目录

[第一章 软件工程学概述 3](#_Toc497313321)

[1.1 软件危机 3](#_Toc497313322)

[1.2 软件工程 3](#_Toc497313323)

[1.3 软件生命周期 4](#_Toc497313324)

[1.4 软件过程 5](#_Toc497313325)

[第二章 可行性研究 11](#_Toc497313326)

[2.1 可行性研究的任务 11](#_Toc497313327)

[2.2 可行性研究过程 11](#_Toc497313328)

[2.3 系统流程图 11](#_Toc497313329)

[2.4 数据流图（DFD） 12](#_Toc497313330)

[2.5 数据字典 13](#_Toc497313331)

[2.6 成本/效益分析 14](#_Toc497313332)

[第三章 需求分析 15](#_Toc497313333)

[3.1 需求分析的任务 15](#_Toc497313334)

[3.2 与用户沟通获取需求的方法 16](#_Toc497313335)

[3.3 分析建模与规格说明 17](#_Toc497313336)

[3.4 实体-联系图 17](#_Toc497313337)

[3.5 数据规范化 17](#_Toc497313338)

[3.6 状态转换图 17](#_Toc497313339)

[3.7 其他图形工具 18](#_Toc497313340)

[3.8 验证软件需求 19](#_Toc497313341)

[第4章 形式化说明技术 19](#_Toc497313342)

[4.1 概述 19](#_Toc497313343)

[4.2 有穷状态机 20](#_Toc497313344)

[4.3 Petri网 20](#_Toc497313345)

[4.4 Z语言 21](#_Toc497313346)

[第5章 总体设计 21](#_Toc497313347)

[5.1 设计过程 21](#_Toc497313348)

[5.2 设计原理 22](#_Toc497313349)

[5.3 启发规则 23](#_Toc497313350)

[5.4 描绘软件结构的图形工具 24](#_Toc497313351)

[5.5 面向数据流的设计方法 25](#_Toc497313352)

[第6章 详细设计 28](#_Toc497313353)

[6.1 结构程序设计 28](#_Toc497313354)

[6.2 人机界面设计 29](#_Toc497313355)

[6.3 过程设计的工具 31](#_Toc497313356)

[6.4 面向数据结构的设计方法 33](#_Toc497313357)

[6.5 程序复杂程度的定量度量 35](#_Toc497313358)

[第7章 实现 36](#_Toc497313359)

[7.1 编码 36](#_Toc497313360)

[7.2 软件测试基础 38](#_Toc497313361)

[7.3 单元测试 39](#_Toc497313362)

[7.4 集成测试 40](#_Toc497313363)

[7.5 确认测试 42](#_Toc497313364)

[7.6 白盒测试技术 42](#_Toc497313365)

[7.7 黑盒测试技术 45](#_Toc497313366)

[7.8 调试 46](#_Toc497313367)

[7.9 软件可靠性 47](#_Toc497313368)

[第8章 维护 49](#_Toc497313369)

[8.1 软件维护的定义 49](#_Toc497313370)

[8.2 软件维护的特点 49](#_Toc497313371)

[8.3 软件维护过程 50](#_Toc497313372)

[8.4 软件的可维护性 51](#_Toc497313373)

[8.5 预防性维护 52](#_Toc497313374)

[8.6 软件再工程过程 53](#_Toc497313375)

[第9章 面向对象方法学引论 54](#_Toc497313376)

[9.1 面向对象方法学概述 54](#_Toc497313377)

[9.2 面向对象的概念 55](#_Toc497313378)

[9.3 面向对象建模 55](#_Toc497313379)

[9.4 对象模型 56](#_Toc497313380)

[9.5 动态模型 58](#_Toc497313381)

[9.6 功能模型 58](#_Toc497313382)

[9.7 3种模型之间的关系 60](#_Toc497313383)

[第10章 面向对象分析 60](#_Toc497313384)

[10.1 面向对象分析的基本过程 60](#_Toc497313385)

[10.2 需求陈述 61](#_Toc497313386)

[10.3 建立对象模型 61](#_Toc497313387)

[10.4 建立动态模型 64](#_Toc497313388)

[10.5 建立功能模型 65](#_Toc497313389)

[10.6 定义服务 65](#_Toc497313390)

[第11章 面向对象设计 66](#_Toc497313391)

[11.1 面向对象设计的准则 66](#_Toc497313392)

[11.2 启发规则 67](#_Toc497313393)

[11.3 软件重用 67](#_Toc497313394)

[11.4 系统分解 69](#_Toc497313395)

[11.5 设计问题域子系统 70](#_Toc497313396)

[11.6 设计人机交互子系统 70](#_Toc497313397)

[11.7 设计任务管理子系统 71](#_Toc497313398)

[11.8 设计数据管理子系统 72](#_Toc497313399)

[11.9 设计类中的服务 73](#_Toc497313400)

[11.10 设计关联 74](#_Toc497313401)

[11.11 设计优化 74](#_Toc497313402)

[第12章 面向对象实现 75](#_Toc497313403)

[12.1 程序设计语言 75](#_Toc497313404)

[12.2 程序设计风格 76](#_Toc497313405)

[12.3 测试策略 77](#_Toc497313406)

[12.4 设计测试用例 78](#_Toc497313407)

[第13章 软件项目管理 79](#_Toc497313408)

[13.1 估算软件规模 79](#_Toc497313409)

[13.2 工作量估算 80](#_Toc497313410)

[13.3 进度计划 81](#_Toc497313411)

[13.4 人员组织 83](#_Toc497313412)

[13.5 质量保证 84](#_Toc497313413)

[13.6 软件配置管理 85](#_Toc497313414)

[13.7 能力成熟度模型 87](#_Toc497313415)

黑色页码 《软件工程导论》（第5版）张海藩 清华大学出版社

# 第一章 软件工程学概述

## 1.1 软件危机

1.1.1 软件危机的介绍

软件危机，又称软件萧条（depression）或软件困扰（affliction），指一系列在软件开发和维护过程中所遇到的行业特有的严重问题。

软件危机的代表问题：  
（1）对软件开发成本和进度的估计常常很不准确  
（2）用户对已完成的软件系统不满意的现象经常发生，交流难充分  
（3）软件产品的质量往往靠不住，质检没做到  
（4）软件常常不可维护，软件重用还只是一个目标  
（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比例逐年上升  
（7）软件开发生产率提高的速度，远远跟不上计算机应用普及深入的趋势

软件工程学即研究解决这些问题的学科。从管理和技术两方面研究如何更好的开发与维护软件。

1.1.2 产生软件危机的原因

对用户要求没有完整准确认识就匆忙着手编程是许多软件开发工程失败的主要原因之一。而用户也不是第一时间就能明确自己全部的需求。开发人员不仅需要时时刻刻和用户沟通，也需要自己做大量深入细致的调查研究工作。

软件维护属于不得不进行的软件后期（开发完成后）更新与修改工作。其工作性质艰巨而复杂。提高软件的可维护性也是软件工程的研究课题之一。

1.1.3 消除软件危机的途径

软件不仅仅是程序，它是一个完整的配置，包括程序（可执行指令序列），数据（数据结构）以及相关文档（图文资料）的完整集合。

软件工程支撑环境：软件开发的各个阶段也可以使用其他的软件工具来辅助并加速这些过程，所有的软件工具的整体称为软件工程支撑环境。

## 1.2 软件工程

1.2.1 软件工程的介绍

软件工程具有下述的本质特性：  
（1）软件工程关注于大型程序的构造  
多人合作用时半年以上的程序（或系统）称为大型程序  
（2）软件工程的中心课题是控制复杂性  
（3）软件经常变化，尽可能预判将来可能的变化  
（4）开发软件的效率非常重要  
（5）和谐地合作是开发软件的关键  
（6）软件必须有效地支持它的用户  
（7）在软件工程领域中通常由具有一种文化背景的人替具有另一种文化背景的人创造产品

1.2.2 软件工程的基本原理

7条原理是确保软件产品质量和开发效率的原理的最小集合：  
（1）用分阶段的生命周期计划严格管理  
严格按照计划进行工作与管理  
（2）坚持进行阶段评审  
以便尽早发现软件开发过程中的错误  
（3）实行严格的产品控制  
对于开发过程中的无法避免的需求改变，按照严格规定进行基准（或基线）配置管理，或称进行变动控制，只有获得批准的才能进行修改  
（4）采用现代程序设计技术  
（5）结果应能清楚地审查  
（6）开发小组的人员应该少而精  
（7）承认不断改进软件工程实践的必要性

1.2.3 软件工程方法学

定义：在软件生命周期的全过程中使用的一整套技术方法的集合称为方法学（methodology），也称为泛型（paradigm）。

软件工程方法学包括三要素：（1）方法，完成软件开发的各项任务的技术方法；（2）工具，软件工程支撑环境；（3）过程，一系列规定了工作步骤的任务框架

软件工程方法学分类：

（1）传统方法学

也称生命周期方法学或结构化泛型。使用结构化技术——结构化分析、设计、实现。按照软件生命周期的过程划分阶段，然后顺序完成每个阶段任务。

缺点：这种技术要么面向行为（对数据的操作），要么面向数据，把数据和行为彼此分离。当软件规模庞大且对软件的需求是模糊的或随时间变化而变化的时候很难胜任。

（2）面向对象方法学

特点：  
①把对象作为融合了数据及操作的统一的软件构件。  
②把所有对象都划分成类  
③有继承现象  
④对象具有数据封装性

## 1.3 软件生命周期

定义（概括）：分三个时期，软件定义，软件开发，运行维护（或软件维护）

（1）软件定义时期  
又称系统分析，进一步分为：问题定义，可行性研究和需求分析

（2）开发时期  
进一步分为：总体设计，详细设计（模块设计），编码和单元测试，综合测试；前两个阶段称系统设计，后两个称系统实现

（3）维护时期  
每一次维护的本质都是一次简化版的定义开发过程

## 1.4 软件过程

软件过程描述了为了开发出客户需要的软件，什么人who在什么时候when做什么事what以及怎样做how以实现某一特定的具体目标。

生命周期模型（或过程模型）来描述软件过程，该模型规定了把生命周期划分成哪些阶段以及各个阶段的执行顺序。

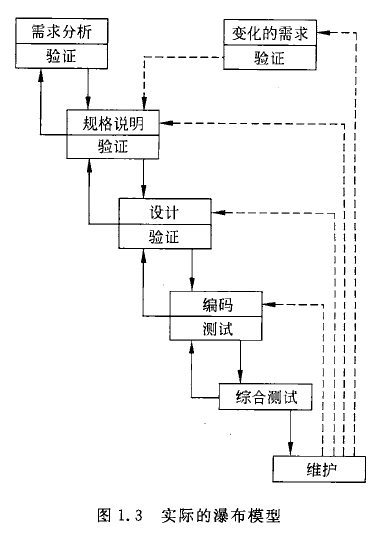
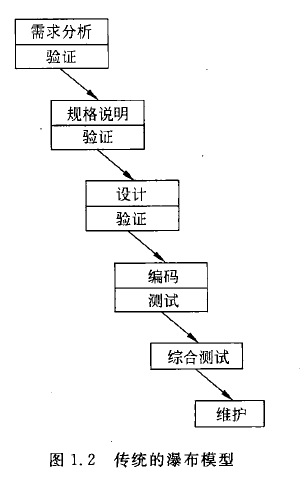
下述模型中有几点约定：a）“规格说明”作为一个阶段独立出来；b）问题定义和可行性研究归并入需求分析；c）总体和详细设计合并为“设计”

1.4.1 瀑布模型

传统瀑布模型特点：  
（1）阶段间具有顺序性和依赖性  
a）前一阶段工作完成之后，才能开始后一阶段  
b）前一阶段的输出文档就是后一阶段的输入文档  
（2）推迟实现的观点  
清楚地进行逻辑设计，尽可能推迟程序的物理实现  
（3）质量保证的观点  
a）每个阶段都必须完成规定的文档  
b）每个阶段结束前都要对所完成的文档进行评审

实际的瀑布模型带有反馈环，允许后面阶段发现前面阶段的错误，返回改正后继续进行。实线代表开发过程，虚线代表维护过程。

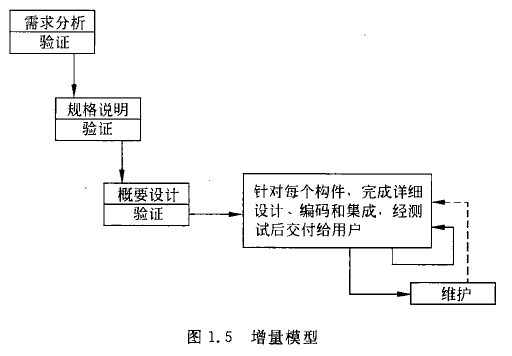
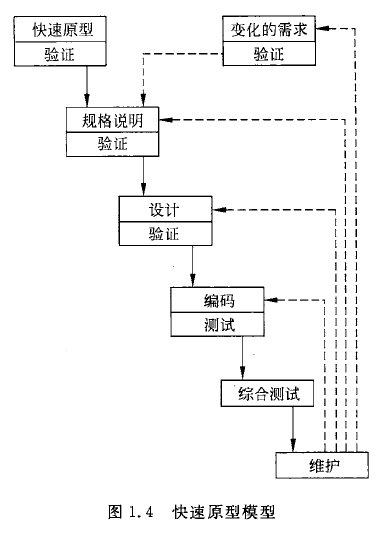
缺点：瀑布模型是基于文档的，在最终产品成型前，用户只能通过文档来了解产品。



1.4.2 快速原型模型

优点：  
（1）原型系统已经通过与用户交互而得到验证，据此产生的规格说明文档理应正确地描述了用户的需求，因此后续阶段无需较大返工  
（2）开发人员通过建立原型系统已经学到很多东西

原型系统需要尽可能地快速建造出来并且可以快速修改，其内部结构并不重要。可以使用UNIX Shell和超文本或第四代语言（4GL）构建快速原型。大部分情况下，确定了用户需求之后，原型系统即可抛弃，除了一些通用的用户界面等。



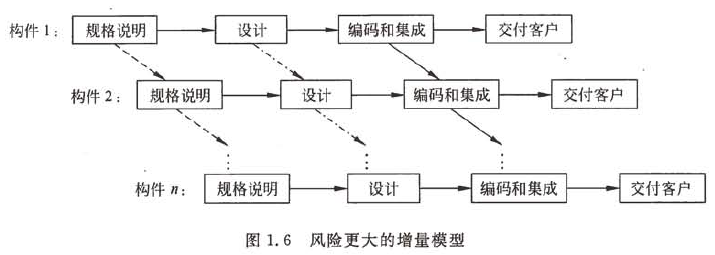
1.4.3 增量模型

也称渐增模型，把软件产品作为一系列增量构件来设计开发，每个构件由多个相互作用的模块构成。由满足基本需求开始，逐渐添加提供高级功能的构件。分解方法因产品特点和开发团队而异，但必须满足每个新构件集成到基本系统中时必须可测试。

优点：  
（1）能在较短时间内向用户提交可完成部分工作的产品  
（2）逐步增加产品功能可以使用户有较充裕的时间学习和适应新产品

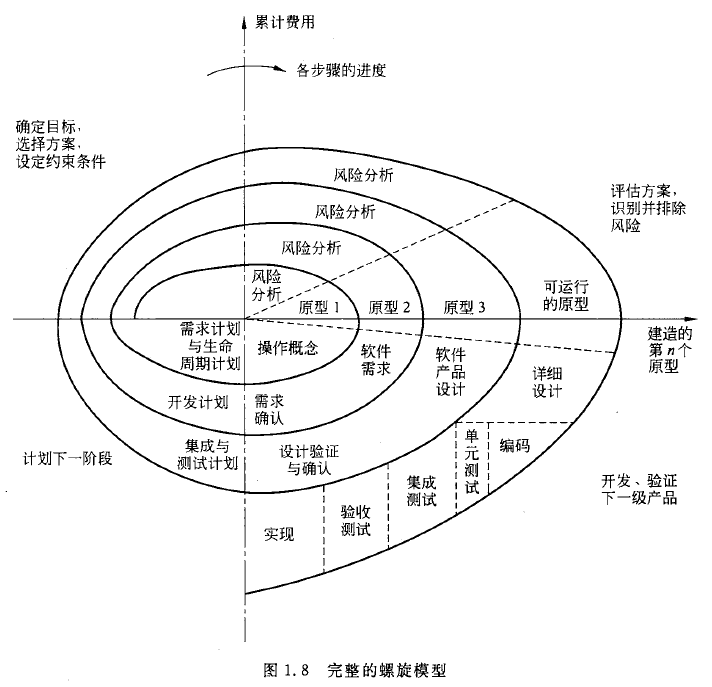
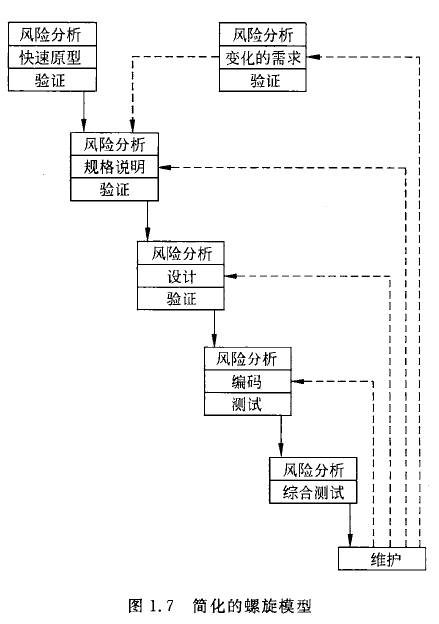
困难：把每个新构件集成到原有体系中时，必须不破坏原有产品。这对于软件的设计要求更高，即要求它必须是结构开放的。这也便于后期进行维护。

进一步，每个构件可以独立设计制造，并行开发。但是风险更大，可能多个构件最终无法统一集成到一起。



1.4.4 螺旋模型

概述：在每个阶段之前都增加了风险分析过程的快速原型模型

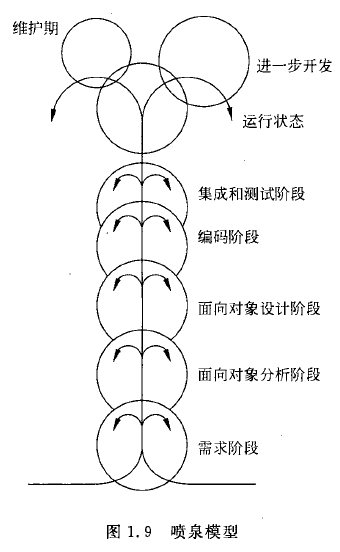


螺旋模型每个周期对应于一个开发阶段，每个象限代表一个阶段的一个步骤，处于相同象限的步骤之间是类似的。

优点：  
（1）对可选方案和约束条件的强调有利于软件重用  
（2）减少了过多测试或测试不足的风险

局限：  
（1）主要适用于内部开发的大规模软件项目，方便风险评估中止项目  
（2）风险分析需要成本也需要专业技术支持

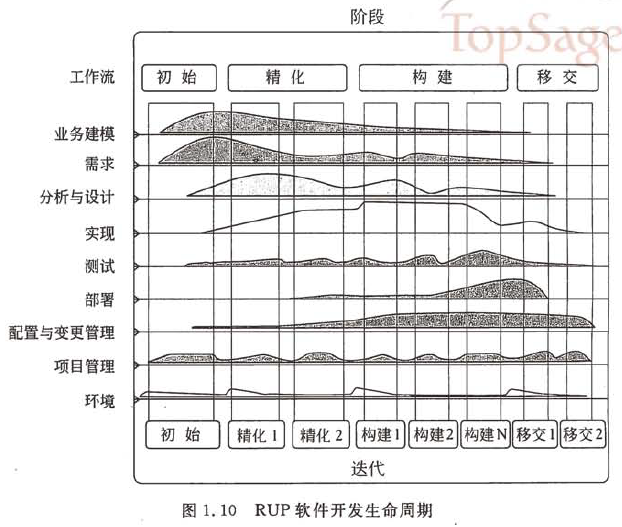
1.4.5 喷泉模型



面向对象的软件过程的一种模型，OOP软件过程不注重生命周期的各阶段划分，因为统一使用一套从一而终逐渐加强深化的对象。通过多次反复迭代相同开发步骤精益求精，一步步完善所开发系统但是又不破坏之前建立的本质。

为了避免使用喷泉模型开发时过程过分无序，应该把一个线性过程（比如快速原型模型）作为总目标。

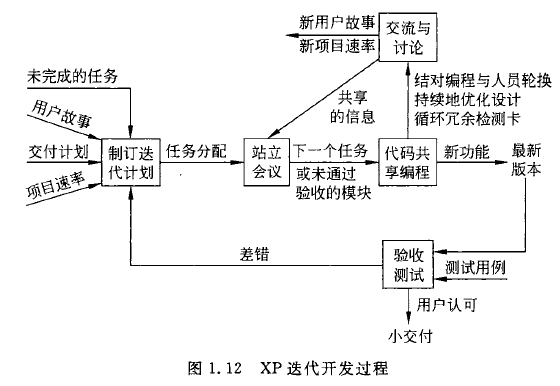
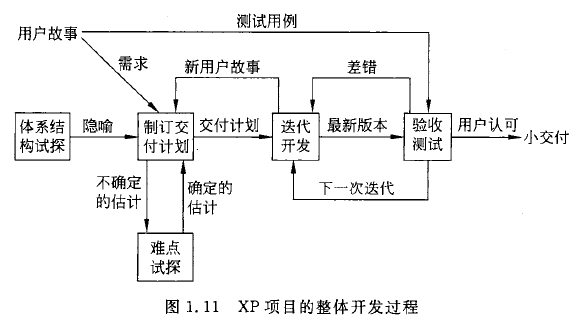
1.4.6 Rational统一过程（Rational Unified Process，RUP）



最佳实践（6条最有效的软件开发经验）：  
（1）迭代式开发  
允许在每次迭代过程中需求有些许变化，每个迭代过程以完成可执行版本结束  
（2）管理需求  
RUP采用用例分析来捕获需求，并由其驱动设计和实现。  
（3）使用基于构件的体系结构  
RUP提供了使用现有的或新开发的构件定义体系结构的系统化方法  
（4）可视化建模  
UML可视化建模语言  
（5）验证软件质量  
RUP使用内建在贯穿于整个开发过程的、由全体成员参与的所有活动中的质量评估  
（6）控制软件变更  
RUP描述了如何控制、跟踪和监控修改，以及确保迭代开发的成功

RUP软件开发生命周期  
（1）核心工作流  
9个核心工作流，前六个为核心过程工作流程，后三个为核心支持工作流程  
a）业务建模：  
b）需求  
c）分析与设计  
d）实现  
e）测试  
f）部署  
g）配置与变更管理  
h）项目管理  
i）环境  
（2）工作阶段（RUP生命周期）  
a）初始阶段：建立业务模型，定义最终产品视图，确定项目范围  
b）精化阶段：设计确定系统体系机构，制定计划，确定资源需求  
c）构建阶段：开发构件，集成，测试  
d）移交阶段  
（3）RUP迭代式开发  
RUP同时采用迭代和渐增的方式开发软件。对于每个构件（或一部分需求）都进行一次完整的生命周期，并向用户交付产品的一个可运行版本。每个生命周期的四个工作阶段也使用迭代方式完成不同工作，不同阶段完成的工作有的是类似种类的，但是投入的程度因阶段不同而异。

1.4.7 敏捷过程与极限编程



敏捷软件开发宣言由4个简单的价值观声明组成：  
（1）个体和交互胜过过程和工具  
团队合作沟通以及交互能力比单纯的软件编程能力（包括编程工具与环境）更重要  
（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  
开发团队与客户密切合作胜过单纯的合同说明  
（4）响应变化胜过遵循计划

根据敏捷价值观提出的软件过程统称为敏捷过程，其中最重要的是极限编程（eXtreme Programming，XP）。指把好的开发实践运用到极致，广泛用于需求模糊且经常改变的场合。

极限编程的有效实践：  
（1）客户作为开发团队的成员  
（2）使用用户素材  
用户素材：正在进行的关于需求的谈话内容的助记符  
（3）短交付周期  
（4）验收测试  
（5）结对编程  
结对编程：两名开发人员在同一台电脑上共同编写解决同一个问题的代码，通常一人编码，另一人审查与测试，保证代码正确性与可读性。  
（6）测试驱动开发  
先设计测试方案再编程  
（7）集体所有  
开发团队每个成员都有更改代码的权利  
（8）持续集成  
一天之内多次集成系统，不断回归测试  
（9）可持续的开发速度  
对于开发人员工作时间的规定  
（10）开放的工作空间  
方便全体参与者交流讨论  
（11）及时调整计划  
（12）简单的设计  
（13）重构  
代码重构：不改变系统行为的前提下，重新调整和优化系统的内部结构，以提高性能  
（14）使用隐喻

用户故事：类似于用例，但相对简单，通常仅描述功能需求

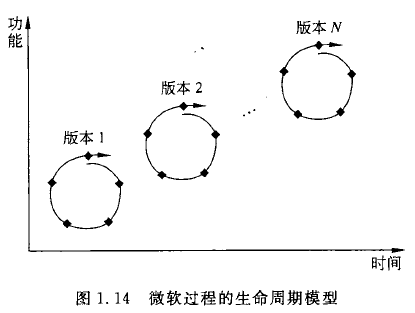
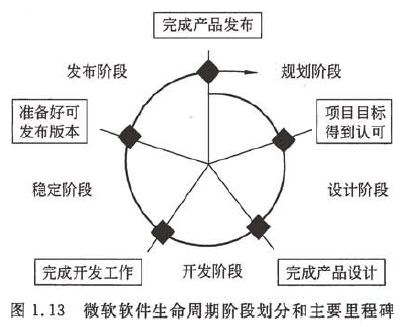
项目速率：实际开发时间和估计时间的比值

站立会议：与会人员站立，缩短会议时间，提高工作效率

1.4.8 微软过程

微软过程准则  
（1）项目计划应该兼顾未来的不确定因素  
（2）用有效的风险管理来减少不确定因素的影响  
（3）经常生成并快速测试软件的过渡版本  
（4）采用快速循环，递进的开发过程  
（5）用创造性的工作来平衡产品特性和产品成本  
（6）项目进度表应该具有较高稳定性和权威性  
（7）使用小型项目组并发地完成开发工作  
（8）在项目早期把软件配置项基线化，项目后期则冻结产品  
（9）使用原型验证概念，对项目进行早期论证  
（10）把零缺陷作为追求目标  
（11）里程碑评审会的目的是改进工作，切忌相互指责

微软软件生命周期（5个阶段）  
（1）规划阶段  
a）确定产品目标  
b）获取竞争对手的信息  
c）完成对客户和市场的调研分析  
d）确定新版本产品应该具备的主要特性  
e）确定相对前一版本，新版应该解决的问题和新增功能



（2）设计阶段  
（3）开发阶段  
（4）稳定阶段  
产品测试和调试  
（5）发布阶段

微软过程模型  
每个生命周期发布一个递进的软件版本，各个生命周期持续、快速地迭代循环

# 第二章 可行性研究

可行性研究的目的：用最小的代价在尽可能短的时间内确定问题是否能够解决

## 2.1 可行性研究的任务

实质上是进行一次简化了的系统分析和设计过程，然后评估时间、金钱和人力成本

可行性研究的方面：  
（1）技术可行性  
（2）经济可行性  
（3）操作可行性  
（4）法律，社会等方面

可行性研究的成本一般占预期工程总成本的5~10%

## 2.2 可行性研究过程

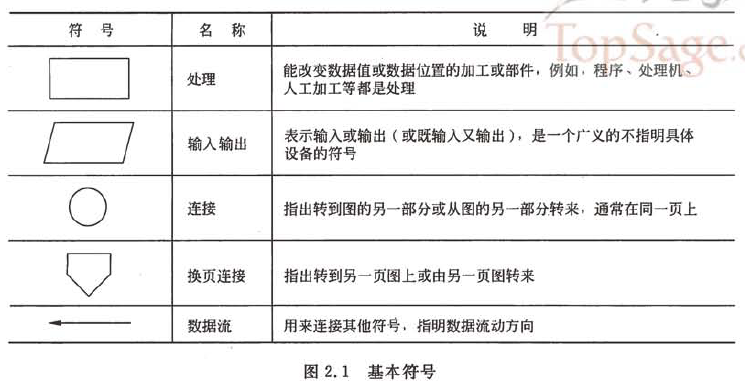
步骤：  
（1）复查系统规模和目标  
（2）研究目前正在使用的系统  
即如果用户委托的被开发系统有一个“前身”正在被用户使用，需要了解将要被替代的系统的特点和经济效益等。要了解的是“旧”系统能做什么，有什么外部接口和限定条件会继续留给新系统，不要过分注重旧系统的实现细节（如果需要耗费大量时间）  
（3）导出新系统的高层逻辑模型  
由现有系统的物理系统导出现有系统的逻辑模型，设想目标系统的逻辑模型  
（4）进一步定义问题  
前四步构成一个循环。本步和用户复查问题定义，工程目标等，进行确认。  
（5）导出和评价供选择的解法  
从技术角度出发考虑解决问题的不同方案，导出较高层次（抽象）的若干个物理解法；进而根据几种可行性评估初步评价一些方案，对于3个方面都可行的候选方案初步制定实现进度表（只是估计生命周期每个阶段的工作量）  
（6）推荐行动方针  
推荐一个最好的解法。主要考虑的因素是经济效益  
（7）草拟开发计划  
包括工程进度表、各种资源的需求情况（什么时候使用和时长）、生命周期每个阶段的成本，需求分析阶段的进度表和成本估计  
（8）书写文档提交审查  
把所有步骤工作结果写成文档

## 2.3 系统流程图

系统流程图是概括描绘物理系统的传统工具。它只是物理数据流图，不是程序流程图，不负责描绘对数据进行加工处理的控制过程。

系统流程图的习惯画法是使信息在图中从顶向下或从左向右流动

2.3.1 符号





2.3.2 例子

2.3.3 分层

对于复杂系统的描绘可以采用分层方式，先描绘一个大致总样，然后对每个关键元素再进行系统流程图描绘，可以分别存储这些多层流程图。

## 2.4 数据流图（DFD）

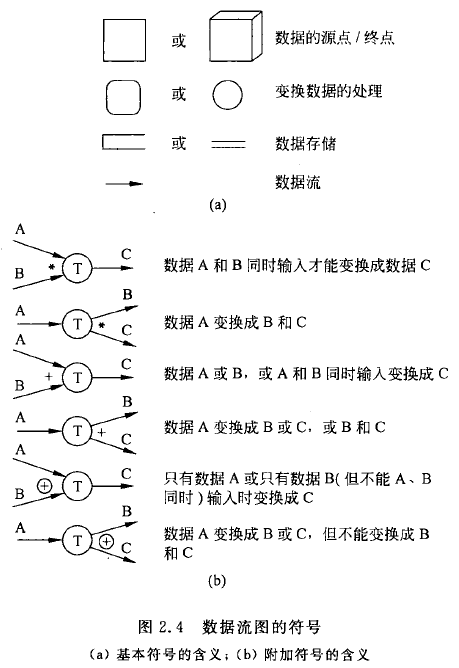
定义：描绘信息流和数据从输入移动到输出的过程中所经受的变换，是系统逻辑功能的图形表示，不描绘具体的物理部件。和程序流程图中的控制流有本质不同。

2.4.1 符号

数据流中的处理元素、数据存储元素，都不一定是一个单独的确定的程序或者元件，具体的数据处理程序和存储的细节不在DFD中表示

在DFD中也忽略出错处理、内务细节处理（如打开关闭文件）等，只是描绘“做什么”，不是“怎么做”

在DFD中，如果需要画同一个事物并且使用相同的符号（即重复的元素）n次，则在这个符号的一个角上画n-1条短斜线做标记



2.4.2 例子 p42/64

（1）先根据问题提取出4种成分，即上图（a）中

（2）做出基本系统模型

（3）把基本系统模型细化，添加数据存储，分解数据处理，当进一步分解将涉及如何具体实现一个功能时就应该停止

2.4.3 命名 p45/67

数据流的成分命名原则

（1）为数据流（或数据存储）命名

（2）为处理命名

（3）沿用实际的数据源/终点名称

2.4.4 用途

数据流图用以描述现有系统和目标系统的构想，当一张数据流图中包含的处理多于5-9个时，应该考虑分层绘制。数据流图描述系统的功能流程（不过不描述操作细节），系统流程图描述系统的物理实现机制。

数据流图可以进一步辅助物理系统设计，根据不同“处理”的定时要求（是否同一时间完成，或积攒至一定程度批量处理），可以在数据流图上以虚线画出自动化边界，每组边界内部可能意味着一个不同的物理系统。不同的自动化边界划分提供了不同的系统设计初步方案，便可进行可行性评估。

## 2.5 数据字典

数据流图和数据字典共同构成系统的逻辑模型，也是系统的规格说明。数据字典是关于数据的信息的集合，也是对数据流图中包含的所有元素的定义的集合。

2.5.1 数据字典的内容

包含对4类元素的定义：  
（1）数据流  
（2）数据流分量（数据元素  
（3）数据存储  
（4）处理

对“数据处理”的定义使用IPO图或PDL描述更方便

除了定义，还需要包括数据的其他信息：  
（1）一般信息（名字，别名，描述）  
（2）定义（数据类型，长度，结构）  
（3）使用特点（值域，使用频率，使用方式——输入、输出、本地或条件值）  
（4）控制信息（来源，用户，使用权，改变权等）  
（5）分组信息（父结构，从属结构，物理位置——记录、文件和数据库等）

2.5.2 定义数据的方法

定义的原则就是自顶向下分解数据为数据元素，当分解到不需要进一步定义每个元素时即可。

由数据元素组成数据的方式，可以分为4种类型：  
（1）顺序 两个或多个分量的顺次连接  
（2）选择 从两个或多个可能的元素中选取一个  
（3）重复 把指定的分量重复零次或多次  
对于重复类型，一般标明重复次数的上下限，如果上下限一致表示重复次数确定  
（4）可选 一个分量可有可无（重复零次或一次）  
可选是重复的特例，但是为了清晰简洁

数据元素组成算符：  
（1）= 等价于或定义为  
（2）+ 和，连接两个分量  
（3）[…|…|…] 或，从括号中列出的若干分量选择一个  
（4）{} 重复括号内的分量  
都可以表示上下限  
（5）() 括号里的分量可选

定义数据举例：p48/70

2.5.3 数据字典的用途

统一数据元素的命名术语，便于开发团队沟通一致

开发数据库的第一步

2.5.4 数据字典的实现

大型软件系统开发过程中数据字典使用CASE“结构化分析与设计工具”来维护，人工维护过于困难

数据字典举例：p49/71

## 2.6 成本/效益分析

2.6.1 成本估计

应使用几种不同的估算技术以便互相参考校验  
（1）代码行技术  
根据经验和历史数据估计实现一个功能需要的源程序代码行数，用每行代码的平均成本（取决于软件复杂程度和工资）计算即可  
（2）任务分解技术  
和程序设计思想类似，将项目分解进行人力估算（人力乘以平均工资）  
典型环境下各个开发阶段需要使用的人力百分比p50/72  
（3）自动估计成本技术  
采用软件工具完成这项工作，但是需要丰富的历史数据和数据库系统支持

2.6.2 成本/效益分析的方法

第一步是估计开发成本、运行费用和新系统将带来的经济效益。

经济效益：效益的来源有两种，一种是带来的收入或营业额，一种是带来的公司运营总成本的减少。效益不是净收入，因此并不减去当前项目成本。

运行费用：系统的操作费用（操作人员数目，工作时间，消耗物资）和维护费用

成本效益分析时一般假定软件生命周期为5年（取下限便于准确估算）

1.货币的时间价值  
考虑通货膨胀和利率的基本因素  
2.投资回收期  
即根据收益计算出的能够回本的时间  
3.纯收入  
将未来的每年的纯收入换算至同一年进行比较  
4.投资回收率  
即投资回报的比率，不是简单的纯收入比支出，而是将项目资金运转过程和银行储蓄进行横向比较，以便估计投资回收率是否高于银行利率。  
将项目投资运转过程套入银行储蓄体系：把数目等于投资额的资金存入银行，每年年底从银行取出等于当年预期效益的资金，在总时间达到系统寿命时，银行账户刚好取空，即无法再创造任何经济价值。  
P为投资额，F为每年效益，j为投资回收率，n为系统寿命（一般假设为5）

# 第三章 需求分析

需求分析的任务：“系统必须做什么”  
对目标系统提出完整清晰准确的要求，是软件定义时期最后一个阶段

《软件需求规格说明书》是需求分析阶段结束的文档

需求分析的结构化分析方法的准则：  
（1）必须理解并描述问题的信息域，根据这条准则应该建立数据模型  
（2）必须定义软件应完成的功能，这条准则要求建立功能模型  
（3）必须描述作为外部事件结果的软件行为，这条准则要求建立行为模型  
（4）必须对描述信息，功能和行为的模型进行分解，用层次的方式展示细节

## 3.1 需求分析的任务

3.1.1 确定对系统的综合要求

1.功能需求

2.性能需求  
指定系统必须满足的定时约束或容量约束，包括响应时间，磁盘容量，安全性等

3.可靠性和可用性需求  
指定系统的容错率容故障性等

4.出错处理需求  
分两种：第一是系统对环境错误应该怎样响应，第二种是系统本身错误的检测；应该对系统本身错误的检测集中在系统关键部分，应该尽可能少，而主要精力集中于开发正确的系统

5.接口需求  
描述系统与它的环境通信的格式。包括：用户接口需求，软件、硬件接口需求，，通信接口需求等。

6.约束  
设计约束或实现约束，即开发过程中应该遵守的限制条件。包括：精度，工具语言，工程标准，硬件平台等。

7.逆向需求  
说明软件系统不应该做什么，只考虑哪些能够澄清真实需求消除误解的情况

8.将来可能提出的要求  
现在暂时不属于开发范畴，但将来可能会提出，应该在开发过程中对这种情况做准备

3.1.2 分析系统的数据要求  
采用建立数据模型的方法，即数据结构。可以使用数据字典（不直观）或图形工具描述数据结构

3.1.3 导出系统的逻辑模型  
通常用数据流图，实体-联系图，状态转换图，数据字典和主要的处理算法（IPO图）描述

3.1.4 修正系统开发计划  
根据进一步分析获得对系统的进一步了解，可能会修正既定的开发计划

## 3.2 与用户沟通获取需求的方法

3.2.1 访谈

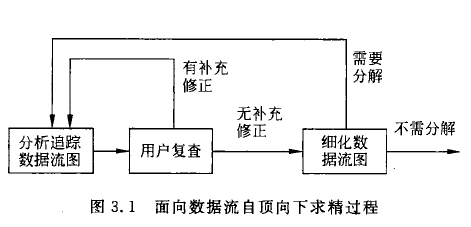
形式：  
正式访谈：分析员提出事先准备好的问题  
非正式：自由交流意见和想法  
问卷调查：调查大量人员意见，书面回答比口头回答更准确

情景分析技术  
由分析员使用目标系统（构想的）解决某个具体问题，然后由客户对结果进行评价从而获知客户可能未提及的具体要求

情景分析技术的用处：  
（1）为用户演示目标系统行为，让用户进一步思考和提供他们的需求与设想  
（2）便于用户理解，也能激发用户参与需求分析的积极性

3.2.2 面向数据流自顶向下求精

结构化分析方法就是面向数据流自顶向下逐步求精进行需求分析的方法，需求分析需要把数据流和数据存储定义到元素级。一般从数据流的输出端往回着手分析系统必须具备的数据组成元素以及算法。这个过程中进一步细化数据流图，添加更多的数据元素信息到数据字典，用IPO图简明描述算法。然后反复和用户推敲复查迭代操作过程。



3.2.3 简易的应用规格说明技术

典型过程：p60/82

这个技术其实是一种面向团队的需求收集方法，而不是开发者单向的需求分析流程

3.2.4 快速建立软件原型

特点：  
“快速”  
原型只要不会产生误解，不必在乎缺陷  
“容易修改”

方法与工具：  
（1）第四代技术  
使用这种技术可以快速生成可执行代码  
（2）可重用的软件构件  
使用以往已有的软件构件来装配原型  
（3）形式化规格说明和原型环境  
形式化规格说明语言代替自然语言规格说明技术，从而可以使用软件工具翻译成可执行代码。形式化规格说明语言也便于修改和设计。

## 3.3 分析建模与规格说明

3.3.1 分析建模

数据模型（或信息模型）：使用实体-联系图

功能模型：以数据流图为基础

行为模型：状态转换图（状态图）

3.3.2 软件需求规格说明

需求分析阶段得出的最主要文档，可以使用自然语言书写，也可以使用形式化说明技术

## 3.4 实体-联系图

和关系型数据库设计中的E-R模型几乎完全一致

3.4.1 数据对象

定义：是对软件必须理解的复合信息的抽象，复合信息是指具有一系列不同性质或属性的事务

数据对象只有数据而没有操作，即不是OOP范型中的对象。

3.4.2 属性

3.4.3 联系

3.4.4 实体-联系图的符号

## 3.5 数据规范化

和关系型数据库设计中的数据规范化内容一致

范式（normal forms）的级别特点：  
（1）范式级别越高，存储同样数据就需要分解成更多的表  
（2）范式级别越高，数据存储结构与实际问题结构之间的匹配程度也下降，需求变化时数据的稳定性较差  
（3）范式级别越高，表越多，访问性能和速度下降

## 3.6 状态转换图

状态图描绘系统的状态及引起系统状态转换的事件来表示系统的行为

3.6.1 状态

定义：状态是任何可以被观察到的系统行为模式，一个状态代表系统的一种行为模式。状态规定了系统对事件的响应方式

状态图中定义的状态：  
初态——一个状态图只有一个初态  
中间状态——  
终态——可以有多个

3.6.2 事件

定义：事件是引起系统做动作或（和）转换状态的控制信息

3.6.3 符号

初态——实心圆

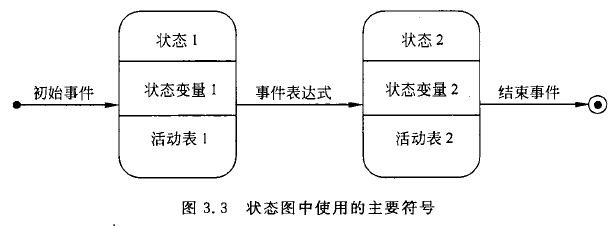
终态——同心圆，内圆实心

中间状态——圆角矩形，从上至下包括：状态名称（必须）、状态变量和值（可选）、活动表（可选）

活动表语法格式：  
事件名（参数表）/动作表达式  
a）事件名：可任取，也有3种标准选择（entry, exit, do）表示进入或退出或处于该状态的动作  
b）参数表根据需要添加，无要求  
c）动作表达式描述应做的具体动作

状态转换：  
使用箭头表示，箭头上方描述触发转换的事件表达式，如果未指定事件表达式则表示源状态内部活动执行完后自动转换

事件表达式的语法：  
事件说明[守卫条件]/动作表达式  
a）事件说明的语法：  
事件名（参数表）  
b）守卫条件：是一个布尔表达式，当a）事件发生和（或）布尔表达式为真时，执行状态转换；守卫条件和事件说明都是可选的，都是事件表达式的一种“条件”。  
c）动作表达式描述状态转换过程中执行的操作



3.6.4例子p67/89

## 3.7 其他图形工具

3.7.1 层次方框图p68/90

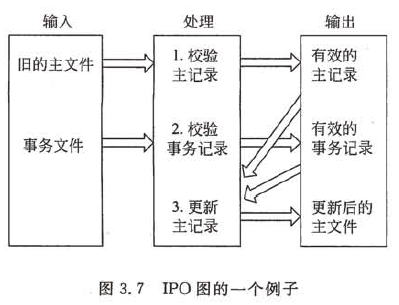
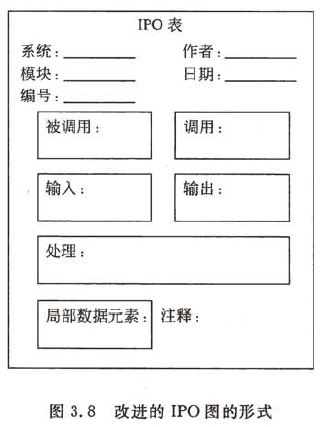
即一种树形结构分层分析图，每一层将上一层的细节细化

3.7.2 Warnier图 p69/91

基于大括号的横向树形结构图，加入了重复和条件约束的信息，比层次方框图高级

3.7.3 IPO图

IPO图是输入，处理，输出图的简称，也用来描述这些内容，下图是原始的IPO图示例

改进的IPO图（或IPO表）比原始的图更加有用，用来描述数据处理或主要算法的内容，IPO图也可以进一步在后续设计阶段继续完善，作为设计阶段的文档。

## 3.8 验证软件需求

3.8.1 从哪些方面验证软件需求的正确性

（1）一致性：所有需求必须一致，各需求间彼此不能矛盾

（2）完整性：需求必须完整，规格说明书应该包括用户需要的每一个功能或性能

（3）现实性：也就是需求是否可以实现

（4）有效性：必须证明需求是正确有效的，确实可以解决用户的问题

3.8.2 验证软件需求的方法

（1）一致性验证

自然语言的需求分析——只能用人工验证，但是无法保证效果

形式化陈述语言——可以使用软件工具验证一致性

（2）现实性验证

根据以往经验判断，或采用仿真或性能模拟技术辅助分析

（3）完整性和有效性

通过让用户试用原型系统来由用户验证完整性与有效性

3.8.3 用于需求分析的软件工具

软件工具需要满足的要求：  
（1）必须有形式化的语法（或表），因此计算机可以自动处理  
（2）使用这个软件工具可以导出详细文档  
（3）必须提供分析（测试）规格说明书的不一致性和冗余性的手段，还有能够产生一组报告指明对完整性分析的结果  
（4）使用这个软件工具之后，能够改进通信状况

RSL（需求陈述语言）；ASSM（抽象系统语义模型）数据库

PSL/PSA（问题陈述语言/问题陈述分析程序）；CADSAT（计算机辅助设计和规格说明分析工具）

# 第4章 形式化说明技术

## 4.1 概述

4.1.1 非形式化方法的缺点

（1）矛盾

（2）二义性

（3）含糊性：过于宽泛和概括的描述

（4）不完整性：对于需求或功能的叙述不完整，没有覆盖所有可能情况的具体对应方法

（5）抽象层次混乱：在非常抽象的陈述中混进关于细节的低层次陈述

4.1.2 形式化方法的优点

（1）能够简洁地准确地描述物理现象，对象或动作的结果

（2）可以在不同的软件工程活动之间平滑地过渡

（3）提供了高层确认的手段，即可以被科学的证明

4.1.3 应用形式化方法的准则

（1）应该选用适用于当前项目的形式化说明技术

（2）应该形式化，但不要过分形式化。形式化技术还不适用于描述系统的所有方面，如用户界面。

（3）应该估算额外的成本

（4）应该有形式化方法顾问随时提供咨询

（5）不应该放弃传统的开发方法，把多种方法集成一起是可能的

（6）应该使用自然语言注释形式化的规格说明书

（7）不应该放弃原有的保证质量标准的行动

（8）不应该盲目依赖形式化方法

（9）软件测试的重要性并没有因此降低

（10）应该重用软件构件，同时形式化方法说明的软件构件由于有清晰的定义和接口而有了更好的重用性

## 4.2 有穷状态机

是一种表达规格说明的形式化方法

4.2.1概念

状态转换图和状态转换表都可以描述有穷状态机，即状态机只是一个技术概念，需要转换图或转换表等其他表达技术来表达。p78/100

有穷状态机包括5个部分：可以表示为一个5元组(J,K,T,S,F)  
（1）状态集J：有穷的非空集  
（2）输入集K：有穷的非空集  
（3）转换函数T：从(J-F)×K×P到J的转换函数  
（4）初始态S：  
（5）终态集F：

为了对一个系统进行规格说明，把5元组扩展为6元组，添加一个谓词集P，转化规则形式为：  
当前状态[菜单]+事件[所选择的项]+谓词→下一个状态

谓词可以是一个状态也可以是一个事件，可以理解为不同于当前状态和触发事件之外的来自于系统的额外约束条件。

4.2.2 例子p79/101

4.2.3 评价

优点：比数据流图技术更精确；可以很容易地转变为设计或程序代码

缺点：系统规模增大，核心的三元组数量会激增；而且也没有处理定时需求的能力

## 4.3 Petri网

4.3.1 概念

Petri网可以有效地描述并发活动，以及状态和时间因素的联系

包含4种元素，四元组C=(P,T,I,O)  
（1）一组位置P：有穷位置集  
（2）一组转换T：有穷转换集，且T和P不相交  
（3）输入函数I：T→P∞由转换到位置无序单位组bags的映射  
（4）输出函数O：T→P∞由转换到位置无序单位组的映射

带标记的Petri网是一个5元组C=(P,T,I,O,M)，标记也称为权标token。p83/105  
M：P→{非负整数集（容量和P一致）}

另一个扩展是禁止线的引入p83/105

4.3.2 例子

只是一个简单例子p84/106

## 4.4 Z语言

使用Z语言需要具备集合论，函数，数理逻辑等方面的知识。

4.4.1 简介

Z语言描述的最简单的形式化规格说明含有4个部分：  
（1）给定的集合，数据类型及常数  
（2）状态定义  
（3）初始状态  
（4）操作

4.4.2 评价

Z语言是应用得最广泛的形式化语言

特点：  
（1）可以比较容易地发现用Z写的规格说明的错误  
（2）Z说明符非常精确，Z规格说明也十分严格使用Z说明符  
（3）使用Z语言可以降低软件开发费用，增加了规格说明书写时间，但减少了开发过程的总时间  
（4）可以用自然语言改写Z规格说明便于用户阅读，比直接使用自然语言更正确  
（5）如果要证明规格说明的结果正确，需要比较高的数学水平，但是编写Z规格说明不需要非常高的要求  
（6）Z语言不适于说明并发性

# 第5章 总体设计

## 5.1 设计过程

设计过程分为两个阶段：  
（1）结构设计——模块划分与关系设计，总体设计  
（2）过程设计——模块的处理过程设计，详细设计

总体设计又称概要设计或初步设计，分为两个主要阶段：  
（1）系统设计阶段：确定系统的具体实现方案  
（2）结构设计阶段：确定软件结构

设计过程的9个步骤：  
（1）设想供选择的方案  
常用数据流图的不同处理分组来预想不同的解决方案  
（2）选取合理的方案  
至少选取高中低三种成本的方案，每个合理的方案需要具备4个资料  
a）系统流程图  
b）组成系统的物理元素清单  
c）成本/效益分析  
d）该方案的进度计划  
（3）推荐最佳方案  
除此之外还要指定一个详细的实现计划——工程网络  
（4）功能分解  
把复杂的功能分解  
（5）设计软件结构  
软件结构（由模块组成的层次系统）可以用层次图或结构图描绘，或者由细化到适当程度的数据流图映射出软件结构  
（6）设计数据库  
（7）制定测试计划  
在开发早期就应该考虑测试问题，便于在设计开发过程中提高后续可测试性  
（8）书写文档  
a）系统说明：所选方案的4个基本资料（见第2条），精化数据流图，软件结构描绘，模块算法描绘（IPO图等），模块间接口关系，需求、功能、模块三者间的交叉参照关系  
b）用户手册：需求分析阶段就应该产生初步的用户手册  
c）测试计划：测试策略，方案，预期结果，进度计划等  
d）详细的实现计划  
e）数据库设计结果  
（9）审查和复审  
项目组审查，用户复审

## 5.2 设计原理

5.2.1 模块化

模块的定义顾名思义，但按照模块的定义，过程、函数、子程序和宏都可以作为模块。此外还有对象和方法（或服务）。

模块化：把程序划分成独立命名且可独立访问的模块，每个模块完成一个子功能。

模块化程度不能过高或过低，对于每个项目有一个理想的区间，程序复杂程度度量和启发式规则可以帮助决定合适的模块数目

模块化不仅便于理解和测试以及修改重用，也便于分工协作（不同难度的模块由不同经验等级的人负责）与管理。

5.2.2 抽象

抽象只是一个逐层分解模块化和逐步求精的共通理念，即一切都是在不同的抽象层次到最终具体的层次。

5.2.3 逐步求精

Miller法则：一个人在任何时候都只能把注意力集中在7±2个知识块上

逐步求精就是从上至下从抽象到具体，优先解决优先级较高的问题，最后将所有问题解决。

5.2.4 信息隐藏和局部化

信息隐藏原理：每个模块内包含的信息（过程和数据）对于不需要这些信息的模块而言，是不能访问的。

局部化：一种目的为信息隐藏的技术，比如在模块中使用局部数据元素

这是一种很接近OOP原理的思想，只是这里的“对象”是模块。

5.2.5 模块独立

类似于对象（类）之间的独立，模块的独立程度可以由两个定性标准度量：

（1）耦合  
耦合衡量不同模块彼此间互相依赖（连接）的紧密程度，取决于模块间接口的复杂程度。设计中追求尽可能松散耦合的系统。  
耦合程度：  
a）数据耦合——模块之间通过参数交换信息，交换的仅仅是数据；是低耦合，最佳  
b）控制耦合——传递的有控制信息；中等耦合，少用  
c）特征耦合——接受数据的模块只用到了传输的数据结构的一部分数据；数据不安全，少用  
d）公共环境耦合——两个或多个模块通过一个公共数据环境相互作用；复杂程度取决于耦合模块数目；如果只有两个模块使用公共环境，是低耦合或介于中低之间的耦合，可用  
e）内容耦合——最高程度的耦合，应坚决避免  
①一个模块访问另一个的内部数据  
②一个模块不通过正常入口转到另一个内部  
③两个模块有一部分程序代码重叠（汇编程序）  
④一个模块有多个入口，即多个功能

模块入口/出口：并非是一个准确定义的术语，不能理解为接口也不能理解为调用，只能理解为模块内部的数据流入口和出口。入口出口数目决定了模块的功能个数。单一功能的简单模块只有一个流程显然只有一个入口/出口。

（2）内聚  
内聚衡量一个模块内部各个元素彼此结合的紧密程度，理想内聚模块只做一件事。高内聚最佳，往往意味着模块间耦合松散。内聚比耦合更重要！（内聚种类按0-10评分）  
a）低内聚  
①偶然内聚：一个模块完成一组任务，任务之间的关系势必很松散；常见做法是，完成一个程序后，将出现在不同地方的相同语句组成一个模块；各元素没有实质联系，不宜使用,0  
②逻辑内聚：一个模块完成的任务在逻辑上属于相同或相似的一类；不同功能混用相同代码，不宜使用,1  
③时间内聚：一个模块包含的任务必须在同一时段内执行；可用,3  
b）中内聚  
①过程内聚：模块内的处理元素相关，且以特定次序执行；使用程序流程图设计软件常产生此类模块,5  
②通信内聚：模块中所有元素都使用同一个输入数据和（或）产生同一个输出数据,7  
c）高内聚  
①顺序内聚：模块内的处理元素同一个功能密切相关，且这些处理必须顺序执行；通常一个处理元素的输出作为下一个的输入；使用数据流图设计模块时产生,9  
②功能内聚：模块内所有处理元素属于一个整体，完成一个单一的功能；最高内聚10

## 5.3 启发规则

即一些经验规则，不是基本原理，不一定普适

1.改进软件结构提高模块独立性  
反复分析已有结构，通过模块分解或合并来降低耦合提高内聚；  
如：合并多个模块公有的子功能为一个模块

2.模块规模应该适中  
一个模块最好能写在一页纸内，通常不超过60行语句；  
分解后的模块又不能降低其独立性；如果只有一个模块调用该模块，就应该将它合并到上级模块中

3.深度、宽度、扇入扇出都应适当  
深度：软件结构中控制的层数  
宽度：软件结构内同一个层次上的模块总数的最大值；对宽度影响最大的是模块的扇出  
扇出：一个模块直接控制（调用）的模块数目；好的设计中平均扇出是3-4（上限是5-9）；  
 可适当增加中间层次的控制模块以减少扇出，可分解下级模块或合并其至上级模块  
 以增加扇出  
扇入：表示有多少上级模块直接调用该模块；合理情况下，扇入越大表示该模块共享程  
 度越高，有好处；  
通常顶层扇出比较高，中层扇出较少，底层扇入到公共的实用模块中，底层模块高扇入

4.模块的作用域应该在控制域内  
模块之间的关系和软件结构可以用类似树状层次图来表示  
控制域：层次图中某个模块的子树即为其控制域，即直接或间接从属于它的模块  
作用域：实际上受到该模块内判定影响的所有模块的集合  
这条准则即说明，模块应该只决定和影响自己的子模块，不应该影响其他模块的进程；修改时可以采用更改模块结构的方式，或者将有问题的指令移到合适和顶层模块中去

5.力争降低模块接口的复杂程度  
接口设计的原则：使信息传递简单明了并且和模块的功能一致  
举例：  
 用单独的基本类型变量来代替元素不多的数组类型以传递参数，数组类型相对更难  
 理解和调用，并且每个变量的参数名称都可以很好的反映参数的作用

6.设计单入口单出口的模块  
单入口单出口即不出现内容耦合

7.模块功能应该可以预测  
定义：输入的数据相同就产生同样的输出，如此的模块功能即可以预测；如果模块带有  
 内部存储器（如标记），其功能很可能受到其影响而不仅仅取决于输入，尤其是内  
 部存储器对于上级模块不可见  
同时，模块的功能范围不能过于狭窄，应具有适当灵活性

## 5.4 描绘软件结构的图形工具

5.4.1 层次图和HIPO图

软件结构层次图和数据结构层次方框图不同：  
结构层次图中一个框代表一个模块，连线代表调用关系（不能直观理解为包含或组成关系，父子模块是两个不同模块，父模块调用子模块完成自己的功能，同时子模块也可以有自己的功能）；  
层次方框图中连线代表组成关系

结构层次图很适合自顶向下设计软件过程

HIPO（层次图加输入/处理/输出图）图是一种IBM开发的升级版的层次图（H图），另外还外加有IPO图。  
H图范例：p103/125 有编号的层次图，编号规则与数据流图一致  
IPO图：每个模块有一个IPO图表来对应描述

5.4.2 结构图

一种类似层次图的样式的升级版图形工具

典型结构图范例：p103/125

符号：  
空心圆做尾的箭头——数据信息  
实心圆做尾的箭头——控制信息  
选择调用p104/126  
循环调用p104/126

层次图和结构图都并不严格表示模块的调用次序，虽然很多人习惯按照调用次序从左到右画模块（有时因此会导致交叉线）。另外这两种图形工具都没有表明模块中除了调用其他模块的信息还是否有其他有效成分。

通常还是使用层次图外加IPO或数据字典作为描绘软件结构的文档，使用结构图不甚清晰。

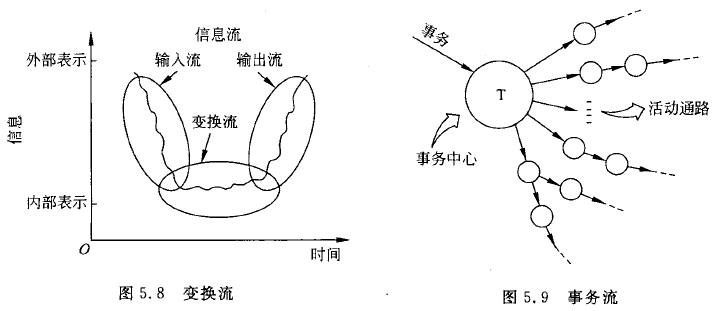
## 5.5 面向数据流的设计方法

结构化设计方法（SD方法）就是基于数据流的设计方法，因为任何软件系统都可以用数据流图表示，所以面向数据流设计法理论上可以设计任何软件的结构

5.5.1 概念

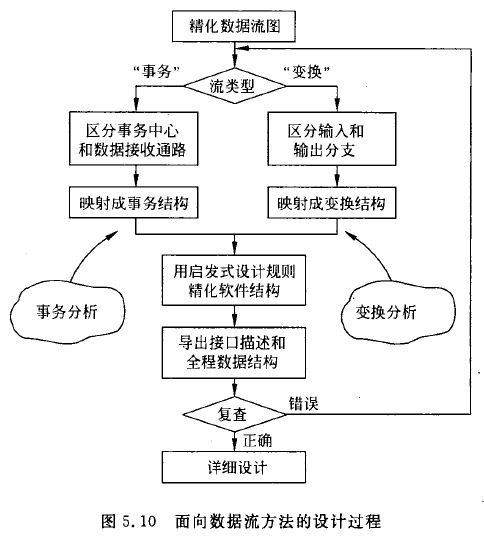
SD方法把信息流映射成软件结构，信息流类型决定映射方法

信息流类型有：  
（1）变换流  
信息沿输入通路进入系统，变成内部形式，经加工处理之后再沿输出通路变换成外部形式离开系统。这种信息流叫做变换流，狭义上讲，所有信息流都是变换流，因为基本系统模型表达的就是变换流。

  
（2）事务流  
一种特殊的变换流，处理T根据输入信息的类型在若干个动作序列中选取一个来执行，也就是有根据的对信息流处以不同操作，这种称为事务流。  
处理T称为事务中心，完成以下任务：  
①接收输入数据（也称事务）  
②分析事务类型  
③根据类型选取一条活动通路

设计过程：

任何设计过程都不是机械的不变的，应该首先遵从人的判断力和创造精神。



5.5.2 变换分析

例子p106/128

变换分析即将变换流映射成软件结构的设计步骤：

（1）复查基本系统模型  
确保系统的输入数据和输出数据符合实际

（2）复查并精化数据流图  
确保数据流图给出目标系统的正确逻辑模型，还应使数据流图中每个处理都代表一个规模适中相对独立的子功能。

（3）确定数据流图具有变换特性还是事务特性  
根据数据流图中占优势的属性，确定数据流全局特性是变换还是事务。把具有和全局特性不同特点的局部区域孤立出来，以后可以按照这些子数据流的特点精化按照全局特性得出的软件结构。  
其实，只要没有明显的总体事务中心，就可以认为信息流具有变换流的总特征，尽管内部有很多事务流样式的小分叉结构也无妨。

（4）确定输入流和输出流的边界，从而孤立出变换中心  
按照变换流的组成结构（3部分）来在数据流图中划分，具体而言如何划分边界并不是一定的，有的微小区别（某个数据流路径上一个处理框的区别）并不十分影响最终结构

（5）完成第一级分解  
软件结构代表对控制的自顶向下分配，分解就是分配控制（将处理分配给模块）的过程。  
第一级分解就是设计软件结构层次中第一层（不算根节点层）的模块，将输入流、输出流和变换中心分别设计成三个大模块，对于较复杂系统可以使用更多的模块完成这些控制，但是应尽量使第一级控制模块数目取最小值。  
位于软件结构最顶层的控制模块Cm协调下述从属的控制功能：  
①输入信息处理控制模块Ca，协调对所有数据的接收  
②变换中心控制模块Ct，管理对内部形式的数据的所有操作  
③输出信息处理控制模块Ce，协调输出信息的产生过程

（6）完成第二级分解  
第二级分解把数据流图中每个处理映射成软件结构中一个适当的下级模块（注意这些模块可能是第二层或更低层）。  
方法：从变换中心的边界开始逆输入通路向外移动，沿途所遇每个处理映射成软件结构中Ca控制下的一个底层模块（同一路径上先遇到的处理模块将是后遇到的处理模块的父模块，不同路径对应着Ca的不同子分支，注意不同子树之间可能共享节点模块）；同理沿输出通路向外移动，映射Ce控制下的模块；最后将变换中心的处理映射成Ct的子模块。  
上述方法只是通用办法，实际上不一定严格遵循这种一对一映射关系。  
软件结构初步设计完成后，除了层次图还要为每个模块写一个说明，共同构成设计规格说明雏形，包括：  
①进出该模块的信息（接口描述）  
②模块内部的信息  
③过程陈述，包括主要判定点及任务等  
④对约束和特殊特点的简短讨论

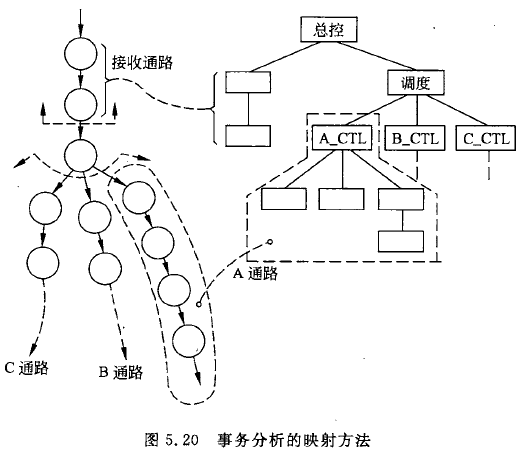
（7）使用设计度量和启发式规则对第一次分割得到的软件结构进一步精化  
设计度量就是模块独立的耦合与内聚度量。根据这些因素将初步结构精化改进。需要改进并非是因为前面步骤不足，而是因为有些问题注定在改进过程中通过全局分析才能发现。

5.5.3 事务分析

当数据流图有一个明显的“发射中心”（事务中心）时，应采用事务分析方法，虽然这种数据流形式仍然可以使用变换分析

事务分析步骤和变换分析大致类似，主要差别在于数据流到软件结构的映射：  
事务流映射成的软件结构包括一个接收分支和一个发送分支（第一层只有两大类分支），分别对应数据流图中接收通路和输出通路区域（即除去事务中心）。接收分支映射方法和变换分析中的输入结构类似。发送分支结构第一层是一个调度模块，第二层是每个分支输出通路的总控模块，第三层以下即每条分支通路的处理所映射的模块（类似变换分析中的输出结构）。

注意：同一条数据流通路不一定映射成一串从高到低的模块（只有确定是输入系统或输出系统的数据流才这么处理），应该根据这条通路上的处理类型来设计模块的结构关系，比如再次调用变换分析。应用不同的分析方法处理数据流图的不同部分，可以得到“构件”，进而构造完整结构。  
映射过程中，不是每个处理元素都映射成模块，有些附加模块充当控制模块（比如总控模块，调度模块等）。应该根据需要决定是否保留或分解或添加（如中间层次控制模块）这些控制模块以最优化软件结构和设计度量。



5.5.4 设计优化

应该在设计的早期阶段，如结构设计阶段，对软件尽可能进行精化。结构简单的软件既表示设计风格优雅，又表示效率高。因此优化的目的就是在有效模块化前提下（设计度量满足），使用最少量的模块，以及满足信息要求前提下，使用最简单的数据结构。

对于时间是决定性因素的场合，可能有必要在详细设计阶段甚至实现阶段进行优化，可以采取下述合理方法：  
（1）在不考虑时间因素的前提下开发并精化软件结构  
（2）在详细设计阶段选出最耗费时间的那些模块，仔细地设计它们的处理过程（算法），以提高效率  
（3）使用高级程序设计语言编程  
（4）在软件中孤立出那些大量占用处理机资源的模块  
（5）必要时重新设计或用依赖于机器的语言重写上述大量占用资源的模块的代码，以提高效率

——“先使它能工作，再使它快起来。”

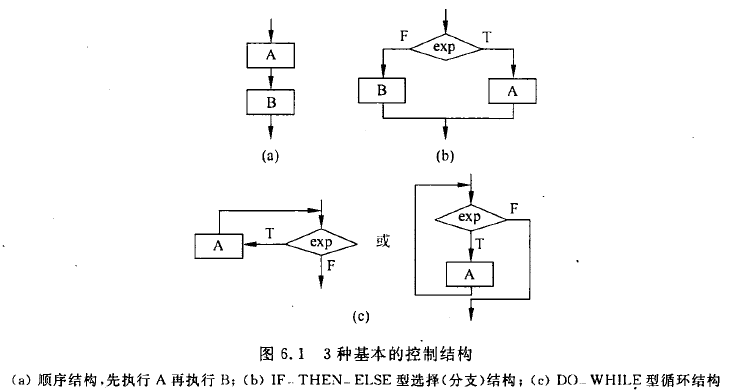
# 第6章 详细设计

长期使用的软件系统，人读程序的时间往往比写程序的时间长，所以必须考虑程序有两个读者——人和计算机

详细设计目标：  
（1）逻辑上正确地实现每个模块的功能  
（2）设计出的处理过程应尽可能简明易懂

## 6.1 结构程序设计

理论上最基本的控制结构有两种——顺序结构和循环结构；也有说法是3种基本结构就能实现任何单入口单出口程序，但是第三种“选择”结构可以被前两种实现

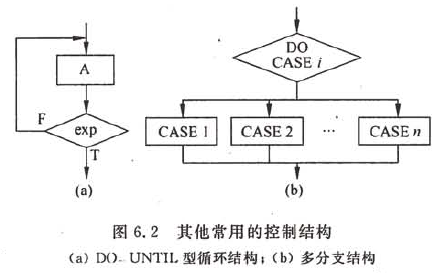


结构程序设计“经典”定义：如果一个程序的代码块仅仅通过顺序，选择和循环3种基本控制结构进行连接，并且每个代码块只有一个入口和一个出口，则称这个程序是结构化的。满足这个定义的称为“经典的结构程序设计”。

新定义：尽可能少用GO TO语句的程序设计方法。最好仅在检测出错误时才使用GO TO语句，而且应该总是使用前向GO TO语句。

还有其他常用控制结构，虽然不是基本结构，但为了实际使用方便，也可以用在结构化设计中，此时称为“扩展的结构程序设计”。

另外还可以有LEAVE或BREAK结构用于跳出循环，提高效率。实质上它们是受限制的前向GO TO语句，此时称为“修正的结构程序设计”。



## 6.2 人机界面设计

人机界面设计时接口设计的一个重要的组成部分，尤其对于交互式系统来说。

6.2.1 设计问题

人机界面设计中常见的4个问题

（1）系统响应时间  
定义：从用户完成某个控制动作到软件给出预期的响应之间的时间，它有两个属性，长度和易变性  
长度：过长过短的响应时间都不利，过短的响应时间会逐渐迫使用户加快操作而致错误  
易变性：系统响应时间相对于平均响应时间的偏差。比长度更重要，长度往往取决于任务复杂程度，但时间变化过大易造成用户认为系统工作异常。

（2）用户帮助设施  
常见的帮助设施分为集成的和附加的两类：  
①集成的——设计在软件系统之中的帮助设施，对用户最近完成的操作敏感（有记忆），可以给用户最快最方便最相关的支持  
②附加的——系统建成后再添加到软件中的，实际上是一种联机查询用户手册。  
帮助设施设计需要解决的问题：  
a）在什么阶段，给用户提供什么程度的帮助信息？部分功能的帮助信息/全部功能的帮助信息  
b）用户怎样请求帮助？帮助菜单/特殊功能键/HELP命令  
c）怎样显示帮助信息？独立窗口/指出参考文档/屏幕特定位置提示  
d）用户怎样返回到正常的交互方式？返回按钮/功能键  
e）怎样组织帮助信息？平面结构（所有信息都通过关键字访问）/信息的层次结构/超文本结构

（3）出错信息处理  
即向用户报错的机制设计，错误信息通报不当会导致用户不知道如何处理  
出错与警告信息机制应具有的属性：  
a）信息应该用用户可以理解的术语描述问题  
b）信息应该提供有助于改正错误的意见  
c）信息应该指出错误可能造成的后果，以便用户检查，提高用户知情程度减少损失  
d）信息应该伴随听觉和视觉上的不寻常提示，如铃声和红色闪烁  
e）信息不能带有指责色彩，不能责怪用户

（4）命令交互  
即原始的命令行交互方式，仍然是很多高级用户偏爱的交互方式  
设计命令行交互的问题：  
a）是否每个菜单选项都有对应的命令  
b）采用何种命令形式？控制序列/功能键/输入命令  
c）使用和掌握命令的难度，忘记了怎么办？  
d）用户是否可以定制或缩写命令？  
“命令宏机制”：可以使用自定义的名称代表一个命令序列，调用时无需再依次分别输入这些具体的命令  
命令（如快捷键热键）是有习惯和约定俗称性的，不应按照自己的解释来设计和绑定命令

6.2.2 设计过程

用户界面设计也是一个迭代过程，具体包含创建原型、用户试用、原型评估和修改原型等。用于界面设计和快速原型开发的软件工具称为用户界面工具箱或用户界面开发系统。

创建用户界面的设计模型之后，可以运用下述评估标准进行早期复审：  
（1）系统及其界面的规格说明书长度和复杂程度，预示了用户学习系统所需要工作量  
（2）命令或动作的数量、命令的平均参数个数或动作中单个操作的个数，预示系统的交互时间和总体效率  
（3）设计模型中包含的动作，命令和系统状态的数量，预示了用户学习系统时需要记忆的内容数量  
（4）界面风格，帮助设施和出错处理协议，预示了界面的复杂程度及用户接受界面的程度

6.2.3 人机界面设计指南（经验）

（1）一般交互指南

一般交互指南涉及信息显示，数据输入，系统整体控制。  
①保持一致性。为人机界面中的菜单选择，命令输入，数据显示以及其他功能使用一致格式  
②提供有意义的反馈。向用户提供视觉和听觉的反馈，保证用户和系统之间建立双向通信  
③在执行有较大破坏性的动作前要求用户二次确认，比如删除或覆盖文件或终止程序等  
④允许取消绝大多数操作，UNDO和REVERSE功能有时可以避免大量时间浪费，提高工作效率  
⑤减少两次操作之间必须记忆的信息量。不能期望用户能记住下一步操作中需使用的大量字符  
⑥提高对话，移动和思考的效率。应该尽量减少用户按键次数，减少鼠标移动距离（比如类似功能或经常连续使用的按键位置的设计），尽量让用户通过直觉能够发现系统的功能和用法（不是不依赖手册与教程，这样能够便于用户记忆操作方法）  
⑦允许犯错误。系统应该能保护自己不受严重错误的破坏  
⑧按照功能对动作进行分类，据此设计屏幕布局  
⑨提供对用户工作内容敏感的帮助设施  
⑩用简单动词或动词短语作为命令名，不宜过长

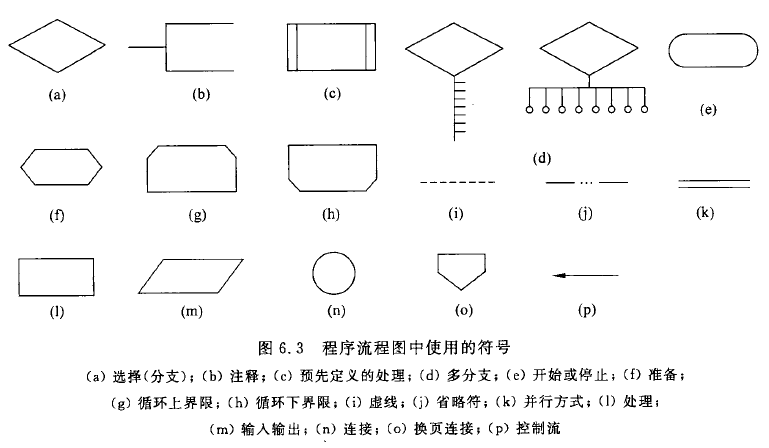
（2）信息显示指南  
不同种方式“显示信息”：文字、图形和声音，位置、移动和大小，颜色、分辨率和省略  
①只显示与当前工作内容有关的信息  
②用便于用户迅速吸收信息的方式表达数据，如图形或图表代替表格  
③使用一致的标记，标准缩写和可预知的颜色，显示含义明确  
④允许用户保持可视化的语境。如果对所显示的图形缩放，原始图像的全图应该缩小显示在屏幕一侧以方便用户知道当前图像在原图的位置  
⑤产生有意义的出错信息  
⑥使用大小写，缩进和文本分组以帮助视觉理解。人机界面显示的信息大部分仍是文字，可以通过布局和形式的处理让用户阅读更加方便，比如编译器的自动标识关键字功能  
⑦使用窗口分隔不同类型信息  
⑧使用“模拟”显示方式表示一些数字信息，不仅仅提供绝对和相对的信息，也提供了提醒和警示的作用。比如用进度条或管状条表示一些数值信息可以更加直观、  
⑨高效率使用显示屏。即多个窗口同时显示时的分配设计，也和显示器尺寸有关。

（3）数据输入指南  
①尽量减少用户的输入动作，如击键次数。可以用鼠标从预定义的一组输入中选一个；用“滑动标尺”在给定的值域中指定输入值；利用宏把一次击键转变成更复杂的输入数据集合。  
②保持信息显示和数据输入之间的一致性。显示的视觉特征应该与输入形式一致。便于确认输入。  
③允许用户自定义输入。高级用户可能希望自己定义专用的命令或者略去某些警告信息或确认信息，人机界面应该提供这样的选择功能。  
④交互应该是灵活的，并且可调整成用户最喜欢的输入方式  
⑤使在当前语境中不适用的命令不起作用，自检机制  
⑥让用户控制交互流，比如给予用户适当的控制动作及其顺序的能力，以及不退出程序就能从错误中恢复  
⑦对所有输入动作都提供帮助  
⑧消除冗余的输入。比如不要求用户指定输入数据的单位（可以自行推断或者在某处全局设置）；尽量多的提供默认值备选，以省去输入麻烦；绝对不要求用户提供程序可以自动获得或计算出的信息（也许也应该是可选的）

## 6.3 过程设计的工具

过程设计工具：描述程序处理过程的工具，可分为图形、表格和语言3种。指明控制流程、处理功能、数据组织以及其他方面的实现细节。

6.3.1 程序流程图

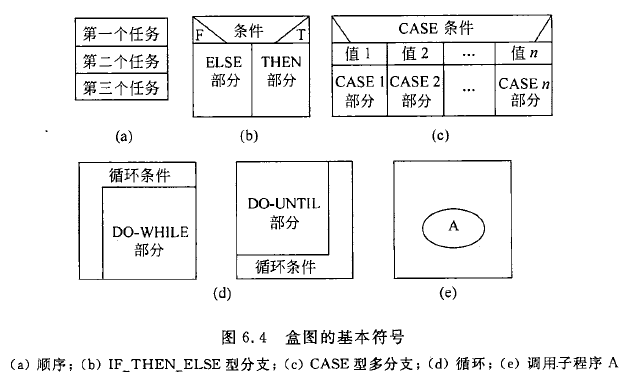


使用广泛，历史悠久，但是问题较多，逐渐弃用

缺点：  
（1）本质上不是逐步求精的好工具，过早注重程序控制流程而不是全局结构  
（2）用箭头代表控制流，因此不受任何约束可随意转移控制  
（3）程序流程图不易表示数据结构

6.3.2 盒图

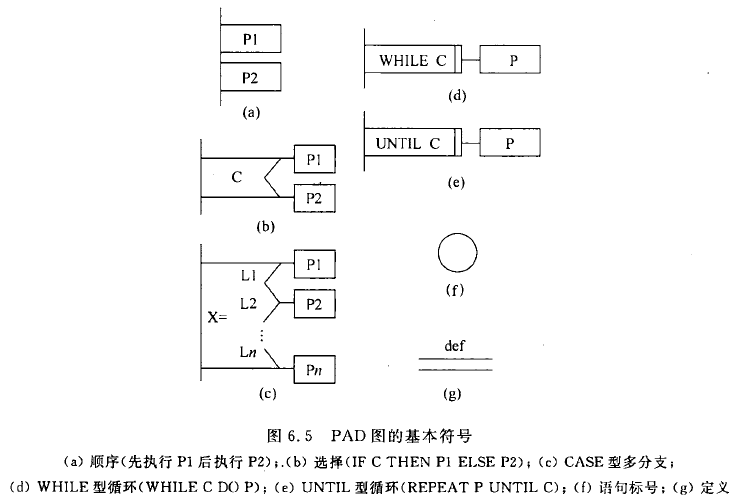
盒图又称N-S图（两个发明人的名字首字母），具有如下特点：  
（1）功能域明确，标明在图上  
（2）不可任意转移控制（盒图没有箭头）  
（3）很容易确定局部和全程数据的作用域  
（4）很容易表现嵌套关系，也可以表示模块的层次结构



6.3.3 PAD图

PAD（problem analysis diagram）称为问题分析图，采用二维树形结构图表示程序控制流，将这种图翻译成代码比较容易。PAD面向高级程序设计语言，针对FORTRAN，COBOL和PASCAL等语言都提供了一整套相应符号，更加便于转换成对应代码。

PAD图的优点：  
（1）使用表示结构化控制结构的PAD符号设计的程序必是结构化程序。  
（2）PAD图从上至下从左至右扩展，每一层的结构保持在同一竖线上  
（3）可以使用软件工具自动完成PAD图到高级语言源程序转换  
（4）既可用于表示程序逻辑，又可以表示数据结构  
（5）PAD图支持逐步求精的方法使用，有对应的符号和机制



6.3.4 判定表

判定表具有其他描述方法不具有的清晰表示复杂条件组合与应做动作之间的关系的功能，即如算法中包含多重嵌套的条件选择时。但是它不适于作为一种通用的设计工具，只是用来清楚地描述一套比较复杂的处理规则。

一般也没有什么常用方法能够同时清楚地表示顺序（用来设计流程）和重复（比较复杂的算法）的处理特性。

判定表是一个二维表格，被水平竖直两条线分成四个区域：左上表示所有可能条件，左下是所有可能的动作，右上是可能条件的不同组合，右下则是动作和条件的对应。右半部整体每一列都代表一种条件组合和对应的处理选择。（例p128/150）

6.3.5 判定树

判定树是判定表的变种（另一种表示方法），形式易于理解直观。但是和判定表一样条件变多时，简洁性下降（例p129/151）

6.3.6 过程设计语言（PDL）

也称为伪代码，PDL与伪代码都是一个笼统名称（不是特指一种设计描述技术），表示用正文形式表示数据和处理过程的设计工具。

PDL有严格的关键字外部语法，用于定义控制结构和数据结构。但表示实际操作和条件的内部语法通常是灵活自由的。因此它是一个混杂语言。

PDL语言的通常特点：  
（1）有关键字的固定语法用于描述结构化控制结构，数据说明和模块，这保证了结构清晰，不至于完全不知所云  
（2）使用自然语言描述处理特点，作为内部描述，自由  
（3）数据说明的手段应该包括从简单到复杂的数据结构  
（4）模块定义和调用的技术，应该提供各种接口描述模式

PDL的通常优点：  
（1）可以作为注释直接插在源程序中，促使修改程序代码的同时修改文档内容  
（2）可以使用任何文本处理软件编辑PDL  
（3）有些可以自动处理PDL的程序用于代码转换生成

缺点：显然不如图形图表工具直观，对于复杂的逻辑描述和纯代码的样式区别不大

## 6.4 面向数据结构的设计方法

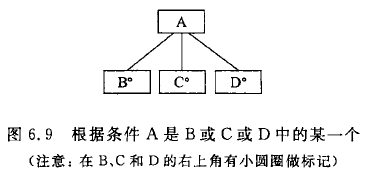
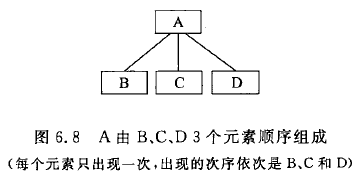
面向数据结构的设计方法注重的是对程序处理过程的设计描述，并不重视软件结构或模块的概念。因此应该在完成结构设计（面向数据流）之后，再用这个方法详细设计每个模块的处理过程。

面向数据结构的设计方法主要包括两个：Jackson方法和Warnier方法

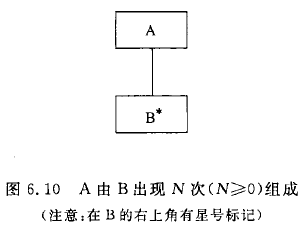
6.4.1 Jackson图

数据结构种类繁多，但是数据元素彼此间的逻辑关系只有顺序、选择和重复3类，即3种逻辑数据结构

（1）顺序结构  
数据由一个或多个数据元素组成，每个元素按照确定次序出现一次



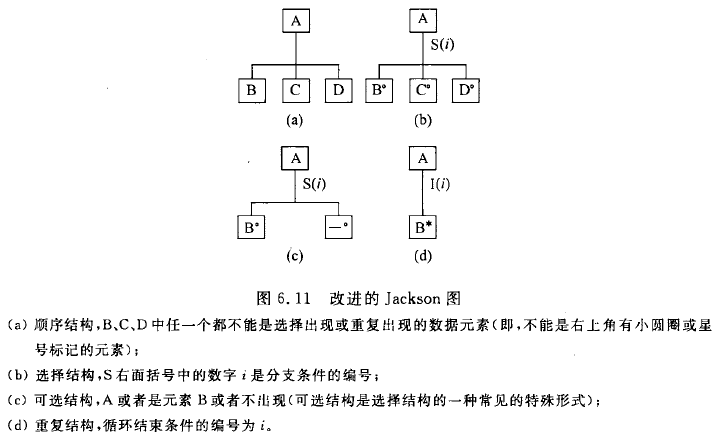
（2）选择结构  
数据包含两个或多个数据元素，每次使用时按一定条件从这些数据元素中选一个



（3）重复结构  
根据使用时的条件由一个数据元素出现零次或多次构成

优点：  
①便于表示层次结构，对结构进行自顶向下分解  
②形象直观，可读性好  
③既能表示数据结构，也能表示程序结构（类似的3种）

6.4.2 改进的Jackson图



改进的Jackson图比原图多了一些必要细节，连线变为比较容易绘制的横竖线。

Jackson图实质上和层次方框图类似，不同于结构层次图。每个方框只由下级方框实现和组成，本身没有特殊或独立含义。

6.4.3 Jackson方法

Jackson结构程序设计方法5步骤：  
（1）分析并确定输入输出数据的逻辑结构，并用Jackson图描绘这些数据结构  
（2）找出输入数据机构和输出数据结构中有对应关系的数据单元，比如有直接因果关系，在程序中可以同时处理的数据单元（对于重复出现的数据单元，重复次序和次数都必须相同方可称为有对应关系）  
（3）用以下3规则从数据结构Jackson图导出程序结构Jackson图  
①为每对有对应关系的数据单元，按照它们在数据结构图中的层次在程序结构图的相应层次画一个处理框（如果这对数据单元在输入数据结构和输出数据结构中所处的层次不同，则和它们对应的处理框在程序结构图中所处的层次与它们之中在数据结构图中的层次靠下的那个对应）  
②根据输入数据结构中剩余的每个数据单元所处的层次，在程序结构图的相应层次分别为它们画上对应的处理框  
③根据输出数据结构中剩余的每个数据单元所处的层次，在程序结构图的相应层次分别为它们画上对应的处理框  
描绘程序结构的Jackson图应该综合输入输出数据结构的层次关系而导出。改进的Jackson图规定在顺序结构中不能有重复出现或选择出现的元素，因此可能需要增加中间层次的处理框。  
（4）列出所有操作和条件（分支条件与循环结束条件），并且把它们分配到程序结构图的适当位置  
（5）用伪码表示程序  
Jackson方法中使用和Jackson图完全对应的伪码，以下是3种基本结构对应伪码（和上文中举的例图对应）：  
①顺序结构：seq和end是关键字  
A seq  
 B  
 C  
 D  
A end  
②选择结构：select、or、end是关键字，cond是执行条件  
A select cond1  
 B  
A or cond2  
 C  
A or cond3  
 D  
A end  
③重复结构（2种形式）：iter、until、while、end是关键字，cond是条件  
A iter until（或while）cond  
 B  
A end

Jackson方法使用范例：p133/155

上面介绍的Jackson方法本体处理较简单的数据处理系统时十分方便。但是对于比较复杂的程序会遇到几个问题：输入数据可能有错，条件不能预先测试，数据结构冲突等问题。这些问题需要借助一些比较复杂的辅助技术。

## 6.5 程序复杂程度的定量度量

6.5.1 McCabe方法

（1）流图

McCabe方法根据控制流的复杂程度定量度量程序的复杂程度，这样度量出的结果称为程序的环形复杂度

流图：也称程序图，实质上是退化版的程序流程图，仅仅描绘程序的控制流程，不表现对数据的具体操作以及分支循环的条件。

所有方法表示的过程设计结果都可以翻译成流图，程序流程图/PDL/复合条件转换成流图的简单例子，见p138/160

复合条件：条件中包含多个布尔运算符，如逻辑OR，AND，NAND，NOR等

（2）计算环形复杂度的方法

环形复杂度定量度量程序的逻辑复杂度

由流图计算环形复杂度的方法：  
①流图中的区域数等于环形复杂度  
②流图G的环形复杂度V(G)=E-N+2，其中E是流图中边的条数，N是结点数  
③流图G的环形复杂度V(G)=P+1，其中P是流图中判定结点的数目

（3）环形复杂度的用途

实践表明，模块规模以V(G)≤10为宜，过高的环形复杂度导致程序难以测试，易出问题

6.5.2 Halstead方法

Halstead方法根据程序中运算符和操作数的总数来度量程序复杂度

程序长度N=N1+N2，N1为程序中运算符出现的总次数，N2为操作数出现的总次数

详细设计完成后，可以知道程序中使用的不同运算符（包括关键字）的个数n1和不同操作数（变量与常数）的个数n2，由此可以预测出程序长度H=n1log2n1+n2log2n2

预测长度H与实际长度N非常接近

预测程序中包含错误的个数E=Nlog2(n1+n2)/3000，根据从300-12000条语句范围内的程序核实，预测错误和实际错误数相比误差在8%之内

# 第7章 实现

实现包括编码和测试，软件测试是对软件规格说明、设计和编码的最后复审。

软件测试在软件生命周期中横跨两个阶段：每个模块编写完成后就对其进行测试，称为单元测试，这通常由同一个人编写和完成测试，都属于实现阶段。在软件系统整体成型后，对其进行各种综合测试，这是一个独立阶段，由专门的人员完成这项工作。

软件测试的工作量往往占软件开发总工作量的40%以上，对于事关重大的软件测试，其阶段成本可能相当于其他开发步骤总成本的3-5倍。编码完成后，距离开发结束至少还有一半的工作量。

测试只是发现软件中的错误，诊断并改正错误是调试过程，这是测试阶段最困难的工作。测试的结果提供了更准确的估计软件可靠性的数据。

## 7.1 编码

7.1.1 选择程序设计语言

编码可选的语言至少有两种：汇编语言和高级语言。两种方法的语句开发效率大致一致，但是高级语言和汇编指令有一句对多句的对应关系。因此显然后者更优。

一些特殊的需要使用汇编语言编码的情景：  
（1）对程序执行时间和使用的空间都有很严格限制的情况  
（2）需要产生任意的甚至非法的指令序列  
（3）体系结构特殊的微处理机，这类机器上通常不能实现高级语言编译程序  
（4）大型系统中执行时间非常关键或直接依赖于硬件的一小部分代码

选择程序设计语言的实用标准：  
（1）系统用户的要求；如果由用户负责维护，可能需要使用他们熟悉的语言  
（2）可以使用的编译程序；即运行目标系统的环境支持  
（3）可以得到的软件工具；某种语言是否有支持程序开发的软件工具  
（4）工程规模；如果工程规模庞大，可能需要自行设计和实现一种专门的程设语言  
（5）程序员的知识；选用工作人员最熟悉的语言  
（6）软件可移植性要求  
（7）软件的应用领域；通用程序设计语言实际上有针对的应用领域  
FORTRAN——工程和科学计算  
COBOL——商业领域  
C和Ada——系统和实时应用  
LISP——组合问题领域  
PROLOG——表达知识和推理

7.1.2 编码风格

编码应该遵循的规则：

（1）程序内部的文档  
包括恰当的标识符，适当注解和程序的视觉组织  
选取合适的名称和缩写，以及注解解释；将代码的格式划分清晰

（2）数据说明  
对于数据结构的说明（规格说明或注解）风格和原则：  
①数据说明的次序应该标准化（比如按照数据结构或数据类型确定说明的次序）。有次序就容易按序查询。  
②当多个变量名在一个语句中说明时，应该按照字母顺序排列这些变量  
③如果设计时使用了一个复杂的数据结构，则应该用注解说明用程序设计语言实现这个数据结构的方法和特点

（3）语句构造  
编写实际代码和语句布局的规则：  
①不要为了节省空间而把多个语句写在同一行  
②尽量避免复杂的条件测试  
③尽量减少对“非”条件的测试  
④避免大量使用循环嵌套和条件嵌套  
⑤利用括号使逻辑表达式或算术表达式的运算次序清晰直观

（4）输入输出有关输入输出风格的规则：  
①对所有输入数据都进行检验  
②检查输入项重要组合的合法性  
③保持输入格式简单  
④使用数据结束标记，不要要求用户指定数据的数目  
⑤明确提示交互式输入的请求，详细说明可用的选择或边界数值  
⑥当程序设计语言对格式有严格要求时，应保持输入格式一致  
⑦设计良好的输出报表  
⑧给所有输出数据加标志

（5）效率  
效率主要指处理机时间（时间）和存储器容量（空间）两个方面。  
讨论效率的3条原则：  
①效率是性能要求，应该在需求分析阶段确定效率方面的要求  
②效率是靠好设计来提高的（不能全部依靠完善代码）  
③程序的效率和程序的简单程度不冲突，不能牺牲清晰简单性换取效率  
效率问题的三方面：  
a）程序运行时间  
源程序的效率既由算法决定，也由代码风格影响（执行速度和存储器要求）。  
详细设计实现为代码时的规则：  
①写程序之前先简化算术的和逻辑的表达式  
②仔细研究嵌套的循环，确定是否有语句可以从内层往外移（减少循环体中语句数量）  
③尽量避免使用多维数组（空间消耗大）  
④尽量避免使用指针和复杂的表  
⑤使用执行时间短的算术运算  
⑥不要混合使用不同的数据类型  
⑦尽量使用整数运算和布尔表达式  
⑧尽量使用有良好优化特性的编译程序，以自动生成高效的目标代码  
b）存储器效率  
大型计算机中必须考虑操作系统页式调度的特点，使用能保持功能域的结构化控制结构可提高效率  
如果微处理机中要求使用最少的存储单元，则应选用有紧缩存储器特性的编译程序，必要时使用汇编语言  
c）输入输出的效率  
硬件之间通信效率的简单原则  
所有输入输出都应该有缓冲（支持批量IO），以减少用于通信的额外开销  
对二级存储器（如磁盘）应选用最简单的访问方法  
二级存储器的输入输出应该以信息组为单位进行  
如果超高效的输入输出很难被理解，则不应采用

## 7.2 软件测试基础

7.2.1 软件测试的目标

测试的目标/规则/定义：  
（1）测试是为了发现程序中的错误而执行程序的过程  
（2）好的测试方案是极可能发现迄今为止尚未发现的错误的测试方案  
（3）成功的测试时发现了迄今为止尚未发现的错误的测试

7.2.2 软件测试准则

主要的测试准则：  
（1）所有测试都应该能追溯到用户需求。针对用户需求进行的测试，不能满足用户需求的错误严重性更大  
（2）远在测试开始之前就制定出测试计划。需求模型完成后可以着手制定计划，设计模型完成后可以开始设计详细测试方案，编码阶段之前就完成测试的计划与设计  
（3）把Pareto原理应用到软件测试中。该原理说明，测试发现的错误中的80%很可能是由程序中20%的模块造成的。找出问题聚集的模块进行彻底测试  
（4）应该从小规模测试开始，逐步进行大规模测试。先单模块测试，再测试模块簇，最后测试整个系统  
（5）穷举测试是不可能的。不可能把程序所有可能的执行路径都检查一遍。  
（6）应该由独立第三方从事测试工作。系统测试不能由开发软件的工程师负责，他们只负责模块测试。

7.2.3 测试方法

（1）黑盒测试（或功能测试）  
知道了产品应该具有的功能，通过测试检验其是否能够正常运行这些功能。是在程序接口进行的测试

（2）白盒测试（或结构测试）  
知道产品的内部工作过程，通过测试检验产品内部动作是否按照规格说明书的规定进行

7.2.4 测试步骤

以大型软件系统为例

（1）模块测试  
又称单元测试，发现的往往都是模块中编码和详细设计的错误

（2）子系统测试  
着重测试模块的接口，几个已完成单元测试模块相互之间的协调和通信

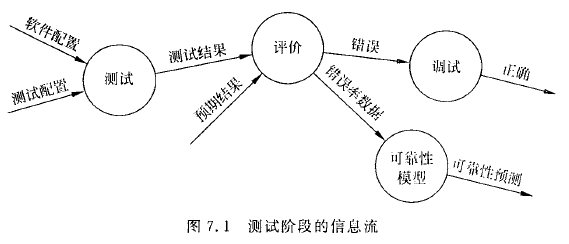
（3）系统测试  
子系统和系统测试又称集成测试（包含有组装模块的步骤）。不仅仅测试设计和编码错误，还应验证是否满足需求说明，以及动态特性是否满足要求

单元测试和集成测试不一定是先后独立的步骤，根据具体采用的方法（见后文），二者可能是共同完成的。不过对每个模块的单独测试或实质上的单独测试和多个模块乃至系统的整体测试都是必须完成的内容。

（4）验收测试  
也称确认测试，是和用户共同参与的测试，内容和系统测试类似，但主要使用实际数据（目标系统将来要现实处理的信息）

（5）平行运行  
对于关系重大的软件产品验收之后仍不立即投产，进行新旧系统平行对比运行（既是测试也是部分投产试验），目的是：  
①可以在准生产环境中运行新系统而又不冒风险  
②用户能有一段熟悉新系统的时间  
③可以验证用户指南和使用手册之类的文档  
④能够以准生产模式对新系统进行全负荷测试，可以用测试结果验证性能指标

7.2.5 测试阶段的信息流



输入信息：  
①软件配置——需求说明书，设计说明书和源程序清单  
②测试配置——测试计划，测试方案（测试使用的输入数据或测试用例，每个用例预定要检验的功能，预期应该得到的输出）

调试由程序编写人员负责

## 7.3 单元测试

单元测试主要使用白盒测试，可对多个模块并行测试。可以使用人工测试和计算机测试两种方法。人或机测试各有优劣，应该相互补充使用。

7.3.1 测试重点

单元测试注重的5个方面：

（1）模块接口  
①测试通过接口的数据流是否可以正确进出  
②测试参数的数目，次序，属性或单位系统与变元是否一致  
③是否修改了只做输入用的变元  
④全局变量的定义和用法在各个模块中是否一致

（2）局部数据结构  
局部数据说明，初始化，默认值等

（3）重要的执行通路  
选择最有代表性，最可能发现错误的执行通路进行测试  
发现由于错误的计算，不正确的比较或不适当的控制流

（4）出错处理通路  
出错处理是程序中或模块中包含的部分，测试这部分内容以发现：  
①对错误的描述是难以理解的  
②记下的错误与实际遇到的错误不同  
③在对错误进行处理之前，错误条件已经引起系统干预  
④对错误的处理不正确  
⑤描述错的信息不足以帮助确定造成错误的原因和位置

（5）边界条件  
边界测试是单元测试中最后也是最重要的任务。边界情况其实就是输入或控制数据的非一般/退化情况（刚好取范围边界值或极值或零值时的情况，循环第一次或最后一次时的情况）。这些边界情况的特殊值最容易暴露软件中的错误。

7.3.2 代码审查

人工测试可以由编程人员非正式进行，也可以由审查小组正式进行，而后者即为代码审查。

审查小组应由4人组成，如果设计和编写是同一人或编写测试是同一人，则应再加一名程序员保证4人规模  
①组长：有能力的程序员，但没有直接参与该项工程  
②程序设计者  
③程序编写者  
④程序测试者

审查会议有两种形式：一种是经典形式的，由设计编写人员先后介绍设计与实现的过程，再根据程序常见错误清单，针对性分析审查程序。另一种是模拟预演式的，由一个人准备测试方案充当测试者，其他人扮演计算机来手动执行程序逻辑，虽然过程肯定是简化的，但是为分析和讨论提供了基础。注意，审查仍然目的是发现错误，改正错误集中在调试阶段。

人工测试的特点和优点：  
①一次审查可以发现许多错误，而计算机测试发现错误后一般需要改正之后才能继续测试，因此只能逐个发现错误  
②人工测试或审查通过质疑和讨论程序逻辑以及设计构想来发现错误。但对于某些类型的错误，计算机测试仍然效率更高。

7.3.3 计算机测试

模块不是独立的程序，不能直接运行以测试，必须开发驱动软件和（或）存根软件来“模拟”模块的运行环境。

驱动程序（Driver）：一个主程序，接收测试数据，，把数据传输给被测试模块

存根程序（Stub）：代替被测试模块所调用的模块，也称为虚拟子程序；使用它所代替的模块的接口，做一些简单的操作

模块的内聚程度高可以简化单元测试过程。

注意：一些复杂的模块没有办法通过简单的单元测试充分暴露问题，因为驱动程序和存根程序不可能编写得很复杂，否则额外增加了开销。可以在集成测试的过程中使用渐增式测试方法完成对这种模块的详尽测试。

## 7.4 集成测试

集成测试是测试和组装软件的系统化技术，目的在于发现与接口有关的问题。子系统和系统测试都属于集成测试。可能发生的接口问题非常多。

由模块组装成程序有两种办法：  
（1）非渐增式测试方法：先分别测试每个模块，再把所有模块按设计要求组合进行测试；先单元测试，再集成测试  
（2）渐增式测试：把下一个要测试的模块同已经测试好的模块结合起来进行测试，每次测试逐个添加模块；实际上同时完成单元测试和集成测试（独立进行的单元测试仍然有意义，且在实现阶段由开发人员完成，并不冲突）

两种办法的特点：  
（1）非渐增式测试一下子要从单个模块转而面向庞大的系统整体，属于“一步到位”的思想。面对的可能错误纷繁复杂，难以定位且数量众多，因而调试过程也很难进行。  
（2）渐增式比较容易定位和改正错误，对接口可以进行更彻底的测试，且可以使用系统化的测试方法。因此应用更普遍。

使用渐增式测试方法有两种集成策略。

7.4.1 自顶向下集成

这种策略是按照结构层次图，从主控模块开始向下逐渐把模块结合起来。类似于遍历树，具体的结合顺序有深度优先和广度优先策略。

把模块结合进软件结构的具体过程，后3步是一个循环过程：  
（1）对主控模块进行测试，测试时用存根程序代替所有直接附属模块  
（2）根据选定的结合策略（广度或深度），每次用一个实际模块代换一个存根程序，并为新进模块添加存根程序  
（3）结合进一个模块的同时进行测试  
（4）为保证加入模块没有引进新的错误，可能需要进行回归测试（重复以前做过的测试）

自顶向下方法能够在测试的早期对主要的控制或关键的抉择进行检验，因为往往这些关键抉择位于上层模块。

深度优先的结合策略可以在早期实现软件的一个完整子功能，并且验证它。在管理上，这提振了开发和用户双方信心

自顶向下法的问题是：为了充分测试高层的模块，需要实际的低层模块参与，而不仅仅是存根程序。

7.4.2 自底向上集成

自底向上集成不需要存根程序

实现自底向上策略的步骤，后3步是一个循环：  
（1）把低层模块组合成实现某个特定的软件子功能的族  
（2）写一个驱动程序（测试用控制程序），协调测试数据的输入和输出  
（3）对由模块组成的子功能族进行测试  
（4）去掉驱动程序，沿软件结构自下向上移动，把子功能族组合起来形成更大的子功能族

7.4.3 不同集成测试策略的比较

自顶向下：  
①优点：不需要测试驱动程序，能够在测试早期实现并验证系统的主要功能，能够在早期发现上层模块的接口错误  
②缺点：需要存根程序，以及与此相联系的困难，不能在早期充分展开人力

自底向上：和自顶向下相反

混合策略：  
（1）改进的自顶向下测试方法  
基本上使用自顶向下的测试方法，但在早期使用自底向上的方法测试软件中的少数关键模块。这种策略具备自顶向下法所有优点，但在测试关键模块时需要驱动程序。  
（2）混合法  
对软件结构中较上层使用自顶向下法，对结构中较下层使用自底向上法。当被测试的软件中关键模块比较多时，这种方法可选

7.4.4 回归测试

定义：重新执行已经做过的测试的某个子集，保证测试和调试过程中引入的程序变化（程序变化包括调试等行为对代码进行的修改，也包括集成测试等过程中逐渐添加模块式的修改）没有带来非预期的副作用

回归测试可以人工进行，也可以使用自动化的捕获回放工具自动进行。该工具可以捕获测试用例和实际运行结果，然后回放（重新执行用例）并比较软件变化前后的结果。

回归测试用例集包括下述3类不同的测试用例：  
①检测软件全部功能的代表性测试用例  
②专门针对可能受修改影响的软件功能的附加测试  
③针对被修改过的软件成分的测试

回归测试不应该也不可能完全的重复全部测试用例，因为每次修改之后都应执行回归测试，那样效率较低。应该精心设计选取能够测试程序的主要功能中的一类或若干部分类错误的用例。

## 7.5 确认测试

验证（verification）：保证软件正确地实现某个特定要求的一系列活动

确认（validation）：为了保证软件确实满足了用户需求而进行的一系列活动

软件有效性：如果软件的功能和性能如同用户所合理期待的那样，就是有效的。需求规格说明即是软件有效性的标准，也是进行确认测试的基础。

7.5.1 确认测试的范围

确认测试必须有用户积极参与或者以用户为主进行，必要时对用户提前进行培训

确认测试通常使用黑盒测试法。

测试计划：包括要进行的测试的种类及进度安排

测试过程/方案：规定了用来检测软件是否与需求一致的测试方案

测试的方向：保证软件能满足所有功能要求，能达到每个性能要求，文档资料准确而完整，另外还有安全性，可移植性，兼容性和可维护性等

如果发现错误，往往是需求分析阶段的差错，解决起来比较困难，需要充分协商

7.5.2 软件配置复查

确认测试的一个重要内容是复查软件配置，保证其所有成分齐全并且合乎要求

7.5.3 Alpha和Beta测试

软件产品如果是专门为某个客户定制开发的，那么验收测试就由最终用户进行

软件产品如果是面向公众发售的，则使用Alpha测试和Beta测试过程。这个测试过程仍然需要最终用户群体（其中的部分）参与进行。

Alpha测试：由用户在开发者的场所进行，并且在开发者对用户的“指导”下进行测试。开发者负责记录发现的错误和遇到的问题。Alpha测试是在受控的环境中进行的

Beta测试：由软件的最终用户在一个或多个客户场所进行，而开发者通常不在测试的现场。测试过程中的问题和建议由用户记录与反馈，开发者再据此对产品进行必要修改。Beta测试是软件在开发者不能控制的环境中的“真实”演练。

## 7.6 白盒测试技术

测试方案：包括具体的测试目的，要测试的功能，应该输入的测试数据和预期结果

测试用例：测试数据和预期输出结果

设计测试方案的基本目标是确定一组最可能发现某个错误或某类错误的测试数据。

设计测试数据的技术很多种，各有优劣，应该联合分情况具体应用。

7.6.1 逻辑覆盖p162/184

含义：对一系列测试过程的总称，这组测试过程逐渐进行越来越完整的通路测试。测试数据执行时所通过的路径覆盖所有程序逻辑的程度可以划分成不同等级。

从覆盖源程序语句的详尽程度分析，有以下不同的覆盖标准：

（1）语句覆盖  
含义：选择足够多的测试数据，使被测程序中每个语句至少执行一次。  
语句覆盖是很弱的逻辑覆盖标准，也就是对程序逻辑覆盖很少。一般的条件判定也只选择一种情况测试（进入条件体的情况）。

（2）判定覆盖  
含义：又叫分支覆盖，满足语句覆盖，而且每个判定的每种可能的结果都应该至少执行一次（对于多判定的组合情况不考虑）。  
判定覆盖比语句覆盖强，但对于程序逻辑的覆盖程度仍然不高，因为没有考虑多分支条件之间的组合情况

（3）条件覆盖  
含义：满足语句覆盖，而且使判定表达式中的每个简单条件都取到各种可能的结果。  
判定表达式中的简单条件是指一个单一的布尔变量或一个关系表达式。关系表达式是由一个关系算符（<，≤，=，≠，＞，≥）连接的两个算术表达式。判定表达式可能是简单条件或复合条件。复合条件由两个或更多简单条件、布尔算符（OR|，AND&，NOT）和括号组成。特别的，不含关系表达式，只有布尔变量和（或）布尔算符的条件称为布尔表达式。  
对于每个条件只考虑满足与不满足两种情况，且每种情况只要覆盖至少一次即可，亦即仍不考虑组合情况。  
条件覆盖通常比判定覆盖强，因为前者考虑每个条件，而后者考虑条件表达式结果。但是因为不考虑组合情况而选取特殊值导致仍存在很大任意性。有时满足条件覆盖的情况可能还不满足判定覆盖。

（4）判定/条件覆盖  
含义：同时满足判定和条件覆盖的标准。  
根据上面的分析，判定/条件覆盖有时和单纯的条件覆盖效果一样

（5）条件组合覆盖  
含义：选取的测试数据使得每个判定表达式中条件的各种可能组合都至少出现一次  
这种覆盖标准考虑了单个判定表达式中的所有条件的组合情况，但是不考虑多个判定表达式结果之间的组合情况。也就是说，如其名曰，是条件组合不是判定组合。  
条件组合覆盖标准包含了前4种标准，是最强的，但仍然不一定是完全覆盖。

从对程序路径的覆盖程度分析，有以下几种覆盖标准：

程序路径由流图表示，流图是连通的有向图

（6）点覆盖  
图论含义：如果连通图G的子图G’是连通的，且包含G所有结点，则G’是G 的点覆盖  
测试含义：测试数据使的程序执行路径至少经过流图的每个结点一次。点覆盖和语句覆盖标准相同。

（7）边覆盖  
图论含义：如果连通图G的子图G’是连通的，且包含G的所有边，则G’是G的边覆盖  
测试含义：测试数据使得程序执行路径至少经过流图中每条边一次。通常边覆盖和判定覆盖一致。

（8）路径覆盖  
含义：测试数据使程序的每条可能路径都至少执行一次（若程序图中有环，则要求每个环至少经过一次）

7.6.2 控制结构测试

白盒测试技术很多是根据程序的控制结构设计测试数据的技术，以下是常用的控制结构测试技术

（1）基本路径测试p166/188

由Tom McCabe提出，该测试方法满足语句覆盖标准和条件覆盖标准

步骤如下：  
①根据过程设计结果画出相应的流图  
②计算流图的环形复杂度  
③确定线性独立路径的基本集合  
线性独立路径：是指至少包含一条在定义该路径前不曾用过的边  
程序的环形复杂度=程序中独立路径的数量  
判定节点：流图中产生分支的结点  
④设计可强制执行基本集合中每条路径的测试用例

注意：某些流图中的独立路径实际中不能以独立的方式测试，而是必须作为另一个路径的一部分来测试。这只是一个现象，往往不需要特殊处理。

（2）条件测试  
（有关条件的概念，见“条件覆盖”部分知识）  
条件测试方法着重测试程序中的每个条件。条件出错可以按照其成分类型进行归因分类  
条件成分的类型：布尔算符，布尔变量，布尔括号，关系算符和算术表达式  
条件测试策略优点：  
①容易度量条件的测试覆盖率  
②程序内条件的测试覆盖率可指导附加测试的设计  
③条件测试不仅仅能检测程序条件中的错误，也能顺带检测程序中的其他错误  
a）分支测试  
最简单的条件测试策略，对于复合条件C，C的真分支和假分支以及C中的每个简单条件都应该至少执行一次（满足条件覆盖）  
b）域测试  
要求对一个关系表达式执行3个或4个测试。分别要使关系表达式两端的算术表达式之间满足大于，等于或小于的关系。如此可以发现关系算符的错误。  
让两个算术表达式之间不等，但差别尽量小的测试用例可以发现算术表达式中的错误  
c）穷举策略（未命名）  
对包含n个变量的布尔表达式进行2n种测试，可以发现所有种类的条件错误，但n较大时效率显然不高  
d）BRO（branch and relational operator）测试  
具体方法和相关概念见p169/191，只是简要介绍  
如果条件中所有布尔变量和关系算符都只出现一次而且没有公共变量，则BRO测试保证可以发现条件中的分支错和关系算符错。  
条件C的条件约束D定义为C的各个简单条件的输出组合，亦即条件C的各种内部取值情况。BRO方法通过测试针对条件而指定的几种输出组合（具体见书中或资料，但肯定少于穷举情况），来验证条件的某些成分的正确性（不同情况下能够验证的成分也许不同）。  
（3）循环测试  
一种白盒测试技术，专注于测试循环结构的有效性。  
结构化的程序中通常循环类型只有3种，分别有对应的测试方法（选取测试用例来激发程序的哪些行为来测试）：  
①简单循环：（n为允许通过循环的最大次数）  
跳过循环；只通过循环一次；通过循环两次；通过循环m次，其中m<n-1；通过循环n-1,n,n+1次。  
②嵌套循环：循环体内部包含另一循环  
③串接循环：多个循环体前后串连，不存在包含关系  
如果各个循环彼此独立，采用简单循环测试方法；如果不独立（比如前一个循环计数器值为下一个循环的初始值），采用嵌套循环测试方法。

## 7.7 黑盒测试技术

黑盒测试着重测试软件功能。黑盒不能取代白盒测试，二者是互补的，彼此可以发现不同类型的错误

黑盒测试试图发现的错误类型：  
（1）功能不正确或遗漏功能  
（2）界面错误  
（3）数据结构错误或外部数据库访问错误  
（4）性能错误  
（5）初始化和终止错误

白盒测试在测试过程的早期阶段进行，黑盒主要用于测试后期。

设计黑盒测试方案时要考虑的问题：  
（1）怎样测试功能的有效性？  
（2）哪些类型的输入可构成好测试用例？  
（3）系统是否对特定的输入值特别敏感？  
（4）怎样划定数据类的边界？  
（5）系统能够承受什么样的数据率和数据量？  
（6）数据的特定组合将对系统运行产生什么影响？

黑盒测试技术的测试用例可以实现的作用：  
（1）能够减少为达到合理测试所需要设计的测试用例的总数  
（2）能告诉人们，是否存在某些类型的错误，而不仅仅指出与特定测试相关的错误是否存在

7.7.1 等价划分

这是一种黑盒测试技术，这种技术把程序的输入域（亦即所有可能的输入数据）划分成若干个数据类，这些数据类称为“等价类”，并分为有效等价类和无效等价类。从每个等价类中只取一组数据作为测试用例（认为它能代表其所在类的所有输入数据）。

划分和确定输入数据的等价类有以下的启发式规则，这些规则同样适用于输出数据，进而可以反过来指导输入数据的有效类划分：  
（1）若规定了输入值的范围，则可划分出一个有效等价类（输入值在此范围内）和两个无效等价类（小于或大于该范围边界的输入值）  
（2）若规定了输入数据的个数，则类似地也可以划分出一个有效类和两个无效类  
（3）若规定了输入数据的一组值，而且程序对不同输入值做不同处理，则每个允许的输入值是一个有效的等价类，此外还有一个无效的等价类（所有不允许的输入值）  
（4）若规定了输入数据必须遵循的规则，可以类似地划分出一个有效类（符合规则）和若干个无效类（从各种不同角度违反规则）  
（5）若规定了输入数据为整型，则可以划分出正整数，零和负整数3个有效类  
（6）若程序的处理对象是表格，则应该使用空表，以及含一项或多项的表。  
（7）在划分无效的等价类时必须考虑编译器的检错功能，没必要设计检测编译器自己就能发现的错误

根据等价类设计测试方案的步骤：（因为程序发现一个错误后就一般不再检查其他错误，因此无效类的覆盖必须一一进行）  
（1）设计一个新的测试方案以尽可能多地覆盖尚未被覆盖的有效等价类，重复这一步骤直到所有有效类都被覆盖  
（2）设计一个新测试方案，使它覆盖一个而且只覆盖一个尚未被覆盖的无效类，重复直到所有无效类都被覆盖

等价划分法设计测试方案举例：p173/195

7.7.2 边界值分析

该方法即检测程序处理边界情况的表现。经常和等价划分技术联合使用，因为等价划分技术提供了参考边界以及范围内的典型（或随机）测试用例，而边界值分析考虑这些范围的边界附近的测试用例。

确定边界情况也是需要经验与创造性，但是可以通过参考输入输出等价类的划分边界。选取的测试值也要包含刚好等于，刚刚小于和大于边界值的3种情况。

7.7.3 错误推测

除了系统化的等价划分和边界值分析技术，不同类型不同特点的程序有时会有自己特殊的易出错问题。更有时，单独的测试数据能正常工作，但它们组合在一起却会出错。

错误推测实际上就是依靠直觉和经验对程序可能有的问题和错误进行推测，设计出对应的测试方案的过程

经验表明，程序中已经发现的错误数目往往和尚未发现的错误数成正比。另一种说法就是，系统中的错误往往集中在少数模块上。应该对发现错误或错误较多的模块着重测试。

等价划分和边界值一般针对的是单个输入数据的测试结果。没有考虑多个输入数据的组合效应。对输入组合形式的测试用例的设计可以使用判定表或判定树来辅助，为判定表的每一列（一种组合）至少设计一个能覆盖到的测试用例。

对于输入数据的组合形式设计，可能需要对代码有所了解，或者结合代码检查。了解模块运作原理，设计合理的测试用例。

## 7.8 调试

调试也称纠错。调试就是把软件错误和内在原因联系起来的尚未被人深入认识的智力过程。

7.8.1 调试过程

调试过程的结果：  
①找到问题原因，并改正，然后执行回归测试，以防修改产生副作用  
②没找到原因，进行新的设想并设计测试验证

软件错误的特征：  
（1）症状和其原因可能在程序中相距甚远。如果耦合程度高，这种情况可能性更大。  
（2）当改正了另一个错误之后，症状可能暂时消失了  
（3）症状可能实际上并不是错误引起的，比如舍入误差  
（4）症状可能是由不易跟踪的人为错误引起的  
（5）可能是由定时问题而不是由处理问题引起  
（6）可能很难重新产生完全一样的输入条件，比如实时应用系统。难以重现问题（不是不可能）因而难于纠错  
（7）症状可能时有时无，在嵌入式系统中常见  
（8）症状可能是由分布在许多任务中的原因引起的，这些任务运行在不同的处理机上

7.8.2 调试途径

无论如何，调试是一个同时需要系统化分析、直觉与运气的过程。

常见的调试途径有：

（1）蛮干法  
应当作为其他方法都失败时的最后方法。  
这种方法尝试印出内存内容，激活对运行过程的跟踪，在程序中添加临时的输出语句用来分析错误。通过对程序运行产生的大量数据进行挖掘来找出错误。

（2）回溯法  
适用于小程序（可能是模块）  
从发现症状的地方开始，人工沿程序的控制流往回追踪分析源程序代码，以图纠错。程序规模过大时，路径数目变多，难以回溯。

（3）原因排除法

包括对分查找法、归纳法和演绎法

①对分查找法  
思路：若已知每个变量在程序内若干个关键点的正确值，可以用赋值语句或输入语句在程序中点（1/2处）附近“注入”这些变量的正确值，然后运行程序并检查输出。若结果正确，则错误在前半部分，反之在后半部分。如其名，形如二分查找。

②归纳法  
思路：从个别现象推断出一般性结论的思维方法。首先把和错误有关的数据组织起来进行分析，以便发现可能的错误原因。然后导出对错误原因的所有假设，并利用已有的数据来证明或排除这些假设。必要时可以设计更多测试用例，以获得更多数据。

③演绎法  
思路：从一般原理或前提出发，经过排除和精化的过程推导得出结论。首先设想所有可能的出错原因，然后试图用测试来排除每一个假设的原因。发现确实原因后，可对数据进行细化以准确定位错误。

如果靠个人力量无法发现和解决错误，可以向同行求助。这也是必要的工程操作。

在改正错误之前，应该考虑的3个问题：  
（1）是否同样的错误也在程序其他地方存在？  
很多错误是错误的逻辑思维模式造成的，循着这种思维模式可以定位一些其他类似错误  
（2）将要进行的修改可能会引入的“下一个错误”是什么？  
改正错误之前应该仔细研究源程序与设计文档，以评估逻辑和数据结构的耦合程度。如果要做的修改位于高耦合段，修改时必须谨慎  
（3）为了防止今后出现类似错误，应该做什么？  
这是一个思考总结，系统化提升的过程。

## 7.9 软件可靠性

7.9.1 基本概念

①软件可靠性的定义：软件可靠性是程序在给定的时间间隔内，按照规格说明书的规定成功地运行的概率。因此，运行时间的增加，在一定条件下，会导致程序故障概率的必然增加；可靠性也随着被考察时间的加大而减少。

以下的定义由IEEE规定给出——

错误：由开发人员造成的软件差错bug

故障：由错误引起的软件的不正确行为

②软件的可用性：程序在给定的时间点，按照规格说明书的规定，成功运行的概率

对于任何其故障可以修复的系统，应该同时使用可靠性和可用性衡量它的优劣程度

平均无故障时间MTTF：修复一个故障平均需要用的时间，取决于维护人员水平和对系统熟悉程度，也和系统的可维护性有关

平均维修时间MTTR：系统按规格说明书规定成功运行的平均时间，主要取决于系统中潜伏的错误数目，和测试关系密切

系统的稳态可用性：,,, Tdown为软件系统故障停机时间tdi的总和，同理Tup是正常运行时间tui总和

系统稳态可用性另一种表示：

7.9.2 估算平均无故障时间的方法

MTTF是一个重要的质量指标，从用户角度来说往往是重要需求

（1）符号

ET——测试之前程序中错误总数

IT——程序长度（机器指令总数）

τ——测试（包括调试）时间

Ed(τ)——在0至τ期间发现的错误数

Ec(τ)——在0至τ期间改正的错误数

（2）基本假定

①在类似的程序中，单位长度里的错误数ET/IT近似为常数。统计数字表明  
0.5\*10-2≤ET/IT≤2\*10-2  
亦即，在测试之前每1000条指令中大约有5-20个错误

②失效率正比于软件中剩余的（潜藏的）错误数，MTTF与剩余错误数成反比

③假设发现的每一个错误都立即正确地改正了，即调试过程中没有引入新的错误，因此Ec(τ)=Ed(τ)  
剩余错误数Er(τ)= ET - Ec(τ)，单位长度程序中剩余错误

（3）估算平均无故障时间MTTF

根据基本假定

其中K为常数，是一个经验数值。统计表明，K典型值为200。上式可以评价软件测试的进展情况。

改写上式，根据对于MTTF的要求，估计需要改正多少错误之后，测试方可告一段落

（4）估计错误总数ET的方法

①植入错误法

测试之前由专人（非测试人员）在程序中随机植入一些错误，数目为Ns。测试之后，发现ns个植入的错误，以及n个原有的错误。假定发现植入错误与原有错误的能力相同，则可以线性估计原有错误总数为

缺点：植入的错误很可能和原有错误类型不一致，所以很难假定发现二者的能力相同。

②分别测试法

改进的植入错误法，不引入新的人为错误。同时进行两组测试，在一组测试中为发现的原有错误添加标记，将被标记的错误看成是上一方法中植入的错误而评估另一组测试

设：

τ=0时，错误总数为B0

τ=τ1时，测试组1发现的错误数为B1，2组发现的错误数为B2

τ=τ1时，两组测试发现的相同错误数为bc

估计出错误总数为

可以每隔一段时间进行多次分别测试，将结果取平均作为正式的ET估计值。之后便可结束分别测试工作，两组人员的工作可以结合，因此增加的测试成本并不多。

# 第8章 维护

软件工程的主要目标之一就是提高软件的可维护性，减少维护所需工作量与成本。维护阶段已经越来越占据主要的开发总成本，趋势可能会导致维护期间开发组织无力开发新的软件。

## 8.1 软件维护的定义

软件维护：软件已经交付使用之后，为了改正错误或满足新的需要而修改软件的过程。

软件维护一般可具体由4项活动定义组成：

①改正性维护  
程序使用期间，用户发现程序错误，报告给维护人员修改。占维护工作17%-21%。

②适应性维护  
为了和变化了的环境（硬件，操作系统等）适当地配合而进行的修改软件的活动。必要又经常的维护工作，占18%-25%。

③完善性维护  
软件使用期间，用户提出增加新功能或改进意见。占维护工作大部分50%-66%。

④预防性维护  
为了改进未来的可维护性和可靠性，给未来的改进奠定更好的基础。占较少的维护活动4%。

## 8.2 软件维护的特点

8.2.1 结构化维护与非结构化维护差别巨大

（1）非结构化维护

非结构化指的是软件配置的唯一成分是程序代码。维护代价极大！

（2）结构化维护

①评价设计文档，确定软件重要的结构特点、性能特点以及接口特点

②估量要求的改动将带来的影响，并且计划实施途径

③修改设计并且对修改进行仔细复查

④编写相应代码

⑤基于测试说明书的信息进行回归测试

⑥交付修改后的软件

8.2.2 维护的代价高昂

维护过程带来的代价（定性）：  
①维护费用远高于开发费用，而且生产率也较开发过程大幅度下降（开发过程的每条指令开发成本远低于维护过程的每条指令成本）  
②因可用资源必须供维护任务，耽误乃至丧失开发新程序的良机  
③维护的改动在软件中引入潜伏的错误  
④对于维护的需求无法及时满足时引起用户不满（维护过程无法完全完整地重复开发过程的设计讨论分析过程，因而有一定简化劣化趋势）  
⑤人事调度上引起的混乱

维护工作可以分成两种类型：  
生产性活动——分析评价，修改设计和编写代码  
非生产性活动——理解代码，解释数据结构、接口特点和性能等

维护工作量的模型：M=P+K\*exp(c-d)  
M维护总工作量；P生产性工作量；K是经验常数，c为软件复杂程度，d为维护人员对软件的熟悉程度

8.2.3 维护的问题很多

与软件维护有关的部分问题，都可或多或少归因于软件定义和开发方法的不当，即软件工程学在前期工作中的应用问题：

（1）理解别人的代码非常困难

（2）软件配置不足，文档资料不合格。

（3）要进行维护时，没有原来参与开发的人员讲解和咨询

（4）软件在设计时，没有考虑将来的修改。

（5）软件维护工作令人沮丧无趣。

## 8.3 软件维护过程

软件维护本质上是修改和压缩了的软件定义和开发过程。大致步骤如下：

（1）维护组织的建立

维护组织包括维护管理员A，系统管理员B，变化授权人C组成。可以是一个松散的组织，但必须明确个人的责任与分工。

每个维护要求通过A转交给熟悉该产品的B进行评价（或AB共同评价）。最终由C决定应该进行的活动。

（2）维护报告

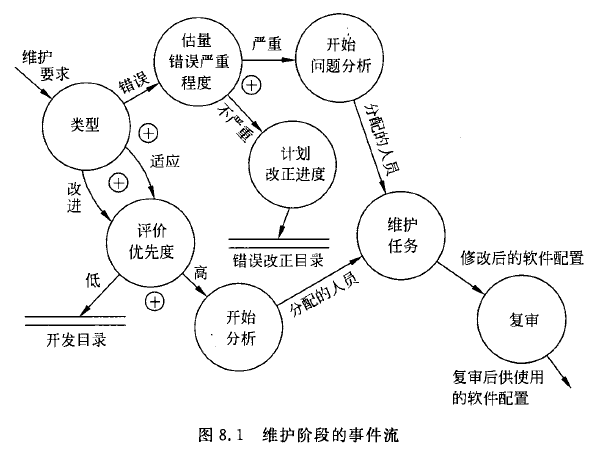
维护报告是标准化格式表达的软件维护要求。包括维护要求表和软件修改报告。

一般维护人员提供给用户空白的维护要求表（或称软件问题报告表），填写时根据具体要求提供必要的信息：  
①改正性维护——完整描述的致错误出现的环境（输入数据，输出数据和其他信息）  
②适应性或完善性维护——简短的需求说明书

根据维护要求表，维护组织AB应制定一个软件修改报告，包含以下信息：  
①满足维护要求表中提出的要求所需要的工作量  
②维护要求的性质  
③这项要求的优先次序  
④与修改有关的事后数据

将软件修改报告提交C审查批准。

（3）维护的事件流



完成软件维护任务之后，应该进行处境复查，思考如下问题：  
①在当前处境下设计、编码或测试的哪些方面能用不同方法进行？  
②哪些维护资源是应该有而事实上没有的？  
③对于此项维护工作什么是主要的障碍？  
④要求的维护类型中有预防性维护吗？

（4）保存维护记录

每一次维护不仅仅要（可能）修改系统代码和修改文档，也要记录下当次修改的有关信息。因为系统代码和文档永远（应该）是最新的，无法反应过去进行的修改，更无法用于分析维护阶段的信息。

值得记录的数据：

①程序标识；②源语句数；③机器指令条数；④使用的程序设计语言；

⑤程序安装的日期；⑥从安装以来程序运行次数；⑦从安装以来程序失效次数；

⑧程序变动的层次和标识；⑨因程序变动而增加的源语句数；

⑩因程序变动而删除的源语句数；⑪每个改动耗费的人时数；⑫程序改动的日期；

⑬软件工程师姓名；⑭维护要求表的标识；⑮维护类型；⑯维护开始和完成的日期；

⑰累积用于维护的人时数；⑱与完成的维护相联系的纯效益

可利用这些数据构成一个维护数据库的基础为每项维护工作收集这些信息记录

（5）评价维护活动

对维护工作的度量所采取的角度：  
①每次程序运行平均失效的次数  
②用于每一类维护活动的总人时数  
③平均每个程序、每种语言、每种维护类型所做的程序变动数  
④维护过程中增加或删除一个源语句平均花费的人时数  
⑤维护每种语言平均花费的人时数  
⑥一张维护要求表的平均周转时间  
⑦不同维护类型所占的百分比

根据定量度量结果，可以做出关于开发技术、语言选择、维护工作量规划、资源分配等方面的决定，也可以据此分析评价维护任务

## 8.4 软件的可维护性

定义：维护人员理解、改正、改动或改进这个软件的难易程度

8.4.1 决定软件可维护性的因素

（1）可理解性  
定义：对于外来读者理解软件的结构、功能、接口和内部处理过程的难易程度  
模块化良好、详细文档、结构化设计、高级程序设计语言都有助于可理解性

（2）可测试性  
和可理解性、文档、软件结构复杂度、可用的测试调试工具等有关

（3）可修改性  
耦合、内聚、信息隐藏、局部化、控制域与作用域关系等有关

（4）可移植性  
定义：把程序从一种计算环境（硬件配置和操作系统）转移到另种计算环境的难易程度  
把与硬件、操作系统以及其他外部设备有关的程序代码集中到特定模块中，有助

（5）可重用性（reuse）  
定义：同一事物不做修改或稍加改动就在不同环境中多次重复利用  
①可重用的软件结构在开发时经过多次严格测试，重用次数越多，可靠性越高，改正性维护需要的少  
②很容易修改可重用的软件结构使之再次应用在其他环境，非改正性维护更容易

8.4.2 文档

文档可以分为两大类——用户文档和系统文档。前者描述系统功能和使用方法，不注重描述机理；后者描述系统整个细节。

软件文档应该满足的整体要求：  
①必须描述如何使用系统  
②必须讲怎样安装和管理系统  
③必须讲系统需求和设计  
④必须讲系统的实现与测试

（1）用户文档  
实际上就是用户手册和使用说明书，并且附带详细的目录，并包含以下内容：  
①功能描述——系统的功能  
②安装文档——如何安装以及怎样使系统适应特定的硬件配置  
③使用手册——如何使用，通过例子说明常用的系统功能，说明遇到常见错误时（不需要维护）如何恢复与重启  
④参考手册——详细描述用户可以使用的所有系统设施与使用方法，解释系统产生的所有出错（报错）警告信息的含义。参考手册重在完整，通常使用形式化描述技术  
⑤操作员指南——若有系统操作员，可详细说明其工作内容和帮助

（2）系统文档（略）

8.4.3 可维护性复审

可维护性复审是所有软件工程阶段末审查步骤的重要部分。在不同的阶段结束期，从不同的角度考虑和确认软件系统可维护性，也应从现在和将来的不同角度出发分析。

测试阶段结束时进行的可维护性复审是最正式的，也称为配置复审。保证软件配置的完整性等。

## 8.5 预防性维护

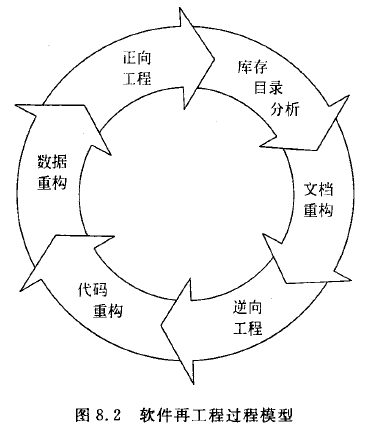
对于没有使用软件工程方法开发的“老程序”的维护方法：  
①反复多次修改尝试，蛮力法  
②仔细分析程序，尽可能多掌握程序工作细节  
③在理解原有设计的基础上，用软件工程方法重新设计开发需要维护的部分  
④用软件工程方法作为指导，对程序全部重新设计开发（逆向工程和再工程）

软件再工程是一种预防性维护措施。目的在于用更新的方法理念重新开发或修改原有的系统（这么做的必要并不一定是因为需要改正性或非改正性维护），以使其更加容易支持未来的需求和操作（降低成本）。

预防性维护的益处和便于预防性维护的事实：  
①维护代价是开发代价的14-40倍。因此预防性维护或再工程有低成本高回报的期望  
②重新设计软件结构有助于将来的维护  
③由于现有程序可作为软件原型，开发生产率会高于平均水平  
④现有程序被使用的经验有助于更快完成需求分析  
⑤采用支持逆向工程或再工程的工具，可以使一部分工作自动化  
⑥完成再工程的过程顺带建立了完善的软件配置

预防性维护一般都是主动采取，而不是被动。考虑到其他维护需求的重要性，预防性维护所占比例很低，只有条件具备时（开发与维护空闲时期）才会被考虑。

## 8.6 软件再工程过程



（1）库存目录分析

库存目录即软件组织对其拥有和开发的软件系统的保存和记载。应该包含以下针对每个应用系统的信息：  
①系统名称；②最初构建日期；③已做过的修改次数；④过去18个月的错误报告；  
⑤用户数量；⑥安装它的机器数量；⑦系统复杂程度；⑧文档质量；  
⑨整体可维护性等级；⑩预期寿命；⑪未来36个月的预期修改次数；  
⑫业务重要程度

有3类程序可能成为预防性维护对象，目前的技术问题和成本问题导致不能将所有老程序执行再工程：  
①预定使用多年的程序  
②当前正在成功地使用着的程序  
③在最近的将来可能要做重大修改或增强的程序

分析库存目录时所需考虑的因素（也是用来将老程序按照优先级排序的依据）：  
①业务重要程度；②寿命；③当前可维护性；④预期的修改次数

（2）文档重构

因具体情况不同，有几种可能的处理方法：  
①若程序相对稳定，很快即将弃用，也不会有什么改动，可以不做文档  
②采用边使用或修改边建文档的方法，从使用频率高的部分开始逐渐完善文档  
③如果一定需要立刻全部重构，则仍然应该设法减小这个工作量

（3）逆向工程  
定义：分析程序以在比源代码更高的抽象层次上创建出程序的某种表示过程，亦即由实现结果恢复设计结果的过程。  
逆向工程的辅助工具从现有程序中提取有关数据，体系结构和处理过程的设计信息，如CASE

（4）代码重构  
代码重构的含义局限于模块之内，即重构单个模块内部代码（设计细节和局部数据结构）以符合结构化设计理念。  
使用重构工具可以分析和自动重构代码，之后进行回归测试和更新文档。

（5）数据重构  
即数据结构的重构，始于逆向工程活动。重新定义设计数据模型和属性，并进行复审。这种重构也是较大范围的修改。

（6）正向工程  
也称为革新或改造。包括重新分析设计现有系统的现有功能，也包括加入新的功能。

# 第9章 面向对象方法学引论

20世纪60年代后期出现的Simula-67编程语言，首次引入了类和对象的概念。

## 9.1 面向对象方法学概述

9.1.1 面向对象方法学的要点

（1）认为客观世界是由各种对象组成的，复杂的对象可以由简单的对象组合而成。  
OOP用对象分解取代了传统方法的功能分解

（2）把所有对象都划分成各种对象类（class），每个class都定义一组数据和方法。数据以表示对象的静态属性，或状态信息。

（3）按照子类与父类的关系，把对象类们组成一个层次结构的系统（继承机制）。

（4）对象之间彼此仅能通过传递消息互相联系。

仅使用对象和消息，则这种方法称为基于对象（object-based）；仅使用对象和类，称为基于类的方法（class-based）；同时使用4种元素的才是OOP

9.1.2 面向对象方法学的优点

（1）与人类习惯的思维方法一致  
数据和程序在计算机内部是分开存放的。容易发生调用错误。而OOP的理念强调数据模型的建立（类的建立），而不是算法（计算机的操作过程），虽然后者仍然有价值。抽象和继承机制为基础的设计过程很良好的符合了人类认识事物的过程。

（2）稳定性好  
传统方法基于算法，开发过程基于功能分析和功能分解。当功能需求变化时，稳定性差。OOP基于对象，无论功能如何变化，现实中的对象相对稳定，则OOP程序相对稳定。

（3）可重用性好  
对象本身就是理想的模块和可重用软件成分，具有很强的自含性与独立性。另外继承机制使得对象可以系统化的修改和扩充，对象类的特点和集成电路IC的特点类似。

（4）较易开发大型软件产品  
每个对象都相当于一个微型程序，更加便于管理和开发

（5）可维护性好  
原因如下（总结上文）：  
①OOP软件稳定性好  
②OOP软件比较容易修改  
③OOP软件比较容易理解——解空间结构和问题空间的结构一致  
④易于测试和调试——测试集中在派生的新类中

## 9.2 面向对象的概念

9.2.1 对象

对对象的操作称作服务或方法（成员函数）

（1）对象的形象表示

（2）对象的定义  
①对象是具有相同状态的一组操作的集合  
②对象是对问题域中某个东西的抽象，这种抽象反映了系统保存有关这个东西的信息或与它交互的能力  
③对象::=<ID,MS,DS,MI>。ID为对象的标识或名字，MS是对象中的操作集合，DS是对象的数据结构，MI是对象受理的消息名集合（即对外接口）

（3）对象的特点  
①以数据为中心。操作围绕数据来设置，不设置无关的操作  
②对象是主动的。向对象发送消息才能让它执行操作  
③实现了数据封装。抽象数据类型  
④本质上具有并行性  
⑤模块独立性好

9.2.2 其他概念

（1）类class

（2）实例instance  
实例就是一个具体的对象。

（3）消息message  
3个组成部分：接受消息的对象、消息选择符（消息名）、若干个变元

（4）方法method  
成员函数

（5）属性attribute  
即数据成员。如果有字段的定义（C#），则略有不同

（6）封装encapsulation  
对象具有封装性的条件：  
①有一个清晰的边界  
②有确定的接口（即协议）  
③受保护的内部实现

（7）继承inheritance

（8）多态性polymorphism  
子类对象可以像父类对象那样使用，或同样的消息既可以发送给父类对象也可以发送给子类对象，接受消息的对象不同做出的反应不同

（9）重载overloading  
函数重载和运算符重载；重载是静态联编（先前联编），多态性是动态联编（滞后联编）

## 9.3 面向对象建模

用面向对象方法开发软件，通常需要建立3种形式的模型：  
①对象模型——描述系统数据结构  
②动态模型——描述系统控制结构  
③功能模型——描述系统功能

3种模型彼此不独立，从不同的侧面描述系统。尽管不同问题下，3种模型重要性不同，而且多数时候，对象模型始终最关键；但是3种模型都是必不可少的。

## 9.4 对象模型

对象模型表示静态的、结构化的系统的数据性质。

建模语言或工具由记号（模型符号）和规则（语法、语义、语用）组成。

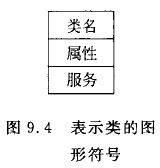
通常使用UML提供的类图来建立对象模型，UML中类的实际含义包括其本身和它实例化出的对象。UML也是OOP的标准建模语言。

9.4.1 类图的基本符号

类图描述类和类与类之间的静态关系。类图是创建其他UML图的基础。一个系统可以由多张类图描述，一个类也可以出现在几张类图中。

（1）定义类

类命名的准则：（描述性、简洁、无二义性）  
①使用标准术语和习惯的术语命名，不要随意创造，要一目了然  
②使用具有确切含义的名词，而不是空洞和模糊的词  
③必要时用名词短语



（2）定义属性

格式：  
可见性 属性名：类型名=初值{性质串}

可见性即可访问性——公有的（用+表示），私有的（-），保护的（#）；没有默认的可见性，即未定义时只表示未定义

类型名——基本数据类型或自定义

性质串——列出该属性所有可能的取值，如枚举类型；也可用来说明属性其他性质

类作用域属性（或类变量）——C++中称作静态数据成员，在该属性全部定义语句下加下划线表示

（3）定义服务

格式：  
可见性 操作名（参数表）：返回值类型{性质串}

参数表——逗号分隔的形参序列，参数格式如下  
参数名：类型名=默认值

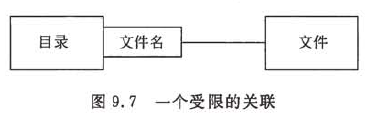
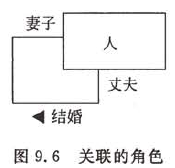
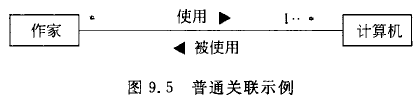
类作用域操作——静态函数成员，加下划线表示

9.4.2 表示关系的符号

类与类之间的关系：关联、泛化（继承）、依赖和细化4种

（1）关联p218/240

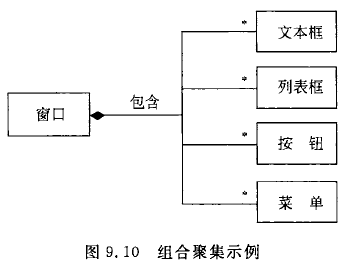
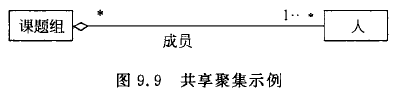
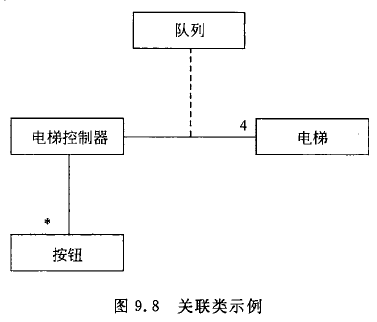
①普通关联  
最常见的关联关系，具体符号和数字标记使用方法见书



②关联的角色  
可以为对象在关联关系中扮演的角色（功能）命名或标识

③限定关联  
限定关联通常用在一对多或多对多关联中，可以把模型中的重数从一对多变成一对一，或多对多简化成多对一。换句话说，限定关联的关键词往往是关联关系中的某个处于相对较低级别（数量较多的一方）的类的标识，如名称、ID、代号等。关键词是依附在某个对象之上的，一般依附在级别较高、关联关系中数量较少的一方。

④关联类  
关联类即用来描述关联的类，也有属性、操作等元素。关联中每个连接与关联类的一个对象相联系。这种关联类的存在也类似E-R图模型的特点。



（2）聚集p220/242

也称为聚合，是关联的特例。表示类与类之间的关系是整体与部分的关系，如包含、组成、部分等。聚集有一般情况下的，也有两种特殊的情况：

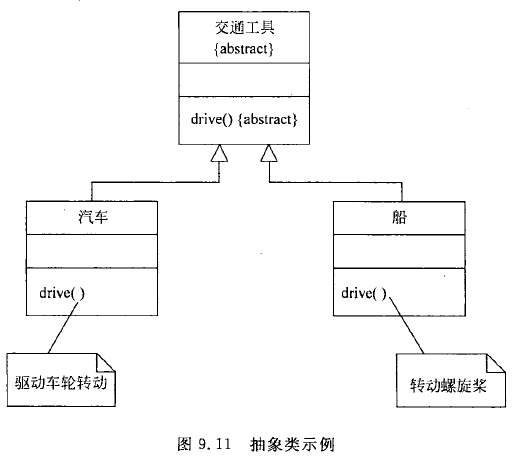
①共享聚集  
定义：在聚集关系中处于部分方的对象可同时参与多个处于整体方对象的构成  
类似E-R模型中的多值情况

②组合聚集（简称为组成）  
定义：部分类完全隶属于整体类，二者共存，若整体不存在了部分也会消失

（3）泛化

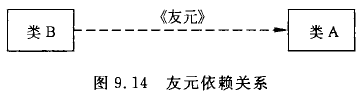
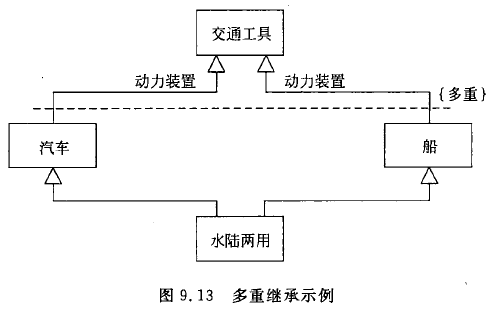
即通常意义上的继承关系。基类称为通用元素，派生类称为具体元素。

①普通泛化  
和继承基本相同。抽象类在类名下方附加一个{abstract}标记，抽象操作的后面跟随一个{abstract}性质串



折角矩形是用来在UML中作注释的标记

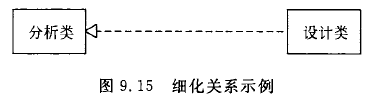
②受限泛化  
即给泛化关系附加约束条件，进一步说明泛化关系的使用方法或扩充方法。  
预定义的约束有4种：  
多重——表示多重继承中，一个子类可以同时多次继承同一个上层基类，即结果是子类中包含相同的多个成员。  
不相交——一般默认的继承关系，表示多重继承中，一个子类不能多次继承同一个基类，即只能保留其一套成员（注意这里并没有哪种约束禁止多重继承）  
完全——父类的所有子类都在类图中穷举出来了，符号是{完全}  
不完全——一般默认的继承关系，表示类图中已有的子类不（一定）是父类的全部子类



（4）依赖和细化

①依赖关系  
表示两个类或对象之间的语义连接关系，即有依赖关系的两个类中，至少有一个类是不独立的。比如：  
——一个类用另一个类的对象作为操作参数  
——一个类用另一个类的对象作为数据成员  
——一个类向另一个类发消息  
——友元关系

②细化关系  
描述同一个事物在不同抽象层次上的关系。注意并不是描述同一开发阶段中基类和派生类之间的关系，而是用来协调和描述不同开发设计阶段模型之间的相关性的。



## 9.5 动态模型

动态模型表示瞬时的、行为化的系统的“控制”性质，它规定了对象模型中的对象的合法变化序列。是基于事件共享而互相关联的一组状态图的集合。

动态模型就是描述对象状态和行为的。  
状态——是由对象的一个或多个属性值（但不一定是全部属性）的取值决定的，当然具体的对象具体的情境不同情况也不同。  
事件——多个对象之间相互作用可能会导致彼此状态变化。每个触发行为称作一个事件。  
响应——包括改变对象自己的状态和形成新的触发行为

每个类的动态行为用状态图（状态图是UML提供和定义的）描绘，各个类的状态图通过共享事件合并起来，从而构成系统动态模型。

## 9.6 功能模型

功能模型一般由一组数据流图组成。不过数据流图在OOP中不如结构分析设计中重要，但是仍然有理解问题域的作用。

UML提供的用例图也可以进行需求分析和建立功能模型，这样的系统功能模型也称为用例模型。用例模型的优点是针对每个或每类用户进行功能说明，比传统的方法更清晰准确。

用例模型的建立也同样需要开发和用户双方的反复讨论并达成共识。

9.6.1 用例图p225/247

用例图包含的模型元素：系统、行为者、用例以及用例之间的关系

（1）系统  
系统被看做是一个黑盒，用方框表示

（2）用例  
用例是可以被行为者感受到的，系统的一个完整的功能。因此它不仅仅包括系统内部完成该功能的运转，也包括和行为者之间的通信与联系。  
用例具有的特征：  
①用例代表某些用户可见的功能，实现一个具体的用户目标  
②用例总是被行为者启动的，并向行为者提供可识别的值  
③用例必须是完整的  
用例是一种功能，不是一个功能的某次执行的具体实例。具体实例称作脚本。

（3）行为者  
指与系统交互的人或其他系统，代表外部实体。使用用例与目标系统交互。  
行为者是一种角色，而不是一个具体的个人或事物。人有时可以充当多种行为者角色。行为者和用例之间的连线称为通信联系。行为者可以分为主行为者和副行为者，也可以分成主动行为者和被动行为者，依据具体情况需要。  
一般对于复杂的系统，先列出行为者清单，然后根据每个行为者确定用例，这样比直接尝试确定所有用例更容易。

（4）用例之间的关系  
用例之间的关系主要有扩展和使用两种。是泛化（类继承）关系的两种不同形式  
①扩展关系  
向一个用例中添加一些动作后构成另一个用例，二者即为扩展关系，扩展后的叫做扩展用例。映射为子类继承父类。  
②使用关系  
一个用例使用另一个用例，称作使用关系。虽然名曰使用关系，但是并不对应对象的调用。因为用例代表类，因此使用关系其实映射为类使用接口。接口或抽象类即对应抽象用例，代表了多个用例之间共同的一些操作。

通常描述一般行为变化时采用扩展关系，在两个或多个用例中出现重复描述又想避免这种重复时，可以采用使用关系。

9.6.2 用例建模

用例模型由若干个用例图组成。创建用例模型的步骤包括：定义系统，寻找行为者和用例，描述用例，定义用例之间的关系，确认模型

（1）寻找行为者  
通过请用户回答一些问题来发现行为者，进一步确定用例：  
①谁将使用系统的主要功能（主行为者）？  
②谁需要借助系统的支持来完成日常工作？  
③谁来维护和管理系统（副行为者）？  
④系统控制哪些硬件设备？  
⑤系统需要与哪些其他系统交互？  
⑥哪些人或系统对本系统产生的结果（值）感兴趣？

（2）寻找用例  
通过请被确认的行为者回答问题来获取用例：  
①行为者需要系统提供哪些功能？行为者自身需要做什么？  
②行为者是否需要读取、创建、删除、修改或存储系统中的某类信息？  
③系统中发生的事件需要通知行为者吗？行为者需要通知系统某些事情吗？从功能观点看，这些事件能做什么？  
④行为者的日常工作是否因为系统的新功能而被简化或者提高了效率？  
⑤系统需要哪些输入输出？输入来自何处？输出到哪里去？  
⑥当前使用的系统（可能是人工系统）存在的主要问题是什么？

一个用例必须至少与一个行为者相关联。允许不知行为者的用例暂时存在。

## 9.7 3种模型之间的关系

总结如下：

（1）动态模型针对每个类建立，描述类实例的生命周期或运行周期

（2）状态转换对应着行为的发生，行为在数据流图中被映射成处理，在用例图中被映射成用例，在类图中对应类的服务。

（3）有时功能模型中的一个处理或用例对应多个类服务，也有一个服务对应多个用例的情况。

（4）数据流图中的数据存储、数据源点终点，通常是对象模型中的对象

（5）数据流图中的数据流，往往是对象模型中属性值，也可能是整个对象

（6）用例图中的行为者，可能是对象模型中的对象

（7）功能模型中的处理或用例可能产生动态模型中的事件

（8）对象模型描述了数据流图中的数据流，数据存储以及数据源/终点的结构

# 第10章 面向对象分析

无论开发方法是不是OOP，分析阶段的工作都是类似的，也就是获取系统需求的过程。OOP方法的软件需求规格说明是由三种模型组成的（上一章），面向对象分析即OOA。

## 10.1 面向对象分析的基本过程

10.1.1 概述

OOA由于需要抽象问题对象和建模，必须有很深的领域知识，所以需要该领域专家的密切配合。

OOA过程可以利用以前针对类似问题进行的分析结果，借助OOA的稳定性和可重用性提高效率。

10.1.2 三个子模型与五个层次

三个模型对应于系统的3个要素，即静态结构（对象），交互次序（动态），数据变换（功能）。对象模型一般是最主要的，动态模型在问题涉及交互作用和时序时（例如用户界面及过程控制）非常重要；功能模型在解决运算量很大的问题时（高级语言编译、科学与工程计算）重要。

大型系统的对象模型通常由下述5个层次组成：主题层（分解大型系统成若干个概念范畴）、类与对象层、结构层（类或对象之间的关系）、属性层和服务层。  
主题层的作用在于落实7±2原则（人的短期记忆能力限于一次记忆5-9个对象）。通过控制可见性和指导读者的注意力（借助主题层划分范围）来帮助读者理解系统。

根据5个层次，建立对象模型也有5项主要活动：找出类与对象，识别结构，识别主题，定义属性，（建立动态模型，建立功能模型），定义服务。建立动态功能模型之后才可能完整建立对象服务。这些活动不是顺序的步骤，可以自由支配，循环反复精化。但这个顺序仍然具有一定的思路指导意义。

## 10.2 需求陈述

10.2.1 书写要点

需求陈述的内容包括：问题范围，功能需求，性能需求，应用环境及假设条件。完整的需求分析过程是构造出完整的3种模型和规格说明。但着手创建模型前，应该有一个大致的需求陈述。

10.2.2 例子

## 10.3 建立对象模型

对象模型由于侧重的是问题实质组成，所以优先于另外2个模型建立，当然完整的对象模型的获得肯定又需要另外2个模型的辅助。模型建立的过程也是反复精化的过程，同时也应注意搜集分析研究以前针对相似问题的分析结果以便重用。

10.3.1 确定类与对象

（例子p236/258）

（1）找出候选的类与对象  
对象既可能是物理实体也可能是抽象概念。  
大多数客观事物可分为以下几类，以此为指导参照可以列出问题域的候选类：  
①可感知的物理实体  
②人或团体组织的角色，如医生教师、财务处等  
③应该记忆的事件，如飞行、演出、访问、事故等  
④两个或多个对象的相互作用（一般具有交易或接触的性质），如购买、纳税、结婚等  
⑤需要说明的概念，如政策、法规等  
还有一种简单的非正式分析方法，可用于初步分析类和对象的划分：  
依据需求陈述，将其中的名词作为类与对象的候选者，形容词作为确定属性的线索，动词作为服务（操作）的候选者。

（2）筛选出正确的类与对象  
筛选依据的标准：  
①冗余  
去掉重复的类，保留最具有描述力名称的类  
②无关  
即去掉在需求陈述中提及但是和当前研究问题无关的类  
③笼统  
去掉一些笼统概括性的词汇，这些词汇要么可以被已经记录的其他更具体的词替代，要么可以通过自行提炼来具体化，要么系统干脆无需记忆这些信息（无关）  
④属性  
需求陈述中有些名词应该被作为对象的属性，应该把这些词分离出来以后参考。但是对于有些独立性很强的性质，还是应该仔细考虑将其作为类而不是属性。  
⑤操作  
需求陈述中有一些词汇既能做名词也能做动词，在不同的环境和问题下，应该做不同的处理，要么作为对象的方法，要么作为一个类，需要慎重考虑。总而言之，当前问题下本身具有属性需独立存在的操作，应该改作为类和对象。  
⑥实现  
暂且去掉一些和设计与实现阶段相关的类和对象。这些类和对象都和如何实现目标系统有关，和问题的宏观逻辑与功能相比只是内部细节。常见的一些这样的类和对象往往都和“日志”，“网络通信”等有关。

10.3.2 确定关联

（例子p238/260）

分析确定关联的时候，会触及问题域边缘情况，也可能会发现一些新的类和对象。

（1）初步确定关联  
需求陈述中的描述性动词或动词词组，通常表示和描述了关联关系。其他隐含的关联仍可能存在于陈述中，应该和专家一起讨论。  
①直接提取动词短语得出关联  
②需求陈述中隐含的关联  
③根据问题域知识得出的关联

（2）筛选  
按照以下标准筛选  
①已删去的类之间的关联  
在找出类和对象过程中因各种原因去掉的类之间的关联，要么也应该删去，要么用其他的类来表示这个关联  
②与问题无关的或应在实现阶段考虑的关联  
③瞬时事件  
关联是问题域的静态结构，有的瞬时事件通过动作（操作或服务）连接了两个类，但是并不表示这两个类有稳定的关系。比如和用例有关的类与和行为者有关的类之间也许只是单纯的瞬时操作关系，并不具有静态关联性。  
④三元关联  
3个以及更多对象之间的关联，大多可以分解为二元关联或用词组描述成限定关联。  
⑤派生关联  
应该去掉那些可以用其他关联定义的冗余关联。比如有ABC三个类，并且已确定AB和BC的关联，那么对于AC之间的关联到底是否值得存在或是否本质上就是前两个关联的组合，应该考虑。

（3）进一步完善  
①正名  
选择含义明确的名称  
②分解  
为了能够适用于不同的关联，必要时应该分解以前确定的类和对象  
③补充  
分解之后的类和对象会有新的关联  
④标明重数  
初步判定关联类型和重数（以后可能经常更改）

10.3.3 划分主题

对于大型复杂系统的开发，在初步识别类与对象及其关联之后，就要划分主题，便于理解和分析整个系统。非常复杂的系统，可能反复几次划分主题和分析结构（类和对象）

确定主题的原则：  
①应该按问题领域而不是功能分解法来确定主题  
②使不同主题内的对象之间相互依赖和交互最少

10.3.4 确定属性

分析阶段不要用属性来表示对象间的关系，暂且只使用关联即可清晰地表达模型。这并非是设计实现阶段，不需要将所有属性确立下来。

确定属性的过程：

（1）分析  
需求陈述中用名词词组表示的内容暗示属性的存在，形容词往往表示可枚举的具体属性。仍需要借助领域知识和常识确定属性。分析阶段只需要考虑与具体应用直接相关的属性（不要考虑超出问题范围的），忽略纯粹用于实现的属性。先找出最重要的属性，后续添加其他的，属性在这个阶段其实并不十分影响系统结构。

（2）选择  
常见的几种情况：  
①误把对象当做属性  
若某个实体的独立存在比它的值更重要，则应该把它作为对象。具体情况下相同实体作为对象或属性应该根据情况考虑。具有自身独立性质，且在当前问题下有意义的实体，才是对象。  
②误把关联类的属性当做一般对象的属性  
若某个性质依赖于某个关联链的存在，则该性质是关联类的属性。特别是多对多关联中，关联类属性很明显。  
③把限定误当成属性  
若把某个属性值固定以后能减少关联的重数，则应该考虑把这个属性重新表述成一个限定词（见限定关联部分）。限定关联在设计实现阶段是否作为类的属性并不是由这里决定的，实际上可以用属性实现限定关联也可以用关联对象。  
④误把内部状态当成了属性  
如果某个性质是对象的非公开内部状态，则应该从对象模型中删除这个属性。这些非公开内部状态是和具体实现对象功能有关的细节（见后文），为了对象模型的简洁可以省略这些细节而不影响对对象模型的理解和解读。  
⑤过于细化  
分析阶段忽略那些对大多数操作没有影响的属性  
⑥存在不一致的属性  
类应该是简单而且一致的。如果得出一些看起来与其他属性毫不相干的属性，则应该考虑把类分解。

10.3.5 识别继承关系

这个步骤并非（完全）是找出已建立对象和类之间的继承（泛化）关系，而是对已建立的类和对象进行知识抽取或抽象。亦即抽象出一些已建立的类的父类，这个过程既有效组织了当前目标系统的结构（有些不同的类可能归纳为同一父类），也提取了必要的知识体系便于日后重用。因此这个过程需要专业知识领域的专家配合，也需要借助以往的经验。

一般使用两种方式建立继承关系：

（1）自底向上  
抽象出现有类的共同性质（属性）泛化出父类，即归纳思维过程。

（2）自顶向下  
把现有类细化成更具体的子类，即演绎思维过程。带有形容词修饰的名词词组往往暗示了一些具体类。但是分析阶段应该避免过度细化。

如果使用多重继承机制，为了避免混乱和复杂，应该选取一个主要父类提供主要的属性和行为，次要的父类只是补充。

10.3.6 反复修改

（例子p244/266）

实际工作中，建模步骤不一定严格按照一定次序进行。有经验的分析人员可以自行决定工作流程以适合自己的工作方式。

## 10.4 建立动态模型

例子p247/269

大致过程：  
①编写典型交互行为的脚本  
②从脚本中提取出事件，确定触发事件的动作对象和接受事件的目标对象  
③排列事件发生的次序，确定对象可能有的状态以及状态间的转换关系，绘制状态图  
④比较所有对象状态图，确保匹配和一致性

10.4.1 编写脚本

定义（建立动态模型过程中）：系统在某一执行期间内出现的一系列事件。在这里脚本的定义和它本身的含义是非常接近的，脚本原意是戏曲话剧等演艺艺术的台词或情节剧本，在这里它要描绘用户（或其他外部设备）对系统交互行为过程中，用户与系统发生的每一个有意义的动作的序列。脚本实际上就是系统行为的准确描述，当然根据使用目的不同，也可能有专门针对某一对象而不是全部系统的脚本。

首先编写正常情况的脚本，再考虑特殊边界情况，最后考虑出错情况，道理类似于测试用例。描述事件序列时，注意描述清事件的三个组成部分（动作对象，目标对象，参数——交换的信息值）。

10.4.2 设想用户界面

大多数交互行为可以分为两部分：应用逻辑（程序内部逻辑）和用户界面（与用户交互）。动态模型着重表示应用系统的控制逻辑，但分析阶段不能忽略用户界面，可以简单创建用户界面的原型，供用户评价。这个阶段重要的是确立信息交换方式（保证全部必要的信息交换，同时界面友好），而不是界面细节。

10.4.3 画事件跟踪图

例子p249/271

事件跟踪图是脚本和状态图之间的桥梁，前者由于自然语言的关系，有时不够简明且存在表达问题。跟踪图即明确事件及事件与对象关系的工具。

（1）确定事件

事件包括系统与用户（或外部设备）交互的所有信号、输入、输出、中断、动作等。传递信息的对象的动作也是事件，大多数对象到对象的交互行为都对应着事件。

注意，一般两个事件参数不同（尽管触发接受对象相同），它们对控制流的导向不同，也就是两个独立的事件。但有时这样的两个或更多事件对控制流的导向相同或相似，就应该把它们作为同一个事件（它们的参数不同在这种情况下对于系统行为没有影响）。

对于事件的分类（两个事件是否可以作为同一事件）的彻底完成需要借助最终状态图的分析，亦即两事件到底是否可以合并取决于目标系统的特点，因此这里只是一个初步分类。

（2）画出事件跟踪图

例子p250/272

事件跟踪图实质上是扩充的脚本（即一个脚本对应一个事件跟踪图），是简化的UML顺序图。  
在事件跟踪图绘制过程中，只会选择一些主动对象（有触发事件的行为，可以有接受事件的行为）作为竖线，其他的对象（被动对象不发送事件）往往是事件的参数。这些主动对象中和系统内部有关的对象才会被绘制状态图。

有时对于被动对象的界定可能有争议。如果该对象不存在可变的状态，即状态属性确定后不再更改，就没有必要将其设计考虑为主动对象。

10.4.4 画状态图

一张状态图描绘一类对象的行为，但是不是任何一个类都需要有一张状态图描绘它的行为。那些仅响应与过去历史无关的输入事件，或把历史作为不影响控制流的参数的类，状态图并不必要（也可以根据该对象是否存在可变的“状态”来判断）。其实只有具有重要交互行为的类（事件跟踪图中竖线涉及的类）才可能值得绘制状态图。另外，系统外部的对象（比如用户等），虽然被当作对象来分析，但是并不是系统内部需要实现的机制，无需为它们绘制状态图。

由事件跟踪图绘制状态图的步骤：  
①选取一条竖线（即一个对象），并先只考虑所有指向该竖线的事件  
②把这些事件作为状态图中的有向边，标以事件名  
③两个事件之间的间隔作为一个状态，并为其命名  
④两个指向竖线的事件之间从竖线射出的事件线，表示对象在当前状态下所做的行为  
⑤再根据其他脚本的事件跟踪图继续完善该对象的状态图，一般来说不同的脚本对应不同的用例情况，对一个特定对象来说有时仅仅只是从某一分支点之后才开始有差异  
⑥特别注意边界情况、特殊情况与出错处理情况的脚本

10.4.5 审查动态模型

各个类的状态图通过共享事件合并为系统动态模型。审查应注意系统级的完整性和一致性。

审查动态模型的方法：  
每个事件都应该既有发送对象又有接受对象，有时发送和接受是同一对象  
没有前驱或没有后继的状态应该着重审查，若其不是交互序列的起点或终点，则必有错  
审查每个事件，跟踪其对系统中各个对象所产生的效果，以保证和脚本匹配

## 10.5 建立功能模型

功能模型表明系统中数据之间的依赖关系和有关的数据处理，由一组数据流图组成。其中处理可以用IPO图表、伪码等多种方式进一步描述。

通常在建立了前两个模型之后（只是建立不是完成了前两个）再建立功能模型。

10.5.1 画出基本系统模型图

基本系统模型由若干个数据源点或终点，以及一个处理框（只有一个）组成，处理框代表了系统加工、变换数据的整体功能。基本系统模型显然并不细化表达系统的功能，只是指明目标系统的边界以及与外部世界之间的交互事件的参数。

10.5.2 画出功能级数据流图

把基本系统模型中的唯一处理框分解成若干个处理框，以描述系统加工、变换数据的基本功能，就是数据流图。

10.5.3 描述处理框功能

例子p255/277

描述时注意不是实现功能的算法，而是功能作用。有两种性质的描述：  
①说明性描述：  
规定了输入值和输出值之间的关系，以及输出值应遵循的规律  
②过程性描述：  
通过算法说明“做什么”。劣于前一种，但是万不得已也可以使用

## 10.6 定义服务

在确定类中应有的服务时，既要考虑该类实体的常规行为，又要考虑在本系统中特殊需要的服务

（1）常规行为

在分析阶段可以认为，类中定义的每个属性都是可以访问的，即每个类中都定义了读写该类每个属性的操作。这些就是常规行为，但是没必要在类图中显式表达。

（2）从事件导出的操作

状态图中发往对象的事件也就是该对象接收到的消息，因此该对象必须有由消息选择符指定的操作，这个操作修改对象状态并启动相应的服务。

（3）与数据流图中处理框对应的操作

要结合数据流图和状态图正确地确定对象应该提供的服务。数据流图的每个处理框都与一个对象（或多个）的操作相对应。

（4）利用继承减少冗余操作

利用继承机制以减少每个类所需定义的服务数目，结合相关知识，尽量抽取出相似类的公共属性和操作，以建立其父类。如果继承层数更多，需要在不同的层次上正确定义各个服务。

# 第11章 面向对象设计

分析是建立问题域模型的过程，而设计是建立求解域模型的过程，即将需求转变为系统实现的方案。具体到OOP而言，OOD就是扩充OOA的模型。

生命周期方法学将设计分为总体设计和详细设计。OOP也可以分为系统设计和对象设计，概念上来说，系统设计确定实现系统的策略和目标系统的高层结构，对象设计确定解空间中的类，关联，接口形式以及实现服务的算法。但二者实际上界限很模糊。

## 11.1 面向对象设计的准则

（1）模块化  
对象就是模块

（2）抽象  
规格说明抽象：类封装了数据和操作，使用者无需知道操作符的算法和数据元素的具体表示  
参数化抽象：描述类的规格说明时，不具体指定要操作的数据类型，而是吧数据类型作为参数。如模板机制。

（3）信息隐藏

（4）弱耦合  
对象不可能是完全孤立的，尽管这样最好。当两个对象必须相互联系时，应该通过类的公共接口实现耦合，而不应该依赖于类的具体实现细节。  
对象之间的耦合种类：  
①交互耦合  
对象之间的耦合通过消息连接来实现，为了使耦合松散需要遵守以下准则：  
•尽量降低消息连接的复杂程度，减少消息中的参数个数  
•减少对象发送或接收的消息数  
②继承耦合  
继承耦合即派生类和基类之间的耦合，应该尽量提高这种耦合，即派生类尽量全部使用基类的属性和操作。

（5）强内聚  
OOD中存在3种内聚：  
①服务内聚：一个服务应该完成一个且仅一个功能  
②类内聚：一个类只有一个用途，其属性和服务都是该用途必须的  
③一般-特殊内聚：即继承关系应该符合领域知识，同时也要符合常识

（6）可重用  
①尽量使用已有的类（包括开发环境提供的类库）  
②创建设计新类时，对其接口考虑将来的可重用性

## 11.2 启发规则

（1）设计结果应该清晰易懂  
①用词一致：不仅名副其实，也要符合习惯  
②使用已有的协议：类库中已有的或其他人建立的  
③减少消息模式的数目：如果自行创建消息模式  
④避免模糊的定义：类的用途和它的名字一致，且用途单一

（2）一般-特殊结构的深度应适当  
一般一个中等规模（约100个类）的系统中，类等级层次数目应保持为7±2个。

（3）设计简单的类  
类的定义不超过一页纸或两屏，则使用它是比较容易记忆的。注意：  
①避免包含过多的属性  
②有明确的定义  
③尽量简化对象之间的合作关系  
④不要提供太多服务——一个类提供的公共服务不超过7个  
对于大型软件系统，如果类设计得简单，则必须有大量的类来组建系统。因此有必要划分主题来帮助建立清晰的结构

（4）使用简单的协议  
一般消息中的参数不要超过3个（也不是绝对的规定）。消息过于复杂，而且通过这种消息构建的关联是紧耦合的。

（5）使用简单的服务  
一般服务只有3-5行源程序语句，也比较容易用一个简单句子描述它的功能。如果一个服务过分复杂，可以考虑是否可以用继承派生的方式简化它。

（6）把设计变动减至最小  
即提高设计质量，避免日后过多的修改，提高可重用性。

## 11.3 软件重用

11.3.1 概述

（1）重用  
也叫再用或复用，软件重用可分为3个层次：  
①知识重用  
②方法和标准重用  
③软件成分重用

（2）软件成分的重用级别  
也可以分成3个级别：  
①代码重用  
•调用库中的模块  
•源代码剪贴：最原始问题最多  
•源代码包含：include稍好，但是修改原类后也要重新编译新程序  
•继承：没有什么问题，新程序的子类和类库分离，修改类库一般不需重新编译新程序  
②设计结果重用  
重用某个软件系统的设计模型（即求解域模型），适合把一个系统移植到不同的平台上  
③分析结果重用  
重用某个系统的分析模型，适合需求未改变，系统体系结构改变了的场合

（3）典型的可重用软件成分  
①项目计划  
软件项目计划的基本结构和许多内容都可以跨项目重用  
②成本估计  
③体系结构  
创建一组类属的体系结构模板，作为可重用的设计框架，也称作领域体系结构  
④需求模型和规格说明  
⑤设计  
⑥源代码  
⑦用户文档和技术文档  
⑧用户界面  
GUI软件经常被重用，也是最广泛重用的软件成分  
⑨数据  
包括内部表、列表和记录结构，以及文件和完整的数据库  
⑩测试用例  
设计或代码重用后，相关的测试用例也可重用

11.3.2 类构件

类构件即OOP中的类，是理想的可重用软构件。

（1）可重用软构件应具备的特点  
①模块独立性强  
②具有高度可塑性  
必须提供为适应特定需求而扩充或修改已有构件的机制  
③接口清晰、简明、可靠

（2）类构件的重用方式  
①实例重用  
最基本的实例重用方式即使用构造函数创建对象的正常使用方式。  
实例重用的另一种形式：用几个简单的对象作为成员创建一个更复杂的类。  
②继承重用  
即使用继承来对已有的类构件进行重用或修改。  
为了提高继承重用的效果，需要设计一个合理的、有一定深度的类构件继承层次结构。每个层次相对于前一层次只加入少量新属性与服务，这样的“渐变”扩充继承方式有利于以后有选择地在合适的层次上根据需要重用。  
③多态重用  
是建立在继承重用的基础上的，有了多态性的支持无需修改代码或消息语句形式就可以访问任何基类或派生类中的同名函数。  
为了使用多态性，设计类构件时，有一些种类的操作需要特别留意，它们和当前类的特点密切相关，可能影响重用性的操作：  
•与表示方法有关的操作。例如，不同实例的比较、显示、擦除等。  
•与数据结构、数据大小等有关的操作。  
•与外部设备有关的操作。例如，设备控制。  
•实现算法在将来可能会改进（或改变）的核心操作。  
因此这些操作需要和其他操作区别对待，因为势必要在派生类中改写，应作为适配接口。适配接口可分为转换接口和扩充接口。转换接口是每个类构件在重用时都必须重新根据自身特点来定义的服务的集合（C++中的纯虚函数）；扩充接口是每个类构件重用时可选择更改或直接继承的服务的集合（C++中的普通虚函数）。

11.3.3 软件重用的效益

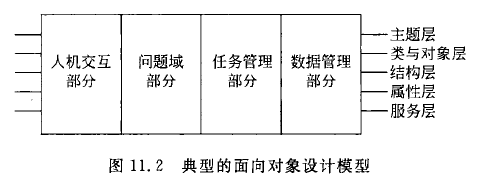
（1）质量  
重用的代码接受的测试与验证充分，错误率较低，多次重用会更低

（2）生产率  
效率的提高

（3）成本  
软件重用带来的净成本节省可以用下式估算：  
C=Cs-Cr-Cd  
Cs是项目从头开发无重用所需成本；Cr与重用相关成本；Cd交付给客户的软件实际成本  
与重用相关联的成本主要包括：  
•领域分析与建模的成本  
•设计领域体系结构的成本  
•为便于重用而增加的文档的成本  
•维护和完善可重用的软件成分的成本  
•为从外部获取构件所付出的版税和许可证费用  
•创建（或购买）及运行重用库的费用  
•对设计和实现可重用构件的人员的培训费用

## 11.4 系统分解

OOD模型，即求解域的对象模型也有5个层次，类似OOA。整个模型或系统可以分成4个主要部分，即4个子系统（按照功能划分）。具体项目可以根据情况自行进一部分解或整合。



（1）子系统之间的两种交互方式  
①客户-供应商关系（Client-supplier）  
客户子系统调用供应商子系统，后者无需了解前者接口。  
②平等伙伴关系（peer-to-peer）  
双方都可以调用彼此，关系更加复杂。  
尽量使用简单易理解的单向交互方式。

（2）组织系统的两种方案  
①水平层次组织  
每一层是一个子系统，上层基于下层，下层提供上层必要的服务（上下层类似客户供应商）。同一层内的对象彼此间相互独立，不同层的对象可能有关联。  
•封闭式：每层子系统仅使用其直接下层提供的服务，降低层次依赖性复杂程度，易改  
•开放式：某层子系统可以使用处于其下面任何一层子系统提供的服务，减少每层需要定义的服务数，更加高效，但不符合信息隐藏原则  
②垂直块状组织  
把软件系统垂直分成若干独立的，弱耦合的子系统，一个子系统相当于一块，提供一种类型服务。  
两种组织形式往往混合使用来组织和表示系统结构。

（3）设计系统的拓扑结构  
典型的拓扑结构有管道形、树形、星形等。应该采用简单的拓扑结构，减少子系统之间的交互数量。

## 11.5 设计问题域子系统

OOD过程中，可能对OOA得出的问题域模型做的补充与修改：

（1）调整需求  
一是用户需求或外部条件发生变化，二是分析模型有错误

（2）重用已有的类  
代码重用从设计阶段开始。若没有可重用的类而必须开发新类，则必须考虑未来重用性  
重用已有类的典型过程：  
①选择有可能被重用的已有类，标出这些候选类中对本问题无用的属性和服务，尽量重用那些能使无用的属性和服务降到最低程度的类  
②在被重用的已有类和问题域类之间添加泛化关系，即从被重用类中派生问题域类  
③标出问题域类中从已有类继承来的属性和服务  
④修改与问题域类相关的关联，必要时改为与被重用的已有类相关的关联

（3）把问题域类组合在一起  
OOD中经常通过引入一个根类把问题域类组合在一起（没有更先进的组合机制可用时）。

（4）增添一般化类以建立协议  
和上一个类似，使用额外的一般化抽象类抽象出多个类的公共协议，具体的服务内容仍可以在具体类中仔细定义。

（5）调整继承层次  
一般根据实际需要，以及编程语言对多重继承的支持性有几种情况：  
①使用多重继承机制p271/293  
为了避免属性和服务的命名冲突，可以使用阔菱形模式代替窄菱形模式的多重继承方式。  
②使用单继承机制  
把多重继承结构简化成单一的单继承层次结构，因此需要在某些具体类中重复定义某些属性和服务。

（6）ATM系统实例

## 11.6 设计人机交互子系统

详细设计指南见6.2节

OOD角度补充设计人机交互子系统的策略：

（1）分类用户  
设计者应该深入现场分析用户的习惯喜好，思考几个问题：  
•用户必须完成哪些工作？  
•设计者能够提供什么工具来支持这些工作的完成？  
•怎样使这些工具使用起来更方便？  
并对将来可能与系统交互的用户分类：  
•按技能水平分类（新手、初级、中级、高级）  
•按职务分类（总经理、经理、职员）  
•按所属集团分类（职员、顾客）

（2）描述用户  
记录有关用户的以下信息：  
•用户类型  
•使用系统欲达到的目的  
•特征（年龄，性别，受教育程度，限制因素等）  
•关键的成功因素（需求，爱好，习惯等）  
•技能水平  
•完成本职工作的脚本

（3）设计命令层次  
命令层次定义：实质上是用过程抽象机制组织起来的、可供选用的服务的表示形式。  
简而言之，用鼠标在常用windows界面的菜单栏中通过一系列移动、单击、子窗口选取等操作完成的一项命令下达即一个命令层次。  
①研究现有的人机交互含义和准则  
现有的普适GUI标准就是windows标准  
②确定初始的命令层次  
先从对服务的过程抽象着手，再进一步修改它们，以适合具体应用环境的需要  
③精化命令层次  
•次序：仔细选择服务的名称，并在命令层的每一部分内把服务排好次序。按照常用程度和用户习惯排序  
•整体-部分关系：寻找在服务中存在的整体-部分模式，有助于在命令层中组织服务  
•宽度和深度：命令层次的宽度和深度不应该过大，导致记忆复杂  
•操作步骤：应该用尽量少的单击、拖动和击键组合来表达命令，应该为高级用户提供简洁的操作方法

（4）设计人机交互类  
windows环境下的MFC提供了这些类的基类

## 11.7 设计任务管理子系统

（1）分析并发性

动态模型（状态图）是分析并发性的主要依据。若两个对象彼此间不存在交互（交互的对象肯定不是并发的），或者它们同时接受事件，则两个对象在本质上是并发的。

控制线：一条遍及状态图集合的路径（路径方向和次序并不存在意义，每一条路径也并不一定是涵盖某个系统中所有非并发对象的，每个路径中的对象有一些共同的特点决定了处理这个控制线的方式），在这条路径上每次只有一个对象是活动的。所有非并发的对象可以归并到一条控制线中。在计算机系统中用任务（task）实现控制线，任务也称进程（process）。多个任务的并发执行称为多任务。

并发行为即不同的任务必须同时发生的不同行为，可以在不同的处理器上实现，也可以在单个处理器上利用多任务操作系统仿真实现。

（2）设计任务管理子系统

设计的过程包括确定各类任务并把任务分配给适当的硬件或软件去执行。

①确定事件驱动型任务  
职责举例：通信工作，事件通常是表明某些数据到达的信号  
工作过程：睡眠状态，接收到中断并唤醒，接收数据并处理，通知有关对象，睡眠状态

②确定时钟驱动型任务  
职责举例：周期性地获得数据，周期性通信  
工作过程：设置唤醒时间，睡眠，接收中断唤醒，处理信息，通知有关对象，睡眠

③确定优先任务  
分为高优先级和低优先级任务，专门处理具有特定优先级的服务，这些服务被独立处理

④确定关键任务  
即关键的处理和服务被独立出来，由单独的任务完成以满足高可靠性要求

⑤确定协调任务  
当系统中存在3个以上任务时，就应该增加一个任务专门来协调工作

⑥尽量减少任务数  
便于理解和维护

⑦确定资源需求  
通过性能计算和比较来决定实现子系统或系统的方式（还有成本和未来可维护性等），有硬件和软件两种方式。专用硬件可以提高某种特定功能的性能，软件实现可能需要额外的编程实现与存在性能劣势。

## 11.8 设计数据管理子系统

数据管理子系统用于系统存储或检索对象，建立在DBMS之上，但是并不受其类型（关系型或OOP型）影响。

11.8.1 选择数据存储管理模式

（1）文件管理系统  
是操作系统的一个组成部分，使用它长期保存数据成本低而且简单，但是使用起来并不灵活而且存在操作系统之间的差异性（缺少统一标准）。

（2）关系数据库管理系统  
优点：  
•提供了各种基本的数据管理功能  
•为多种应用提供了一致接口  
•标准化SQL语言  
缺点：  
•运行开销大  
•不能满足高级应用的需求：支持的数据结构比较简单，操作也比较简单  
•与程序设计语言的连接不自然：SQL支持面向集合操作，是非过程性语言，但大多数高级语言都是过程性的，每次只能处理一个记录

（3）面向对象数据库管理系统  
有两种设计途径：  
①扩展的关系数据库管理系统  
增加了抽象数据类型和继承机制以及其他OOP相关服务  
②扩展的面向对象程序设计语言  
增加了和数据库交互的有效机制，支持“永久对象”方法（准确存储对象全部信息，而不是复制再重建的方式），也就便于共享对象

11.8.2 设计数据管理子系统

（1）设计数据格式

①文件系统  
•定义第一范式表：列出每个类的属性表，把属性表规范成第一范式  
•为每个第一范式表定义一个文件  
•测量性能和需要的存储容量  
•修改原设计的第一范式，以满足性能和存储需求  
必要时可以把泛化结构的属性压缩在单个文件中，或者使用某种编码值表示一些属性组合以减少空间需求，但是会增加数据处理时间

②关系数据库管理系统  
•定义第三范式表：列出列出每个类的属性表，把属性表规范成第三范式  
•为每个第三范式表定义一个数据库表  
•测量性能和需要的存储容量  
•修改原设计的第三范式，以满足性能和存储需求

③面向对象数据库管理系统  
扩展的关系数据库——使用和上一个相同的方法  
扩展的面向对象程序设计语言——不需要规范化属性的步骤

（2）设计相应的服务

如果某个类的对象需要存储起来，则在这个类中增加一个属性和服务，用于完成储存对象自身的工作（服务中具体如何存储该对象，是由服务自行定义的操作存储还是调用数据管理子系统是两种不同的选择）。这个属性和服务无需在OOP的模型中显式表示，仅仅在文档中记录即可。

①文件系统

被存储的对象需要知道打开哪些文件，怎样把文件定位到正确的记录上，怎样检索出旧值并且怎样更新它们

还应该定义个对象服务器类ObjectServer和其对象，以提供以下服务：  
•通知对象保存自身或保存需要长期存储的对象状态  
•检索已存储的对象（查找，读值，创建并初始化对象），以便把这些对象提供给其他子系统使用

②关系数据库管理系统

被存储的对象应该知道访问哪些数据库表，怎样访问需要的行，检索旧值和更新旧值。

同样需要一个对象服务器类和对象。

③面向对象数据库管理系统

扩展的关系数据库途径和②相同

扩展的面向对象程序设计语言途径：无需增加服务，这种数据库管理系统已经提供了存储对象的行为，只需给需要长期保存的对象加个标记即可

11.8.3 例子p280/302

## 11.9 设计类中的服务

11.9.1 确定类中应有的服务

确定操作的目标对象的规则：（服务要定义在目标对象所属的类中）  
①如果某个处理的功能是从输入流中抽取一个值，则该输入流就是目标对象  
②如果某个处理具有类型相同的输入流和输出流，而且输出流实质上是输入的另一种形式，则该输入输出流为目标对象  
③如果某个处理从多个输入流得出输出值，则该处理是输出类中定义的一个服务  
④若某个处理把对输入流处理的结果输出给数据存储或动作对象，则该数据存储或动作对象就是目标对象

若一个处理涉及多个对象，则要把服务定义在起主要作用的对象中，有两个经验规则：  
①若处理影响或修改了一个对象，则最好把该处理与处理的目标联系在一起  
②考察处理涉及的对象类及这些类之间的关联，从中找出处于中心地位的类。若整个关联结构呈星形，则中心类就是处理的归属

11.9.2 设计实现服务的方法

（1）设计实现服务的算法  
•算法复杂度应较低，效率应较高  
•容易理解和实现  
•易修改

（2）选择数据结构  
能够方便有效实现算法的物理数据结构，不是分析阶段中的系统逻辑结构

（3）定义内部类和内部操作  
添加的一些中间类来方便实现算法，以及定义一些低层操作来分解复杂的高层操作。

## 11.10 设计关联

（1）关联的遍历

使用关联的方式：单向遍历和双向遍历

使用原型开发时，原型中的所有关联都应该是双向的，便于增加新的行为以扩充原型

（2）实现单向关联

使用指针（重数一元）或指针集合（重数多元）实现单向关联

（3）实现双向关联

3种方法  
①只用属性实现一个方向的关联，当需要反向遍历时就执行一次正向查找。适用于两个方向遍历的频度相差很大时，而且需要尽量减少存储开销和修改开销  
②两个方向的关联都用属性实现，修改一个属性必然导致对应的一个属性必须修改。适用于访问次数远远多于修改次数的情形。  
③用独立的关联对象实现双向关联，必须为每个一对一关系创建一个关联对象，则一对N就要有N个。

（4）关联对象的实现  
对于一对一和一对多的关联关系，关联对象可以和参与关联的对象合并（1对1任意，1对N只能和N端对象合并），合并之后的形式类似于第一种实现双向关联的方法。对于多对多的情况，关联对象不能合并。

## 11.11 设计优化

11.11.1 确定优先级

优化之前需要站在全局角度正确确定系统各项质量指标的优先级，因为并不是系统的每个质量指标都是同等重要的，而且优化过程也很难保证所有质量指标都被优化。虽然这只是一个模糊定性的方案设计，但是仍然很重要。

11.11.2 提高效率的几项技术p284/306

（1）增加冗余关联以提高访问效率

即针对经常执行并且开销大、命中率低的查询或其他操作建立特殊的关联或索引。虽然增加了存储开销以及运行开销，但是从实际应用的总体而言提高了效率。

（2）调整查询次序

即改进算法的一种，通过调整查询次序，在多条件串连查询中先查询预期结果个数较少的条件，以提高整体查询效率。

（3）保留派生属性

通过某种运算而从其他数据派生出来的数据，是一种冗余数据。但是当运算表达式比较复杂时，可以把这类数据作为属性保存起来以避免重复计算。这种属性依赖于基本对象因此必须根据它更新。

11.11.3 调整继承关系

（1）抽象与具体

设计类继承时，很少使用自顶向下法，而是首先创建一些满足具体用途的类，然后对其进行归纳。归纳后的通用类可以再用来派生出需要的其他具体类。在进行了一些具体化的工作之后，可能要对某些派生出的有相似性的具体类再次归纳出抽象级别较低的抽象类，以形成整体的多层继承结构。即这也是一个循环精化的过程。

对现有类进行归纳抽象时注意：  
①不能违背领域知识和常识  
②应该确保现有类的协议接口不变，保持一致性

（2）为提高继承程度而修改类定义

常见情况下，各个现有类之间只是存在相似而并不存在完全相同的属性和操作（或很少）。这时需要修改现有类的定义，才能定义一个基类供其子类从中继承。

对于有时抽象出的基类暂时只有一个派生类的情况，无需置疑，因为只有抽象过程有明显的实际意义，以后就很有可能重用。

（3）利用委托实现操作共享

操作和行为的区别在于，操作可理解为对象的方法或服务，而行为是对象的全部操作的集合。只是为了共享一部分操作而继承一个行为并不相似的基类是没有意义的。因为继承会导致其他不需要的操作遗留下来，这会导致误操作和出错的可能。

委托即把一类对象或其指针作为另一类对象的属性，从而在两类对象之间建立组合关系。受托的对象是委托对象的实际本体以及操作的实际执行者，委托对象把这些操作封装起来，外界无法直接接触受托对象，因而不会误调用不需要的服务。

因此委托用于不相似的对象之间部分操作的共享。

# 第12章 面向对象实现

## 12.1 程序设计语言

12.1.1 面向对象语言的优点

（1）一致的表示方法  
即始终使用和支持类和对象及其相关的理论概念

（2）可重用性  
一致的表示方法增加了可重用的范围，从分析设计乃至实现结果

（3）可维护性  
问题域语义的贯彻表达使得即便脱离文档的帮助也能比较容易的读懂程序

12.1.2 面向对象语言的技术特点

有两大类面向对象语言，一类是纯面向对象语言，如Smalltalk和Eiffel等。另一类是混合型面向对象语言，即在过程语言的基础上增加OOP机制，如C++。

（1）支持类与对象概念的机制  
所有的OOP语言都允许用户动态创建对象，并可以使用指针引用它。相应地，管理内存的方法也有两种，一种是编程语言的运行机制自动管理内存与垃圾回收，另一种给予程序员编写释放内存的代码，如析构函数。

（2）实现整体-部分（即聚集）结构的机制  
两种实现方法：使用指针和独立的关联对象。

（3）实现一般-特殊（即泛化）结构的机制  
包括实现继承的机制，和解决多重继承（如果支持）中名字冲突的机制

（4）实现属性和服务的机制  
实现属性的机制应考虑：  
•支持实例连接的机制  
•属性的可见性控制  
•属性值的约束  
实现服务的机制应考虑：  
•支持消息连接  
•控制服务可见性的机制  
•动态联编（多态性）

（5）类型检查  
弱类型：编程语言仅要求每个变量或属性隶属于一个对象，如Smalltalk（无类型）  
强类型：语法规定每个变量或属性必须准确地属于某个特定的类，如C++，Eiffel  
强类型的优点：有利于编译时发现程序错误；增加了优化的可能性  
强类型语言适合开发可靠的软件产品，弱类型适合快速开发原型

（6）类库  
类库中一般包括实现通用数据结构的类（称为包容类），以及实现各种关联的类。也有的提供独立于具体设备的接口类，还有实现窗口系统的用户界面类等。

（7）效率  
面向对象语言并不存在效率低的缺点，只要拥有完整的类库。

（8）持久保存对象  
有些面向对象语言没有提供直接存储对象的机制，有的提供了。

（9）参数化类  
即模板机制

（10）开发环境  
基本的软件工具：  
•编辑程序  
•编译或解释程序（编译后的代码速度快效率高，适合开发正式产品；解释后的代码速度慢效率低，但适合调试）  
•浏览工具（类库编辑和浏览工具，联想功能等）  
•调试器debugger  
开发大型系统需要系统构造工具和变动控制工具

12.1.3 选择面向对象语言

（1）将来能否占主导地位

（2）可重用性

（3）类库和开发环境  
语言、开发环境和类库共同决定了可重用性

（4）其他因素p294/316

## 12.2 程序设计风格

良好的OOP程设风格包括传统的和适应OOP的新准则。

12.2.1 提高可重用性

（1）提高方法的内聚  
一个方法完成一个功能

（2）减小方法的规模  
代码长度不超过一页纸，否则应该分解

（3）保持方法的一致性  
功能相似的方法应该有一致的名字、参数特征（参数个数，类型和次序），返回类型，使用条件与出错条件等

（4）把策略与实现分开  
方法（服务）有两种类型：  
•策略方法：检查系统状态，做出决策，提供变元，管理全局资源，处理出错情况，不直接完成计算或实现复杂算法  
•实现方法：负责完成具体操作，不做出是否执行这个操作的决定，也不知道为何执行，出错后只返回执行状态不采取行动  
实现方法有可能在其他系统中重用，应该和策略方法分离，后者显然依赖于系统的功能

（5）全面覆盖  
即提高方法的鲁棒性，能够覆盖所有的输入组合与情况

（6）尽量不使用全局信息  
不使用全局信息是降低耦合度的一项主要措施

（7）利用继承机制

①调用子过程  
最简单的方法：将公共代码分离出成为一个公用方法，以被其他方法调用。公用方法一般定义在基类中以便继承。

②分解因子  
将类似但不同的方法（来自类似但不同的类）分解成相同的部分和不同的部分（共3部分）。相同的部分作为公用方法在基类中定义，不同的部分作为派生类中的方法分别定义（但名称相同）并被公用方法调用。具体的过程是在基类中定义公用方法以及定义要在派生类中实现的“虚方法”，公用方法调用虚方法但基类中不必实现虚方法；派生类分别实现不同的虚方法，根据多态性，基类公用方法在派生类中将调用被派生类实现的虚方法。

③使用委托  
继承不适用时（不存在一般-特殊关系）为了重用代码，可以使用委托，见前一章节

④把代码封装在类中p296/318

12.2.2 提高可扩充性

（1）封装实现策略

（2）不要用一个方法遍历多条关联链  
方法不能太复杂

（3）避免使用多分支语句（慎用）  
多分支语句如DO\_CASE可以用来调查对象的内部状态，然后针对性做出响应行为。但不要用其来检验对象类型，应该合理利用多态性。

（4）精心确定公有方法  
公有方法修改代价大，私有方法修改代价低

12.2.3 提高健壮性

健壮性即鲁棒性，是在硬件故障、输入数据无效或接近有效值边界、操作错误等情况下，系统能做出适当反应的能力。鲁棒性严格来说和效率之间存在一定冲突，毕竟需要增加程序逻辑来提高它。

可行办法：

（1）预防用户的操作错误  
检查所有用户输入，并做出恰当提示

（2）检查参数的合法性

（3）不要预先确定限制条件  
先使用动态内存分配机制创建未预先设定限制条件的数据结构。当确定了数据结构的容量限制等因素后再行优化。

（4）先测试后优化  
先测试性能再计划进行提高效率的优化。

## 12.3 测试策略

此节介绍除了传统测试的方法外的OOP测试新特点。

12.3.1 面向对象的单元测试

由于单元的概念改变了，即最小测试单元是类和对象，并且由于有继承和多态性等OOP特点，传统的对于某个操作的独立测试不再适用。应该把操作作为类的一部分来测试，无论这些操作是否同名。

12.3.2 面向对象的集成测试

由于不存在层次的控制结构，所有类和对象之间有着复杂的交互关系，传统的自顶向下或自底向上以及渐增式等集成方法都不适用。

OOP集成测试的策略，即如何逐渐添加类至被测试集合中：

（1）基于线程的测试（thread based testing）。把响应系统的一个输入或一个事件所需要的类集成起来进行测试。分别集成测试每个线程，必要时进行回归测试。

（2）基于使用的测试（use based testing）。首先测试几乎不使用服务器类的独立类，然后再测试使用独立类的下一层此的依赖类。然后逐渐添加依赖类的个数直到构造整个系统。

集群测试（cluster testing）：OOP集成测试的一个步骤，用特殊的测试用例检查一群相互协作的类，力图发现协作错误。

12.3.3 面向对象的确认测试

主要依据动态模型和描述系统行为的脚本来设计确认测试用例。目的和传统方法一致，都是针对用户可见和可识别的功能、动作和输出进行测试，以获得用户确认。

## 12.4 设计测试用例

和传统的方法（黑白盒）不同，OOP测试关注设计适当的操作序列以检查类的状态。

12.4.1 测试类的方法

这即是小型测试或单元测试，注重单个类中的方法的测试。

（1）随机测试

一个类的若干方法在实际情况的限制下一般也有多种调用序列的可能。首先可以先确认最小最短的方法调用序列（可能对应系统的某个功能或处理行为），然后列出可能的向序列中添加方法的情况（不必枚举），随机选取几种情况作为测试用例。

（2）划分测试（partition testing）

类似于传统的等价划分方法，把类的输入输出情况分类，然后设计用例针对每个类别测试。

划分类别的方法：

①基于状态的划分  
根据类操作改变类状态的能力来划分类操作，分成状态操作和非状态操作两类，分别测试这两种操作。（注意最小测试序列中的操作可能会改变对象状态，在这里可以忽略它们，只考虑其他的操作的分类）

②基于属性的划分  
根据类操作使用的属性来划分类操作，对于一个给定属性，可以把操作分成3类：  
•使用该属性的操作  
•修改该属性的操作  
•不使用也不修改该属性的操作  
然后为每个类别分别设计测试序列。

③基于功能的划分  
根据类操作完成的功能划分类操作，比如常见的分法：初始化操作、计算操作、查询操作、终止操作等。然后为每个类别分别设计用例。

（3）基于故障的测试（fault based testing）  
与传统的错误推测法类似，首先推测错误（依据经验），然后针对性的设计用例。

12.4.2 集成测试方法

集成测试必须考虑类之间协作的测试。方法类型上也类似单类测试有随机测试、划分测试、基于情景的测试和行为测试等。

（1）多类测试

使用下列步骤，得出多个协作类的随机测试用例：  
①对每个客户类，使用类操作符列表来生成一系列随机测试序列。这些操作符向服务器类实例发送消息（即基于单个类的操作生成一些随机测试序列）  
②对所生成的每个消息，确定协作类和在服务器对象中的对应操作符（即根据得到的单类测试序列找到每个方法中调用的其他类的方法，这些方法的类就是协作类或服务器类）  
③对服务器对象中的每个操作符（已经被来自客户对象的消息调用），确定传递的消息  
④对每个消息，确定下一层被调用的操作符，并把这些操作符结合进测试序列中（即重复前面的步骤把所有协作和参与的类的方法都找出来参与测试）

划分测试方法类似单类划分测试法，只是需要扩充测试序列以包括那些通过发送给协作类的消息而被调用的操作。多类划分测试也可以根据与特定类的接口来作为划分准则，比如A与B、C协作，可以按照AB和AC的方式划分。

（2）从动态模型导出测试用例  
依据类的状态图导出测试用例，测试用例的集合必须在合理限定（如果存在，即是系统的需求）条件下覆盖所有该类的可能状态（节点）以及所有可能的状态转换（边）。目的是测试类的所有行为。  
对于有其他类协作的行为，加入其他类中的方法至测试序列即可。  
显然动态模型只是一个测试方案制定的一种指导，并不代表只按照这个方法就能得出所有应该尝试的测试用例。

# 第13章 软件项目管理

软件项目管理第一步是项目计划活动，而其基础就是估算工作量和完成期限。

## 13.1 估算软件规模

13.1.1 代码行技术

这种方法依赖于以往开发类似产品的经验和历史数据，来进行估计实现功能和系统所需的源程序行数。

最好由多名经验丰富的软件工程师分别估计出程序的最小规模a，最大规模b和最可能的规模m，并计算3种规模的均值。再使用下式计算程序规模的估计值

当程序较小时，软件规模常用单位是代码行数LOC，程序较大时，使用千行代码数KLOC

优点：容易计算，普适（所有软件都有代码）

缺点：仅仅考虑了代码或源程序的工作量，不同语言实现同一个系统规模不同，不适用于非过程语言

13.1.2 功能点技术

使用功能点FP为单位度量软件规模

（1）信息域特性

信息域的5个特性：  
①输入项数Inp：用户向软件输入的项数，给软件提供面向应用的数据，注意查询单独计数，不计入输入项中  
②输出项数Out：软件向用户输出的项数，向用户提供面向应用的信息，注意报表内的数据项不单独计数  
③查询数Inq：查询是一次联机输入，导致软件以联机输出方式产生某种即时响应  
④主文件数Maf：逻辑主文件数目，即数据的一个逻辑组合，可能是大型数据库的一部分或一个独立的文件  
⑤外部接口数Inf：机器的可读的全部接口的数量

（2）估算功能点的步骤

①计算未调整的功能点数UFP  
把产品信息域的每个特性都给予一个单独的合理的评级（简单，平均，复杂）。对照图表（p307/329）给每个特性一个对应其评级的功能点数。使用下式计算UFP，ai是信息域特性系数，即对应特性评级的功能点数。

②计算计数复杂性因子TCF  
这一步骤度量14种技术因素（图表p307/329）对软件规模的影响程度。Fi代表这些因素，其取值[0,5]，取值越大表示影响程度越大。  
使用下式计算技术因素对软件规模的综合影响程度DI和TCF

DI取值[0,70]，TCF取值[0.65,1.35]。

③计算功能点数FP

优点：考虑全面，与具体的编程语言无关

缺点：主观因素影响太大

## 13.2 工作量估算

工作量是软件规模（KLOC或FP）的函数，其单位是人月pm。大多数估算模型是从有限个相似项目样本集中总结的，没有任何一个估算模型适用于所有类型的软件与开发环境

13.2.1 静态单变量模型

（1）面向KLOC的估算模型  
4种模型和估算公式p308/330

（2）面向FP的估算模型  
2种模型和估算公式p308/330

这类模型的常数是根据应用领域具体情况的经验数值，有必要根据实际情况调整。

13.2.2 动态多变量模型

也称软件方程式，由4000+当代软件项目中收集的生产率数据推导而来。

其估算模型形式为

其中E是以人月或人年为单位的工作量；t是以月或年为单位的项目持续时间；B是特殊技术因子（详情见p309/331），P是生产率参数，反映了一些工作条件对工作量的影响（详情见p309/331）

该模型说明，开发一个确定的软件，即LOC固定，若延长项目持续时间，可降低完成项目所需总工作量。

13.2.3 COCOMO2模型

COCOMO是构造性成本模型constructive cost model的缩写，COCOMO2是原始的该模型的修订版。它给出了3个层次的软件开发工作量估算模型，3个模型对软件细节的考虑逐级增加，可以在项目不同开发阶段有针对性的使用。

（1）应用系统组成模型  
主要用于估算构建原型的工作量

（2）早期设计模型  
适用于体系结构设计阶段

（3）后体系结构模型  
适用于完成体系结构设计之后的软件开发阶段。  
软件开发工作量E的公式为

其中E以人月为单位，a是模型系数，b是模型指数，fi是成本因素对工作量的影响，称为工作量系数（它类似功能点技术中的ai，表格见p310/332）  
a的典型值为3.0，这是一个经验数值，应根据历史经验确定  
b的计算公式为

Wi（i为[1,5]）为5个分级因素的评分，评分从5（甚低）到0（特高）共6级。分级因素如下：  
①项目先例性：对于开发组织来说该项目的新奇程度，可能和经验有关也可能是创新  
②开发灵活性：为了实现预先确定的外部接口需求和为了及早开发出产品需要增加的工作量  
③风险排除度：反映重大风险已被消除的比例，和指定了重要模块接口的比例密切相关  
④项目组凝聚力：和出身文化背景差异有关，也和组队经验有关。  
⑤过程成熟度：反映按照能力成熟度模型度量出的项目组织的过程成熟度。

## 13.3 进度计划

进度安排的过程是先把估算的工作量分配给特定的软件工程任务，并初步规定各项任务的起止日期，从而将估算出的项目工作量分布于计划好的项目持续期内。随着项目进行，初期的宏观进度安排的每个条目被精化成一个详细进度表。

13.3.1 估算开发时间

由估算工作量E反过来计算估计开发工作时间T（正常开发时间）的模型：  
4种，见p312/334

T并非与E存在和人数相关的反比关系，即人越多干得越快。存在以下两大原因：  
①小组规模变大，消耗于组员之间的通信时间与开销变大，个人生产率下降  
②开发过程中若增加小组成员，则一段时间内不仅不会增加人力数，反而会降低原小组成员的工效

研究项目组规模与项目组总生产率的关系p313/335，结论是存在一个最佳的项目组规模Popt使得项目组总生产率最高。

Brooks规律：向一个已经延期的项目增加人力，只会使得它更加延期

Boehm经验：软件项目的开发时间最多可以减少到正常开发时间的75%

13.3.2 Gantt图

能很形象的描绘任务分解情况，以及每个子任务（作业）的开始和结束时间。

缺点：  
①不能显示地描绘各项作业彼此间的依赖关系  
②进度计划的关键部分不明确，难于判定哪些部分应该是主要的部分  
③计划中有潜力的部分和潜力大小不明确，往往造成潜力的浪费

13.3.3 工程网络

制定进度计划时另一种常用的图形工具，比Gantt图更加高级，包含其优势解决其劣势。但工程网络图作为一个分析工具确实较强，但也较复杂，可以用于专业人员进行分析；Gantt图简单直观，使用广泛，适用于和非管理专业人士交流讨论。

工程网络图中的基本图形元素和含义：  
•带箭头的边——作业：即一项任务，是消耗时间和资源的事务。一般用两个事件编号表示一个作业，如1-2  
•带有编号的圆圈——事件：即一项作业开始或结束。这里的事件仅仅表示一个明确的时间点，不是一个持续一段时间并消耗资源的事务。  
•虚线箭头边——虚拟作业：实际上是一种依赖关系，并不是耗时耗资源的实际作业。被指向的事件需要依赖于箭头发源的事件才能进行某项作业。但在模型中虚拟作业和实际作业的处理方法相同。

例子p316/338，下一节讲扩展工程网络图

13.3.4 估算工程进度

扩展的工程网络图及其图形元素和含义p316/338：  
•编号圆圈右上角数值——最早时刻EET：完成相关作业达到该事件（时间点）的最早时刻，起始事件从0开始  
•编号圆圈右下角数值——最迟时刻LET：完成相关作业达到该事件所允许的最迟时刻，前提是没有任何作业拖延最终事件的时刻  
•箭头边上方的数值——持续时间：即作业预计需要的时间  
•加粗箭头边组成的路径——关键路径  
•箭头边下方的括号中的数值——机动时间

计算EET的规则：  
①所有的事件从起始事件开始正向计算  
②考虑进入当前事件的所有作业  
③对于每个作业都计算它的持续时间与其起始事件的EET之和  
④选取所有和中最大值作为当前事件的EET

计算LET的规则：  
①所有的事件从结束事件开始反向计算  
②结束事件的EET和LET必然相同  
③考虑离开当前事件的所有作业  
④从每个作业的结束事件的LET减去该作业的持续时间  
⑤选取所有差中最小值作为该事件的LET

13.3.5 关键路径

关键路径：一个工程网络图中一系列EET和LET相同的事件组成的贯穿项目始终的路径，同时路径上的所有作业的机动时间为0。显然顾名思义，关键路径上的事件必须准时发生，路径上的作业也是一系列分支任务中最占用时间的部分，因此决定了最终项目的时间。

关键路径指示项目中最重要的一些任务（从决定工期的角度讲），如果想要缩短工期，则指示了需要采取措施的任务。

13.3.6 机动时间

定义：一个非关键路径上的作业所允许的灵活使用的时间，即允许超出其实际标准所需时间的时间。

计算方法：作业的可用全部机动时间等于它的结束事件的最迟时刻减去它的开始事件的最早时刻，再减去本作业的持续时间。

注意：机动时间在实际情况下并非是全部独立的，即如果一个作业真的耗费掉了它全部的机动时间（当然也包括它自己的持续时间），那么和它相关的某些作业将不再具有机动时间，尽管这些作业在第一个作业没有耗费那么多机动时间的情况下也曾具有机动时间。

理论上分析可知，处于工程网络中的同一支线路径上的所有作业共享同一个机动时间。支线即非关键路径的路径，起始于关键路径上的某事件，终止于关键路径上的另一事件（必然晚于前一事件），中间不具有关键路径的事件。可以理解为，支线即主线任务的分支任务，主线任务每个时间点是确定的，在某两个确定时间点之间的支线任务必然共享机动时间。

## 13.4 人员组织

软件项目组的组织方式很多，根据项目特点，组织经验甚至个人喜好，都有可能决定使用哪一种组织形式。常见的有以下3种。

13.4.1 民主制程序员组

特点：小组成员完全平等，通过协商做出技术决策

优点：  
①人数少，减少通信成本损耗  
②容易确定小组的质量标准，更容易被所有人遵守  
③组员间关系密切，能够互相学习，学术空气浓厚  
④组员对于发现程序错误持积极的态度，有助于更快速提高代码质量

适宜和不适宜性：  
•当开发的软件和遇到的问题技术难度较高时适用  
•当组内多数成员是经验丰富技术熟练的程序员时适用（多数成员技术不高时不适用）

注意事项：  
小组人数应该比较少，2-8人为宜

13.4.2 主程序员组（传统的组织结构）

特点：一个技术熟练管理能力强的人做主程序员，领导负责协调组内全部工作。  
•专业化。该组每名成员仅完成他们受过专业训练的那些工作  
•层次性。主程序员指挥每名组员工作，并对项目全面负责。

分工：  
①主程序员——  
体系结构设计和关键或复杂部分的详细设计，指导其他程序员完成详细设计和编码，处理所有程序员之间的接口问题，对所有成员的工作成果复查和质量负责  
②后备程序员——必要时能够接替主程序员工作  
设计测试方案，分析测试结果及独立于设计过程的其他工作  
③编程秘书——  
项目有关的全部事务性工作，维护资料库和项目文档，编译、链接、执行源程序和测试用例

适宜和不适宜性：  
•分工情况十分不合理，主程序员少有人达到标准，后备程序员和编程秘书的职责难以让人积极接受。

13.4.3 现代程序员组

特点：由传统主程序员组改进而来，分割了传统主程序员的职责与权力

分工：  
①技术组长——  
负责小组技术活动，全部代码审查工作，项目质量负责  
②行政组长——  
所有非技术性事务的管理决策，对程序员的业绩评价（于常规调度会议上了解），不能参加代码审查工作（以防程序员不愿发现代码错误）  
③取消编程秘书工作——技术进步和设备进步导致  
被分解为每个程序员的职责

注意事项：  
①对于技术组长和行政组长职责划分与纠纷由更高层管理人员负责  
②当项目规模增大时，可以适当增加管理结构的中间层次。比如将程序员分成若干小组，每个小组有一个技术组长，每个技术组长向项目经理汇报（相对应的，也有一套类似的非技术管理组织结构）。  
③分散决策：即采用民主方式决定和解决问题，但是仍然存在管理关系和集中指导。在主程序员组遇到技术难关需要集体智慧攻关时有效。

## 13.5 质量保证

13.5.1 软件质量

定义：软件与明确地叙述的功能和性能需求、文档中明确描述的开发标准以及任何专业开发的软件产品都应该具有的隐含特征相一致的程度。（总共3个要点）

影响软件质量的主要因素（分3类）：  
①产品修改：可理解性、可维修性、灵活性、可测试性  
②产品转移：可移植性、可再用性、互运行性  
③产品运行：正确性、健壮性、效率、完整性、可用性、风险  
具体因素定义见p325/347

13.5.2 软件质量保证措施

软件质量保证software quality assurance，SQA；主要措施有：  
①基于非执行的测试（复审或评审）  
②基于执行的测试（即一般的软件测试）  
③程序正确性证明

参加SQA的工作人员分工：  
•软件工程师：正式的技术复审和计划软件测试方案  
•SQA小组：计划，监督，记录，分析和报告

（1）技术复审的必要性

技术复审把控编码之前产生的错误，大部分是规格说明错误或设计错误，大大降低后续开发成本和后续修改的成本。包括两种形式，走查和审查。

（2）走查walkthrough

走查组由4-6名成员组成，分工如下：  
•1位被审查的小组的成员（被审查部分的工作的参与者）  
•1位被审查的小组的管理员  
•1位客户代表  
•1位下阶段开发组的代表  
•1位SQA小组的代表（组长）

工作流程：  
①将被走查部分工作的相关材料预先分发给每位成员  
②成员仔细研究材料列出2张表：一张是不理解的术语，另一张是认为不正确的术语  
③小组共同走查文档，只求发现错误，不求更正，时间最长不要超过2h

走查方式：  
•参与者驱动法——参与者根据列表，提出问题，被走查部分工作的负责人员来解释问题或承认问题  
•文档驱动法——被走查的文档编写者向其他成员解释文档相关部分的内容，其他成员在此期间根据自己的列表或临时发现的问题提问

注意事项：走查组成员最好是经验丰富的高级技术人员

（3）审查inspection

分工：4人  
•组长——管理人员和技术负责人  
•当前阶段项目组代表  
•下阶段开发工作的项目组代表  
•SQA小组代表

工作流程：  
①综述——由文档编写者成员之一向审查组综述文档，之后分发文档  
②准备——审查组仔细阅读文档，列出发现的错误类型，并按发生频率把错误类型分级，以辅助审查  
③审查——审查组审查整个文档，目的在于发现错误而不是改正。每次审查会不超过90分钟。审查组组长应该在一天之内写出一份关于审查的报告  
④返工——文档作者负责解决在审查报告中列出的所有错误及问题  
⑤跟踪——组长必须确保所有问题都得到圆满解决。如果审查过程中返工量超过5%，则应该再对文档全面审查一遍

特点：正规的审查过程会仔细划分错误类型，并将其应用在后续阶段的审查和未来项目的审查中

（4）程序正确性证明

基本思想：证明程序能完成预定的功能。正确性证明和软件测试不冲突，即便是“正确的”软件也可能有错误，可能是设计或实现上的。

基本原理：若在程序的若干个点上，提出关于程序变量及它们的关系的断言，那么在每一点上的断言都应该永远是真的（这是对于理想中正确的程序而言的）。其中按照程序流程，第一个断言必须是关于程序输入的断言，最后一个断言必须是关于程序输出的断言。如果能够证明每两个点之间的程序确实能够将前一个点的断言转变成后一个点的断言，并且输入断言和输出断言都为真，再加上程序不包含死循环，就可以证明程序正确性。

大型程序必须使用自动系统辅助证明程序正确性，但是这些系统仍处于开发阶段。

## 13.6 软件配置管理

软件配置管理是软件整个生命期内管理变化的一组活动，包括：  
①标识变化  
②控制变化  
③确保适当地实现了变化  
④向需要知道这类信息的人报告变化

软件配置管理不同于软件维护，它贯穿整个项目始终，直到软件废弃。它的目标是使软件变化更正确且更容易被适应，在必须变化时减少所需花费的工作量。

13.6.1 软件配置

（1）软件配置项

软件配置的内容，或软件过程的全部输出信息，可分为主要的3类：  
计算机程序、描述文档、数据、（软件工具和开发环境）

软件配置管理即专门用于管理变化的软件质量保证活动，保证每个软件配置项彼此完全一致。

（2）基线

定义：已经通过了正式复审的规格说明或中间产品，可以作为进一步开发的基础，只有通过正式的变化控制过程才能改变它。简而言之就是通过了正式复审的软件配置项。

建立了基线之后，必须应用特定的正式的过程来评估、实现和验证每个变化，称为规程。

理想的基线（软件配置）应该包括软件工具和开发环境，因为这些事物的变化（版本更新等）很可能会导致工程结果不同。

13.6.2 软件配置管理过程

软件配置管理主要有5项任务：

（1）标识软件配置中的对象

软件配置对象分为两类：  
•基本对象——分析、设计、编码或测试过程中创建出来的“文本单元”，如一个模块的源程序清单、一组测试用例  
•聚集对象——基本对象和其他聚集对象的集合

每个配置项对象的标识特征：名字、描述、资源表和“实现”。名字必须是无二义性的字符串，而且在整个软件生命周期内都能标识该对象（尽管其可能发生变化）。

（2）版本控制

版本控制过程联合使用规程和工具，以管理在软件过程中所创建的配置对象的不同版本。

版本控制的具体应用：为每个配置对象的每个版本赋予一个“属性”，然后使用者通过描述一组属性来指定和构造所需要的配置。这里的属性可以单纯的就是配置对象的版本号，也可以是复杂的布尔变量串——标明当前版本是否具备哪些特点或功能。

（3）变化控制

典型流程：（在配置项成为基线之后的过程）  
①接到变化请求  
②评估变化在技术方面的得失、可能的副作用、对其他配置对象和系统功能的整体影响，估算修改成本  
③将评估结果形成“变化报告”供“变化控制审批者”审阅  
④对被批准的变化生成一个“工程变化命令”，描述变化本身、约束、复审和审计标准  
⑤将要修改的对象从项目数据库中提取出来，进行修改并应用适当的SQA活动  
⑥把修改后的对象提交进数据库，并用适当的版本控制机制创建该软件的下一个版本

从数据库中提取和提交配置对象涉及变化控制的两个主要功能——访问控制和同步控制。访问控制决定软件工程师访问和修改特定配置对象的权限，同步控制保证多名软件工程师进行的修改可以正确地并发处理。

配置项称为基线之前，仅需应用非正式的变化控制。即在不影响变化申请者工作范围之外的系统配置的情况下，变化申请者可以任意进行合理修改。

（4）配置审计

包含两部分内容：技术复审和软件配置审计，二者互为补充。

技术复审的内容见前部分（13.5.2节等）

软件配置审计评估配置对象特有的一些需要考虑的问题（不在一般复审考量中）：  
•修改遵循了软件工程标准  
•配置项中显著标明了所做的修改  
•注明了修改日期和修改者  
•适当更新了所有相关的软件配置项  
•遵循了标注变化、记录变化和报告变化的规程

（5）状态报告

状态报告的目的不仅仅是给管理者审阅，而是保证所有相关工作人员知晓目标系统的所有变化，保持信息沟通及时顺畅。

报告的内容：  
•发生了什么事  
•谁做的这件事  
•这件事的发生时间  
•将影响哪些其他事物

## 13.7 能力成熟度模型

能力成熟度模型capability maturity model，CMM，目的用于评价软件机构的软件过程能力高低的模型。既可以用于评价一个软件机构的业务能力，也可以用于一个软件机构的内部改进。

能力成熟度模型重在提升软件过程的管理方面的因素，因为技术因素并不是大多数软件项目失败的原因，管理混乱制约了技术的发挥。

CMM将软件过程的有序程度进化过程分成5个等级，5个等级逐级提高并提供了软件过程改进工作重心的优先次序。即对每个等级而言，下一个等级都指示了其理想的下一步改进工作的方向和重点。

（1）初始级p332/354

（2）可重复级

（3）已定义级

（4）已管理级

（5）优化级