白盒测试

张程
Email：bootan@cqu.edu.cn
QQ:80463125

白盒测试

概述

- 也称结构测试或逻辑驱动测试，通过了解软件系统的内部工作过程，设计测试用例来检测程序内部动作是否按照规格说明书规定的正常进行，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
- 基于一个应用代码的内部逻辑知识，即基于覆盖全部代码、分支、路径、条件，使用程序设计的控制结构导出测试用例

测试原则

- 保证一个模块中所有路径至少被测试一次
- 所有逻辑值都要测试真(true)和假(false)两种情况
- 检查程序的内部数据结构是否有效
- 检查上、下边界及可操作范围内运行所有循环

白盒测试依据

- 软件需求报告
- 软件需求规格说明
- 程序设计文档
- 软件界面设计
- 编码规范
- 开发命名标准

白盒测试的类别

- 软件公用问题的测试；
- 语言测试；
- 业务对象测试；
- 数据类型测试；
- 界面测试；
- 数值对象测试；
- 数据管理对象测试

测试类别-软件公用问题的测试

- 检查代码与设计对照表
  - *按软件需求检查CRC设计文档是否完全地实现了所有CRC卡中规定的内容，CRC设计文档完备、没有错误*
  - *按软件需求检查UI设计文档是否完全地实现了所有的UI设计的规定要求UI设计完备、没有错误*
  - *按软件需求检查编码对照表设计文档是否完全地实现了软件所规定的内容，完备、没有错误*
  - *检查代码名，代码位数，代码含义，姓名，编号，删除，追加，修改是否实现设计的规定要求*
- 检查是否按软件需求创建了所需的数据库，数据库的内容正确、完备、没有错误。
- 检查调用程序参数返回值
  - *检查调用程序参数返回值的类型、个数、顺序及返回值是否正确？没有错误。*
- 检查调用程序公用接口
  - *程序公用接口的没有错误。数据类型、个数、顺序及返回值正确*

测试类别-软件公用问题的测试（续）

- 检查子系统的设计
  - *主要功能，主要流程，输入内容，输出内容，交接口方式*
- 检查数据库设计
  - *说明数据库主要内容，数据库的逻辑划分，数据库的安全措施，数据库更新备份与恢复方式*
- 检查系统目标
- 检查数据元素的结构
- 检查用户访问
- 检查结构上的分析
- 检查程序是否冗余
  - *对于程序中的大量重复内容，是否使用了专门的类来实现?*
- 检查代码整体规范
  - *代码是否自始至终使用了《程序员开发手册》和《编码规范》中要求的格式、调用约定、结构等?*
- 检查类命名

测试类别- Java语言的测试检查

- 检查JAVA语言的下标是否有下标变量越界错误?
- 检查JAVA语言的除数是否包含有除零（n/0）错误的可能?
- 检查字符串;
- 检查字符串连结符"+";
- 检查浮点值、整型值应用没有错误;
- 检查switch语句的应用没有错误;
- 检查if语句的应用没有错误;
- 检查循环语句的应用没有错误;
- 检查数值范围是否存在溢出错误

测试类别-数据类型的测试检查

- Null转化
  - *在设置值对象VO时，在VO内部是否将空串" "将转化null，数值型数据(整数、浮点数)null转为0.#。*
- 检查控件数据类型的转换
  - *编辑控件数据类型是否与表中对应字段数据类型一致。*
- 检查单精度型、双精度型控件的范围控制
- 检查小数位数的设置
- 检查禁止输入字符的设置

测试类别- SQL语句的测试检查

- SQL系统数据库内在的数据库文件集表
- 检查SQL数据库对象
- 检查SQL语句书写规范
- 检查SQL语句
- 检查容量规划
- 检查局部变量和全局变量的定义。
- 检查if语句的定义
- 检查数据类型
- 检查数据完整性
- 检查临时表
- 检查集合的合并。集合合并的内容正确、完备、没有错误。
- 检查隔离等级

测试类别- SQL语句的测试检查（续）

- 检查安全性管理
- 检查应用程序的安全性
- 检查调度作业
- 检查指定作业响应
- 检查报警管理
- 检查备份
- 检查数据库还原
- 检查数据传输
- 检查分布式数据
- 检查函数，不允许动态创建函数。

测试类别-界面UI的测试检查

- 测试检查继承类每个界面类都要实现软件所规定的内容，完备、没有错误。
- 测试检查添加按钮
- 测试检查响应按钮
- 测试检查界面标题
- 测试检查其他业务代码在完成业务代码时，可能需要用到帐套编码、单位编码、用户编码等信息。
- 测试检查客户端调用BO对象
- 测试检查对话框须继承和使用的类
- 测试检查表格模型须继承和使用的类
- 界面规范测试检查
- 界面初始化测试检查
- 编辑控件（除功能按钮以外的控件）应用测试检查
- 通用对话框测试
- 参照框测试检查
- 状态栏测试检查
- 业务功能测试检查
- 界面校验测试检查

测试类别-数值对象的测试要求

- 检查数字转换为其字符串表示形式的数值格式化方法，是否完全地实现了软件所规定的内容，没有错误；
- 检查以参数的形式传递对象的接口，是否完全地实现了软件所规定的内容，没有错误；
- 检查参数以合法的方式提供格式化服务，是否完全地实现了软件所规定的内容，没有错误；
- 检查数据库自动生成数值序列功能，是否完全地实现了软件所规定的内容，没有错误；
- 检查布尔对象转换值，null、未定义、0、或false均转换成布尔对象的方法,是否完全地实现了软件所规定的内容，没有错误；
- 检查数值函数对象是否完全地实现了软件所规定的内容，没有错误；
- 检查将非字母、数字字符转换成ASCII码编码函数，是否完全地实现了软件所规定的内容，没有错误；
- 检查将ASCII码转换成字母、数字字符译码函数，是否完全地实现了软件所规定的内容，没有错误等；

测试类别-业务对象的测试检查

- 检查处理应用程序的业务逻辑和业务校验，是否完全地实现了软件所规定的内容，完备、没有错误；
- 检查允许与其它层相互作用的接口，是否完全地实现了软件所规定的内容，完备、没有错误；
- 检查管理业务层级别的对象的依赖，是否完全地实现了软件所规定的内容，完备、没有错误，
- 检查函数之间（考虑复用）功能独立，效率高，功能完整,全局看不雷同；
- 检查业务对象之间不考虑关系,全部都是函数载体等；

测试类别-数据管理对象的测试检查

- 检查数据库的利用效率；
- 检查DMO类中方法的完整性（DMO类中应包含insert()、delete()、update()方法，还可以包括其它的查询方法）；
- 检查数据库连接，是否完全地实现了设计规定的要求，没有错误

静态测试-代码检查

- 是软件静态测试中常用的软件测试方法之一，更容易发现架构以及时序相关等较难发现的问题
- 帮助团队成员统一编程风格，提高编程技能等
- 代码检查被公认为是一个提高代码质量的有效手段
- 代码审查包含对错误代码的检查，和不好的编码风格代码的检查
  - *可靠性：事实证明按照某种标准或规范编写的代码比不这样做的代码更可靠，软件缺陷更少。*
  - *可读性/维护性：符合标准和规范的代码易于阅读、理解和维护。*
  - *移植性：如果代码符合标准，迁移到另一个平台就会轻而易举，甚至完全没有障碍*
- 代码检查的内容
  - *可追溯性、逻辑、数据、接口、文档、注释、异常处理、内存、其它*

静态测试-代码检查过程

- 代码检查策划
  - *项目负责人分配代码审查任务*
  - *确定代码审查策略：依据软件开发文档，确定软件关键模块，作为代码审查重点；将复杂度高的模块也作为代码审查的重点。*
  - *确定代码审查单*
  - *确定代码审查进度安排*
- 代码检查实施
  - *代码讲解：软件开发人员详细向测试人员讲解代码实现情况，测试人员提出问题和建议*
  - *静态分析：采用静态分析工具进行分析，有利于软件测试人员在后续代码审查时对软件建立宏观上认识*
  - *规则检查：采用静态分析工具对源程序进行编码规则检查，对于工具报出的问题再由人工进行进一步的分析以确认软件问题*
  - *正式代码检查*
    - 独立审查：测试人员根据项目负责人的工作分配，独自对自己负责的软件模块进行代码审查。
    - 会议审查：项目负责人主持召开会议，测试人员和开发人员参加；测试人员就独立审查发现的问题和疑问与开发人员沟通，并讨论形成一致意见。
  - *更改确认：开发人员对问题进行处理，代码审查人员对软件的处理情况进行确认，验证更改的正确性*
- 代码检查总结
  - *代码审查工作结束后，项目负责人总结代码审查结果；编写测试报告，对软件代码质量进行评估，给出合理建议。*
  - *详细记录代码审查提出的所有问题及最终结论可以供其他软件项目代码审查借鉴*

静态测试-代码走查

- 是软件静态测试方法之一，是通过对代码的阅读，检查发现程序代码中的问题。
- 具体方法
  - *由测试人员组成小组，准备一批有代表性的测试用例，集体扮演计算机的角色，沿程序的逻辑，逐步运行测试用例，查找被测软件缺陷。*
- 意义
  - *经验表明，代码走查通常能够有效地查找出30%～70%的逻辑设计和编码错误。*
  - *一旦发现错误，通常就能在代码中对其进行精确定位，降低了调试的成本。*
  - *代码走查过程通常可以发现成批的错误，这样错误就可以一同得到修正。*
- 走查小组的组成
  - *由三至五人组成，其中一个人扮演类似代码检查过程中"协调人"的角色，一个人担任秘书的角色，还有一个人担任测试人员。关于小组的组成结构，一般建议包括：*
    - 一位极富经验的程序员；
    - 一位程序设计语言专家；
    - 一位程序员新手（可以给出新颖、不带偏见的观点）；
    - 最终维护程序的人员；
    - 一位来自其他不同项目的人员；
    - 一位来自该软件编程小组的程序员

静态测试-代码走查过程

- 准备阶段
  - *在走查会议前几天分发有关材料，走查小组详细阅读材料，认证研究程序。*
- 生成实例
  - *小组中被指定为测试人员的那个人应提前准备好一些书面的有代表性的测试用例*
- 执行走查
  - *在走查会议期间，每个测试用例都在人们脑中进行推演。也就是说，把测试数据沿程序的逻辑结构走一遍。程序的状态（如变量的值）记录在纸张或白板上以供监视。*
- 形成报告
  - *会后将发现的错误形成报告，并交给程序开发人员。对发现错误较多或发现重大错误，在改正错误后再次进行会议走查。*

走查与检查的比较

	走 查	检 查
准备	通读设计和编码	应准备好需求描述文档、程序设计文档、程序的源代码清单、代码编码标准和代码缺陷检查表
形式	非正式会议	正式会议
参加人员	开发人员为主	项目组成员包括测试人员
主要技术方法	无	缺陷检查表
注意事项	限时、不要现场修改代码	限时、不要现场修改代码
生成文档	会议记录	静态分析错误报告
目标	代码标准规范，无逻辑错误	代码标准规范，无逻辑错误

静态测试-静态结构分析

- 以图形的方式表现程序的内部结构，例如函数调用关系图、函数内部控制流图。
- 静态结构主要分析：
  - *可以检查函数的调用关系是否正确；*
  - *是否存在孤立的函数而没有被调用；*
  - *明确函数被调用的频繁度，对调用频繁的函数可以重点检查。*
  - *编码的规范性；*
  - *资源是否释放*

静态测试-静态质量度量

- 软件质量包括6个方面：
  - *功能性（functionality）*
  - *可靠性（reliability）*
  - *可用性（usability）*
  - *有效性（efficiency）*
  - *可维护性（maintainability）*
  - *轻便性（portability）。*
- 质量度量包括3点：
  - *质量因素（Factors）*
  - *分类标准（criteria）*
  - *度量规则（Metrics）*

动态测试

- 通过运行软件来检验软件的动态行为和运行结果的正确性。
- 动态测试流程是：
  - *选取定义域有效值，或定义域外无效值；*
  - *对已选取值决定预期的结果；*
  - *用选取值执行程序；*
  - *执行结果与对已选取值决定预期的结果相比，不吻合程序有错*
  - *保证每个模块的所有独立路径至少被使用一次；*
  - *对所有的逻辑值均测试true和false；*
  - *上下边界及可操作范围内运行所有循环；*
  - *检查内部数据结构以确保其有效性。*

动态测试方法分类

- 结构性测试
  - *采用语句测试、分支测试或路径测试;*
- 正确性测试
  - *基于产品功能规格说明书、从用户角度针对产品特定的功能和特性所进行的验证活动，以确认每个功能是否得到完整的实现，用户能否正常使用这些功能*

白盒测试常用技术

- 逻辑覆盖法
- 插桩技术
- 基本路径测试法
- 域测试法
- 符号测试
- Z路径覆盖法
- 程序变异测试法

白盒测试技术-逻辑覆盖法

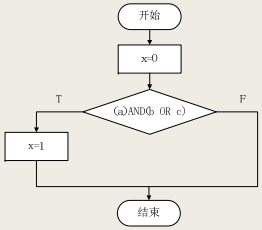
- 测试覆盖率：包括功能覆盖和结构覆盖
  - *功能点覆盖率大致用于表示软件已经实现的功能与软件需要实现的功能之间的比例关系。*
  - *结构覆盖率包括语句覆盖率、分支覆盖率、循环覆盖率、路径覆盖率等等。*
- 逻辑覆盖
  - *语句覆盖：通过选择足够的测试用例，使得运行这些测试用例时，被测程序的每个语句至少被执行一次*
    - 优点：**可以很直观地从源代码得到测试用例，无须细分每条判定表达式**
    - 缺点：**语句覆盖是最弱的逻辑覆盖。不能发现其中的逻辑错误**
  - *判定覆盖：指通过设计足够的测试用例，使得程序中的每一个判定至少都获得一次“真值”和“假值”，或者说使得程序中的每一个分支都至少通过一次.* *判定覆盖包含语句覆盖*
    - 优点：**判定覆盖具有比语句覆盖更强的测试能力。同样判定覆盖也具有和语句覆盖一样的简单性，无须细分每个判定就可以得到测试用例**
    - 缺点：**往往大部分的判定语句是由多个逻辑条件组合而成，若仅仅判断其整个最终结果，而忽略每个条件的取值情况，必然会遗漏部分测试路径。判定覆盖仍是弱的逻辑覆盖**
  - *条件覆盖：对于每个判定中所包含的若干个条件，应设计足够多的测试用例，使得判定中的每个条件都至少取到一次“真值”和“假值”的机会，也就是说，判定中的每个条件的所有可能结果至少出现一次。* *条件覆盖并不能包含判定覆盖*
    - 优点：**增加了对条件判定情况的测试**
    - 缺点：**条件覆盖不一定包含判定覆盖。例如，我们刚才设计的用例就没有覆盖判断M的Y分支和判断N的N分支。条件覆盖只能保证每个条件至少有一次为真，而不考虑所有的判定结果**

白盒测试技术-逻辑覆盖法（续）

- 逻辑覆盖
  - *条件/判定覆盖：指通过设计足够多的测试用例，使得运行这些测试用例时，判定中的每个条件的所有可能结果至少出现一次，并且每个判定本身的所有可能结果也至少出现一次。* *条件/判定覆盖同时包含判定覆盖,和条件覆盖*
    - 优点：**能同时满足判定、条件两种覆盖标准**
    - 缺点：**判定/条件覆盖准则的缺点是未考虑条件的组合情况**
  - *条件组合覆盖：通过设计足够多的测试用例，使得运行这些测试用例时，每个判定中条件结果的所有可能组合至少出现一次。* *条件组合覆盖包含前述4种覆盖*
    - 优点：**条件组合覆盖准则满足判定覆盖、条件覆盖和判定/条件覆盖准则**
    - 缺点：**线性地增加了测试用例的数量**
  - *路径覆盖：使设计的测试用例能覆盖被测程序中所有可能的路径。* *路径覆盖实际上考虑了程序中各种判定结果的所有可能组合，但它并未考虑判定中的条件组合。因此，虽然说路径覆盖是一种非常强的覆盖度量标准，但并不能代替条件组合覆盖*
    - 优点：**这种测试方法可以对程序进行彻底的测试，比前面五种的覆盖面都广**
    - 缺点：**需要设计大量、复杂的测试用例，使得工作量呈指数级增长，不见得把所有的条件组合都覆盖**

语句覆盖

- 语句覆盖，又称行覆盖（LineCoverage）、段覆盖（SegmentCoverage）、基本块覆盖（BasicBlockCoverage），这是最常用也是最常见的一种覆盖方式。其基本思想是设计若干个测试用例，运行被测程序，使程序中每一条可执行语句至少应该执行一次。
- 为了使上述示例程序中的每条语句都能够至少执行一次，可以构造以下测试用例：a = T, b= T,c= T。
- 其实语句覆盖对程序执行逻辑的覆盖率很低，这是语句覆盖法自身最严重的缺陷。
- 例如，判定的第一个运算符&&错写成运算符||，或第二个运算符||错写成运算符&&，这时使用上述的测试用例仍然可以达到100%的语句覆盖，上述的逻辑错误无法被检测出来。因此一般认为语句覆盖是很弱的逻辑覆盖。

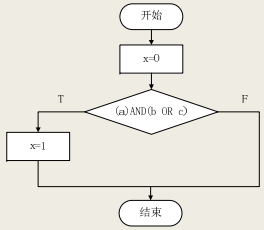


判定覆盖

- 判定覆盖的基本思想是设计若干个测试用例，运行被测程序，使得程序中的每个判定至少都获得一次真值或假值，或者说使得程序中的每一个取真分支和取假分支至少经历一次，因此判定覆盖又称为分支覆盖。
- 除了真假双值判定语句外，还有多值判定语句，如case语句，因此判定覆盖更一般的含义是：使得每一个判定获得的每一种可能的结果至少被满足一次。

  a = T,b = T,c = T

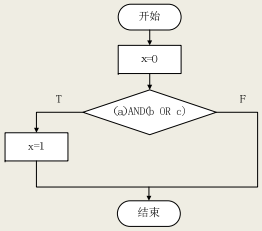
  a = F,b = F,c = F
- 两组测试用例不仅满足了判定覆盖，而且满足了语句覆盖，所以判定覆盖要比语句覆盖更强一些。如表所示，判定的第一个运算符&&错写成运算符||或第二个运算符||错写成运算符&&，这时使用上述测试用例仍可以达到100%的判定覆盖，仍然无法发现上述假设的逻辑错误



条件覆盖

- 在程序设计中，一个判定语句可能是由多个条件组合而成的复合判定。条件覆盖的含义是：构造一组测试用例，使得每一个判定语句中的每个逻辑条件的可能值至少满足一次。
  - a = F,b = T,c = F
  - a = T,b = F,c = T
  - a = F,b = T,c = T
  - a = T,b = F,c = F
- 仔细分析可以发现，上述测试用例在满足条件覆盖的同时，把判定的两个分支也覆盖了。但是否可以说，达到了条件覆盖也就必然实现了判定覆盖呢?
- 可以发现这两组测试用例可以满足条件覆盖，但却不能满足分支覆盖，为达到更高的覆盖率，需要同时兼顾条件覆盖和分支覆盖。

序号	a	b	c	a && (b||c)	条件覆盖%	判定覆盖%
1	F	T	T	F	100	50
2	T	F	F	F		



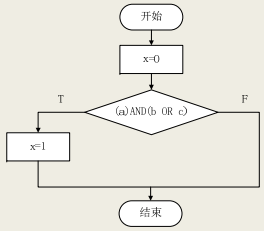
条件/判定覆盖

- 条件/判定覆盖的含义是：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。
- 选用以下的两组测试用例可以符合条件判定组合覆盖标准：

  a = T,b = T,c = T

  a = F,b = F,c = F
- 但是条件判定组合覆盖也存在一定的缺陷。例如，判定的第一个运算符&&错写成运算符||或第二个运算符||错写成运算符&&，如表所示，使用上述测试用例仍然可以达到100%的条件判定组合覆盖，无法发现这些逻辑错误。

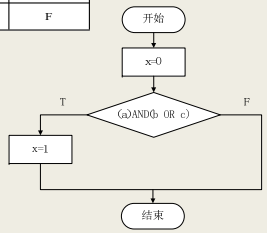
序号	a	b	c	a ||(b||c)	a && (b&&c)	条件判定组合覆盖%
1	T	T	T	T	T	100
2	F	F	F	F	F	



条件组合覆盖

- 设计足够的测试用例，使得每个判定中条件的各种可能组合都至少出现一次。显然满足多条件覆盖的测试用例是一定满足判定覆盖、条件覆盖和条件判定组合覆盖的。
- 判定语句中包含三个逻辑条件，每个逻辑条件有两种可能取值，因此共有23=8种可能的取值组合，对应的测试用例如表所示，这些测试用例能够保证多条件覆盖。

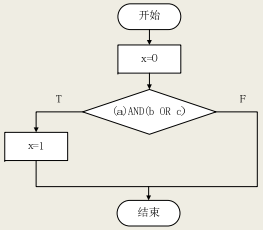
序号	a	b	c	a&&(b||c)
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F



修正条件判定覆盖

- 当程序中的判定语句包含多个条件时，运用条件组合覆盖方法进行测试，其条件取值组合数目是非常大的。
- 修正条件判定覆盖在条件组合的基础上进行数据的挑选，其挑选数据的要求是：程序的判定被分解为通过逻辑操作符（and、or）连接的bool条件，每个条件对于判定的结果值是独立的。
- 8条满足条件组合覆盖的测试用例基础上，按照修正条件判定覆盖的要求选择需要的测试用例，选择结果如表所示。

序号	a	b	c	a&&(b||c)	a	b	c
1	T	T	T	T	✓		
2	T	T	F	T	◇	✓○○	
3	T	F	T	T	○		✓○○
4	T	F	F	F		✓○○	✓○○
5	F	T	T	F	✓		
6	F	T	F	F	◇		
7	F	F	T	F	○		
8	F	F	F	F			



白盒测试技术-逻辑覆盖法（续）

- 面向对象的覆盖
  - *继承上下文覆盖*
  - *基于状态的上下文覆盖*
- 测试覆盖准则
  - *ESTCA覆盖准则*
  - *现行代码序列与跳转LCSAJ*

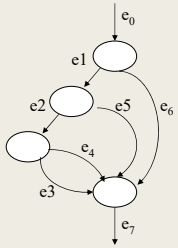
白盒测试技术-逻辑覆盖法（续II）

- 谓词覆盖准则
  - *一个分支的条件是由谓词组成的。单个谓词称为原子谓词，例如a!=0、mid>0等都是原子谓词。*
  - *原子谓词通过逻辑运算符可以构成复合谓词，常见的逻辑运算符包括“与”、“或”、“非”等。*
- 谓词覆盖准则包括：
  - *原子谓词覆盖准则*
  - *分支—谓词覆盖准则*
  - *复合谓词覆盖准则*

白盒测试技术-逻辑覆盖法（续II）

- 原子谓词覆盖准则
  - *在软件测试中，每个复合谓词所包含的每一个原子谓词都至少获得一次“真”值和一次“假”值*

```
main()
 { int i,j,k,match;
 scanf(“%d%d%d”, &i, &j, &k);
    if(i<=0||j<=0||k<=0||i+j<=k||i+k<=j||j+k<=i) match=4;
      else if(i==j&&i==k&&j==k) match=1;
           else if(i==j||i==k||j==k) match=2;
                else match=3;
    printf(“match=%d\n”,match);
    }
```



白盒测试技术-逻辑覆盖法（续II）

- 原子谓词覆盖准则
  - *在软件测试中，每个复合谓词所包含的每一个原子谓词都至少获得一次“真”值和一次“假”值*

```
main()
 { int i,j,k,match;
 scanf(“%d%d%d”, &i, &j, &k);
    if(i<=0||j<=0||k<=0||i+j<=k||i+k<=j||j+k<=i) match=4;
      else if(i==j&&i==k&&j==k) match=1;
           else if(i==j||i==k||j==k) match=2;
                else match=3;
    printf(“match=%d\n”,match);
    }
```

测试用例	变量	原子谓词								
	i, j, k	i<=0	j<=0	k<=0	i+j<=k	i+k<=j	j+k<=i	i==j	i==k	j==k
用例1	-1,2,2	真	假	假	真	真	假	--	--	--
用例2	2,-1,2	假	真	假	真	假	真	--	--	--
用例3	2,2,-1	假	假	真	假	真	真	--	--	--
用例4	1,1,2	假	假	假	真	假	假	--	--	--
用例5	2,1,1	假	假	假	假	假	真	--	--	--
用例6	1,2,1	假	假	假	假	真	假	--	--	--
用例7	2,2,2	假	假	假	假	假	假	真	真	真
用例8	2,2,3	假	假	假	假	假	假	真	假	假
用例9	2,3,2	假	假	假	假	假	假	假	真	假
用例10	3,2,2	假	假	假	假	假	假	假	假	真
用例11	5,3,4	假	假	假	假	假	假	假	假	假

白盒测试技术-逻辑覆盖法（续II）

- 分支-谓词覆盖准则
  - *在原子谓词覆盖测试基础上还要求每个复合谓词本身也至少获得一次“真”值和一次“假”值*

```
main()
 { int i,j,k,match;
 scanf(“%d%d%d”, &i, &j, &k);
    if(i<=0||j<=0||k<=0||i+j<=k||i+k<=j||j+k<=i) match=4;
      else if(i==j&&i==k&&j==k) match=1;
           else if(i==j||i==k||j==k) match=2;
                else match=3;
    printf(“match=%d\n”,match);
    }
```

测试用例	变量	复合谓词		
	i, j, k	i<=0||j<=0||k<=0||i+j<=k||i+k<=j||j+k<=i	i==j&&i==k&&j==k	i==j||i==k||j==k
用例1	-1,2,2	真	--	--
用例2	2,-1,2	真	--	--
用例3	2,2,-1	真	--	--
用例4	1,1,2	真	--	--
用例5	2,1,1	真	--	--
用例6	1,2,1	真	--	--
用例7	2,2,2	假	真	--
用例8	2,2,3	假	假	真
用例9	2,3,2	假	假	真
用例10	3,2,2	假	假	真
用例11	5,3,4	假	假	假

白盒测试技术-逻辑覆盖法（续II）

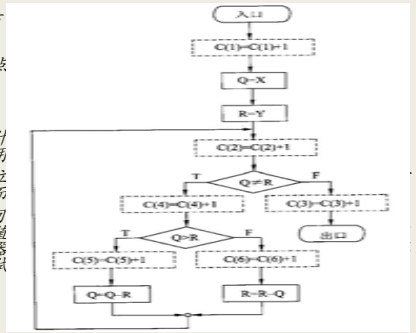
- 复合谓词覆盖准则
  - *在软件测试中，每个谓词中条件的各种可能都至少出现一次*

```
main()
 { int i,j,k,match;
 scanf(“%d%d%d”, &i, &j, &k);
    if(i<=0||j<=0||k<=0||i+j<=k||i+k<=j||j+k<=i) match=4;
      else if(i==j&&i==k&&j==k) match=1;
           else if(i==j||i==k||j==k) match=2;
                else match=3;
    printf(“match=%d\n”,match);
    }
```

测试用例	变量	原子谓词								
	i, j, k	i<=0	j<=0	k<=0	i+j<=k	i+k<=j	j+k<=i	i==j	i==k	j==k
用例1	-1,-1,-1	真	真	真	真	真	真	--	--	--
用例2	-1,-2,2	真	真	假	真	假	假	--	--	--
用例3	2,-1,-2	假	真	真	假	假	真	--	--	--
用例4	2,2,-1	假	假	真	假	真	真	--	--	--
用例5	1,1,2	假	假	假	真	假	假	--	--	--
用例6	2,1,1	假	假	假	假	假	真	--	--	--
用例7	1,2,1	假	假	假	假	真	假	--	--	--
用例8	2,2,2	假	假	假	假	假	假	真	真	真
用例9	2,2,3	假	假	假	假	假	假	真	假	假
用例10	2,3,2	假	假	假	假	假	假	假	真	假
用例11	3,2,2	假	假	假	假	假	假	假	假	真
用例12	5,3,4	假	假	假	假	假	假	假	假	假

白盒测试技术-插桩技术

- 借助于在被测程序中设置断点或打印语句来进行测试的方法，在执行测试的过程中可以了解一些程序的动态信息。这样在运行程序时，既能检验测试的结果数据，又能借助插入语句给出的信息掌握程序的动态运行特性，从而把程序执行过程中所发生的重要事件记录下来。
- 程序插装设计时主要需要考虑三方
  - *需要探测哪些信息；*
  - *在程序的什么位置设立插装点*
  - *计划设置多少个插装点*
- 主要有以下几个应用：
  - *覆盖分析：程序插装可以估计分性，从而设计更好的测试用*
  - *监控：在程序的特定位置设立程序运行时的某些特性，从而*
  - *查找数据流异常：程序插装可范围。掌握了数据变量的取值数据流异常可以用静态分析器竟所有信息的获取是随着测试*



白盒测试技术-断言测试

- 用于检查在程序运行过程出现的一些本“不应该”发生的情况。也就是在一个应该正确的地方，加一条判断来验证程序运行时，它是否真正如当初预料的那样，具有预期的正确性。
- 断言测试就是在程序中插入断言，插入断言的根本目的是用于帮助程序的调试与排错，因此本质上它是属于测试代码，是一种特殊的插桩语句，而不是属于真正的应用程序模块的一部分

白盒测试技术-缺陷种植测试

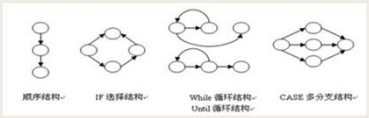
- 一种用来估计驻留在程序中的缺陷数量的技术。
- 工作原理是向一个软件中“种植”缺陷，然后运行测试集，以检查发现了多少个种植的缺陷，还有多少个种植的缺陷没有被发现，以及已经发现了多少个新的非种植的缺陷。然后就可以预测残留的缺陷数量。
- 案例
  - *如果种植了100个种子缺陷，而在测试中只找到75个种植的缺陷，那么种子发现率为75%。如果已经发现了450个真实的缺陷，那么可以通过种子发现率，推出这450个真实的缺陷只代表了现在存在所有真实缺陷的75%。那么，真是的缺陷总数估计为600个。所以还有150个真实的缺陷需要测试出来。*

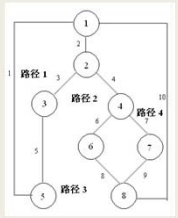
白盒测试技术-基本路径测试法

- 程序的控制流图
  - *对程序流程图进行简化后得到的，它突出表示程序控制流的结构。程序控制流图是描述程序控制流的一种方式*
- 程序环形复杂度：称为圈复杂度，它是一种为程序逻辑复杂度提供定量尺度的软件度量。
  - *可用如下方法来计算环形复杂度V(G)：*
    - 控制流图中区域的数量对应于环形复杂度。
    - V(G) = E-N+2 其中：E是控制流图中边的数量；N是控制流图中的节点数量
    - V(G) = P+1 其中：P是控制流图G中判定节点的数量
- 导出测试用例
- 准备测试用例

白盒测试技术-基本路径测试法

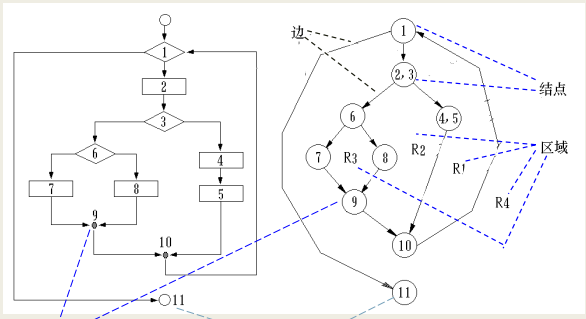
- 程序控制流图





- 计算环形复杂度
  - *节点数量N =8：*
  - *导出条边E=10。用（①、②、③、④、⑤、⑥、⑦、⑧、⑨、⑩编号表示）*
  - *V(G) = E-N+2=10-8+2= 10（条边）- 8（个节点）+ 2 = 4*
  - *导出独立路径用路径1、 路径2、 路径3、路径4编号表示*

白盒测试技术-基本路径测试法



在分支结构中，分支的汇聚处应有一个汇聚结点.

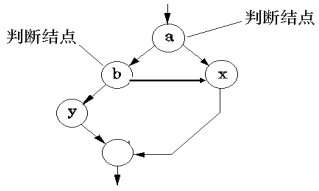
每一条边必须终止于一个结点

白盒测试技术-基本路径测试法

- 关于复合条件的处理
  - *如果判断中的条件表达式是由一个或多个逻辑运算符连接的复合条件表达式，则需要改为一系列只有单个条件的嵌套的判断。*

```
      .
      .
      .
if   a OR b
  then  procedure  x
  else  procedure  y;
      .
      .
      .
```

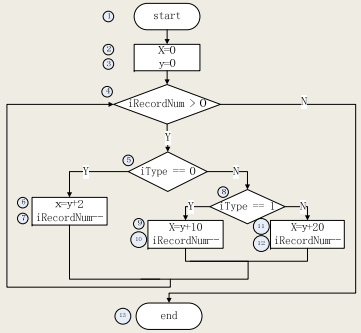
(a)



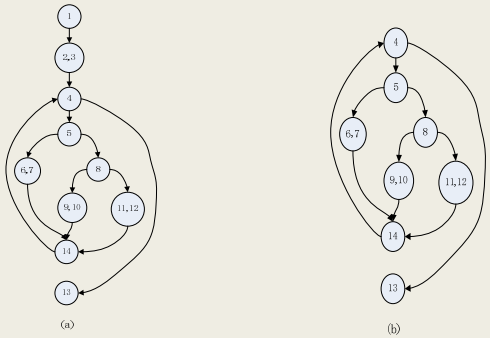
(b)

基本路径测试法示例

- 下面以一个简单的 C 函数为例，说明使用基本路径测试法设计测试用例的过程。



- 1. 以详细设计或源代码为基础，导出程序的控制流图



2. 计算控制流图 G 的环路复杂性 V(G) V(G)=4（区域数）；

或：V(G)= 3（判断节点数）+ 1 = 4；

或：V(G)=12（边的个数）-10（节点个数）+2=4。

3. 导出独立路径（用代码行号表示）

根据控制流图计算得到的环路复杂性，可知该程序的基本路径集中的路径条数为 4，

具体路径描述如下。

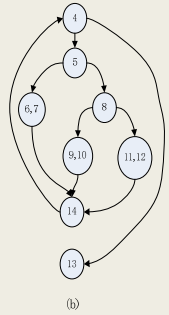
路径 1：4→13

路径 2：4→5→6,7→14→4→13

路径 3：4→5→8→9,10→14→4→13

路径 4：4→5→8→11,12→14→4→13

4.设计测试用例，确保基本路径集中的每条路径都被执行



- 本示例的源程序代码：

```
    int sort(int iRecordNum, int iType)
1   {
2       int x = 0;
3       int y = 0;
4       while (iRecordNum > 0)
5       {
6           if(iType == 0)
7           {
8               x=y+2;
9               iRecordNum--;
10          }else if(iType == 1)
11              x = y+10;
12          else
13              x = y+20;
14      }
15      return x;
16  }
```

此代码有错，试试用测试用例来找出程序中的错误

白盒测试技术-域测试法

- 一种基于程序结构的测试方法，基于对程序输入空间（域）的分析，选择测试点进行测试。主要为：
  - *域错误：程序的控制流存在错误，对于某一特定的输入可能执行的是一条错误路径，这种错误称为路径错误，也叫做域错误；*
  - *计算型错误：对于特定输入执行的路径正确，但赋值语句的错误导致输出结果错误，称为计算型错误；*
  - *丢失路径错误：由于程序中的某处少了一个判定谓词而引起的丢失路径错误。*
- 域测试缺点是：
  - *为进行域测试对程序提出的限制过多；*
  - *当程序存在很多路径时，所需的测试点很多*

白盒测试技术-符号测试法

- 基本思想是允许程序的输入不仅仅是具体的数值数据，而且包括符号值，符号值可以是基本的符号变量值，也可以是符号变量值的表达式。
- 符号测试执行的是代数运算，可以作为普通测试的一个扩充；
- 符号测试可以看作是程序测试和程序验证的一个折衷办法；
- 符号测试程序中仅有有限的几条执行路径；
- 符号测试的缺点是：
  - *分支问题不能控制；*
  - *二义性问题不能控制；*
  - *大程序问题不能控制。*

白盒测试技术-Z路径覆盖法

- 分析程序中的路径是指检验程序从入口开始，执行过程中经历的各个语句，直到出口
- Z路径覆盖对循环机制进行简化，减少路径的数量，使得覆盖所有路径成为可能，简化循环意义下的路径覆盖称为Z路径覆盖（循环简化：限制循环次数，只考虑循环一次或零次情况）
  - *循环简化的目的是限制循环的次数，无论循环的形式和循环体实际执行的次数，简化后的循环测试只考虑执行循环体一次和零次（不执行）两种情况，即考虑执行时进入循环体一次和跳过循环体这两种情况*

白盒测试技术-程序变异测试法

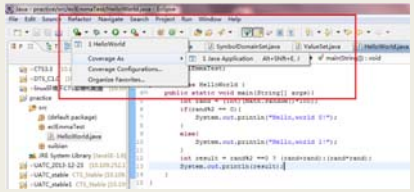
- 程序变异是一种错误驱动测试。错误驱动测试是指该方法是针对某类特定程序错误的，经过多年的测试理论研究和软件测试的实践，人们逐渐发现要想找出程序中所有的错误几乎是不可能的。
- 比较现实的解决办法是将错误的搜索范围尽可能地缩小，以利于专门测试某类错误是否存在
- 对于给定的程序P，先假定程序中存在一些小错误，每假设一个错误，程序P就变成P′，如果假设了n个错误：e_1，e_2，…，e_n，则对应有n个不同的程序：P_1，P_2，…，P_n，这里P_i称为P的变异因子。
- 存在测试数据C_i，使得P和P_i的输出结果是不同的。因此，根据程序P和每个变异的程序，可以求得P_1，P_2…，P_n的测试数据集C={C_1，C_2，…，C_n}。运行C，如果对每一个C_i，P都是正确的，而Pi都是错误的，这说明P的正确性较高。如果对某个C_i，P是错误的，而P_i是正确的，这说明P存在错误，而错误就是e_i。

白盒测试技术-程序变异测试法（续）

- 把一种语法结构改变成另一种语法结构，保证转换后的程序的语法正确，但不保持语义的一致。
  - *表达式$a<b$ → $a>b$，$a==b$，$a\neq b$，$a\geq b$，$a\leq b$*
  - *把变量设成某常量C*
  - *把常量C_1换成C_2*
- 程序强变异测试
  - *变异测试的缺点是它需要大量的计算机资源来完成测试充分性分析。对于一个中等规模的软件，所需的存储空间也是巨大的，运行大量变异因子也导致了时间上巨大的开销。*
- 程序弱变异测试
  - *弱变异和强变异有很多相似之处。*
  - *主要差别：弱变异强调的是变动程序的组成部分，根据弱变异准则，只要事先确定导致P与P'产生不同值的测试数据组，则可将程序在此测试数据组上运行，而并不实际产生其变异因子。*
  - *主要优点是开销较小，效率较高。*

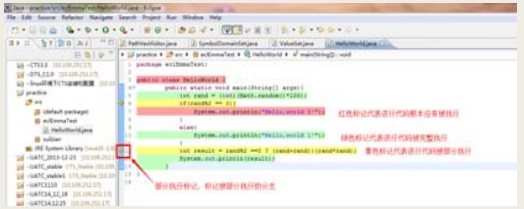
白盒测试工具-Emma

- Emma是一个开源的、面向Java程序的测试工具
- 它通过对编译后的Java字节码文件进行插装，在测试执行过程中收集覆盖率信息，并支持通过多种报表格式对覆盖率结果进行展示。
- EclEmma可以看作是Emma的一个图形界面，其具有Emma的基本功能，使用方式更加友好
- 测试过程
  - *选择被测程序(practice/src/EclEmmaTest/HelloWorld.java)*
  - *选择Coverage as→Java Application*

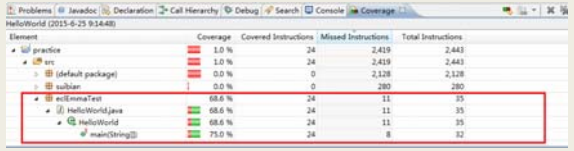


白盒测试工具-Emma（续）

- 测试过程
  - *得到覆盖测试结果*

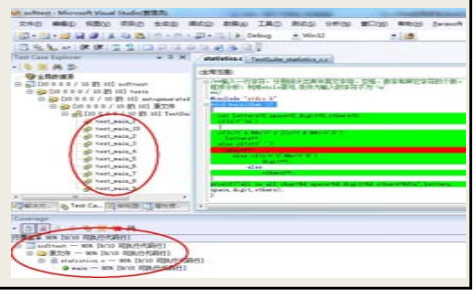


- Coverage单独视图
  - *EclEmma还设计了单独的视图来统计对代码的覆盖情况*



白盒测试工具-C++test

- C++test是一个C/C++单元级测试工具，自动测试C/C++类、函数或部件。
- 不需要编写测试用例、驱动程序或桩函数。
- 静态代码检测
- 单元测试



白盒测试工具-JUnit

- JUnit是一个Java语言的单元测试框架，多数Java的开发环境都已经集成了JUnit作为单元测试的工具。
- JUnit支持两种运行单个测试的方法：静态和动态方法
- JUnit的优势
  - *不用为单元测试编写重复的程序代码*
  - *JUnit的测试用例可以被组织成测试组合（TestSuite）*
  - *JUnit的测试结果容易收集*