第一章　概述

软件的定义

Software is a set of instructions, data or programs used to operate computers and execute specific tasks.

软件是按照特定顺序组织的计算机数据和指令的集合。

软件工程与计算机科学的对比

Computer science: focusing on computer hardware, compilers, operating systems, and programming languages.

Software engineering: a discipline（学科） that uses computer and software technologies as a problem-solving tool.

Software engineer means the application of a systematic, measurable and disciplined approach to the development, operation, and maintenance of software. That is, the application of engineer to software.

软件工程是用计算机或计算机技术来求解实际问题的研究或实践。计算机科学家研究计算机的结构及其理论，以及计算机的功能，而软件工程是计算机科学的一部分，因为软件工程师应用计算机科学研究的结果来构建工具和技术以满足客户的需求。

软件的好坏标准

软件产品的质量（The quality of the product）

衡量标准：McCall’s quality model

软件开发过程的质量（The quality of the process）

衡量标准：Capability Maturity Model（CMM）

ISO9000

Software Process Improvement and Capability Determination(SPICE)

商业环境的质量（The quality of the product in the context of the business environment）

衡量标准：Return on Investment（ROI）投资回报率

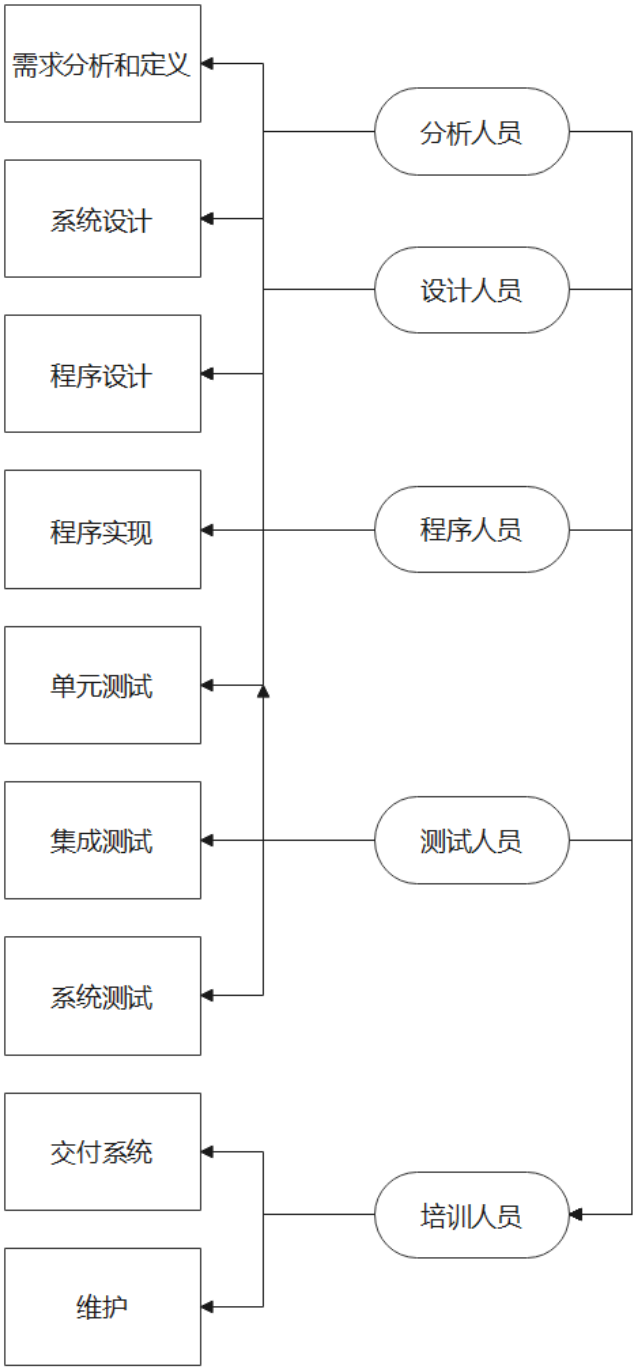
软件的使用

Customer: the company, organization, or person who pays for the software system

Developer: the company, organization, or person who is building the software system

User: the person or people who will actually use the system

软件团队的人员


需求分析和定义
系统设计
程序设计
程序实现
单元测试
集成测试
系统测试
交付系统
维护
分析人员
设计人员
程序人员
测试人员
培训人员

第二章 过程和生命期

过程的定义

A series of steps involving activities, constraints, and resources that produce an intended output of some kind

In many software projects, we should capture the development process as a collection process models, rather than focusing on a single one.

对开发流程建模的原因（Reasons for Modeling a Process）

为了形成共识，进而将不同的软件过程组合为模块

为了寻找不同模块直接矛盾、冗余、被省略的部分

寻找和评估达到过程目标的适当活动

将开发过程中的特定情况建模为一般过程

生命期的定义

When the process involves the building of some product, we sometimes refer to the process as a life cycle

瀑布模型（Waterfall model）

是一个项目开发的抽象构架，从系统需求分析到产品发布维护，设计一系列阶段顺序展开，每个阶段都会产生循环反馈，从而将软件生存周期的各项活动规定按照固定顺序连接成若干阶段的工作，形如瀑布流水。

优点：

- 模型简单，使客户容易理解
- 为项目提供了按照阶段划分的检查点
- 精确定义了每位开发人员的不同职能
- 可强迫开发人员采用规范化的方法
- 说明了每个主要阶段完成的里程碑或可交付产品

缺点：

- 由于瀑布模型几乎完全依赖于书面的规格说明，很可能导致最终开发出的软件产品不能真正满足用户的需要
- 只适用于项目开始时需求就已经确定的情况，无法应对开发过程中需求的变更
- 将软件开发视为一个制造过程而不是一个问题求解的过程

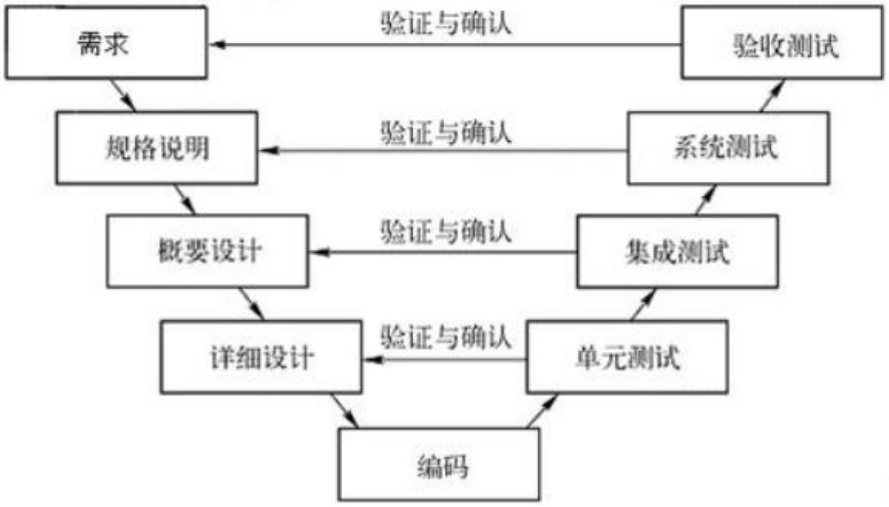
适用情况：

- 开发已经有明确规定的确定项目
- 需求一旦确定便不再发生变化
- 组织具有明确分工的项目
- 重新开发一个与之前相同的项目，即重复开发

V 模型（V model）

优点：

- 使测试与需求相对。采用参与测试来验证程序设计，采用集成测试来验证体系结构，采用验收测试来确认需求
- 验证过程中发现问题即可在执行后续测试步骤前重新执行左边步骤对软件修正
- 瀑布的焦点通常是文档和工件，而 V 模型的焦点是活动性和正确性



原型模型（Prototyping model）

在需求分析阶段对软件进行初步的分析和定义，快速开发出简单的软件原型并向用户展示，用户对软件原型进行测评后进一步提出进要求，开发人员的原型修改到用户满意为止。因此原型模型的开发更能符合需求，适合市场导向（market-driven）的产品

优点：

- 很好地满足客户的需求
- 减少开发风险与不确定性

阶段性开发：增量与迭代（Phased development: increments and iterations）

增量：将系统功能划分为诸多子系统，先开发具有一个小功能的子系统，之后不断在子系统上增加新的功能来逐渐满足所有需求

迭代：第一次提交的产品就是一个完整的系统，之后在该系统的基础上不断修改，不断开发出新的版本，但是每个版本都是完整可运行的版本。

优点：

- 逐步增加产品功能使用户有充足的时间学习和适应新产品
- 一部分一部分的开发，能在较短时间内就向用户提供一个有用功能的产品
- 项目的风险较低，能较好满足客户需求

螺旋模型（Spiral Model）

瀑布模型与原型模型的组合，既可以周期性地验证客户的需求，有可以分阶段进行开发工作，能保证软件在遇到较大风险时停止，大大增强了软件的风险防控能力。但是过程很复杂，需要开发人员有丰富的风险评估能力和专业知识，适合大型项目的开发。

优点：

- 将开发活动与风险管理相结合以最小化控制风险
- 较好地满足用户的需求

缺点：

- 开发成本高，对开发人员要求高

敏捷开发（Agile Process）

以用户为核心，在软件构建初期将其切分成多个子项目，每个项目分别开发、测试。更注重软件的开发而不是写文档；更注重与用户交流，了解用户的各种需求；擅长应对各种变化，没有计划的约束。

极限编程（Extreme Programming）

是敏捷开发的一种形式，基本特点与敏捷开发相同，适合小团队性的开发。

第三章 项目管理

什么是项目?

A project is a temporary endeavor undertaken to create a unique product, service, or result.

项目是完成某些特定指标的一次性人物，具有唯一性、一次性、整体性、多目标性等特点。同时软件项目与其他项目产品相比，有着不可见性、复杂性、一致性、灵活性等特点。

什么是项目管理?

Project Management is the process of using proven tools and techniques to manage the scope, time and cost of a project.

The application of knowledge, skills, tools, and techniques to project activities in order to meet or exceed stakeholder needs and expectations.

Activity：takes place over a period of time

Milestone：An objectively identifiable point in a project（项目中客观可识别的点）

项目可交付部分

文档

展示软件功能

展示软件的子系统

展示软件的精确度

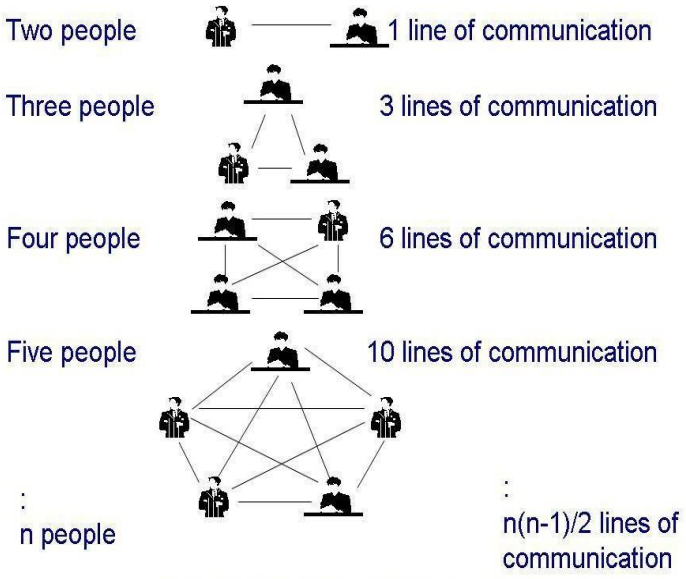
展示软件的可靠性、性能、安全性

工作分解结构（Work Breakdown Structure WBS）

就是把一个项目，按一定的原则分解，项目分解成任务，任务再分解成一项项工作，再把一项项工作分配到每个人的日常活动中，直到分解不下去为止。即：项目→任务→工作→日常活动。

成员交流

如果一个工作中有 n 个人，那么一共有 n（n-1）/ 2 条交流线



工作风格

Extrovert（外向）: tell their thoughts

Introvert（内向）: ask for suggestions

Intuitive（直觉）: base decisions on feelings

Rational（理智）: base decisions on facts, options

项目组织

根据项目人员的性格、团队管理的方式、人员的数量等，可将团队分为高结构化团队和低结构化团队。

Chief programmer team（高结构化形式）

项目中所有人必须与领导直接交流，而成员之间可以没有交流

适合工作类型：①高确定性工作②重复性工作③大型项目

Egoless approach（低结构化形式）

没有直接的统一领导，成员之间相互交流

适合工作类型：①不确定性的工作②采用新技术，新编程语言的开发工作③小型项目

风险管理

Risk management is concerned with identifying risks and drawing up plans to minimize their effect on a project

风险管理关注识别风险并制定计划以尽量减少其对项目的影响

Risk impact（风险影响）：与事件相关的损失

Risk probability（风险概率）：风险发生的可能性

Risk control（风险控制）：我们可以控制的程度

风险评估

包括以下活动：

风险识别（Risk Identification）

风险分析（Risk Analysis）

风险排序（Risk Prioritization）

关键路径（Critical path）

从开始到结束得所有路径中，所用时间最长的为关键路径（有可能有多条相同的）。

最早开始时间：

在关键路径上，从开始到该任务的最早执行的时间

第一个结点（开始结点）的最早执行的时间是 1，不是 0.

其他结点的最早开始时间为：[从开始结点到 Activity 前者结点经过的最长的时间总长]+1

最晚开始时间：

关键路径的总时间-反向得出该任务的时间

关键路径上的 Activity 的最晚开始时间是同最早开始时间。

其他结点的最晚开始时间为：关键路线的总时长 - [从结束结点到 Activity 前者结点经过的最长的时间总长]+1

松弛时间（Slack time）：

最晚开始时间-最早开始时间

关键路径上活动的 Slack time 为 0.

Precursors 前驱结点：

是 Activity（前者-后者）中前者的前驱结点。

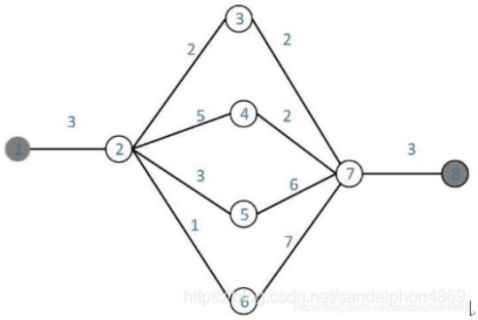
例题：

2-3 的最早开始时间为 3+1=4，最晚开始时间为=15-(2+2+3)+1=9

2-4 的最早开始时间为 3+1=4，最晚开始时间为=15-(5+2+3)+1=6

3-7 的最早开始时间为 2+3+1=6，最晚开始时间为=15-(2+3)+1=11

5-7 的最早开始时间为 3+3+1=7，最晚开始时间为=15-(3+6)+1=7



Activity	Earliest start time	Latest start time	Slack	Precursors
1-2	1	1	0	-
2-3	4	9	5	1-2
2-4	4	6	2	1-2
2-5	4	4	0	1-2
2-6	4	5	1	1-2
3-7	6	11	5	2-3
4-7	9	11	2	2-4
5-7	7	7	0	2-5
6-7	5	6	1	2-6
7-8	13	13	0	3-7、4-7、5-7、6-7

The critical path(s): 1→2→5→7→8 (15)

注意关键路径最后算出来的时间是所有边上的时间的和，不包括最开始的加一

从关键路径向后推最晚开始时间时，要记得加一

这样做的原因是贴近实际，实际生活中最早只能从一号开始工作，而不能从零号开始

第四章 需求分析

什么是需求?

A requirement is an expression of desired behavior

Requirements focus on the customer needs, not on the solution or implementation

需求侧重于软件的功能和行为，根据需求我们可以构建软件的功能行为模型

需求定义（Requirements Definition）

一份关于软件功能的完整清单，是给用户看到，用户希望实现的功能。what the customer wants

需求规范（Requirements Specification）

将需求整理为规范性的文档。what the designers need to know

用户、设计者、测试人员是需求文档的直接使用者，而程序员不是。

需求的种类

功能需求（Functional requirement）

根据所需活动描述所需行为，指的是软件必须要实现的功能，描述开发人员要做什么

质量需求（Quality requirement）

描述软件必须所具备的特性，一般是确定的测试目标，例如性能、精度、反应速度、兼容性、安全性。通俗的说，不属于具体功能的目标都是质量需求

设计约束（Design constraint）

对设计决策的描述，例如平台或者接口组件的选择

工艺约束（Process constraint）

对用于构建系统过程中，可利用的技术和资源的限制

需求的可测试性（testable）

The requirements are testable if they could clearly demonstrate whether the eventual product meets the requirement.

软件最终需要是可测试的，即我们需要通过测试证明软件是否满足了用户的需要。为了帮助需求可测试化，有以下方法：

需求的描述应该的准确的、定量的，而不是模糊的

例如：The system handles files of up to 2MB in size（good requirements）

The system should be constructed so that it will be easy to add new functionality in the future（unverifiable）

The shell should be easy to use（bad requirement）

在需求文档中用专有名词替换代词、确保需求中每个名词都有准确的定义

需求的建模符号

实体关系图（Entity Relationship Diagrams ER 图，也叫 ERD）

提供了一种描述实体、属性、联系的方法，用来对现实世界的概念进行建模

实体：用矩形表示，矩形框内填写实体的名字

属性：用椭圆表示，用无向边与对应实体相连，表示实体的属性

联系：用菱形表示，与不同的实体相连表述互相的关系

它对应与 UML 中的**类图（class diagram）**

事件跟踪（Event Traces）

关于现实世界实体的事件序列，描述对象之间发送信息的时间顺序，即对象的动态合作

对应 UML 中的**序列图（sequence diagram）**

状态机（State Machines）

描述了系统与其环境之间所有的对话

对应 UML 中的**状态图（state diagram）**

数据流图（Date Flow Diagrams，DFD）

描述了系统数据流程的图形工具，标志了一个系统的逻辑输入与输出，以及输入到输出的加工处理。它是从数据的角度，对系统整体功能的一种描述，是控制流与数据流的综合，在一定程度上反应了客户的需求

对应 UML 中的**用例图（use case diagram）**

第五章 系统设计

什么是设计?

Design is the creative process of figuring out how to implement all of the customer's requirements; the resulting plan is also called the design

Design principles are guidelines for decomposing our system's required functionality and behavior into modules.

好的设计应具备什么特点?

① 构建独立性② 异常标识和处理③ 防错和容错技术

过程分解模块

Modularity, also called separation concern, in the principle of keeping separating the various unrelated aspects of the system.

将相关信息分解，建立层次结构，并在相应的模块中增加细节。无论是下面那种分解方式，其背后的设计思想都是将事件进行模块化的分解。

Starting with a high-level depiction of the system's key elements and creating lower-level looks at how the system's features and functions will fit together

从对系统关键元素的高级描述开始，然后创建较低级别的视图，了解系统的特性和功能将如何组合在一起

模块分解（Modular decomposition）

根据软件的不同功能，将软件划分为不同的组件，每个组件执行单一的功能，保证模块直接的独立性

面向数据的分解（Data-oriented decomposition）

考虑数据如何储存、利用、加工，围绕数据对软件进行建模

面向事件的分解（Event-driven decomposition）

考虑功能直接按照事件进行交互，而不是进行直接的数据通信，所有通信信息将会被包装成事件

Outside-in-design decomposition

根据客户的要求，从需求出发，分析所需的部件，即从整体分析到细节的决定

Object-oriented decomposition

类似于面向对象程序设计的原理

Process-oriented decomposition

考虑系统的并发性，将进程作为系统执行任务的基本单位

软件构架的 4+1 视图模型

逻辑视图（A logical view）

将系统中的关键抽象显示为对象或对象类，主要描述系统的功能需求，我们可以用 UML 的类图描述逻辑视图。

进程视图（A process view）

显示系统在运行时如何由交互进程组成，侧重于系统的运行特性，关注非功能的需求，如性能、可用性，服务于系统集成人员，方便后续测试。

开发视图（A development view）

它显示了软件如何分解以进行开发，服务于软件编程人员，描述各个模块如何组织和管理。

物理视图（A physical view）

显示系统硬件以及软件组件如何在系统中的处理器之间分布。

视图的相关使用用例或场景

常用的设计风格（architecture style）

管道-过滤器模式（Pipes and Filter）

每个模块都有一组输入与输出，每个模块从输入端接收数据流，内部处理后按照标准顺序将数据送至输出端，这种模式成为过滤器，而各个过滤器之间连接的导管，起数据传递的作用，成为管道。

过滤器就是对数据进行处理的模块，而管道就是传递数据的通道，注意管道是支持并行的（concurrent execution），此模式可以描述并行的系统。

客户端-服务器模式（Client/Server）

客户端可向服务器发出请求，发送请求的过程中要遵循请求/回复协议（request/reply protocol）

服务器针对客户端发来的请求，提供对应的服务功能

这种方式部署难度比较低，但是对网络要求比较高

对等网络（Peer to Peer P2P）

每个组件都充当一个进程的功能，对于其他组件来说，这个组件自己寄宿一个客户端和服务器

任何组件都可以向其他组件发起请求

发布订阅（Publish-Subscribe）

各个组件通过订阅事件来表达对这件事情感兴趣（interest）

当一个组件发布该事件时，订阅了这个事件的组件会接收到通知。

仓库管理（Repositories）

是一种中央数据储存，外界发出对数据的储存、检索、更新等功能，经常用于与库（library）储存相关的设计，例如浏览器、组件库管理等

层次结构（Layering）

分层构架利用抽象，将软件表示为不同的抽象层次，层与层直接相互作用，具有高度抽象的特点

解释器（Interpreter）

为语句定义一个接口，该接口可以解释在特定上下文中的语句，即我们在编程时遇到的解释性语言。

解释器很适合用于开发一种新的语言，而且具有良好的扩展性和灵活性，但是后期维护很困难，而且当语句繁杂时，会引起膨胀，导致难以维护，因此实际中很少用。

注意解释器并不是编译器！编译器用的是管道-过滤器模式

软件的质量属性（Quality Attributes）

可修改性（Modifiability）

Design must be easy to change

性能（Performance）

Performance attributes describe constraints on system speed and capacity:

常见的性能包括：

① 系统响应时间（Response time）：响应请求的速度

② 吞吐量（Throughput）：每分钟可以处理多少请求

③ 负载（Load）：在响应时间和吞吐量开始受到影响之前，它可以支持多少用户？

安全（Security）

安全主要分为两个方面：

① 免疫（Immunity）：阻止外界攻击的能力

② 弹性（Resilience）：从外界攻击中快速回复的能力

可靠性（Reliability）

软件在假定条件下正确执行，则说明系统的可靠的

常见的异常有：

① 没能正确提供应有的服务

② 没有提供服务

③ 数据被破坏

④ 违反系统的属性规定，如安全属性

⑤ 死锁

注意，发生异常后系统无法恢复到之前的状态（restore the system to its previous state）

健壮性（Robustness）

衡量系统在出现故障的情况下的回复能力，故障可以是来自系统外部，也可以来自系统内部，例如系统内部崩溃后的恢复机制，接收到错误输入的应对机制等。

需要注意的是，健壮性描述的是系统出错后也能继续运行的能力，而可靠性描述的是系统出错的概率，一般通过一些可靠性测试来衡量系统的可靠性。

可用性（Usability）

指的是产品对用户来说有效、易学、少错和令人满意的程度，即用户能否用产品完成他的任务，效率如何，主观感受怎样。

第六章 模型设计

模块化（Modularity）

Modularity is the principle of keeping separate the various unrelated aspects of a system, so that each aspect can be studied in isolation

模块化就是将系统各个不相关方面分开，以便研究单独每个方面，也成关注点分离（separation concern）。如果原理使用得当，生成的每个模块都是单一目的的，相对独立与其他模块的。

耦合（Coupling）

表示模块之间的依赖性，根据依赖性强弱可分为：tightly coupled，loosely coupled，uncoupled

内容耦合（Content Coupling）

一个模块直接使用或者修改另一个模块内部的数据，或者通过非正常的入口，直接进入另一个模块的内部。比如在某个组件的分支操作，直接进入到了另一个组件之中（one component branches into the middle of another component）。

共享耦合（Common Coupling）

几个模块对一个公共的数据区域进行数据上的操作

控制耦合（Control Coupling）

一个模块通过传递参数，或者函数的返回值来控制另一个模块的行为

特征 / 标记耦合（Stamp Coupling）

几个模块共同使用一个复杂的数据结构

数据耦合（Data Coupling）

几个模块共享几个数据的值。

特征耦合代表模块直接有更复杂的接口，因为传递相同的数据结构意味着模块对数据的格式和组织形式要达成一致。而如果只是值的传递，而不是数据结构的传递，那么就是数据耦合，它更简单。

内聚（Cohesion）

描述模块内部各部分的关联程度，是衡量一个模块是否只专注做一件事的定性指标（focuses on just one thing）

偶然内聚（Coincidental Cohesive）

模块直接毫无关系，相互之间是松散的

逻辑内聚（Logical Cohesive）

逻辑上相关的功能被放在同一个模块之中，即模块执行任务的逻辑是相似的，尽管他们的功能可能有很大差别

时间内聚（Temporal Cohesive）

一个模块完成的许多功能必须在相近的时间点完成，比如系统的初始化，这些功能通过时间因素关联在一起

过程内聚（Procedural Cohesive）

允许在调用前面的构件或操作之后，马上调用后面的构件或操作，即使两者之间没有数据进行传递

通信内聚（Communicationally Cohesive）

一个模块的所有成分都操作同一数据集或者生成同一数据集

顺序内聚（Sequential Cohesive）

指一个模块中各个处理元素都密切相关于同一功能且必须顺序执行,前一功能元素的输出就是下一功能元素的输入

功能内聚（Functional Cohesive）

最理想的内聚方式。模块的所有成分对于完成单一功能都是必须的

软件的设计应该追求低耦合，高聚合（low coupling， high cohesion）

第七章 软件实现

程序的主要层次

控制结构（control structures）

控制结构主要指程序的分支结构，展现程序的控制流结构。

控制单元可以让代码更易阅读，同时将诸多独立的模块组织成有逻辑联系的程序，还可以在控制结构中加入传递参数的名字或者一定的注释，从而展现各组件之间的耦合性

算法（algorithms）

算法有着共同的目标：性能（速度）。同时更高效的算法也能让编写、理解、测试、修改代码的成本降低

数据结构（data structures）

使用数据结构来将程序中的数据组织起来，从而让程序尽可能保持简单。

程序文档

内部文档（Internal documentation）

内部文档经常以注释的方式包含在程序之中，是可以直接写在程序里面的，写法一般有固定的格式。

内部文档包括以下成分：

标题注释块（header comment block）

组件的名称

组件的编写者

组件在一般系统设计中的位置

编写和修订组件的时间

组件的功能和作用

组件是如何使用其数据结构、算法和控制流程的

有意义的变量名和语句标签

其他程序注释

外部文档（External documentation）

外部文档写在需要使用该软件的人可以阅读有关如何使用该软件的地方

外部文档包括以下成分：

对于要解决问题的描述

对于程序采用的算法的描述

对于数据的描述

第八章 程序测试

测试的目标（Objective of testing）

测试的目标是为了发现错误，测试不是为了证明软件的正确性，而且为了证明软件有错误。只有当一个测试发现了软件的错误时才能说测试是成功的。

一般在测试时，会选择与软件开发团队相独立的测试团队，即让对软件陌生的人对软件进行测试，这样做是因为陌生人对软件的测试更严格、无情。

故障类型（Faults）

算法故障（Algorithmic fault）

算法故障指的是由于组件内部算法或者逻辑发生错误，对于给定输入没有得到正确的输出。

语法（syntax）错误也属于算法故障

计算和精度故障（Computation and precision fault）

由于程序内部公式的实现是错误的而导致

文档故障（Documentation fault）

文档所描述的程序功能和实际得到的程序功能不符合

性能故障（Performance faults）

系统未按照规定的速度运行

容量或边界故障（Capacity or boundary faults）

当数据达到一定大小超出容量限制时，系统无法接受而报错。

程序测试步骤

单元测试（Module test, component test, unit test）

测试各个组件的功能

集成测试（Integration test）

将所有组件组装成一个整体进行测试

系统测试（System test）

对整个系统进行测试，包括以下四个方面：

功能测试（Function test）

按照系统的功能需求进行测试

性能测试（Performance test）

测试除功能外其他的项目，如速度、精度等

验收测试（Acceptance test）

让顾客针对具体软件确认自己的需求

安装测试（Installation test）

在用户的计算机环境下进行安装测试

测试技术——静态测试

也称为代码评审（code view）。以人工的方式对代码进行复审，检查程序的静态结构，重点关注程序的逻辑设计，而不是找到代码中的错误去修改它，主要根据需求定义和设计文档来审核源代码。对于特定类型的错误，研究表明静态测试更有效。静态测试分为两类：

代码走查（walk-through）

程序员向评审小组提交代码及其相关文档，然后评议小组评论其正确性

代码审查（code inspection）

与走查类似但是更正式。按照一个事先准备好的问题清单来检查代码和文档

测试技术——动态测试

在设定的测试数据上执行被测试程序，通过代码的动态结果验证结果的正确性。具有三个要素：程序、测试数据、需求定义。

黑盒测试（black box test）

已知产品应具有的功能，通过测试检验每个功能是否能正常使用，因此又称为功能测试。黑盒测试着眼于程序的外部结构，仅仅考虑输入和输出的关系，注重对程序的对外接口进行测试，主要针对程序的软件页面和软件功能。测试方法如下：

划分等价类（equivalence class partitioning）：把程序的输入域划分为若干子集，每个部分选取少数具有代表性的数据作为测试用例，假设这些数据能代表这一类其他所有数据。

边界测试（boundary test）：对输入、输出的边界数据进行测试

白盒测试（white box test）

这是一种测试用例的设计方法，已知产品内部的工作过程，对程序内部的具体结构进行测试。主要测试方法如下：

语句覆盖（statement coverage）：保证程序中每条语句至少执行一次，最弱的覆盖

条件覆盖（condition coverage）：保证程序中每个判断的真假都至少执行一次

边界覆盖（boundary coverage）：对每个组件的边界条件进行测试

穷举测试（exhaustive test）

列举出程序的所有情况，对于黑盒来说对所有输入的可能值进行测试，对于百合来说，使测试的覆盖率达到百分之百，即每条路径的每条可能都执行一次。

需要指出的是穷举测试是不可能的，这只是一个理性情况而已

集成测试（Integration testing）

将软件的各部分组件连接起来后，会得到软件的整体构架，一般是树形结构。主要测试方法如下：

自底向上的测试（Bottom-up testing）

先测试软件最底部的各个独立组件再向上测试集成的组件。测试每一个集成的组件都要连带其下所属的底层组件一并测试一遍，即将他们看作一个整体进行测试。

需要注意的是这种测试需要为编写驱动（driver components）

自顶向下的测试（Top-down testing）

利用广度优先遍历或深度优先遍历，从根节点开始向下遍历，遍历结点的顺序就是测试的顺序。

三明治测试（Sandwich testing）

混合使用自底向上和自顶向下的测试方式，先采用自顶向下测试单个组件，再利用自底向上将已测试的单个组件组装成一个整体测试。

大爆炸测试（Big-bang testing）

类似于黑盒测试，直接将所有组件组装成一个整体，对整个系统进行测试。

第九章 系统测试

系统测试一般是进行功能测试、性能测试、验收测试、安装测试。

性能测试

压力测试（stress test）

不在常规条件下运行软件，而是在计算机数量较少或系统资源匮乏下测试软件的性能，通常测试的资源包括内部内存、CPU 可用性、磁盘空间、网络带宽

容量测试（volume test）

产生大量测试数据来测试软件某项指标的极限值，在给定时间内能够处理的最大负载量，从而让开发商和用户了解该软件系统的承受能力或提供服务能力。

配置测试（configuration test）

对电脑硬件的测试，通过对被测系统的软硬件环境进行调整，了解各种不同环境对系统性能的影响程度，进而找到系统各项资源的最优分配原则。

兼容性测试（compatibility test）

检查软件之间能否相互正确地交流共享信息。测试内容主要为不同软件版本的兼容性；不同操作系统下软件的兼容性；新旧数据之间的兼容性等。

回归测试（regression test）

回归测试是用于软件新版本的一种测试，在修改了旧代码后，重新对相同的功能进行测试，以确认修改没有引入新的错误或导致其他代码产生错误。

由于在正常情况下，对旧代码的修改往往会引入新的错误，因此回归测试的很必要的

软件的性能指标

MTTF (Mean Time to Failure)平均失效等待时间

MTTR (Mean Time to Repair)平均修复时间

MTBF (Mean Time between Failures)平均失效间隔时间

MTBF=MTTF+MTTR

R=MTTF/(1+MTTF) (**可靠性** Reliability)

A=MTBF/(1+MTBF) (**可用性** Availability)

M=1/(1+MTTR) (**可维护性** Maintainability)

验收测试

基准测试（benchmark test）

由实际的用户或测试系统功能的专门小组进行测试

试点测试（pilot test）

在特定的实验环境下安装测试

α 测试（Alpha test）

在小范围的用户内部测试，相当于内测（in-house）

β 测试（Beta test）

开发组发布软件，要求用户运行并提交反馈意见，即公测（out-house）

并行测试（parallel test）

新旧版本系统并行运行测试

安装测试

一般而言如果进行了验收测试就不需要进行安装测试了。如果在验收测试时测试环境和用户环境有较大差距，那么安装测试是有必要的。

第十章 系统维护

维护的定义

Any work done to change the system after it is in operation is considered to be maintenance.

维护类型

更正性维护（corrective maintenance）

又叫修正错误，对软件系统中发生异常或故障的部分修复，改正软件性能上的缺陷，排除实施中的错误，一般是出现错误后才会进行维护，是被动的

适应性维护（adaptive maintenance）

由于软件会经常出现新版本、新插件等，需要对软件进行改造，例如添加插件、更新版本，这样的维护叫适应性维护

完善性维护（perfective maintenance）

对已有的软件追加新的功能，这些功能一般是在设计阶段没有考虑的功能，为了让软件变得更强大、更高效

预防性维护（preventive maintenance）

是一种主动的维护，一般选择还有较长寿命的，目前仍能正常运行，但是可能要发生变化或调整的软件进行维护，为了给未来的修改和调整奠定更好的基础

圈复杂度（cyclomatic number）

用于衡量一个模块判定结构的复杂度，圈复杂度越大软件越难维护。一般将软件的流程图进行抽象，得到一个无向图，流程被抽象化为结点。

复杂度计算公式为 V = e – n + 2

E 表示边数，n 表示节点数