明中描述的那样共同工作的过程。

(3)功能测试(function test)：对系统进行评估，以确定集成的系统是否确实执行了需求规格说明中描述的功能，其结果是一个可运转的系统。

(4)性能测试(performance test)：测试系统的软硬件性能是否符合需求规格说明文档。其结果是一个确认的系统。

(5)验收测试(acceptance test)：确定系统是按照用户的期望运转的。

(6)安装测试(installation test)：确保系统在实际环境中按照应有的方式运转。

(7)系统测试(system test)：功能测试、性能测试、验收测试和安装测试统称为系统测试。

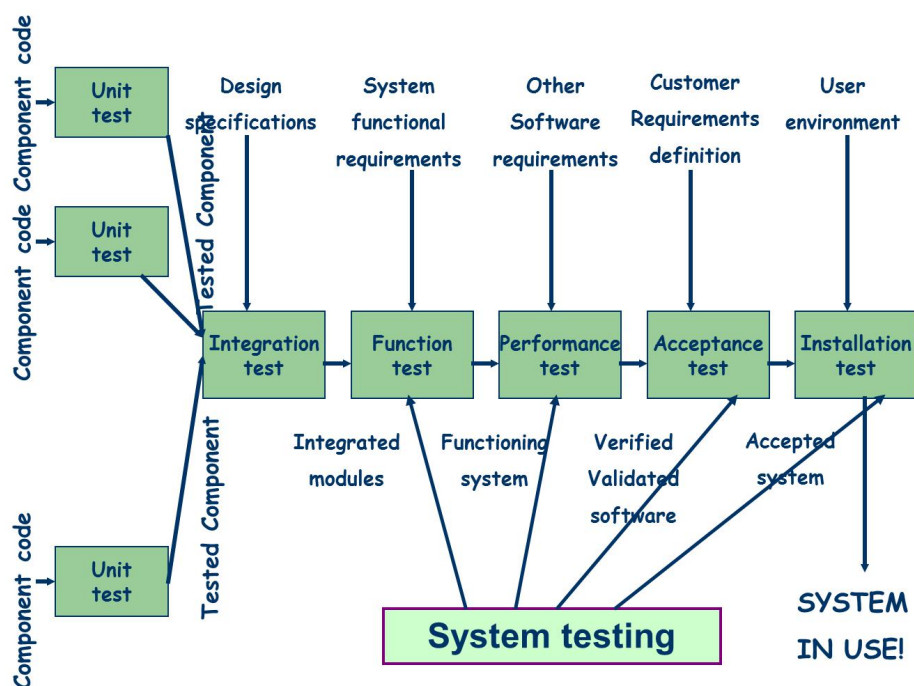


Fig 8.3 Testing steps.

8.2.2 测试的态度

正确的态度应该是将测试看成是一个发现的过程。不仅仅将程序视作问题的解决方案，同样考虑问题本身是否存在问题。当出现故障或者失效的时候，关注的是修改故障，而不是谴责某个开发人员。

8.2.3 谁执行测试——独立的测试团队

(1)开发人员可能担心发现系统故障影响自己的业绩，独立的测试团队避免了故障的个人责任与尽可能多的发现故障的需要之间的冲突；

(2)独立团队因为和代码不是太过紧密，所以能更加的客观，有更多的机会发现细微的故障；

(3)独立的测试团队可以参与软件开发的整个过程，测试可以和编码并行的进行。

8.2.4 测试的方法

1. 黑盒：将测试的对象看作是一个不了解其内容的闭盒，我们的测试就是向闭盒提供输入的数据，并记录产生的输出。测试的目标是确保针对每一种输入，观察到的输出与预期的输出相匹配。黑盒测试参考的文档是系统设计和程序设计阶段的文档。

优点：黑盒测试免于受强加给测试对象内部结构和逻辑的约束。更偏向于功能性的测试。

缺点：黑盒法以 SRS 为依据，有一定的盲目性和不确定性，不可能揭示所有的错误。没办法总是使用这种方式进行完备的测试。不容易找到具有代表性的测试用例证明所有情况下功能都正确。

2. 白盒：将测试对象看作一个白盒，然后根据测试对象的结构用不同的方式进行测试。

优点：可以测试一个模块的细节。

缺点：该法以模块内部逻辑为依据，当内部逻辑过于复杂时，则不能给出好的或合适的测试用例。有时候，对于大量递归、循环和分支的构件，想要测试完所有的分支也是不现实的。

3. 实际测试中，没有必要把黑盒测试和白盒测试严格的区分开来。具体的测试方法的选择受到很多因素的影响。

8.3 单元测试

8.3.1 检查代码

1. 代码评查(code review)：代码走查(code walkthrough)和代码审查(code inspection)。代码走查相对非正式，代码审查相对正式，会事先准备关注问题清单，依据清单比对代码和文档的一致性。

代码评审带来的好处：一个故障在开发过程中发现的越早，它就越容易纠正，所造成的损失也就越小。评审在检测故障方面表现得格外成功。

8.3.2 测试程序构件

1. 测试点(test point)或**测试用例(test case)**: 用于测试程序的输入数据的一个特定选择。测试是测试用例的有限集合。

2. **测试的完全性**: 以一种使人信服的方式来证明测试数据展现了所有可能的行为。

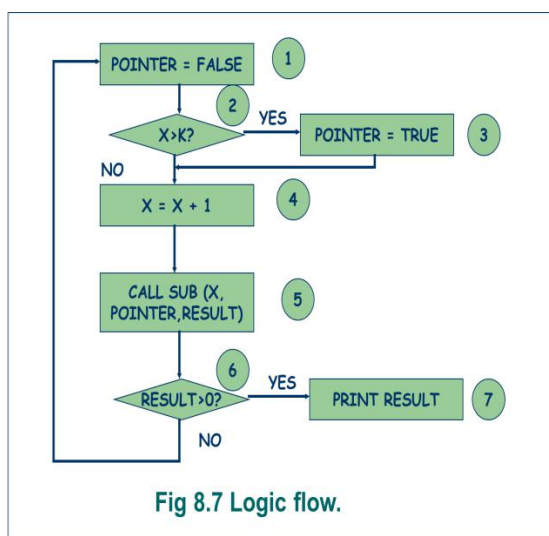
(1) 语句(覆盖)测试: 在某次测试中, 构件中的每条语句至少执行一次。

(2) 分支测试: 对代码中的每个判断点, 每个分支在某次测试中至少选择一次。

(3) 路径测试: 通过代码的每一条不同路径在某个测试中至少执行一次。

(4) 此外还有定义使用的路径测试、所有使用的测试、所有谓词使用/部分计算使用的测试和所有计算使用/部分谓词使用的测试。

例图



对于这个图:

语句覆盖: 选择 X 大于 K 使其产生正的 $RESULT$, 就可以顺序执行 1-2-3-4-5-6-7 语句。

分支覆盖: 图中有两个判断点, 因此使用两个测试用例分别执行 1-2-3-4-5-6-7 和 1-2-4-5-6-1 即可。

路径覆盖: 图中可能路径包括: 1-2-3-4-5-6-7、1-2-3-4-5-6-1、1-2-4-5-6-7、1-2-4-5-6-1 四条路径。

3. 黑白盒测试 (详见本章补充 PPT)

黑盒测试方法:

(1) 等价分类法: 将输入域划分为若干等价类。每一个测试用例都代表了一类与它等价的其他例子。如果测试用例没有发现错误, 那么对应的等价例子也不会发生错误。有效等价类的测试用例尽量公用, 以此来减少测试次数, 无效等价类必须每类一个用例, 以防止漏掉可能发现的错误。

(2) 边界值分析法: 在等价分类法中, 代表一个类的测试数据可以在这个类的允许范围内任意选择。但如果把测试值选在等价类的边界上, 往往有更好的效果, 这就是边界值分析法的主要思想。

(3) 错误猜测法: 猜测程序中哪些地方容易出错, 并据此设计测试用例。更多的依赖于测试人员的直觉和经验。

(4)因果图法：适用于被测试程序有很多输入条件，程序的输出又依赖输入条件的各种组合的情况。

白盒测试方法：

(1)语句覆盖：略；

(2)判定(分支)覆盖：略；

(3)条件覆盖：要求判定中的每个条件均按照“真”、“假”两种结果至少执行一次。

(4)条件组合覆盖：要求所有条件结果的组合都至少出现一次(比如 $A \& \& B$ ，两个条件，那么就有四种条件的组合)。

8.4 集成测试——得到一个正常运作的系统。

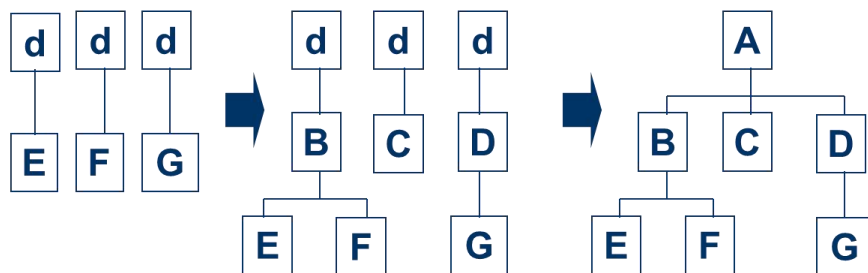
8.4.1 自底向上的集成

1.含义：使用这种测试方法的时候，每一个处于系统层次中最底层的构件先被单独测试，接着测试的是那些调用了前面已测试构件的构件。反复采用这种方法，知道所有的构件测试完毕。

2.构件驱动程序：在测试最底层的构件时，因为没有现成的已测试的构件调用底层的待测试构件，所以我们需要编写特定的代码来辅助集成。构件驱动程序就是调用特定构件并向其传递测试用例的程序。（即代替上层模块的调用程序）

3.优点：很适合面向对象的程序；编写驱动程序很简单。

4.缺点：顶层构件通常是最重要的，但是却是最后测试的。主要故障的测试将会推迟到测试的后期。



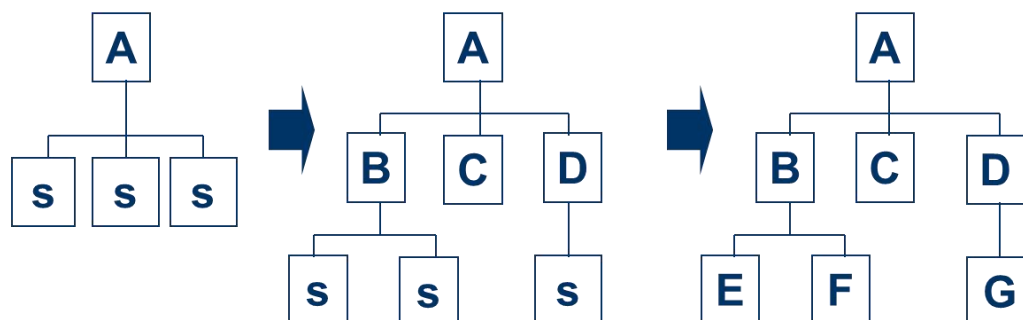
8.2.2 自顶向下的集成

1.含义：顶层构件通常是一个控制构件，是独立进行测试的。然后将被测构件调用的所有构件组合起来，作为一个更大的单元进行测试。重复这样的操作直到所有的构件都被测试。

2.桩(stub)：一种专用程序，用于模拟测试时缺少构件时的活动。桩应答调用序列，并传回输出数据，使测试能够正常的进行下去。（即代替下层模块的应答程序）

3.优点：一些功能性的设计故障或主要问题可以在测试的早期进行处理。

4.缺点：桩不容易编写，桩的正确性可能影响测试的有效性；另一个缺点是可能需要大量的桩。



8.4.3 一次性集成

1. 先测试每一个构件，然后将所有的构件一次性的集成。只适用于小型系统
2. 缺点：有些构件同时需要桩和驱动程序；一次性集成，很难发现失效原因；很难将接口故障和其他故障区分开。

8.4.4 三明治集成

1. 含义：将系统分成三层，目标层处于中间、目标层上有一层，目标层下有一层。在顶层采用自顶向下的方式集成，在较低层采用自底向上的方式集成。测试集中于目标层。
集成策略的选择不仅依赖于系统特性，还依赖于客户的期望。

8.5 测试面向对象的系统

1. 面向对象的测试有独特的特点，需要测试的时候需要一些额外的步骤。例如考察对类和对象进行考察，避免出现对象和类的过度和不足：遗漏对象、不必要的对象等。除此之外传统测试方法适用于功能，单很多方法没有考虑测试类所需的对象的状态。

2 传统测试和面向对象测试的区别：

(1) 测试用例的充分性：对过程语言而言，当系统改变时，我们可以针对改变测试是否正确，并使用原有的测试用例来验证剩余的功能是否同原来一致。但是面向对象的测试中，我们可能需要编写不同的测试用例。

(2) 面向对象趋向于小粒度，并且平常存在于构件内的复杂性常常转移到构件之间的接口上。这意味着，其单元测试较为容易，但是集成测试涉及面变得更加广泛。

(3) 传统测试和面向对象的测试主要集中在：需求分析和验证、测试用例生成、源码分析和覆盖分析。

8.6 测试计划

8.6.1 测试步骤：

- (1)制定测试目标
- (2)设计测试计划
- (3)编写测试用例
- (4)测试测试用例
- (5)执行测试
- (6)评估测试结果

8.6.2 测试计划：

描述我们将以何种方式向用户证明软件运转正确。制定测试计划的时候要充分的了解需求、功能规格说明、系统设计和代码的模块层次结构。测试计划是随着系统本身的开发而制定的。

8.6.3 计划的内容：

针对测试的每一个阶段，测试计划详细描述执行每一步所使用的方法。每一种测试方法或技术还要附有一份详细的测试用例列表。测试计划的内容就是描述我们如何测试以及为什么执行测试。

8.7 补充

- 1.故障播种：在程序中事先插入已知的故障，然后其他小组尽可能多的排查故障。通过排查播种故障占总播种故障的比例来估计遗留的真实故障比例。
- 2.可信度：表示软件无故障的可能性。

四 例题

1.在单元测试中，使用（ ）模拟被测试单元的调用和数据传递动作，（ ）模拟被测试模块的子程序。

答案显示是：构件驱动程序、桩。注意虽然这两个概念是在集成测试中介绍的，但本质上还是用来在集成测试中测试单元模块的。

2.传统测试与 OO 测试有何不同？

答案详见 8.5 小节。

3. 判断题：软件测试只能证明程序有错误，不能证明程序没有错误。

这句话是正确的，我们测试的目的就是找出尽可能的错误，但是没有找出错误并不能代表程序就没有错误。

4. 某城市的电话号码由 3 部分组成。这 3 个部分的名称与内容分别是：

地区码：空白或 3 位数字；

前 缀：非 '0' 或 '1' 开头的 3 位数字；

后 缀：4 位数字。

假定被测程序能接受一切符合上述规定的电话号码，拒绝所有不符合规定的号码，请使用等价类的思路设计测试用例。

表5.1 电话号码程序的等价类划分

输入条件	有效等价类	无效等价类
地区码	空白① 3位数字②	有非数字字符⑤, 少于3位数字⑥, 多于3位数字⑦
前 缀	从200到999之间的3位数字③	有非数字字符⑧, 起始位为'0'⑨, 起始位为'1'⑩, 少于3位数字⑪, 多于3位数字⑫
后 缀	4位数字④	有非数字字符⑬, 少于4位数字⑭, 多于4位数字⑮

解：第一步先划分等价类

第二步：

对于有效等价类可以公用。

对于无效等价类，我们采用单一变量的原则，每次测试一个无效等价类，这里只举几个例子。

测试数据	测试范围	期望结果
() 276—2345	等价类①、③、④	有效
(635) 805—9321	等价类②、③、④	有效

测试数据	测试范围	期望结果
(20A) 123—4567	无效等价类⑤	无效
(33) 234—5678	无效等价类⑥	无效
(7777) 345—6789	无效等价类⑦	无效

第九章 测试系统

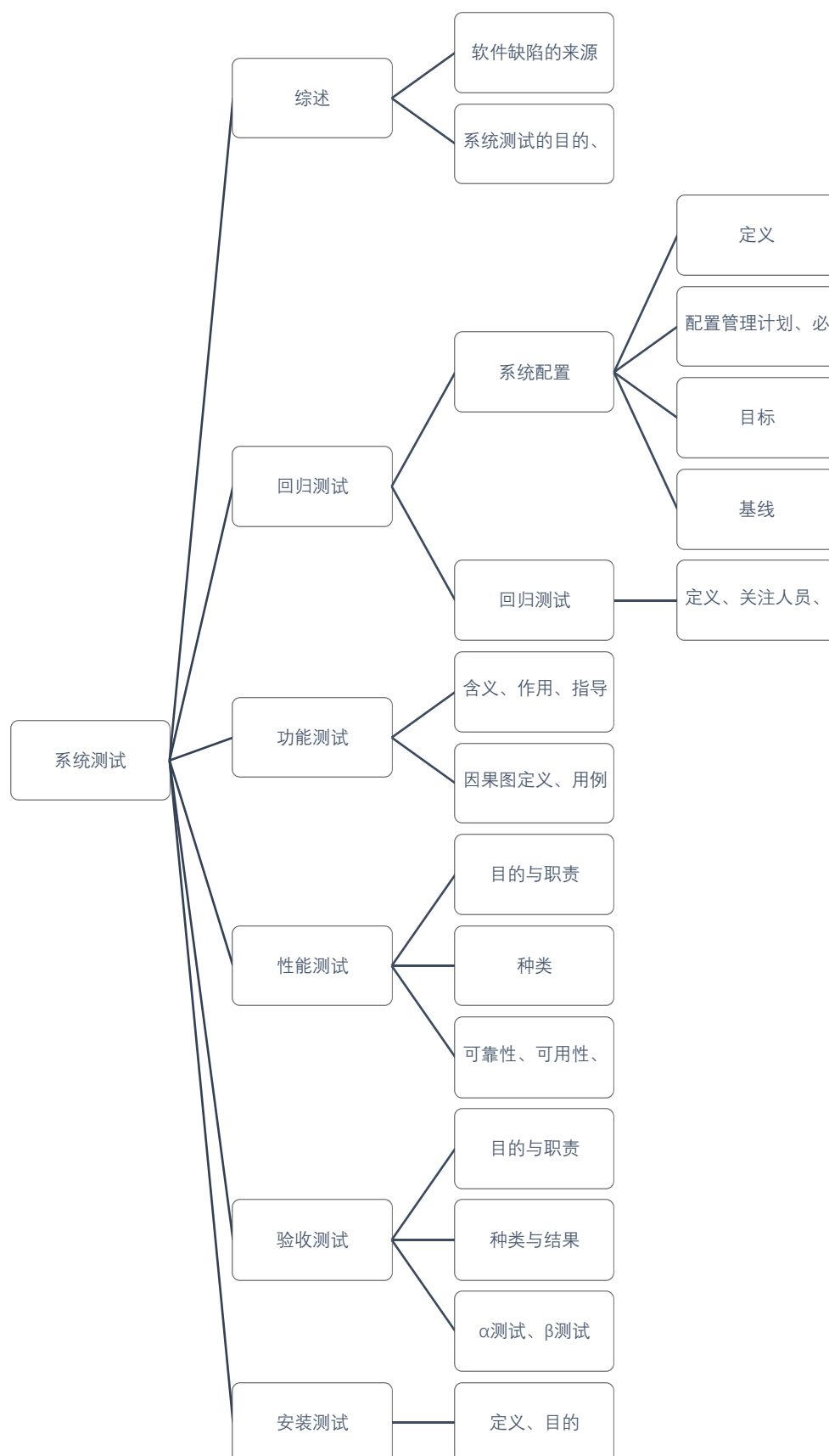
作者：张晓敏

一、章节概述

系统测试与单元测试和集成测试的不同在于，系统测试需要与整个开发团队一起工作、协调你做的工作并且接受测试小组组长的指导；而单元测试时你可以完全控制测试过程——自己设计测试数据、测试样例、运行测试；集成构件时，虽然有时独自工作，但通常是测试小组或者开发团队的一些人合作。

在本章内，我们将会讨论测试系统所包含的功能测试、性能测试、验收测试、安装测试。

二、章节框架



三、知识点解析

9.1 系统测试综述

9.1.1 软件缺陷的来源

(既然我们要讨论系统测试, 那么对软件缺陷的各种来源的讨论自然是必要的了, 毕竟排除所有可能导致软件失效的缺陷就是进行测试的目标嘛。在中文译本课本的 326 页可以找到与本内容相关的图。)

软件缺陷可能存在于软件设计开发过程中的任何一个部分。其中:

- (1)需求分析: 不正确、遗漏或者不清晰的需求;
- (2)系统设计: 对需求设计的误读, 不正确或不清晰的设计规格说明;
- (3)程序设计: 对系统设计的误读, 不正确或不清晰的设计规格说明;
- (4)程序实现: 对程序设计的误读, 不正确的文档, 不正确的语法语义;
- (5)单元/集成测试: 不完全的测试过程, 改正已有故障时引入新故障;
- (6)系统测试: 不完全的测试过程, 改正已有故障时引入新故障;
- (7)维护: 需求变化, 错误的用户文档, 负面的人为因素, 改正已有故障时引入新故障。

9.1.2 系统测试的目的

(1)测试过程应该有足够的完全性, 如果测试过程是不全面的, 故障仍可能检测不到, 所以越早检测出故障越好, 早期检测出的故障更容易改正;

(2)测试过程应该使每一个人都对系统功能感到满意, 包括用户、客户和开发人员, 因为在大型系统中, 必定会存在一些用户允许存在的, 当时不必修复的缺陷存在。

9.1.3 系统测试的主要步骤及目标

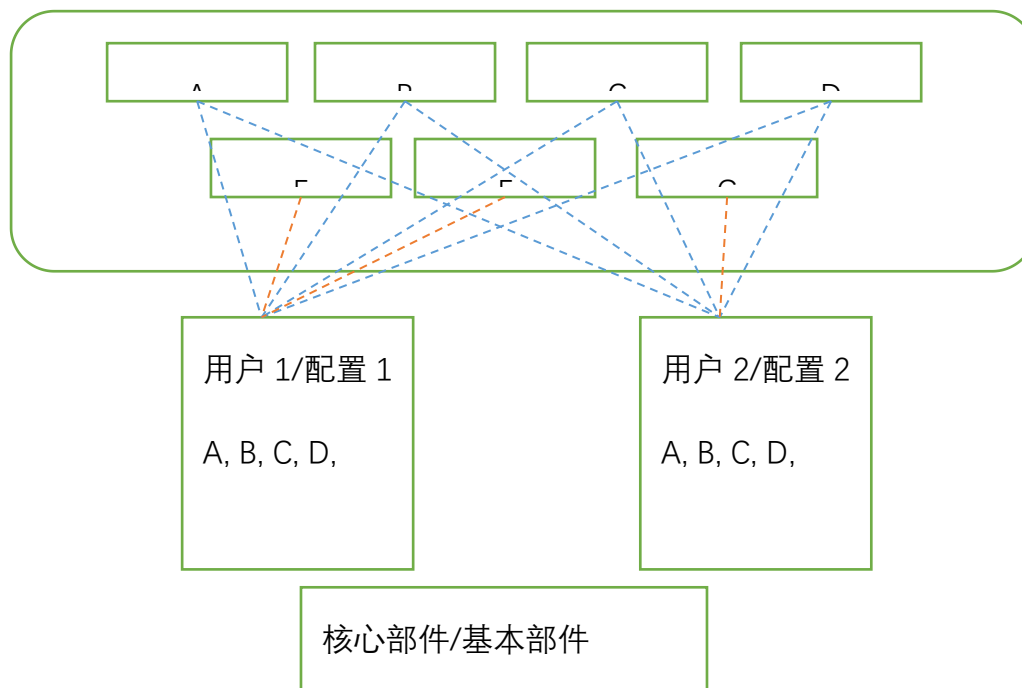
(系统测试的每一步的侧重点都有所不同, 并且一个步骤的成功依赖于它的目标与目的。在中文译文课本课本的 327 页可以找到与本内容相关的图。)

- (1)功能测试——系统功能需求
- (2)性能测试——其他软件需求
- (3)验收测试——客户需求规格说明书
- (4)安装测试——用户环境

9.2 系统配置

9.2.1 系统配置定义

向特定客户交付的系统构建的集合。



9.2.2 配置管理

对系统不同软件配置的管理及控制方法（其中既有开发，也有测试）。通过控制系统差别以降低风险，减少错误。它的重要性在于它协调测试人员与开发人员之间的工作，从而获取有效配置。

9.2.3 配置管理计划

SCM 流程旨在确保在任何时候产品的内容都是已知的、可用的，以及在设计到实施的过程中，产品的功能都是可跟踪的，可以完全控制和保护产品的内容。

9.2.4 配置计划的必要性

如果不使用适当的配置管理计划，我们就经常无法知道哪一个模块是被确定、增强或测试过的。

我们会开发出功能错误或是缺少功能的产品来，而且我们不知道哪些测试正在进行，哪些缺陷已被确定。

我们不得不重新整理已经完成了的工作。

9.2.5 管理计划对系统测试的目标

迅速准确地进行测试。

9.2.6 基线

软件文档和其他资料的集合，它们代表了产品在某一时间点的情况(以及其他参考点)。

9.2.7 配置管理计划关键的功能

- (1)每个产品元素(部件)版本的复件;
- (2)对每一个基线修改的记录;(修改内容的说明:修改位置,修改时间,修改内容等等)
- 谁进行了这个改变;
- (3)他们什么时候改变的;
- (4)改变具体内容是什么;
- (5)为什么他们要进行改变。
- (6)可能涉及的其他功能等等(从涉及的基本文档开始统计)。

9.2.8 版本(version)与改进版本/发布(release)定义

Version:针对特定系统的特定配置

Release:针对旧版本的改进版本

Version n.m=Version n and release m.

9.3 回归测试

9.3.1 回归测试定义

回归测试是用于新的版本或者改进版本的一种测试,以验证与旧版本或改进版本相比,他是否仍然以同样的方式执行同样的功能。

9.3.2 测试小组关注人群

单元测试、集成测试:主要为开发者;

功能测试、性能测试:主要为开发者;

验收测试、安装测试:主要为用户;

(因此,程序员不能参与自己负责的模块相关的测试工作。)

9.3.3 测试小组组成

- (1)专业测试人员:集中于测试开发、方法和过程;
- (2)分析员:以需求创建者的立场参与测试;
- (3)系统设计人员:了解系统运作,可以使测试工作更有目的性;
- (4)配置管理代表:出现失效或变化请求时安排变动,使变动反映在文档、需求、设计、代码或者其他开发制品中。
- (5)用户:对所发布的软件进行评估。

9.4 功能测试

9.4.1 功能测试含义与作用

测试需求设计的功能性需求。

有很高的故障检测概率（因为一项功能测试只面向一小组组件）。

9.4.2 有效的功能测试指导原则

- (1)高故障检测概率；
- (2)使用独立于设计人员和程序员的测试小组；
- (3)了解期望的动作和输出；
- (4)既要测试合法输入，也要测试不合法输入；
- (5)制定停止测试的标准。

9.5 因果图

9.5.1 因果图定义

需求中的 INPUT 称为原因,OUTPUT 称为结果。反映这种因果关系的布尔逻辑图称为因果图。

9.5.2 基于因果图的用例产生过程

(构建过程例: 中文译文版本课本 P336 的水位控制)

- (1)由布尔逻辑与约束条件产生因果图；
- (2)将因果图转化为判定表；
- (3)由判定表产生测试用例。

9.5.3 产生因果图的步骤

- (1)将需求分解为各个单独的功能；
- (2)每个功能分出原因和结果；
- (3)中间节点的产生。

9.6 性能测试

性能测试与需求的质量有密切的关系,需求文档需要足够完备才能确保性能测试的成功进行。因此需求的质量通常可以反映在性能测试的容易度上。

9.6.1 性能测试目的与职责

性能测试所针对的是非功能需求。它需要确保这个系统的可靠性、可用性与可维护性。性能测试由测试小组进行设计和执行并将结果提供给客户。

(所以性能测试通常会设计硬件和软件, 硬件工程师可能会参与测试小组。)

9.6.2 性能测试种类

压力测试(短时间内加载极限负荷, 验证系统能力)

容量测试(验证系统处理巨量数据的能力)

配置测试(测试各种软硬件配置(最小到最大))

兼容性测试(如果它与其他系统交互时)

回归测试(如果这个系统要替代一个现有系统时)

安全性测试

计时测试

环境测试

质量测试

恢复测试

维护测试

文档测试

人为因素测试/可使用性测试

9.6.3 可靠性、可用性与可维护性定义(括号内为取值范围):

可靠性: 一个系统对于给定时间间隔内、在给定条件下无失效运作的概率。(0~1)

可用性: 在给定的时间点上, 一个系统能够按照规格说明正确运作的概率。(0/1)

可维护性: 在给定的使用条件下, 在规定的时间内, 使用规定的过程和资源完成维护活动的概率。(0~1)

9.7 验收测试

9.7.1 验收测试目的与职责

验收测试的目的是使客户和用户能确定我们构建的系统真正满足了他们的需要和期望。

验收测试的编写、执行和评估都是由客户来进行的。只有在请求某个技术问题的答案的时候才会需要开发人员。

9.7.2 验收测试的种类

(1) 基准测试

由用户准备典型测试用例, 在实际安装后的系统运作并由用户对系统执行情况进行评估。

(2) 引导测试(课件译)/试验性测试(课本译)

在假设系统已经永久安装的前提下执行系统。它依赖系统的日常工作进行测试, 相对基准测试不是非常的正式与结构化。

(3) 并行测试

9.7.3 α 测试与 β 测试

α 测试：在向客户发布一个系统之前，先让来自自己组织机构或公司的用户来测试这个系统。在客户进行实际的试验性测试前先试验这个系统。

β 测试：客户的试验成为 β 测试。

9.7.4 验收测试的结果

- (1)让用户能够验证他的需求是怎样被实现的；
- (2)让客户发现需求中模糊的定义。
- (3)改需求
- (4)如果配置管理人员，则可以记录软件所有的变更。

9.8 安装测试

9.8.1 安装测试定义

在用户环境中配置系统，以测试可能因为开发环境与用户环境的不同而导致的问题。

9.8.2 安装测试目的

安装系统的完备性，验证任何可能受场所条件影响的功能和非功能特性。

=====

感谢对智库的支持，如发现本知识见解有错误或对结构内容有你的观点，欢迎扫码给我们留言。



扫码关注公众号
获取最新版本和更多科目知识见解