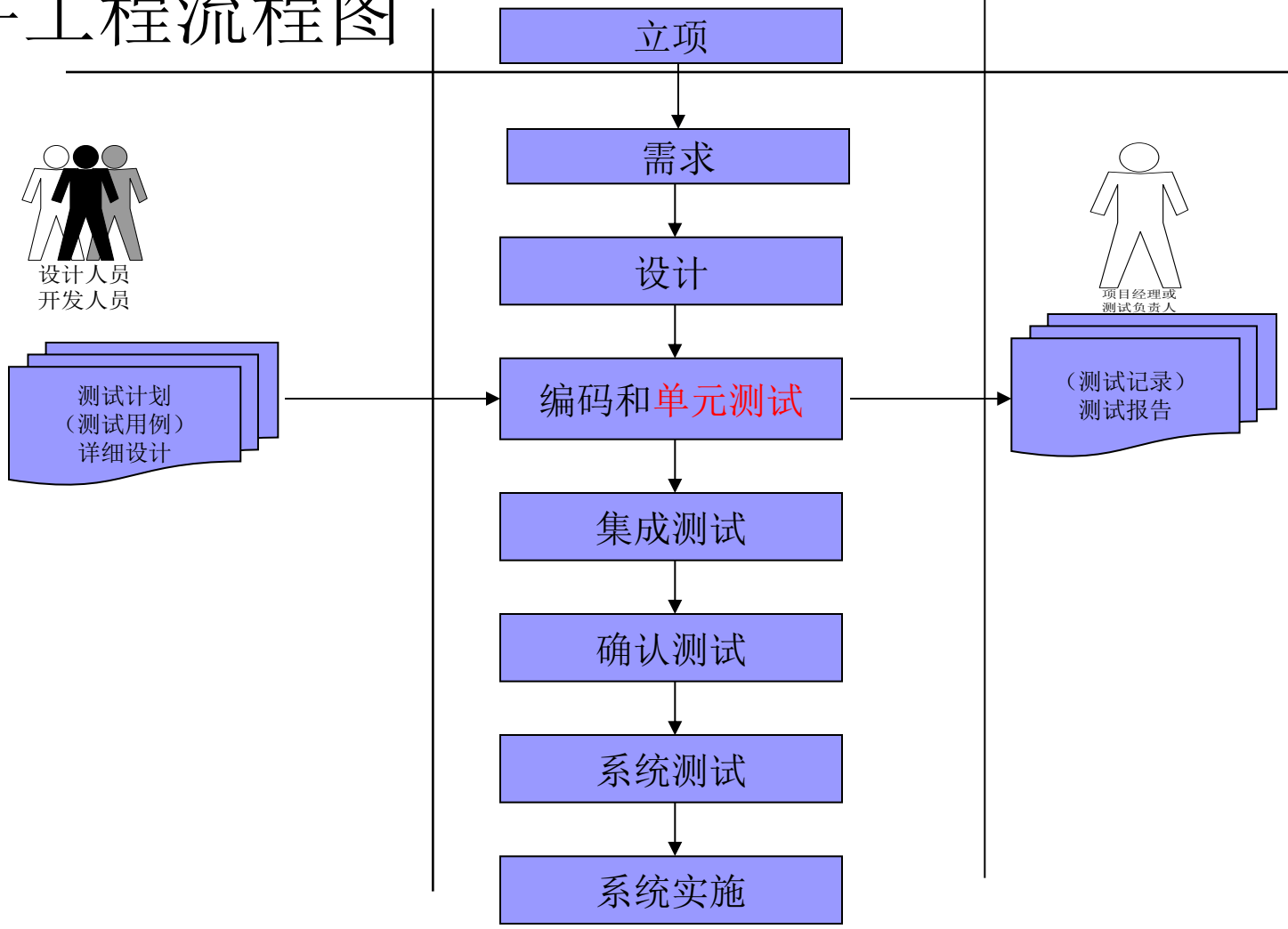
测试培训-单元测试

软件工程流程图


立项
需求
设计
编码和单元测试
集成测试
确认测试
系统测试
系统实施
设计人员
开发人员
测试计划
（测试用例）
详细设计
项目经理或
测试负责人
（测试记录）
测试报告

概述

- 单元测试的目标:确保模块被正确地编码。
- 由谁去做:通常由编程人员执行。
- 怎样去测试:功能测试可以用黑匣测试方法，代码测试可用白匣测试方法。
- 什么时候可以停止:当程序员感到代码没有缺陷时。
- 记录:通常没有记录。

名词解释

- 模块：程序组成部分的最小单元。
- 黑盒测试：黑盒测试也称功能测试或数据驱动测试，它是在已知产品所应具有的功能，通过测试来检测每个功能是否都能正常使用，在测试时，把程序看作一个不能打开的黑盒子，在完全不考虑程序内部结构和内部特性的情况下，测试者在程序接口进行测试，它只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数据而产生正确的输出信息，并且保持外部信息（如数据库或文件）的完整性。黑盒测试方法主要有等价类划分、边值分析、因—果图、错误推测等，主要用于软件确认测试。
- 白盒测试：白盒测试也称结构测试或逻辑驱动测试，它是知道产品内部工作过程，可通过测试来检测产品内部动作是否按照规格说明书的规定正常进行，按照程序内部的结构测试程序，检验程序中的每条通路是否都有能按预定要求正确工作，而不顾它的功能，白盒测试的主要方法有逻辑驱动、基路测试等，主要用于软件验证。

软件测试

- 什么是软件测试：
  - 定义1：软件测试是为了发现错误而执行程序的过程。
  - 定义2：软件测试是根据软件开发各阶段的规格说明和程序的内部结构而精心设计的一批测试用例（即输入数据和与其输出结果），并利用这些测试用例去运行程序，以发现程序错误的过程。
- 软件测试的对象：
  - 软件测试不等于程序测试
  - 软件测试贯穿于软件定义和开发的整个周期。因此，需求分析、概要设计、详细设计，以及程序编码等各阶段所得到的文档，包括需求规格说明书、概要设计说明书、详细设计规格说明书以及源程序，都是测试的对象。

软件测试

- 软件测试分类
  - 按测试用例的设计方法，软件测试分为白盒测试和黑盒测试。
  - 按测试策略和过程，软件测试分为单元测试、集成测试、确认测试和系统测试。
  - 按软件系统工程，测试是软件质量保证的最后的一关。
  - 本培训文档主要内容将围绕软件开发编码阶段运用的单元测试过程加以描述和讨论

单元测试

- 为什么要进行单元测试（一）

1 单元测试是不是太浪费时间了？

- 不经过单元测试，直接进入集成测试，系统正常工作的可能性非常低，大量的时间被花费在跟踪那些简单的Bug上，会导致集成为一个系统时增加额外的工期。
- 编写完整计划的单元测试和编写实际的代码所花费的精力大致相同。但是，一旦完成了这些单元测试工作，很多Bug将被纠正，在确信他们手头拥有稳定可靠的部件的情况下，开发人员能够进行更高效的系统集成工作，这才是真正意义上的进步。
- 调试人员的不受控和散漫的工作方式只会花费更多的时间而取得很少的好处。

2 单元测试仅仅是为了证明这些代码作了什么吗？

- 这是那些没有首先为每个单元编写一个详细设计文档而直接跳到编码阶段的开发人员提出的一条普遍的抱怨。这样的测试完全基于已经写好的代码，这无法证明任何事情。
- 单元测试基于详细设计文档，这样的测试可以找到更多的代码错误，甚至是详细设计的错误。
- 因此，高质量的单元测试需要高质量的详细设计文档。

为什么要进行单元测试（二）

3 我是一个很棒的程序员，是不是可以不进行单元测试呢？

- 每个人都可能犯错误。
- 真正的完整的系统往往是非常复杂的，不能寄希望于没有进行广泛的测试和Bug修改过程就可以正常工作。

4 有集成测试就够了，集成测试将会抓住所有的Bug。

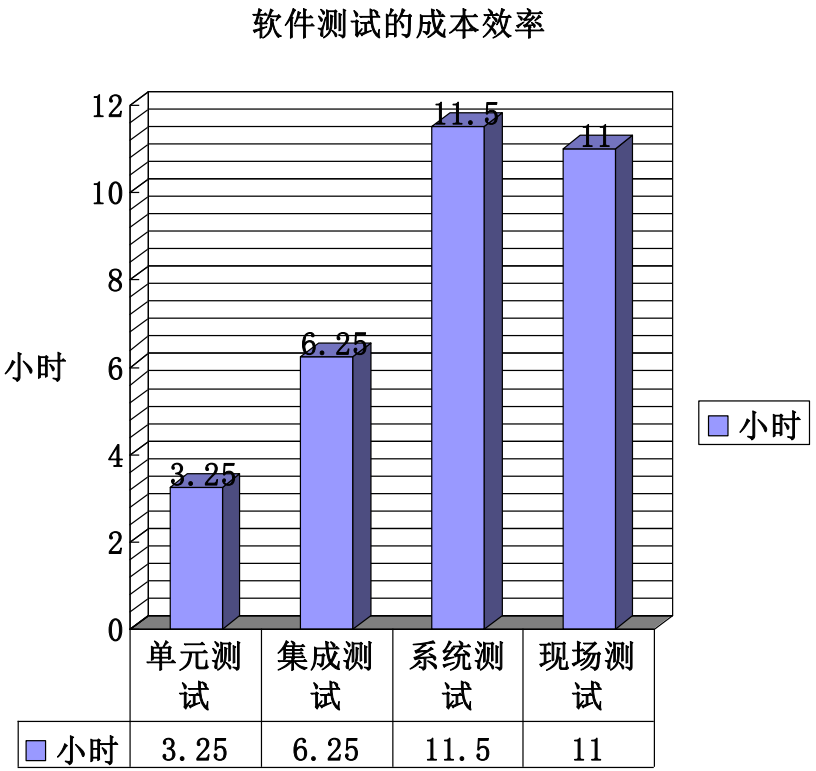
- 系统规模愈来愈大，复杂度愈来愈高，没有单元测试，开发人员很可能会花费大量的时间仅仅是为了使该系统能够运行。
- 任何实际的测试方案都无法执行。
- 在系统集成阶段，对单元功能全面测试的负载程度远远的超过独立进行的单元测试过程。
- 最后的结果是测试将无法达到它所应该有的全面性，一些缺陷将被遗漏，并且很多Bug将被忽略过去。

单元测试

为什么要进行单元测试（三）

5 单元测试的成本效率不高？

- 无论什么时候做出修改都要进行完整的回归测试。
- 在生命周期中尽早的对产品进行测试将使效率和质量得到最好的保证
- Bug修改越晚，费用就越高，单元测试是一个在早期抓住Bug的机会。
- 相比后阶段的测试，单元测试的创建更简单，维护更容易，并且可以更方便的进行重复。
- 从全程的测试费用来考虑，相比复杂且旷日持久的集成测试，或是不稳定的系统，单元测试所需的费用是最低的。



单元测试

- 单元测试的重要性
  - 一个尽责的单元测试方法将会在产品开发的某个阶段发现很多的Bug，并且修改它们的成本也很低。
  - 系统开发的后期阶段，Bug的检测和修改将会变得更加困难，并要消耗大量的时间和开发费用。
  - 无论什么时候做出修改都要进行完整的回归测试，在生命周期中尽早的对产品代码进行测试将是效率和质量得到最好的保证。
  - 在提供了经过单元测试的情况下，系统集成过程将会大大的简化。开发人员可以将精力集中在单元之间的交互作用和全局的功能实现上，而不会陷入充满很多Bug的单元之中不能自拔。
  - 使测试工作的效率发挥到最大化的关键在于选择正确的测试策略，这包含了完全的单元测试的概念，以及对测试过程的良好的管理，还有适当的使用好工具来支持测试过程。

单元测试

- 模块接口的测试项目
  - 调用被测模块的输入参数与模块的形式参数在个数、属性、顺序上是否匹配。
  - 被测模块调用子模块时，它输入给子模块的参数与子模块中的形式参数在个数、属性、顺序上是否匹配。
  - 是否修改了只做输入用的形式参数。
  - 输出给标准函数的参数在个数、属性、顺序上是否正确。
  - 全局变量的定义在各模块中是否一致。
  - 限制是否通过形式参数来传送。

单元测试

- 局部数据结构测试：
  - 模块的局部数据结构是最常见的错误来源，应设计测试用例以检查以下各种错误。
    - 检察不正确或不一致的数据类型说明。
    - 使用尚未赋值或尚未初始化的变量。
    - 错误的初始值或错误的默认值。
    - 变量名拼写错误或书写错误。
    - 不一致的数据类型。

单元测试

➢ 路径测试

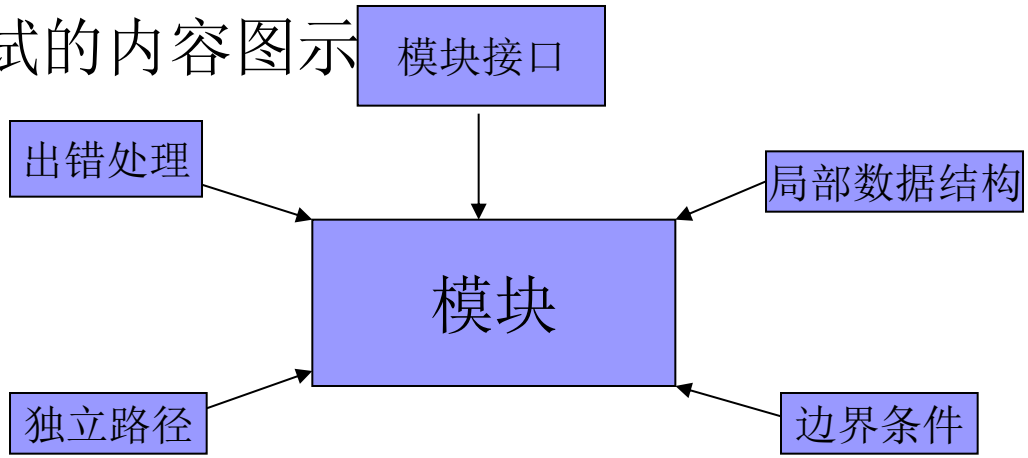
- 对基本执行路径和循环进行测试会发现大量的错误。
- 测试方法：白盒测试和黑盒测试相结合。
- 常见的不正确计算有：
  1. 运算的优先次序不正确或误解了运算的优先次序；
  2. 运算的方式错误（即运算的对象彼此在类型上不相容）；
  3. 算法错误；
  4. 初始化不正确；
  5. 运算精度不够；
  6. 表达式的符号表示不正确等
- 常见的比较和控制流错误有：
  1. 不同数据类型的比较；
  2. 不正确的逻辑运算符或优先次序；
  3. 因浮点运算精度问题而造成的两值比较不等；
  4. 关系表达式中不正确的变量和比较符；
  5. “差1错”，即不正确的多循环或少循环一次；
  6. 当遇到发散的迭代时不能终止的循环；
  7. 不适当的修改了循环变量等。

单元测试

- 错误处理测试
  - 概述：比较完善的模块设计要求能预见出错的条件，并设置适当的出错处理对策，以便在程序出错时，能对出错程序重新做安排，保证逻辑上的正确性。这种出错处理也是模块功能的一部分。
  - 出错处理模块有错误或缺陷的情况有：
    1. 出错的描述难于理解；
    2. 出错的描述不足以对错误定位和确定出错的原因；
    3. 显示的错误与实际的错误不符；
    4. 对错误条件的处理不正确；
    5. 在对错误进行处理之前，错误条件已经引起系统的干预等。

单元测试

- 边界测试
  - 概述：边界上出现错误是常见的，必须设计测试用例检查。
  - 出现边界问题的情况：
    1 在n次循环的第0次、1次、n次是否有错误；
    2 运算或判断中取最大最小值是否有 错误
    3 数据流、控制流中刚好等于、大于、小于确定的比较之时是否出现错误。
- 单元测试的内容图示



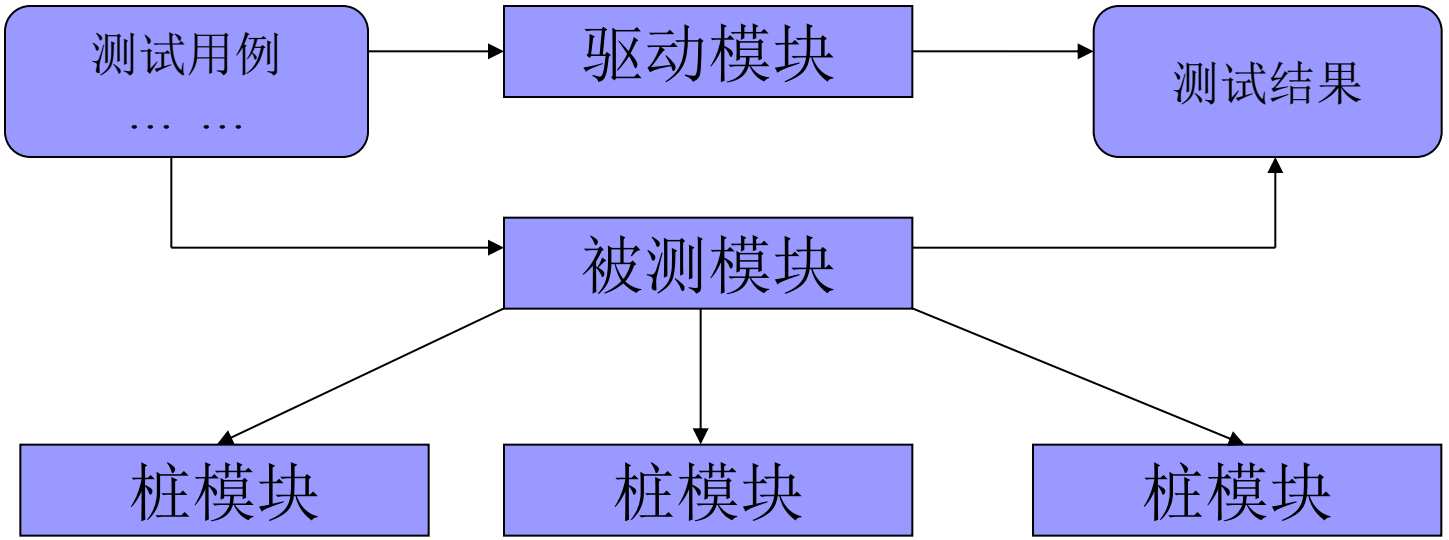
单元测试

- 何时进行单元测试：单元测试在编码阶段进行。
- 何时设计测试用例：在源程序代码编制完成、经过评审和验证、确认没有语法错误之后，就可以开始进行单元测试的测试用例设计。
- 用例设计的依据：软件设计文档。
- 对于一组输入，应该有预期的正确结果。
- 在单元测试时，如果模块不是独立的程序，需要辅助测试模块。辅助测试模块种类：

1 驱动模块（Driver）--所测模块的主程序。
   - 他接受测试数据，把这些数据传递给测试模块，最后在输出实测结果。
   - 当被测试模块能完成一定功能时，也可以不要驱动模块。

2 桩模块（Stub）：用来替代所测模块调用的子模块。

单元测试



单元测试

- 与单元测试相关联的开发活动
  - 代码走读（code review）

    代码走读主要是对程序的编程风格如注释、命名等以及编程时易出错的内容进行检查
  - 静态分析（Static Analysis）

    静态分析就是对模块的源代码进行研读，查找错误或收集一些度量数据，并不需要对代码进行编译和仿真运行。
  - 动态分析（Dynamic Analysis）

    动态分析就是通过观察代码运行时的动作，来提供执行跟踪、时间分析，以及测试覆盖度方面的信息。
  - 单元测试作为无错编码的一种辅助手段在一次性的开发过程中加以运用。

单元测试

- 单元测试的过程分为计划、设计、实现、执行、评审等几个步骤
  - 计划单元测试。确定测试所用资源（包括人力和设备资源），创建测试任务时间表。
  - 设计单元测试。设计单元测试模型，制定测试方案，确认并结构化测试过程。
  - 实现单元测试。参考测试模型和测试方案，制定具体的测试用例，创建可重用的测试脚本。
  - 执行单元测试。根据单元测试的方案、用例对单元进行测试，验证测试的结果并记录测试过程中出现的缺陷。
  - 评审单元测试。对单元测试的结果进行评审，主要从需求覆盖和代码覆盖的角度进行测试完备性的评估。

单元测试

- 分解软件，写测试需求
  - 反复检查并理解各种信息，和用户交流，理解他们的要求。可以按照以下步骤执行：

    1 确定软件提供的主要商业任务。

    2 对每个商业任务，确定完成该任务所要进行的交易。

    3 确定从数据库信息引出的计算结果。

    4 对于对时间有要求的交易，确定所要的时间和条件。这些条件包括数据库大小、机器配置、交易量、以及网络拥挤情况。

    5 确定会产生重大意外的压力测试，包括：内存、硬盘空间、高的交易率。

    6 确定应用需要处理的数据量。

    7 确定需要的软件和硬件配置。通常情况下，不可能对所有可能的配置都测试到，因此要选择最有可能产生问题的情况进行测试，包括：最低性能的硬件、几个有兼容性问题的软件并存、客户端机器通过最慢的LAN/WANF连接访问服务器。

单元测试

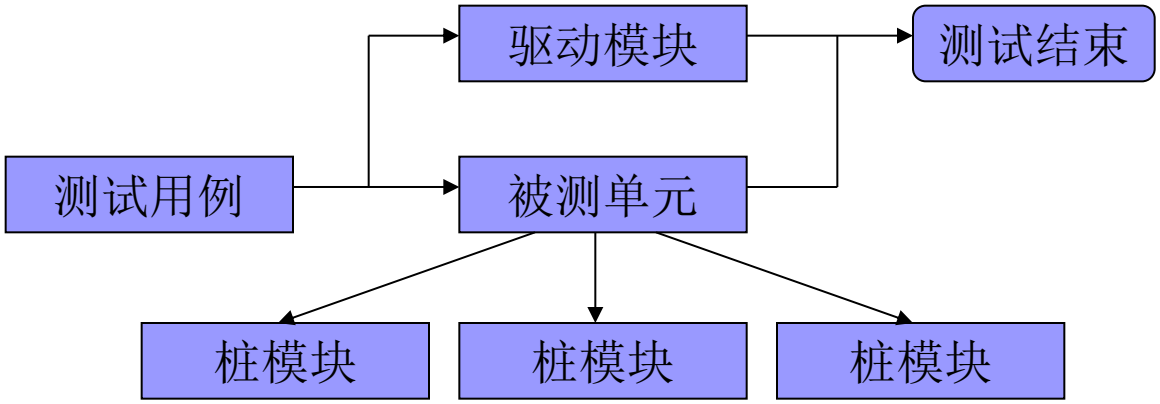
- 步骤执行

8 确定其它与应用软件没有直接关系的商业交易。包括：管理功能，如启动和退出程序配置功能，如设置打印机操作员的爱好，如字体、颜色应用功能，如访问email或者显示时间和日期。

9 确定安装过程，包括订制从哪安装、定制安装、升级安装。

10 确定没有隐含在功能测试中的户界面要求。大多界面都在功能测试时被测试到。如：操作与显示的一致性，如使用快捷键等；界面遵从合理标准，如按钮大小，标签等。

- 单元测试设计（一）
  - 单元测试模型的设计。
  - 测试项目的设计。
- 单元测试模型设计
  - 构造最小运行调度系统，即驱动模块，用于模拟被测模块的上一级模块。
  - 模拟实现单元接口，即单元函数需调用的其他函数接口，即桩模块。
  - 模拟生成测试数据或状态，为单元运行准备动态环境。
  - 对测试过程的支持，对测试结果的保留，对测试覆盖率的记录等。
  - 单元测试环境的示意图如下：



单元测试设计（二）

- 测试项目设计
- 测试项目是测试用例的一个总则，主要是根据测试需求设计测试点，不包含具体实践的用例。
- 在测试项目的设计中，主要从功能覆盖和代码覆盖两个角度进行考虑。

1 功能覆盖属黑盒的范畴，用来指出测试用例是否已经覆盖了程序应该提供的功能。逻辑覆盖率是考核单元测试质量的一个关键指标。

2 代码覆盖也称逻辑覆盖，包括语句覆盖、分支覆盖、路径覆盖，是一种常用的白盒测试方法。

（1）语句覆盖：设计若干测试用例，运行被测程序，使得每一个可执行语句至少执行一次。

（2）分支覆盖：也称判断覆盖，使得程序中每个判断的取真分支和取假分支至少执行一次。

（3）条件覆盖：设的程序中每个判断的每条件的可能取值至少执行一次。

- 覆盖率指标：核心代码覆盖率达到100%，共享资源库的代码覆盖率达到100%，非核心代码覆盖率达到90%。

单元测试

- 单元测试用例的设计
  - 单元测试用例编写原则之一
    - 测试用例设计的根据是详细设计说明书。
    - 测试用例不仅要证明被测单元是否做了它应该做的事情，同时需要证明它是否做了不希望它做的事，即是否满足设计说明书中的要求。
    - 设计测试用例的5个步骤：

      1　为系统运行起来而设计的测试用例

      如果用例可以正常运行，至少可以知道测试环境和测试单元是可用的。

      2　为正向测试而设计用例

      （1）　用例的设计者应该通读相关的设计说明，每一个测试用例就是通过有针对的测试说明书中的一项或多项内容来设计的。

      （2）　当设计到不止一个设计说明书时，最好将测试方案中的测试用例与主要的设计说明中的描述顺序相对应。

      （3）　正向测试用例验证设计说明书所对应的功能项或性能指标能否兑现。

      3　为逆向测试而设计用例

      用例用来验证被测的软件单元有没有做不应该做的事情，可以依靠错误猜测的方法进行测试用例的构造。

单元测试

- 单元测试用例的设计
  - 单元测试用例编写原则之二

    4　为满足特殊需求而设计用例

    用户的特殊要求包括系统的性能、安全性、保密性，尤其对安全和保密性要求较高。

    5　为代码覆盖而设计用例

    设计好的测试用例是可以保证较高的代码测试覆盖率的。

单元测试用例设计方法之一

- 规范（规格）导出法
  - 是根据相干的规范描述来设计测试用例的。
  - 一个比较实际的方法是根据陈述规范所用语句的顺序来相应的为被测单元设计测试用例。
  - 例如，考虑一个计算平方根的函数的规范：
    - 输入：实数
    - 输出：实数
    - 规范：当输入一个0或者比0大的实数时，返回其正的平方根；当输入一个小于0的实数时，显示错误信息“平方根非法—输入之小于0”，并返回0；库函数Print_Line可以用来输出错误信息。
  - 在这个规范中有3个陈述，可以用两个测试用例来对应：
    - 测试用例1：输入4，输出2。对应规范中的第一句陈述（“当输入一个0或者比0大的实数时，返回其正的平方根”）。
    - 测试用例2：输入-1，输出0。对应规范中的第二、第三句陈述（“当输入一个小于0的实数时，显示错误信息“平方根非法—输入之小于0”，并返回0；库函数Print_Line可以用来输出错误信息。”）。
  - 规范导出测试可以应用到保密分析、安全分析、软件故障分析，或其他队单元规范作出补充的文件上去。

单元测试

- 单元测试用例设计方法之二
  - 等价类划分法
    - 等价类划分假定某一特定的等价类中的所有值对于测试目的来说是等价的。所以在每个等价类中找一个之作为测试用例。
    - 是一种正式的测试用例设计方法，基于对被测单元的输入、输出所做的划分，对每一个划分中的所有输入、被测单元有等价的行为。
    - 划分也可以根据软件所能存取的数据确定，包括时间、输入输出顺序、状态。
    - 当软件复杂时，等价类的区分以及相互之间的作用关系也相应变复杂，不能方便的应用这种方法。

单元测试

- 边界值分析法
  - 边界值分析使用与等价类测试方法相同的等价类划分，只是边界值分析假定错误更多地存在于两个划分的边界上。
  - 边界值测试在软件变得复杂的时候也会变得不实用。
  - 边界值测试对于非向量类型的值（如枚举类型的值）也没有意义。
  - 以求平方根函数为例设计测试用例：

用例	输入	输出	描述
1	最小负实数	“非法复制错误”	对应于划分（i）的下边界
2	恰比0小	“非法复制错误”	对应于划分（i）的上边界
3	0	0	划分（i）的上边界之外，划分（ii）的下界，划分（a）的下界
4	恰比0大	正的平方根	划分（ii）的下界
5	最大正实数	正的平方根	划分（ii）的上界，划分（a）的上界

单元测试

分支测试法

- 根据单元中的中止流分支或判断点来设计测试用例。
- 通常用来达到一定的判定覆盖率。
- 分支测试是白盒或结构化测试用例设计技术。
- 给定一个结构的规格说明，说明单元中的控制流，就可以设计测试用例来测试其中分支。
- 对于一个有结构化说明的单元适合于作分支测试。
- 但一个单纯的单元功能规格说明可能导致覆盖遗漏。
- 需要当心的是，当集中注意力于分支上时，测试设计则容易忽略单元功能的整体。
- 分支测试只是基于单元中的判断情况，对于复杂的逻辑表达的判断也不容易使用。

单元测试

- 条件测试法
  - 条件测试的目标是测试在每个逻辑条件的单个成分及他们组合情况下，程序都是正确的。
  - 条件测试包含了许多测试用例设计技术，他们都致力于弥补在遇到复杂逻辑条件是分支测试的弱点。
  - 测试用例用来测试逻辑表达式的单个成分。
  - 在分支测试里，条件测试被用作一种“黑盒”技术，测试设计者要根据单元的规格说明对单元的结构作出猜测。
  - 条件测试更适合做“白盒”测试技术。
  - 条件测试最适合有结构化规格说明的单元，它提供一个全面的测试，包括复杂的测试条件、编成和设计错误多发区与以及分支测试不能达到的区域。

单元测试

条件测试法

- 一个单元也可能有内部边界，它们只能从单元的结构化规格说明找到。
- 如左侧代码事例，对该内部边界值分析包括了3个需要测试的条件：

  用例1：误差恰好大于期望精度

  用例2：误差等于期望精度

  用例3：误差恰好小于期望精度
- 内部边界值测试可以用来发现一些内部错误。如误把‘<’写作‘<=’。
- 内部边界值测试应作为一种补充方法，在其它方法的最后使用。

单元测试

- 单元测试的执行
  - 测试的结论
    - 测试的结论与执行被测试的单元中模拟目标环境下的程序执行精确性相关。保证在估计这些特征时所有的环境因素均被考虑。
    - 例如，所有的隐性输入必须被考虑（即系统时钟，文件状态，单元加载地点），还有实际环境的代表物（即相同的编译器，加载者，操作系统，计算机，输入分布）也是测试环境必须考虑的。
  - 自动测试
    - 自动测试可以大大提高测试效率，便于回归测试。
    - 自动测试有两种方式：商业化的测试工具和自行开发的工具。
    - 自动测试通过自定义的脚本文件将测试用例逐条放入，通过驱动模块读入脚本文件，驱动被测单元执行每条用例，将相应的结果返回驱动模块，驱动模块将结果保存在一个文本文件中，通过与预期的结果比较（也是一个文本文件），可以判断是否所有的用例都通过测试。

- 单元测试的四个过程

1 检查编码是否遵循软件编程规范和标准。
2 自动或手工分析程序
3 设计测试用例并运行测试用例
4 错误跟踪分析

- 从覆盖的角度测试应覆盖

1 功能覆盖
2 输入域覆盖
3 输出域覆盖
4 函数交互覆盖
5 代码执行覆盖

- 按针对性，单元测试可有以下特定的测试：

1 声明测试：检查模块中的所有变量是否被声明。
2 路径测试：每条语句最少执行一次；每个确定语句的每个方向要测试到。
3 循环测试：循环不执行；执行两次；反映执行典型的循环的执行次数；最大循环次数减1；最大循环次数；大于最大循环次数。
4 循环嵌套：外最小值，内所用情况；内最小值，外所有情况；内外最小值；内外最大值；外最大值，内所有情况；内最大值，外所有情况。
5 边界值测试：重点检查大于、小于和等于边界条件的情况。
6 接口测试：检查模块的数据流（输入、输出）是否正确；检查输入参数和声明的自变量的个数，数据类型和输入顺序是否一致；检查全局变量是否正确定义和使用等。
7 确认测试：是否接受有效输入数据（操作），拒绝无效数据（操作）。
8 事务测试：输入--〉输出，错误处理。

单元测试

- 测试模块
  - 结构化测试的原则是每个测试模块有单个功能组成。
    - 可重用性

      考虑到模块的重用性，模块不应该被设计成包含多个功能。
    - 复杂性

      1 多个功能的模块容易变得复杂，复杂的模块容易混乱。

      2 大量的这样的模块有多个入口和多个出口。

      3 多个标志将被使用。这些标志指出将使用哪个功能，在复杂的代码中很容易出错。
    - 可维护性

      模块需要修改时，复杂的模块要比简单的模块难以维护。
  - 测试用例（脚本）的设计也要按上面的原则，每个测试描述只测试单一功能。这是测试很集中，只有两种输出：成功和失败。

单元测试

单元测试工具

- 测试工具分为以下几类：
  - 代码检测工具
  - 覆盖率测试工具
  - 内存检查
  - 性能检查
  - 质量分析工具
- 重用工具有以下几种
  - 代码检查：LINT
  - 覆盖率： TrueCoverage
  - 内存检查： BounderChecker
  - 性能检查： TrueTime

结束

- 测试目的是为了发现软件的缺陷。
- 目前软件单元测试的状况。
- 培训的目标，让更多的人了解测试、会做测试，让更多的人参与测试。