#### 软件测试与维护复习归纳

不保证内容完全无误。

Zhc

2019年5月22日，星期三

1. 描述软件失败的术语

缺点（defect）、偏差（variance）、故障（fault）、失败（failure）、问题（problem）、矛盾（inconsistency）、错误（error）、特殊（feature）、事件（incident）、缺陷（bug）、异常（anomaly）

在这里全都叫缺陷（bug）。

1. 问题的来源
2. 需求定义（requirements definition）：错误的、不完整的和不一致的需求
3. 设计：软件中的重大设计错误
4. 实现（implementation）：编程中的错误和恶意代码
5. 支持系统（support system）：糟糕的（poor）编程语言、有缺陷的编译器及调试器、误导性的开发工具
6. 不充分的软件测试（inadequate testing of software）：不完备的测试、缺乏确认（poor verification）和调试错误
7. 发展（evolution）：草率的再开发或维护、尝试修复旧缺陷时产生了新缺陷、增量升级导致过度复杂（incremental escalation to inordinate complexity）
8. 软件缺陷的定义

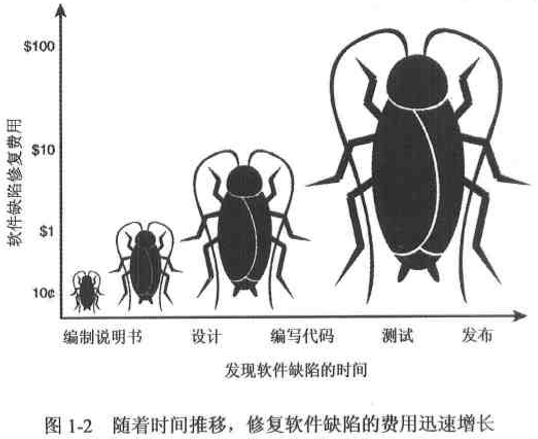
（共5条）

1. 软件未实现产品说明书（specification）要求的功能。
2. 软件出现了产品说明书指明不应该出现的错误。
3. 软件实现了产品说明书未提到的功能。
4. 软件未实现产品说明书虽未明确提及但应该实现的目标。
5. 软件测试员认为软件难以理解、不易使用、运行缓慢或者（从测试员的角度看）最终用户会认为不好。
6. 软件缺陷的构成

导致软件缺陷最大的原因是产品说明书。

（以下为约数）

1. 软件缺陷修复费用与缺陷发现时间的关系



费用指数级地增长——也就是说随着时间的推移，费用呈十倍地增长。

1. 软件测试员的目标

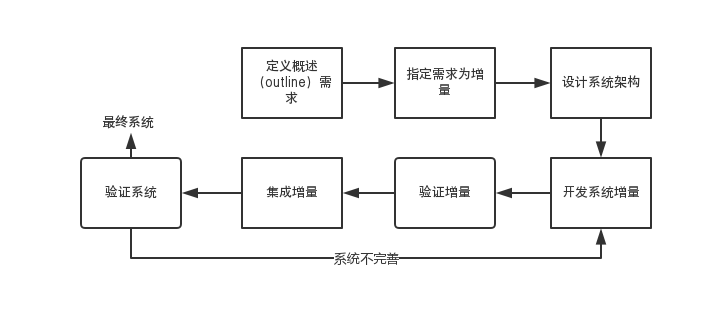
尽可能早地找出软件缺陷，并确保其得以修复。

英文原文：To find bugs as early in the software development processes as possible, and make sure they get fixed.

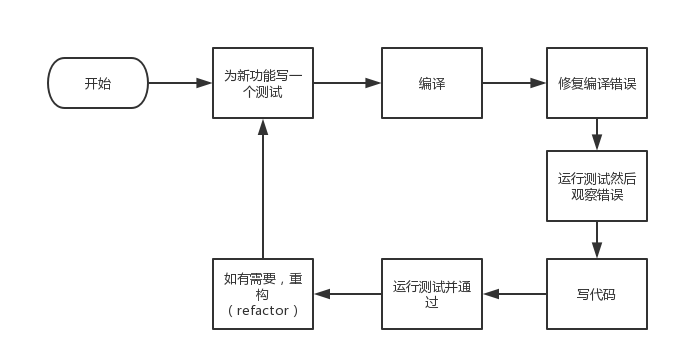
建议：千万不要在无法达到的完美上兜圈子。

英文原文：Be careful not to get caught in the dangerous spiral of unattainable perfection.

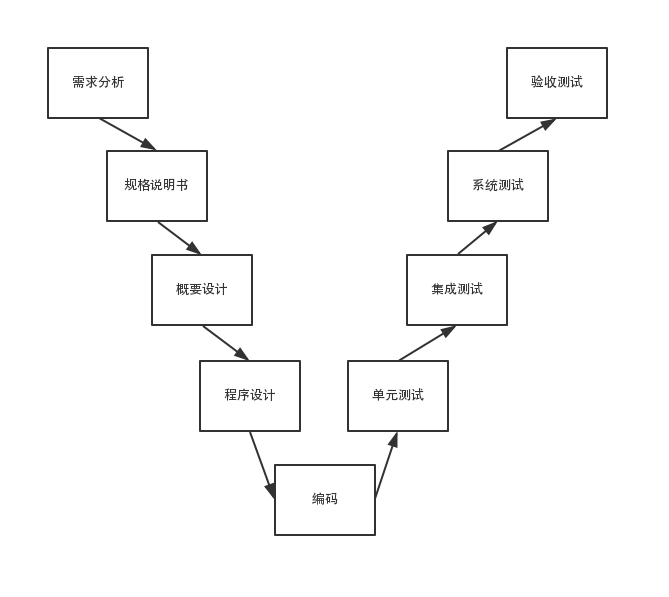
1. 软件包括需求说明文档（requirements specification documents）、设计文档（design documents）、源代码（source code）、测试套件（test suites）及测试计划（test plans）、面向软硬件操作环境的接口（interfaces to hardware and software operating environment）、内部和外部的文档编制（internal and external documentation）、可执行程序及其持久化数据（executable programs and their persistent data）。
2. 软件开发过程
3. 瀑布模型
4. 原型工具：GUI Design Studio
5. 增量开发（Incremental development）



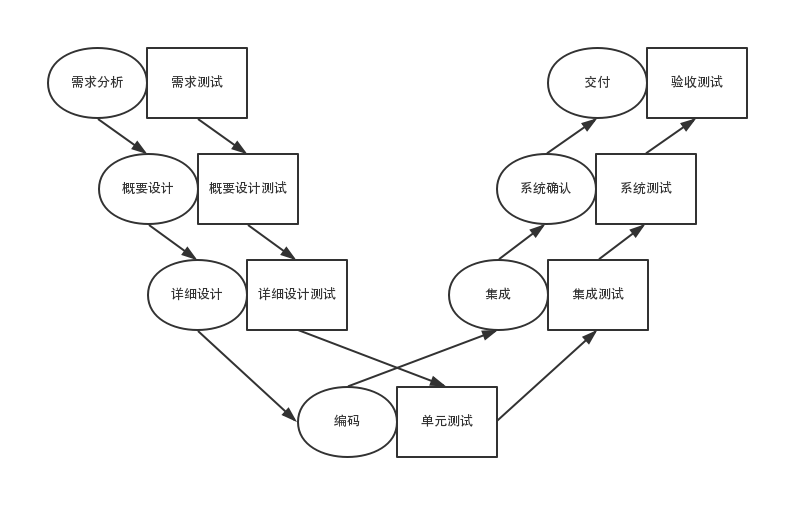
1. 测试驱动开发（Test-Driven Development, TDD）



1. 与测试相关的过程模型
2. V模型



1. W模型

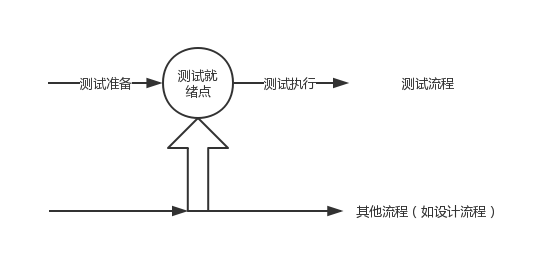


1. H模型

测试流程：

测试准备活动：测试计划、测试设计和测试开发。

测试执行活动：测试运行和测试评估。



测试不仅仅是测试执行，还包括其他活动。

测试是一个独立流程，贯穿整个产品周期，与其他流程并发执行。

测试要尽早准备，尽早执行。

1. 软件测试原则（axiom）

（共9条）

1. 完全测试程序是不可能的（It is impossible to test a program completely）。

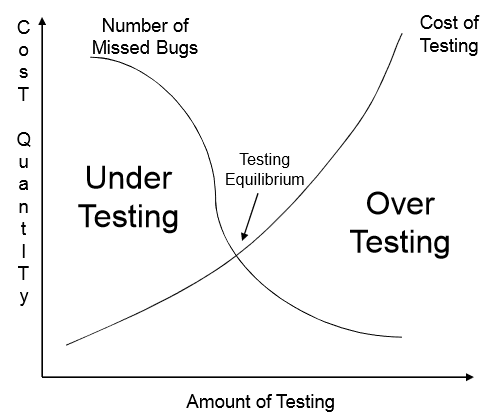
想象一下，如果要完全测试以下的软件，会需要多少测试用例：PowerPoint、计算器、MS Word等等。

保证软件绝对完备（be absolutely sure software works）的唯一方法是执行所有可能的输入并检查其输出。同时，说明书必须正确且完善。

1. 软件测试是有风险的行为（Software testing is a risk-based exercise）。

如果进行太多测试，开发成本会难以承受。

如果测试太少，软件出错的可能性会增大，软件错误会耗费许多时间。



1. 测试无法显示潜伏的软件缺陷（Testing cannot show the absence of bugs）。

对于软件缺陷，软件测试只能证其有，不能证其无。

1. 找到的软件缺陷越多，就说明软件缺陷越多（The more bugs you find, the more bugs there are）。

缺陷成组出现，如果你看到一个，你很可能会发现更多。原因：程序员可能状态不佳；程序员常常犯相同的错；一些缺陷可能只是冰山一角。

1. 并非所有软件缺陷都要修复（Not all bugs found will be fixed）。

不修复已知缺陷的原因：

α 时间不足：有些截止日期不能被推迟（如千年虫问题）

β 不是真的缺陷：说明书可能出错

γ 修复的风险太高

δ 缺陷不值得修复

1. 难以说清什么时候才叫缺陷（It is difficult to say when a bug is indeed a bug）。

如果一个软件中的问题没人发现，这算缺陷吗？

为了让我将其看作一个缺陷，这个缺陷必须是可观测的吗？（Does a bug have to be observable in order for it to me a bug?）

未被发现的缺陷被称作潜在缺陷（latent bug）。

1. 产品说明书从没有最终版本（Specifications are never final）。

基于“变化的目标”说明书建造产品：竞争激烈、发布周期快、软件“易于”改变

1. 软件测试员在产品小组中不受欢迎（Software testers are not the most popular members of a project）。

避免不受欢迎的提示：尽早找出缺陷；控制你的热情（enthusiasm），用专业的方法；不要只报告坏消息

1. 软件测试是一项讲究条理的技术专业（Software testing is a disciplined and technical profession）。

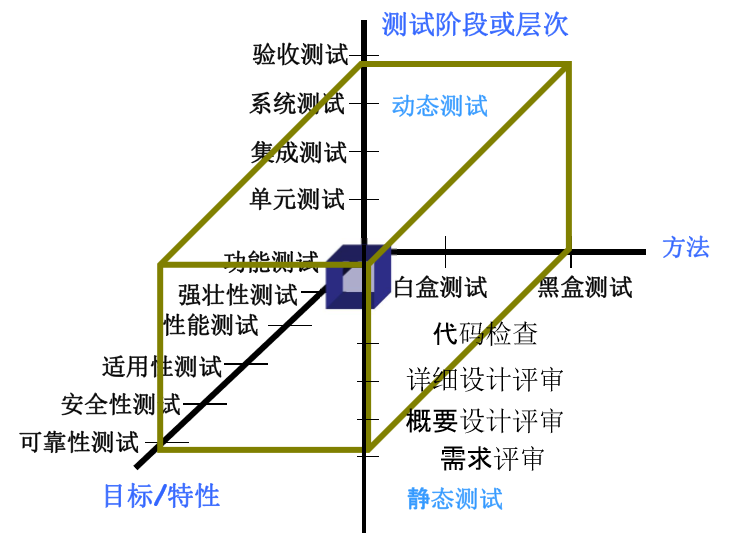
现在构建一个有缺陷的软件非常昂贵。结果是，软件测试已成长为一门成熟的学科：老练的技巧、工具支持、高收入的职业

注：书上还有一条“杀虫剂怪事”，意思是软件测试越多，其对测试的免疫力越强。

1. 补充原则
2. 软件测试必须基于“质量第一”。
3. 事先定义好产品的质量标准，根据测试的结果，对产品的质量进行分析和评估。
4. 第三方进行测试会更客观，更有效。
5. 重视文档，妥善保存一切测试过程文档（测试计划、测试用例、测试报告等）。
6. 软件测试误区

（共5条）

1. 如果发布出去的软件有质量问题，都是软件测试人员的错。
2. 软件测试技术要求不高，至少比编程容易多了。
3. 有时间就多测试一些，来不及就少测试一些。
4. 软件测试是测试人员的事，与开发人员无关。
5. 根据软件开发瀑布模型，软件测试是开发后期的一个阶段。
6. 软件测试分类



1. 假装自己是消费者（Act-Like-A-Customer, ALAC）

二八原则：

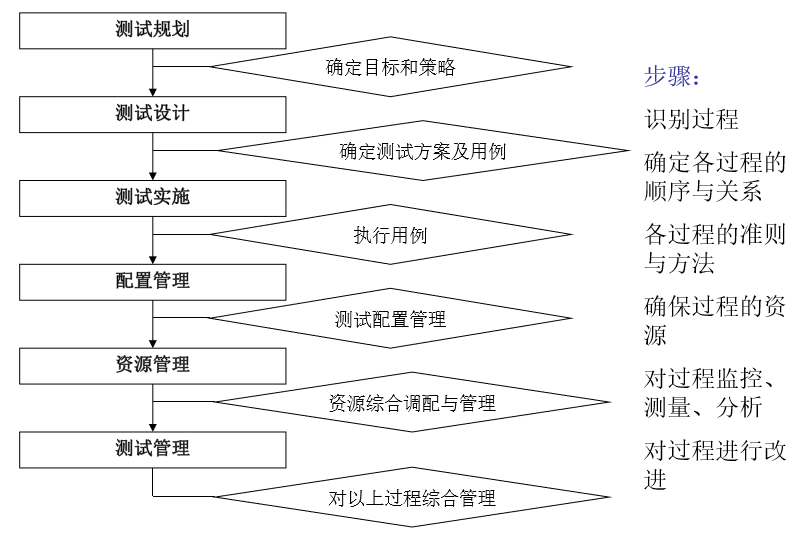
常用功能占（所有功能的）20%，而其缺陷分布占（所有缺陷的）80%。

非常用功能占（所有功能的）80%，而其缺陷分布占（所有缺陷的）20%。

1. 确认、系统与验收测试

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试类型 | 对象 | 目的 | 依据 | 方法 |
| 单元测试 | 模块内部程序错误 | 消除局部模块的逻辑和功能上的错误和缺陷 | 详细设计 | 白盒为主，黑盒为辅 |
| 集成测试 | 模块间的集成和调用关系 | 找出与软件设计相关的程序结构、模块调用关系和模块间接口方面的问题 | 概要设计 | 白盒与黑盒结合 |
| 系统测试 | 整个系统中的软硬件 | 对整个系统进行一系列的整体、有效性测试 | 需求规格说明书、概要设计 | 黑盒 |
| 验收测试 | 用户的实际环境 | 对整个系统进行整体有效测试 | 需求规格说明书 | 黑盒 |

1. 测试与调试
2. 测试发展的初期，测试就是调试，而现在测试是一个系统化工程化的概念，调试的范畴更小一些。
3. 调试不属于测试，是编码阶段的工作，由程序员完成；调试与测试的对象及采用的方法有很大程度上的相似，调试还用到断点控制等排错方法，但其目的却完全不同。
4. 而测试由测试员或程序员完成。
5. 测试是为了找出软件中存在的缺陷；而调试是为了解决存在的缺陷。
6. 成功的测试发现了错误的症状，从而引起调试的进行。
7. 测试管理体系



1. 确认（verification）和验证（validation）

确认是保证软件符合产品说明书的过程；

验证是保证软件满足用户要求的过程。

1. 软件质量保证（Software Quality Assurance, SQA）
2. 定义：通过对软件产品和活动有计划地进行评审和审计来验证软件是否合乎标准的系统工程活动。
3. 与软件测试的关系

SQA是管理工作，其审查对象是流程，强调以预防为主；测试是技术实施工作，其测试对象是产品，主要是以事后检查（文档、程序）为主。

SQA指导测试、监控测试；测试为SQA提供依据；测试是SQA的一个环节、一个手段。

1. 软件测试团队的任务与责任

（共8条）

1. 基本任务：测试计划、测试用例设计、执行测试、评估测试结果、递交测试报告；
2. 尽早地发现问题；
3. 督促相关人员尽快地解决产品中的缺陷；
4. 帮助项目管理人员制定合理的开发计划；
5. 并对问题进行分析、分类总结和跟踪；
6. 提高程序、文档编写的规范性、易读性、可维护性等。
7. 测试团队的基本构成及其职责

（共7条）

1. 测试组长：负责项目的管理、测试计划、任务安排等；
2. 环境配置人员：设置、配置和维护实验室的测试环境；
3. 测试设计人员/资深测试工程师：产品规格说明书、设计的审查、测试用例的设计、技术难题的解决、培训和指导、实际测试任务的执行；
4. 一般（初级）测试工程师：执行测试用例和相关的测试任务；
5. 以项目经理为核心的组织模型
6. 培训考核的项目

公司产品、测试知识与技术、ISO 9001 & CMMI知识、工作态度、测试用例设计、测试用例执行、发现缺陷能力、缺陷描述、问题分析、报告、流程控制

1. 优秀软件测试工程师的必备素质

（共10个）

1. 沟通能力
2. 技术能力
3. 信心
4. 外交能力和幽默感
5. 耐心
6. 很强的记忆力
7. 怀疑精神
8. 洞察力
9. 适度的好奇心
10. 反向思维和发散思维能力
11. 测试策略的概念

测试策略通常是描述测试工程的总体方法和目标。描述目前在进行哪一阶段的测试（如单元测试、集成测试、系统测试）以及每个阶段内进行的测试种类（如功能测试、性能测试、压力测试等）和方法，以确定合理的测试方案使得测试更有效。

1. 测试计划的组成

（共16条）

1. 测试计划标识符（test plan identifier）
2. 介绍（introduction）
3. 测试条目（test items）
4. 待测试特性（features to be tested）
5. 无需测试的特性（features not to be tested）
6. 方法（approach）
7. 条目成败标准（item pass/fail criteria）
8. 暂停标准和继续条件（suspension criteria and resumption requirements）
9. 测试的可交付成果（test deliverables）
10. 测试任务（testing tasks）
11. 环境要求（environmental needs）
12. 职责（responsibilities）
13. 人员和培训要求（staffing and training needs）
14. 进度表（schedule）
15. 风险和事故（risk and contingencies）
16. 批准（approvals）
17. 测试用例（test case）
18. 定义：满足特定目的的测试数据、测试代码和测试规程的集合；是发现软件缺陷的最小测试执行单元；有特殊的书写标准和基本原则。
19. 书写格式

|  |  |  |
| --- | --- | --- |
| 字段名称 | 描述 | |
| 标识符 | 0001 | |
| 测试项 | 学生选课系统登录功能 | |
| 测试环境 | 测试环境要求 | 实际测试环境 |
| 硬件：  软件：  Windows 2000 Server，  IE5.0以上浏览器，  Web server  网络：  Internet  工具：  Rational Robot 2003  等等 |  |
| 输入 | 输入标准 | 实际输入 |
| 1. 输入正确的用户名和密码 2. 输入错误的用户名和密码 | 1. user1, pwd1 2. errorusr, pwd2 |
| 输出 | 输出标准 | 实际输出 |
|  | 1. 正确进入系统 2. 错误提示 | 1. 正确进入系统 2. 错误提示 |
| 测试用例间的关联 |  | |

1. 测试用例设计生成的基本准则

α 测试用例的代表性：能够代表并覆盖各种合理的和不合理、合法的和非法的、边界的和越界的、以及极限的输入数据、操作和环境设置等；

β 测试结果的可判定性：即测试执行结果的正确性是可判定的，每一个测试用例都应有相应的期望结果；

γ 测试结果的可再现性：即对同样的测试用例，系统的执行结果应当是相同的。

1. 测试用例设计的步骤

α 为测试需求确定测试用例

β 为测试用例确定输入输出

γ 编写测试用例

δ 评审测试用例

ε 跟踪测试用例

1. 测试用例的作用

α 避免盲目测试

β 估算测试工作量

γ 减少回归测试的复杂程度

δ 方便地书写软件测试缺陷报告

ε 实施不同级别的测试

总结：软件测试是有组织性、步骤性和计划性的，为了能将软件测试的行为转换为可管理的、具体量化的模式，需要创建和维护测试用例。

1. 测试用例工具

TestDirector、Testlink

1. 测试系统的架构
2. 测试套件（test suite）：一组相关需求的测试集合（collection of tests for a related group of requirements）
3. 测试：一个功能或需求的一个或多个测试用例
4. 测试用例：最小单元
5. 黑盒测试与白盒测试

|  |  |
| --- | --- |
| 黑盒测试 | 白盒测试 |
| 不涉及程序结构 | 考查程序逻辑结构 |
| 用软件规格说明生成测试用例 | 用程序结构信息生成测试用例 |
| 功能测试和系统测试 | 单元测试和集成测试 |
| 某些代码段得不到测试 | 对所有逻辑路径进行测试 |

1. 动态黑盒测试

不深入代码细节测试软件的方法称为动态黑盒测试（dynamic black-box testing）。动态，即程序在运行；黑盒，因为测试时不知道程序如何工作。有时又被称为行为测试（behavioral testing）。

动态黑盒测试需要一个可执行程序（executable program）和产品说明书（或者用户手册）。

测试用例以一个二元组集合的形式表示，如（输入，期望输出）。

测试数据和测试用例：测试数据是测试系统时的输入；测试用例是输入和期望输出。

黑盒测试的特点：程序被当做一个黑盒；实现细节无关紧要；需要站在最终用户的立场上；测试计划可以在早期开始。

测试用例设计技术：

1. 等价类划分方法

等价类划分法是把程序的输入域划分成若干部分，然后从每个部分中选取少数代表性数据当作测试数据。

在每一个等价类中取一个数据作为测试的输入条件，就可以用少量代表性的测试数据取得较好的测试结果。可以减少测试用例的量。

每一类的代表性数据在测试中的作用等价于这一类中的其他值，也就是说，如果某一类中的一个例子发现了错误，这一等价类中的其他例子也能发现同样的错误；反之，如果某一类中的一个例子没有发现错误，则这一类中的其他例子也不会查出错误。

等价类划分有两种不同的情况：

有效等价类：是指对于程序的规格说明来说是合理的、有意义的输入数据构成的集合。利用有效等价类可检验程序是否实现了规格说明中所规定的功能和非功能。

无效等价类：与有效等价类的定义恰巧相反。

设计测试用例时，要同时考虑这两种等价类。因为软件不仅要能接收合理的数据，也要能经受意外的考验。这样的测试才能确保软件具有更高的可靠性。

集合的划分：设是集合的子集。若且，则称是集合的一个划分。保证了测试的完备性和无冗余性。

确定等价类的原则：

α 在输入条件规定了取值范围或值的个数的情况下，则可以确立一个有效等价类和两个无效等价类。

β 在输入条件规定了输入值的集合或者规定了“必须如何”的条件的情况下，可以确立一个有效等价类和一个无效等价类。

γ 在输入条件是一个布尔量的情况下， 可确定一个有效等价类和一个无效等价类。

δ 在规定了输入数据的一组值 ，并且程序要对每一个输入值分别处理的情况下，可确立n个有效等价类和一个无效等价类。

ε 在规定了输入数据必须遵守的规则的情况下，可确立一个有效等价类（符合规则）和若干个无效等价类（从不同角度违反规则）。

ζ 如果确知，已划分的等价类中各个元素在程序中的处理方式不同，则应将此等价类进一步划分成更小的等价类。例如：

if(i>=0&&i<=100)

switch(i){

case 1:…;break;

case 3:…;break;

case 5:…;break;

…

}

根据等价类创建测试用例的步骤：

α 建立等价类表，列出所有划分出的等价类：

|  |  |  |
| --- | --- | --- |
| 分析 | 有效等价类 | 无效等价类 |
| …… | …… | …… |
| …… | …… | …… |

β 为每个等价类规定一个唯一的编号。

γ 设计一个新的测试用例，使其尽可能多地覆盖尚未覆盖的有效等价类。重复这一步，最后使得所有有效等价类均被测试用例所覆盖。

δ 设计一个新的测试用例，使其只覆盖一个无效等价类。重复这一步，使所有无效等价类均被覆盖。

1. 边界值分析法（BVA，Boundary Value Analysis）

边界值分析也是一种黑盒测试方法，是对等价类划分方法的补充。

边界值分析不是从某等价类中随便挑一个作为代表，而是使这个等价类的每个边界都要作为测试条件。

人们从长期的测试工作经验得知，大量的错误是发生在输入或输出范围的边界上，而不是在输入范围的内部。因此针对各种边界情况设计测试用例，可以查出更多的错误。

与等价划分的区别：

α 边界值分析不是从某等价类中随便挑一个作为代表，而是使这个等价类的每个边界都要作为测试条件。

β 边界值分析不仅考虑输入条件，还要考虑输出空间产生的测试情况。

设计方法：

α 确定边界情况（输入或输出等价类的边界）。

β 选取正好等于、刚刚大于或刚刚小于边界值作为测试数据。

γ 一个边界，加上左右值，共三个测试用例。

1. 错误推测方法

通过经验和直觉推测出程序的错误所在；主观、灵感、反向思维，难以复制等等；不是一个系统的方法，用作辅助手段。

策略：先等价类划分，再边界值分析，三错误推测法。不要搞倒了，实际中经常搞倒。

1. 判定表驱动分析方法

后面的因果图方法中会到判定表。判定表（Decision Table）是分析和表达多逻辑条件下执行不同操作的工具。

在程序设计发展的初期,判定表就已被当作编写程序的辅助工具了，因为它可以把复杂的逻辑关系和多种条件组合的情况表达得既具体又明确。

判定表通常由五个部分组成：条件桩、动作桩、条件项、动作项和规则。

下表是一张关于科技书阅读指南的判定驱动表：3个问题，8 种情况，23=8。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 问题 | 你觉得疲倦吗？ | Y | Y | Y | N | N | N | N | N |
| 你对内容感兴趣吗？ | Y | Y | N | N | Y | Y | N | N |
| 书中内容使你糊涂吗？ | Y | N | Y | N | Y | N | Y | N |
| 建议 | 请回到本章开头重读 | x |  |  |  | x |  |  |  |
| 继续读下去 |  | x |  |  |  | x |  |  |
| 跳到下一章去读 |  |  |  |  |  |  | x | x |
| 停止阅读，请休息 |  |  | x | x |  |  |  |  |

判定表的建立步骤：（根据软件规格说明）

α 确定规则的个数。假如有n个条件桩，每个条件桩有两个取值(Y, N)，故有2n种规则。

β 列出所有的条件桩和动作桩。

γ 填入条件项。

δ 填入动作项。得到初始判定表。

ε 就是根据原始判定表的规则，设计测试用例， 要求覆盖所有的原始判定表的规则（一条规则至少一个测试用例）。

1. 因果图法

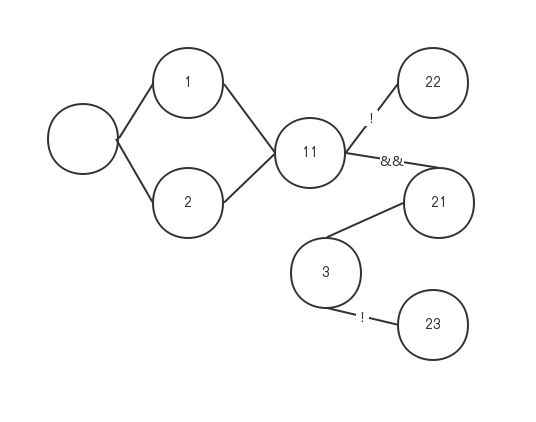
概念：

α 借助图的方式，设计测试用例，被测程序有多种输入条件，输出结果依赖于输入条件的组合；

β 着重分析输入条件的各种组合，每个条件就是一个“因”，这些“因”的组合必然有一个输出的结果，这就是“果”；

γ 与其他的方法相比，更侧重于输入条件的组合。

例子：



|  |  |  |  |
| --- | --- | --- | --- |
| o | || | && | ！ |
| 约束 | 或 | 与 | 非 |

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 原因（条件） | 编号 | 结果（动作） |
| 1 | 第一列字符是A | 21 | 修改文件 |
| 2 | 第一列字符是B | 22 | 给出信息L |
| 3 | 第二列字符是一个数字 | 23 | 给出信息M |
| 11 | 中间原因 |  |  |

因果图方法最终生成的是判定表。它适合于检查程序输入条件的各种组合情况。利用因果图生成测试用例的基本步骤：

α 分析软件规格说明描述中，哪些是原因（即输入条件或输入条件的等价类），哪些是结果（即输出条件），并给每个原因和结果赋予一个标识符。

β 分析软件规格说明描述中的语义。找出原因与结果之间，原因与原因之间对应的关系。根据这些关系，画出因果图。

γ 把因果图转换为判定表。

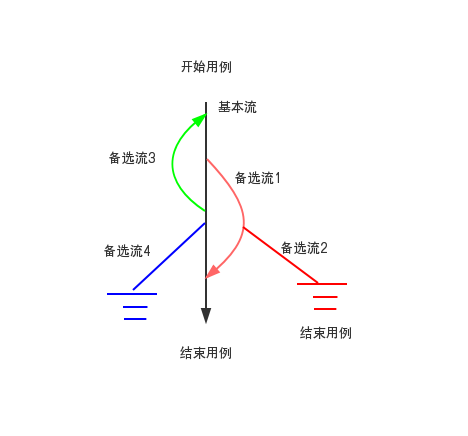
δ 把判定表的每一列拿出来作为依据，设计测试用例。

1. 场景法

现在的软件几乎都是用事件触发来控制流程的，事件触发时的情景便形成了场景，而同一事件不同的触发顺序和处理结果就形成事件流。这种在软件设计方面的思想也可引入到软件测试中，可以比较生动地描绘出事件触发时的情景，有利于测试设计者设计测试用例，同时使测试用例更容易理解和执行。

用例场景用来描述流经用例的路径，从用例开始到结束遍历这条路径上所有基本流和备选流。

基本流和备选流：



上图中经过用例的每条路径都用基本流和备选流来表示，直黑线表示基本流，是经过用例的最简单的路径。备选流用不同的彩色表示，一个备选流可能从基本流开始，在某个特定条件下执行，然后重新加入基本流中（如备选流1和3）；也可能起源于另一个备选流（如备选流 2），或者终止用例而不再重新加入到某个流（如备选流2和4）。

按照上图中每个经过用例的路径，可以确定以下不同的用例场景：

|  |  |
| --- | --- |
| 场景1 | 基本流 |
| 场景2 | 基本流、备选流1 |
| 场景3 | 基本流、备选流1、备选流2 |
| 场景4 | 基本流、备选流3 |
| 场景5 | 基本流、备选流3、备选流1 |
| 场景6 | 基本流、备选流3、备选流1、备选流2 |
| 场景7 | 基本流、备选流4 |
| 场景8 | 基本流、备选流3、备选流4 |

注：为方便起见，场景5、6和8只考虑了备选流3循环执行一次的情况。

场景法的测试用例设计：设计测试用例覆盖所有场景。

场景法步骤总结三步曲：

α 设计场景：通过用例的主事件流和备选事件流的组合给出不同的场景。

β 设计测试用例标准覆盖场景。

γ 根据测试用例标准给出具体的测试数据。

场景法应用场合：一般用于功能测试,基本上所有的功能流程测试都会用到；业务复杂时，针对关键业务场景进行业务的流程测试。

1. 黑盒测试策略
2. 针对功能具体页面中具体的输入域进行细化测试，采用等价类、边界值；
3. 用静态检查法检查按钮、链接、内容、图片等；
4. 针对每个功能的流程，采用场景法，进行该功能（用例）的全方位流程测试；
5. 如果是多个条件的组合可以采用判定表或因果图法；
6. 在任何情况下都必须使用边界值分析方法；
7. 用错误推测法再追加一些测试用例。
8. 白盒测试
9. 控制流测试（Control Flow Testing）

逻辑分支覆盖法：

逻辑覆盖是以程序内部的逻辑结构为基础的设计测试用例的技术。这一方法要求测试人员对程序的逻辑结构有清楚的了解，甚至要能掌握源程序的所有细节。

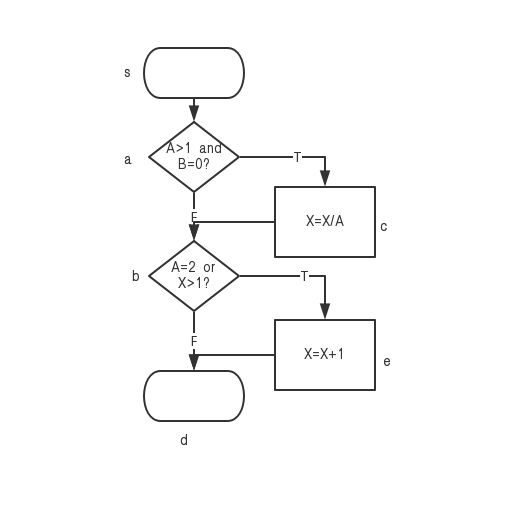
由于覆盖测试的目标不同，逻辑覆盖又可分为：

（共5种）

α 语句覆盖

语句覆盖就是设计若干个测试用例，运行被测程序，使得每一可执行语句至少执行一次。

这种覆盖又称为点覆盖，它使得程序中每个可执行语句都得到执行，但它是最弱的逻辑覆盖，效果有限，必须与其它方法交互使用。



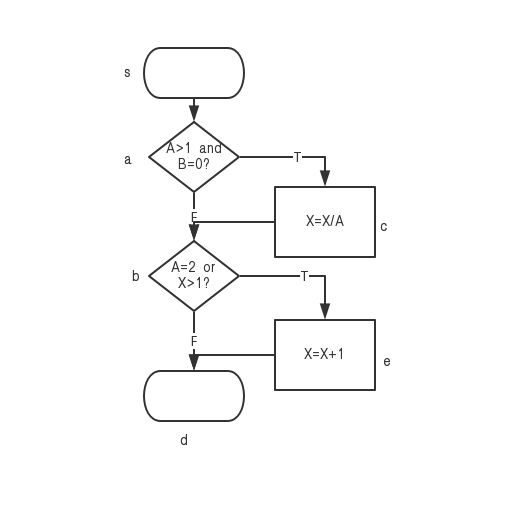
|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 输入（A, B, X） | 路径 | 输出（X） |
| 1 | 2, 0, 4 | sacbed | 3 |

β 判定覆盖

判定覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的取真分支和取假分支至少经历一次。

判定覆盖又称为分支覆盖。

判定覆盖只比语句覆盖稍强一些，但实际效果表明，只是判定覆盖，还不能保证一定能查出在判断的条件中存在的错误。因此，还需要更强的逻辑覆盖准则去检验判断内部条件。



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 测试用例 | 输入（A, B, X） | A>1 and B=0 | A=2 or X>1 | 执行路径 | 预期结果（X） |
| 1 | 2, 0, 3 | T | T | sacbed | 2.5 |
| 2 | 1, 0, 1 | F | F | sabd | 1 |

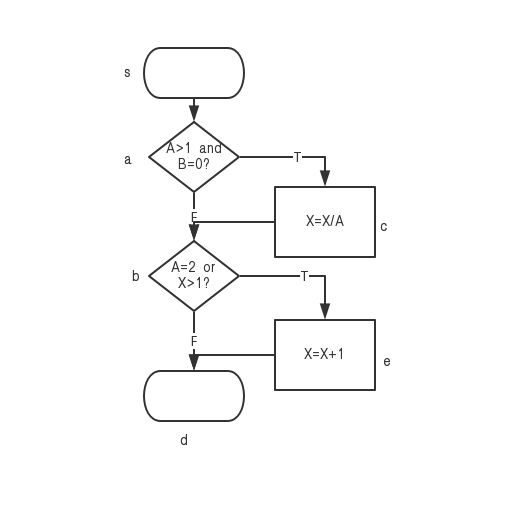
γ 条件覆盖

条件覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的每个条件的可能取值至少执行一次。

条件覆盖通常比判定覆盖强，因为它使判定表达式中每个条件都取到了两个不同的结果，判定覆盖却关心整个判定表达式的值。但也可能有相反的情况：虽然每个条件都取到了不同值，但判定表达式却始终只取一个值。

条件覆盖不一定包含判定覆盖；判定覆盖也不一定包含条件覆盖。

为解决这一矛盾，需要对条件和判定兼顾。

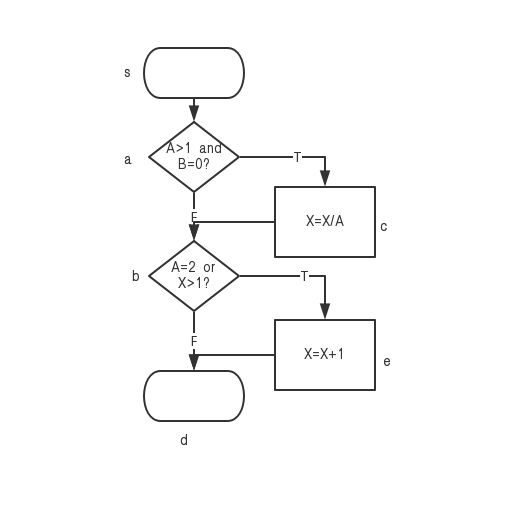


|  |  |
| --- | --- |
| 编号 | 条件 |
| 1 | A>1 |
| 2 | B=0 |
| 3 | A=2 |
| 4 | X>1 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试用例编号 | 输入（A, B, X） | 通过路径 | 各条件真假 | 期望输出（X） |
| 1 | 2, 0, 4 | sacbed | TTTT | 3 |
| 2 | 1, 1, 0 | sabd | FFFF | 0 |

δ 判定-条件覆盖

判定/条件覆盖要求设计足够的测试用例，使得判定中每个条件的所有可能（真/假）至少出现一次，并且每个判定本身的判定结果（真/假）也至少出现一次。



|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 判定 | 编号 | 条件 |
| 1 | A>1 and B=0 | 1 | A>1 |
| 2 | A=2 or X>1 | 2 | B=0 |
|  |  | 3 | A=2 |
|  |  | 4 | X>1 |

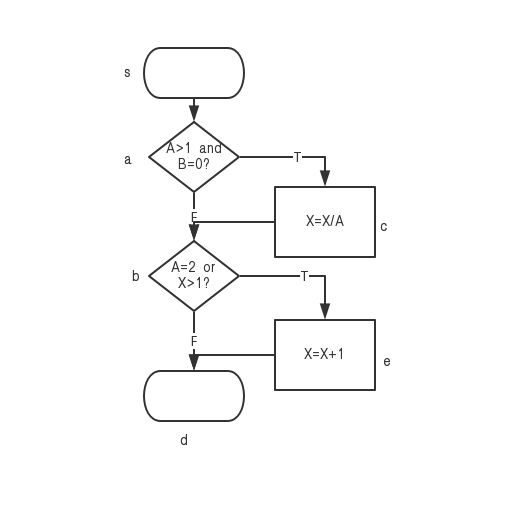
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 测试用例编号 | 输入（A, B, X） | 执行路径 | 条件真假 | 判定真假 | 期望输出（X） |
| 1 | 2, 0, 3 | sacbed | T, T, T, T | T, T | 2.5 |
| 2 | 1, 1, 1 | sabd | F, F, F, F | F, F | 1 |

ε 条件组合覆盖

条件组合覆盖就是设计足够的测试用例，运行被测程序，使得每个判断的所有可能的条件取值组合至少执行一次。

显然，满足“条件组合覆盖”的测试用例是一定满足“判定覆盖”、“条件覆盖”和“判定-条件覆盖”的。

这是一种相当强的覆盖准则，可以有效地检查各种可能的条件取值的组合是否正确。它不但可覆盖所有条件的可能取值的组合，还可覆盖所有判断的可取分支，但可能有的路径会遗漏掉。测试还不完全。



|  |  |  |  |
| --- | --- | --- | --- |
| 条件编号 | 条件 | 条件组合编号 | 条件组合 |
| 1 | A>1 | 1 | A>1, B=0 |
| 2 | B=0 | 2 | A>1, B≠0 |
| 3 | A=2 | 3 | A≤1, B=0 |
| 4 | X>1 | 4 | A≤1, B≠0 |
|  |  | 5 | A=2, X>1 |
|  |  | 6 | A=2, X≤1 |
|  |  | 7 | A≠2, X>1 |
|  |  | 8 | A≠2, X≤1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 输入（A, B, X） | 执行路径 | 条件真假 | 条件组合 | 覆盖分支 | 期望输出 |
| 2, 0, 4 | sacbed | T, T, T, T | 1, 5 | T, T | 3 |
| 2, 1, 1 | sabed | T, F, T, F | 2, 6 | F, T | 2 |
| 1, 0, 2 | sabed | F, T, F, T | 3, 7 | F, T | 3 |
| 1, 1, 1 | sabd | F, F, F, F | 4, 8 | F, F | 1 |

路径法：

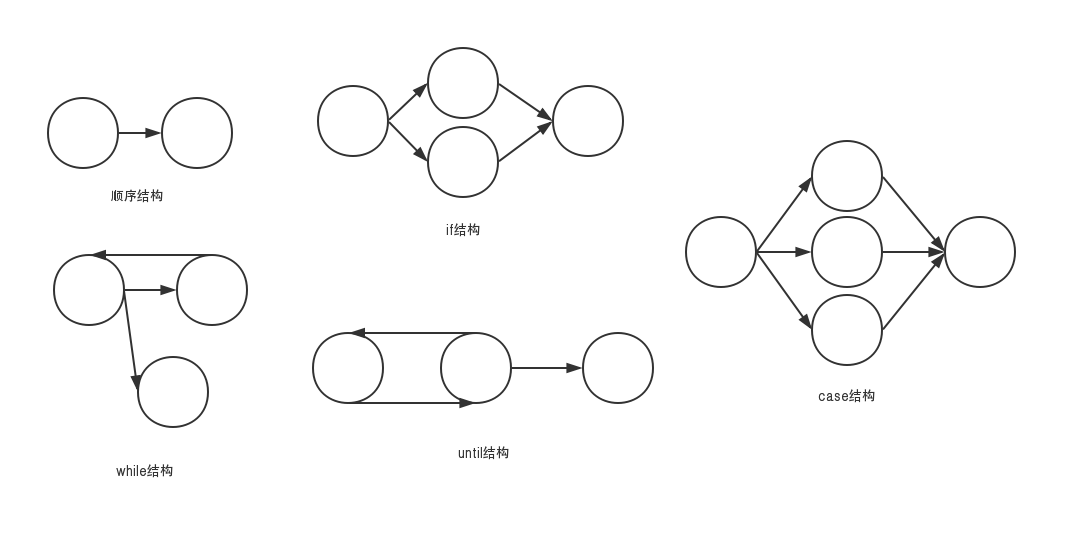
（共2种）

α 路径覆盖

路径测试就是设计足够的测试用例，覆盖程序中所有可能的路径。这是最强的覆盖准则。但在路径数目很大时，真正做到完全覆盖是很困难的，必须把覆盖路径数目压缩到一定限度。

β 基本（独立）路径测试法

流图（flow graph）：利用流图表示控制逻辑。



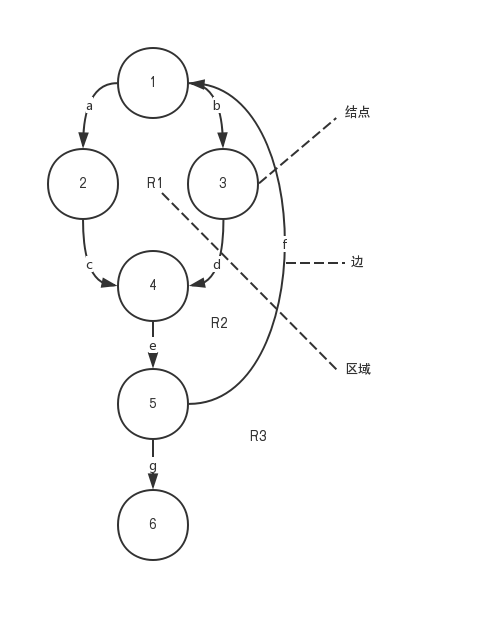
控制流图：

描述程序控制流的一种图示方式。

结点：一个圆圈，辨识单条或多条语句。

边：带箭头的线条，起始于一个结点，终止于一个结点。表示了控制流的方向。

区域：边和结点圈定的部分。



基本路径法：

为解决难题只得把覆盖的路径数压缩到一定限度内。例如，只考虑循环的两个可能性：重复零次和多于零次的重复；对于do-while循环，两种，循环体执行一次和多于一次。

使用线性独立路径。

独立路径：

定义：从入口到出口的路径，至少经历一个从未走过的边，这样形成的路径叫独立路径。

优点：减少路径数量；包含所有的边和结点。

缺点：简化循环结构。

基本路径测试步骤：

1. 根据程序的逻辑结构画出程序框图。
2. 根据程序框图导出流图。
3. 计算流图G的环路复杂度V(G)。
4. 确定只包含独立路径的基本路径集。
5. 设计测试用例。

程序框图→流图→基本路径→测试用例

基本路径数（独立路径数）的计算方法（列出公式）：

1. ，表示区域个数。
2. ，表示边数，表示结点数。
3. ，表示判定结点个数。
4. 数据流测试（Data Flow Testing）

数据流测试使用控制流图来讨论发生在数据上的不合理事情（如异常）。

对数据流异常的考虑会产生可弥补分支或语句测试与完备路径测试的差距的测试路径选择策略。

数据对象分类：

d：定义，创建，初始化

k：杀死，取消定义，释放

u：c：在表达式中使用；p：在谓词中使用

定义-使用路径（du-path）：关于变量v的定义-使用路径（记作du-path）是PATHS(P) 中的路径，使得对某个v∈V，存在定义和使用节点DEF(v, m)和USE(v, n)，使得m和n是该路径的最初和最终节点。

定义-使用关联（def-use associations）：一个三元组(x, d, u)，x变量名，d定义结点，u使用结点。从d到u没有对x的其它定义。

1. 白盒法测试步骤
2. 选择逻辑覆盖标准

显然只要程序不是太复杂，应该尽可能选择一种覆盖程度较深的标准，如条件组合覆盖。

1. 按照覆盖标准列出所有情况

所谓“情况”是指根据所选择的覆盖标准，列出能够满足此标准的可能的路径、条件等。

1. 设计测试用例

能够满足覆盖标准的测试用例往往不止一个，应该设计选择高效的测试用例，即是用最少的用例能够满足覆盖标准。特别要注意：测试用例应由两部分组成：输入数据（测试用例）和预期的输出结果（正确结果），另外不同的覆盖方式，需要加入覆盖的判定、覆盖的条件、条件组合、覆盖的路径或基本路径等等。

1. 验证分析运行结果与预期结果

将运行结果与测试用例中的预期结果进行比较分析，可以帮助找出错误。

1. 单元测试（Unit Testing）

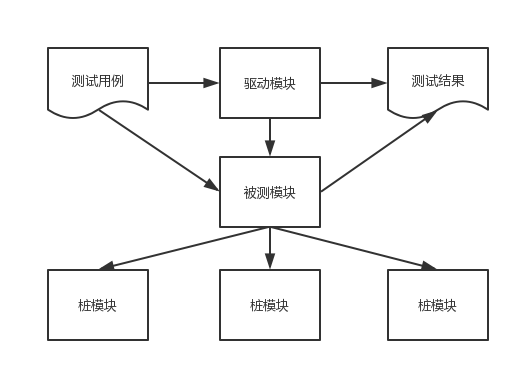
定义：单元测试是对软件基本组成单元进行的测试。

时机：一般在代码完成后由开发人员完成，QA人员辅助。（微软的例子是例外的，根据具体的测试人员的素质与技术有关）。

概念：单元是有明确的功能、性能定义、接口定义的软件设计最小单位——模块，概念已经扩展为组件了。不是函数或类中的方法！而是模块或者类。

单元测试的内容：

单元测试示意图：



单元测试策略：

黑盒：只使用程序说明书；测试对说明书的实现。

白盒：使用结构或程序的其他属性生成测试。

1. 静态白盒测试（Static white-box testing）

静态白盒测试是在不执行软件的条件下有条理地仔细审查软件设计、体系结构和代码，从而找出软件缺陷的过程（Static white-box testing is the process of carefully and methodically reviewing the software design, architecture, or code for bugs without executing it）。

不幸的是，静态白盒测试在实践中很少使用（不像动态白盒测试）。

正式审查（formal review）就是进行静态白盒测试的过程。正式审查可以是简单的一对一会议，也可以是详细严格的代码检验。正式审查由编程团队或测试团队组织。

正式审查的四个基本要素：

α 确定问题（identity problems）：找出软件中的问题，例如遗漏项目、错误等。

β 遵守规则（follow rules）：要审查的代码量、所花费的时间等。

γ 准备（prepare）：每个参与者都应该准备为审查做出贡献。

δ 编写报告（write a report）：总结出审查结果，使报告便于开发小组使用。

1. 非正式代码检查（Informal code inspections）
2. 同事审查（Peer reviews）

一个由程序员和（或）测试员组成的非正式小组充当审查者。参与者应当遵循正式审查的四个基本要素，即使这个审查是非正式的。

1. 走查（Walkthroughs）

一个更正式的过程，其中，代码作者将代码正式展示给一个由程序员和（或）测试员组成的小组。作者逐行阅读代码并解释原因，审查者聆听陈述并提问。参与者应当遵循正式审查的四个基本要素。

定义：采用讲解、讨论和模拟运行的方式进行的查找错误的活动。

注意：

α 引导小组成员在走查前通读设计和编码。

β 限时，避免跑题。

γ 发现问题适当记录，避免现场修改。

δ 检查要点是代码是否符合标准和规范，是否有逻辑错误。

1. 正式代码检查（Formal code inspections）

代码的表述者（presenter）不是代码作者。

其他参与者被称为检验员（inspector）。

有协调员（李红老师课件译为主持人，moderator）来保证检验过程遵守规则及审查顺利进行（assure that the rules are followed and the meeting runs smoothly）。

检验之后会出一份报告，然后程序员做出修改。如有必要，会再次检验。

正式检验可以非常有效地找出代码和设计中的缺陷并日趋流行。

1. 代码检查方式三部曲：走查、审查（review）和检验。
2. 代码审查检查表（Code review checklist）

数据声明错误（Data declaration errors）

数据引用错误（Data reference errors）

计算错误（Computation errors）

比较错误（Computation errors）

子例程参数（Subroutine parameter）

输入/输出错误（Input/Output errors）

控制流错误（Control flow errors）

1. xUnit框架

|  |  |  |
| --- | --- | --- |
| xUint框架概念 |  | 描述 |
| 测试 | TestMethod | 简单说，这些是您的测试。测试预期结果的逻辑，并报告未取得结果（如果有）。请将它看做您的“方法”。 |
| 测试装置 | TestClass | 针对大量测试的一个逻辑分组。请将它看做您的“类”。 |
| 测试套件 | 测试列表 \*\* | 针对大量测试装置的一个逻辑分组。请将它看做您的“类库”。  注：不需要一个属性。 |
| 测试运行器 |  | |

JUnit(Java), CppUnit(C++), NUnit(.Net)

1. JUnit

JUnit是一个开发源代码的Java测试框架，用于编写和运行可重复的测试。是用于单元测试框架体系xUnit的一个实例（用于Java语言）。

JUnit最初是由Erich Gamma和Kent Beck（XP和Refactor的先驱之一）编写的。

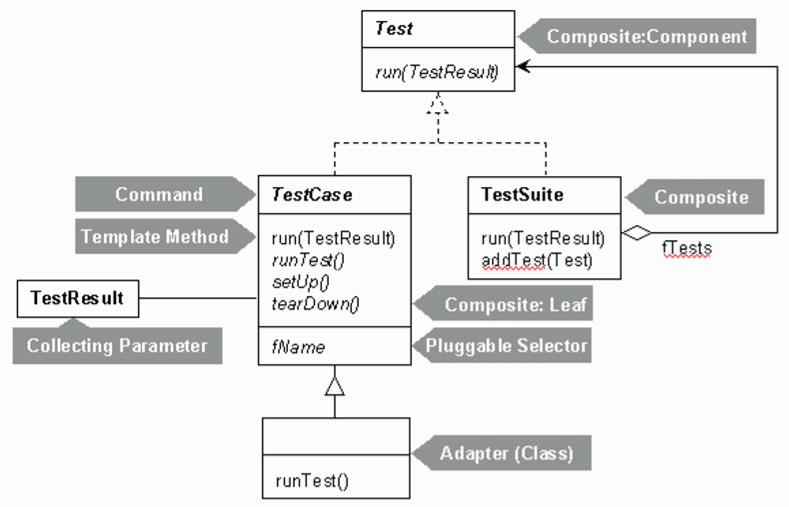
需要说明的是JUnit一般是用来进行单元测试的，因此需要了解被测试代码的内部结构（即所谓的白盒测试），另外JUnit是在XP编程和重构（refactor）中被极力推荐使用的工具，因为在实现自动单元测试的情况下可以大大的提高开发的效率。

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

JUnit Platform是在JVM上启动测试框架的基础。

JUnit Jupiter是JUnit 5扩展的新的编程模型和扩展模型，用来编写测试用例。Jupiter子项目为在平台上运行Jupiter的测试提供了一个TestEngine（测试引擎）。

JUnit Vintage提供了一个在平台上运行JUnit 3和JUnit 4的TestEngine。



JUnit框架提供测试类（TestCase）、测试套件（TestSuite）、测试方法、测试运行器。满足xUnit框架。

JUnit4：

JUnit4使用 Java 5中的注解（annotation），以下是JUnit4常用的几个annotation介绍：

@Before：初始化方法，每开始一个新的测试都要执行一次，setup()

@After：释放资源，每完成一个测试都要执行一次，tearDown()

@Test：测试方法，对应：testAdd()

@Ignore：忽略的测试方法

@BeforeClass：针对所有测试，只执行一次，且必须为static void

@AfterClass：针对所有测试，只执行一次，且必须为static void

JUnit5：

以下是将现有JUnit4测试迁移到JUint Jupiter时必须注意的事项。

org.junit.jupiter.api包中的注解。

org.junit.jupiter.api.Assertions类中的断言。

org.junit.jupiter.api.Assumptions类中的假设。

@Before和@After已经不存在；取而代之的是@BeforeEach和@AfterEach。

@BeforeClass和@AfterClass已经不存在；取而代之的是@BeforeAll和@AfterAll。

@Ignore已经不存在；取而代之的是@Disabled。

@Category已经不存在；取而代之的是@Tag。

@RunWith已经不存在；取而代之的是@ExtendWith。

@Rule和@ClassRule已经不存在；取而代之的是@ExtendWith。

1. 单元测试策略

什么时候测试？XP开发理论讲究TDD，即测试驱动开发，先编写测试代码，再进行开发。

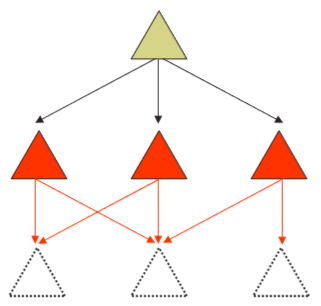
1. 先编写产品的框架，是指先编写类、方法空的实现，需要确定接口。
2. 编译通过然后编写测试类，针对产品类的功能编写测试用例，这时方法名、参数表、返回类型都应该确定下来了。
3. 然后编写产品类的代码，每写一个功能点都运行测试，随时补充测试用例。
4. 编写的测试代码以后需修改的可能性比较小。
5. 集成测试（Integration Testing）

定义：将单元组装起来再进行测试，以检查这些单元之间的接口是否存在问题，如：数据丢失、模块间相互影响、组合后不能实现主功能等。

集成策略（Integration Strategies）：

1. 自顶向下（Top-down）

从最顶层的模块开始；为低层模块使用桩（stub）；每层都完成后，用下一层的模块代替桩。



评价（Issues）：

优点：总是有一个顶层系统；桩可以基于接口说明书来编写。

缺点：可能太晚才发现性能问题；桩可能很贵。

1. 自底向上（Bottom-up）

从最底层的模块开始；为最顶层使用驱动模块（driver）；有一层完成就用上一层模块代替驱动。

评价：

优点：原始功能（Primitive functions）得到最好的测试；驱动通常很便宜。

缺点：最后才有一个完整的系统。

1. 混合策略

对软件结构中较上层，使用的是“自顶向下”法；对软件结构中较下层，使用的是“自底向上”法，两者相结合。

1. 大爆炸（Big bang）

所有组件（component）就绪后一次性把所有组件集成起来测试。

评价：避免了脚手架的花销【avoid cost of scaffolding (stubs or drivers)】；不易发现错误（does not provide any locality for finding faults）。

1. 关键先行（Critical-first）

先集成最重要的模块，然后在增加剩余的部分。

问题：保证最重要的组件能首先工作。

1. Function-at-a-time

几种集成测试方法性能的比较：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 自顶向下 | 自底向上 | 混合策略 | 大爆炸 |
| 集成 | 早 | 早 | 早 | 晚 |
| 基本程序能工作时间 | 早 | 晚 | 早 | 晚 |
| 需要驱动程序 | 否 | 是 | 是 | 否 |
| 需要桩程序 | 是 | 否 | 是 | 否 |
| 计划与控制 | 难 | 容易 | 中 | 容易 |

集成测试的模式：渐增式测试模式与非渐增式测试模式。

1. 非渐增式测试模式：先分别测试每个模块，再把所有模块按设计要求放在一起结合成所要的程序，如大棒模式。
2. 渐增式测试模式：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，测试完以后再把下一个应该测试的模块结合进来测试。
3. 测试环境的五要素

测试环境的基本要素是：软件和硬件。

在基本要素的基础上派生出：网络环境、数据准备、测试工具三要素。

1. 系统测试（System Testing）

系统测试的目标：增强对系统整合成一个整体的信心；保证满足功能性需求和非功能性需求。

定义：在集成测试结束之后，依据系统的需求规格说明书和产品功能说明书对系统的整体功能进行的全面测试，称为功能测试。

采用黑盒测试方法。

功能测试的目的和内容：

（共12条）

1. 程序安装、启动正常，有相应的提示框、错误提示等。
2. 每项功能符合需求。
3. 系统的界面清晰、美观。
4. 菜单、按钮操作正常、灵活，能处理一些异常操作。
5. 能接受正确的数据输入，对异常数据的输入可以进行提示、容错处理等。
6. 数据的输出结果准确，格式清晰，可以保存和读取。
7. 功能逻辑清楚，符合使用者习惯。
8. 系统的各种状态按照业务流程而变化，并保持稳定。
9. 支持各种应用的环境。
10. 能配合多种硬件周边设备。
11. 软件升级后，能继续支持旧版本的数据。
12. 与外部应用系统的接口有效。
13. 回归测试

定义：对修正缺陷后的软件进行再次的测试，不仅测试被修复的软件缺陷是否已经解决，还要测试软件旧有的功能与非功能是否满足要求。

目的：

1. 所做的修改达到了预定的目的，如错误得到了改正，新功能得到了实现，能够适应新的运行环境等；
2. 不影响软件原有功能的正确性。

回归测试的方法：

（共4种）

1. 再测试全部用例。
2. 基于风险选择测试：测试关键的、重要的、可疑的，跳过非关键的、稳定的。
3. 基于操作剖面选择测试。
4. 再测试修改的部分。
5. 冒烟测试

关于冒烟测试，冒烟测试就是在每日build建立后，对系统的基本功能进行简单的测试。这种测试强调功能的覆盖率。简单的说，就是先通过最基本的测试，先保证系统能跑的起来，减少测试部门时间的浪费。

1. 非功能性测试（Non-functional test）
2. 压力测试（Stress test）

压力测试是在一种需要反常数量、频率或资源的方式下，执行可重复的负载测试或强度测试，以检查程序对异常情况的抵抗能力，找出性能瓶颈。

α 稳定性压力测试

稳定性压力测试，也叫可靠性测试（reliability testing），或疲劳测试，是指连续运行被测系统，检查系统运行时的稳定程度。

我们通常用平均故障间隔时间（Mean Time Between Failure, MTBF）来衡量系统的稳定性，MTBF越大，系统的稳定性越强。

稳定性测试的方法也很简单，即采用24\*7（24小时\*7天）的方式让系统不间断运行，至于具体运行多少天，视项目的实际情况而定。

β 破坏性压力测试

通常是指持续不断的给被测系统增加压力，直到将被测系统压垮为止（让问题与薄弱环节快速暴露出来，找出瓶颈）。用来测试系统所能承受的最大压力。

压力测试工具——LoadRunner：

α Virtual User Generator创建脚本：创建脚本，选择协议；录制脚本；编辑脚本；检查修改脚本是否有误。

β 中央处理器（Controller）来调度虚拟用户：创建Scenario，选择脚本；设置机器虚拟用户数；设置Schedule；如果模拟多机测试，设置IP Spoofer。

γ 运行脚本：分析Scenario。

δ 分析测试结果。

1. 容量测试（Capacity test）

容量测试目的是通过测试预先分析出反映软件系统应用特征的某项指标的极限值（如最大并发用户数、数据库记录数等），系统在其极限值状态下还能保持主要功能正常运行。容量测试还将确定测试对象在给定时间内能够持续处理的最大负载或工作量。

1. 性能测试（Performance test）

通过测试确定系统运行期间的性能表现与性能数据，得到如：CPU使用的效率、运行速度、响应时间、占有系统资源等方面的系统数据。

性能测试与瓶颈分析关键步骤：

步骤一：性能测试与数据收集。

步骤二：性能瓶颈分析。

步骤三：性能调优解决方案。

1. 安全测试（Security test）

安全性测试是检查系统对非法侵入的防范能力。安全测试期间，测试人员假扮非法入侵者，采用各种办法试图突破防线。

系统安全设计的准则是，使非法侵入的代价超过被保护信息的价值，此时非法侵入者已无利可图。

1. 容错测试（Recovery test）

容错性测试是检查软件在异常条件下自身是否具有防护性的措施或者某种灾难性恢复的手段。如当系统出错时，能否在指定时间间隔内修正错误并重新启动系统。

1. 验收测试（Acceptance Test）

定义：在软件产品完成了功能测试和系统测试之后、产品发布之前所进行的软件测试活动。

它是技术测试的最后一个阶段，也称为交付测试。

验收测试的过程和主要内容：

前提：系统或软件产品已通过了系统测试的软件系统。

测试内容：主要包括易用性测试、兼容性测试、安装测试、文档（如用户手册、操作手册等）测试等几个方面的内容。

必须编写正式的、单独的验收测试报告。

验收测试必须在实际用户运行环境中进行。

由用户和测试部门共同执行。如公司自开发产品，应由测试人员，产品设计部门，市场部门等共同进行。

其他验收方法：

α测试是指软件开发公司组织内部人员模拟各类用户对即将面市软件产品（称为α版本）进行测试，试图发现错误并修正。

经过α测试调整的软件产品称为β版本。紧随其后的β测试是指软件开发公司组织各方面的典型用户在日常工作中实际使用β版本，并要求用户报告异常情况、提出批评意见。然后软件开发公司再对β版本进行改错和完善。

1. 向前和向后兼容
2. 向后兼容是指可以使用软件的以前版本。
3. 向前兼容指的是可以使用软件的未来版本。
4. 可安装性测试
5. 系统软件安装
6. 应用软件安装
7. 服务器的安装
8. 客户端的安装
9. 产品升级安装

等等

1. 网站测试（Website Testing）

Web测试的内容：

网站文本（Website text）

网站超链接（Website hyperlinks）

网站图像（Website graphics）

网站表单（Website forms）

超文本标记语言（HTML）

易用性测试（Usability testing）

动态内容（Dynamic content）

数据库驱动网页（Database-driven web pages）

服务器性能、负载和安全（Server performance and loading and security）

兼容性测试（compatibility testing）

1. 报告所发现的软件缺陷

（以下标题均为目录）

1. 软件缺陷的描述

准确有效的定义和描述软件缺陷，可以使软件缺陷得以快速修复，节约了软件测试项目的成本和资源，提高产品质量。

α 软件缺陷的基本描述

软件缺陷的描述是软件缺陷报告中测试人员对问题的陈述的一部分并且是软件缺陷报告的基础部分。同时，软件缺陷的描述也是测试人员就一个软件问题与开发小组交流的最初且最好的机会。

一个好的描述，需要使用简单的、准确的、专业的语言来抓住缺陷的本质。

β 软件缺陷属性

软件缺陷属性包括缺陷标识、缺陷类型、缺陷严重程度、缺陷优先级、缺陷产生可能性、缺陷状态、缺陷起源、缺陷来源和缺陷原因。

1. 软件缺陷相关的信息

α 软件缺陷的图片、记录信息

β 分离和再现软件缺陷

1. 软件缺陷的处理和跟踪

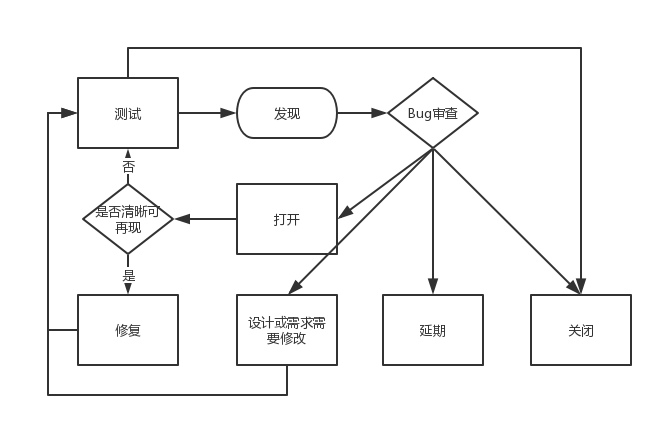
软件缺陷跟踪管理是测试工作的一个重要部分，测试的目的是为了尽早发现软件系统中的缺陷，而对软件缺陷进行跟踪管理的目的是确保每个被发现的缺陷都能够及时得到处理。

软件测试过程简单说就是围绕缺陷进行的，对缺陷的跟踪管理，一般而言需要达到以下目标：

α 确保每个被发现的缺陷都能够被解决，“解决”的意思不一定是被修正，也可能是其他处理方式（例如，延迟到下一个版本中修正或者由于技术原因不能被修正），总之，对每个被发现的BUG的处理方式必须能够在开发组织中达到一致。

β 统计数据：收集缺陷数据并根据缺陷趋势曲线识别测试处于测试过程中的哪个阶段；决定测试过程是否结束，通过缺陷趋势曲线来确定测试过程是否结束是常用并且较为有效的一种方式；收集缺陷数据并在其上进行数据分析，作为组织过程改进的财富。

复杂的软件缺陷生命周期：



缺陷管理工具：

α 常见免费缺陷管理工具：Mantis

β 付费缺陷管理工具：Rational ClearQuest和Testdirector

缺陷跟踪的方法和图表：

1. 测试报告

测试报告是把测试的过程和结果写成文档，并对发现的问题和缺陷进行分析，为纠正软件的存在的质量问题提供依据，同时为软件验收和交付打下基础。

测试报告是测试阶段最后的文档产出物，优秀的测试经理应该具备良好的文档编写能力，一份详细的测试报告包含足够的信息，包括产品质量和测试过程的评价。

测试报告的内容可以总结为以下目录：

1. 首页
2. 引言（目的、背景、缩略语、参考文献）
3. 测试概要（测试方法、范围、测试环境、工具）
4. 测试结果与缺陷分析（功能、非功能）
5. 测试结论与建议（项目概况、测试时间、测试情况、结论性能汇总）
6. 附录（缺陷统计）
7. 软件维护
8. 定义：软件维护是指软件系统交付使用以后， 为了改正错误或满足新的需要而修改软件的过程。
9. 原因：一般来说，要求进行维护的原因大致有以下几种：改正程序中的错误和缺陷；改进设计以适应新的软、硬件环境；增加新的应用范围。
10. 类型：改正性维护、适应性维护、完善性维护和预防性维护。

α 改正性维护：

在软件交付使用后，因开发阶段的问题以及测试得不彻底、不完全，必然会有部分隐藏的错误遗留到运行阶段。

为了识别和纠正软件错误、改正软件功能、非功能（性能）上的缺陷、排除实施中的误使用，应当进行的诊断和改正错误的过程就叫做改正性维护。

β 适应性维护：

在使用过程中，外部环境（新的硬、软件配置）和数据环境（数据库、数据格式、数据输入/输出方式、数据存储介质）可能发生变化。

为使软件适应这种变化，而去修改软件的过程就叫做适应性维护。

γ 完善性维护：

在软件的使用过程中，用户往往会对软件提出新的功能与性能要求。

为了满足这些要求，需要修改或再开发软件，以扩充软件功能、增强软件性能、改进加工效率、提高软件的可维护性。

这种情况下进行的维护活动叫做完善性维护。

δ 预防性维护：

预防性维护即软件再工程，是为了提高软件的可维护性、可靠性等，为以后进一步改进软件打下良好基础。

采用先进的软件工程方法对需要维护的软件或软件中的某一部分（重新）进行设计、编制和测试，称为预防性维护。

各种维护类型和维护工作量的比例

1. 结构化维护VS非结构化维护

软件的开发过程对软件的维护产生较大的影响。

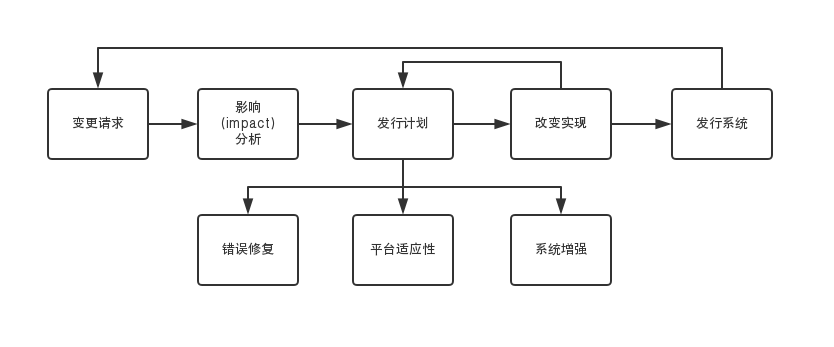
如果采用软件工程的方法进行软件开发，保证每个阶段都有完整且详细的文档，这样维护会相对容易，被称为结构化的维护。

反之，如果不采用软件工程方法开发软件，软件只有程序而欠缺文档，则维护工作变得十分困难，被称为非结构化的维护。

1. 影响软件维护工作量的因素

系统的大小、程序设计语言、系统年龄、先进的软件开发技术和其他因素

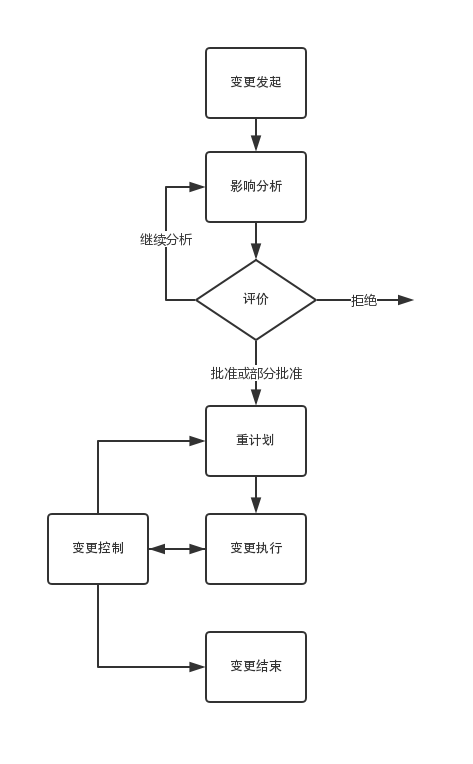
1. 系统演化过程



改变实现：修改软件需求说明、修改软件设计、设计评审、对源程序做必要的修改、单元测试、集成测试（回归测试）、确认测试和软件配置评审等。

1. 整体变更控制

变更流程：



1. 维护的流程（从上往下）
2. 软件可维护性

软件可维护性是指纠正软件系统出现的错误和缺陷，以及为满足新的要求进行修改、扩充或压缩的容易程度。

目前广泛使用的是用如下的七个特性来衡量程序的可维护性：可理解性、可靠性、可测试性、可修改性、可移植性、可使用性和效率。

而且对于不同类型的维护，这七中特性的侧重点也不相同。

α 可理解性

可理解性表明人们通过阅读源代码和相关文档，了解程序功能及其如何运行的容易程度。

β 可靠性

可靠性表明一个程序按照用户的要求和设计目标，在给定的一段时间内正确执行的概率。

γ 可测试性

可测试性表明论证程序的正确性的容易程度。程序越简单，证明其正确性就越容易。而且设计合用的测试用例，取决于对程序的全面理解。

一个可测试的程序应当是可理解的、可靠的、简单的。

δ 可修改性

可修改性表明程序容易修改的程度。

一个可修改的程序应当是可理解的、通用的、灵活的、简单的。

ε 可移植性

可移植性表明程序转移到一个新的计算环境的可能性的大小。或者它表明程序可以容易地、有效地在各种各样的计算环境中运行的容易程度。

ζ 可使用性

从用户观点出发，可使用性定义为程序方便、实用、以及易于使用的程度。

η 效率

效率表明一个程序能执行预定功能而又不浪费及其资源的程度。

这些机器资源包括内存容量、外存容量、通道容量和执行时间。

1. 维护的文档

维护合同、维护方案、维护手册、维护申请表、维护记录和维护报告

参考资料

本笔记参考了李红老师课件、Ron Patton的《软件测试（第一版）》、《软件测试（第二版）》以及Stephen Brown、Joe Timoney、Tom Lysaght和叶德仕的《软件测试：原理与实践》（课本）。