第二章 软件测试技术

1. 黑盒测试技术

1.1. 概念

- 又叫功能测试、性能测试

黑盒测试概念

1. *不考虑程序的内部结构和处理过程，只依据需求说明，对功能进行测试的过程。*
2. *注重软件的功能性需求，主要针对软件界面和软件功能进行测试*

定义

黑盒测试就是把测试对象看做一个不能打开的黑盒子，在完全不考虑程序的内部结构和处理过程的情况下，只依据程序的需求规格说明书，检查程序的功能是否符合他的功能说明。

- 为什么要使用黑盒测试方法：
    - 因为测试数据太庞大，需要挑选典型数据进行测试
- 黑盒测试是以用户的角度，从输入数据与输出数据的对应关系出发进行测试的，又称数据驱动测试
- 黑盒测试是在程序外部接口进行的测试

黑盒测试原则

- 根据程序的重要性和一旦发生故障将造成的损失，来确定测试等级和测试重点
- 认真选择测试策略，以便能尽少使用测试用例发现尽多的程序错误
- 一次完整的软件测试过后，如果程序中遗留的错误过多并且严重，表明改次测试是不足的，测试不足意味着让用户承担隐藏错误带来的危险，测试过度又会带来资源的浪费，因此需要找一个平衡点

黑盒测试策略

1. 首先进行等价类划分，包括输入条件和输出条件的等价划分，将无限测试变成有限测试，这是减少工作量和提高测试效率的最有效方法。
2. 通常都会用到用边界值分析法。经验表明用这种方法设计出测试用例发现程序错误的能力最强。
3. 对于业务流清晰的系统，可以利用场景法贯穿整个测试案例过程，在案例中综合使用各种测试方法。
4. 如果程序的功能说明中含有输入条件的组合情况，则应在一开始就选用因果图法和判定表驱动法。
5. 可以用错误推测法再追加一些测试用例，这需要依靠测试工程师的智慧和经验。
6. 对照程序逻辑，检查已设计出的测试用例的逻辑覆盖程度。如果没有达到要求的覆盖标准，应当再补充足够的测试用例。

1.2. 等价类划分法

概念

- 等价类指某个输入域的子集合

把所有的输入数据划分成若干个子集（等价类），然后从每一个子集（等价类）中选取少量数据设计测试用例的方法。

有效和无效等价类

- 有效等价类：合理数据，关注功能和性能
- 无效等价类：异常数据，关注异常处理

等价类划分法设计测试用例步骤

- 根据需求，划分等价类
- 建立等价类表
- 覆盖等价类表中的全部等价类，设计测试用例原型

软件不能只接受合理的数据，还要经受意外的考验，接受无效的或不合理的数据

等价类划分法设计原则：

- 编号唯一
- 测试用例尽可能多地覆盖有效等价类；（全面地测试软件功能，节省工作量）
- 测试用例只能覆盖一个无效等价类；（方便开发人员定位）
- 覆盖所有的有效和无效等价类

等价类划分法分类

- 弱健壮类型（用的最多）：不考虑组合，考虑异常
    - 从每个有效等价类中选取一个值
    - 对于无效等价类，使用一个无效值，保持其余值都是有效的
- 强健壮类型：考虑组合，考虑异常
- 弱一般类型：不考虑组合，不考虑异常
- 强一般类型：考虑组合，不考虑异常
- 弱：只考虑等价类自身，单缺陷
- 强：考虑了等价类间的相互影响，考虑等价类的组合
- 一般：不考虑无效值
- 健壮：考虑无效值

1.3. 边界值分析法

引入

- 大量错误往往出现在数据范围的边界是，而不是输入、输出的内部
- 由于需求界定不准确、设计不严密、程序员手误等原因

概念

对输入或者输出数据的边界点进行测试的黑盒测试技术，经常用于对等价类划分法进行补充。

边界点：3种：内点、上点、离点

假设给定需求：账号长度[5,11]，要测哪些边界点?

- 内点：范围内部的点
- 上点：边界上的点，无论开闭区间，闭区间在域内，开在域外
- 离点：离上点（边界）最近的点，开在内，闭在外。
    - 对闭区间来说，离点指的是离上点最近的范围外部的点；
    - 对开区间来说，离点指的是离上点最近的范围内部的点。

边界值分析法步骤

1. 确定边界范围
2. 找到边界点
3. 设计测试用例

和等价类的联系

	等价类划分法	边界值分析法
联系	边界值分析是针对输入或输出等价类的边界进行分析	
区别	从某个等价类人选一个作为测试数据	在每个边界上有针对性的选择测试数据

边界值分析法分类

- 一般边界值分析法（最常用）
    - 每次保留一个变量，其它的变量为合法值
    - 保留的变量取遍 `min` 、 `min+` 、 `nom` 、 `max-` 、 `max`
- 健壮边界值分析法
    - 每次保留一个变量，其它的变量为合法值
    - 被保留的变量依次取 `min-` 、 `min` 、 `min+` 、 `nom` 、 `max-` 、 `max` 、 `max+`
    - n 变量测试用例数位 $6n+1$
- 最坏边界值分析法
    - 所有变量取 `min` 、 `min+` 、 `nom` 、 `max-` 、 `max`

    - 五种集合的笛卡尔积
- 最坏健壮边界值分析法
    - 所有变量取 `min-`、`min`、`min+`、`nom`、`max-`、`max`、`max+`
    - 七个集合的笛卡尔积

1.4. 判定表驱动法

概念

用来表示条件和行动的二维表，是分析和表达多逻辑条件下执行不同操作的情况的工具

可以清晰的表达条件、决策规则和应采取的行动之间的逻辑关系

很适合描述不同条件集合下采取行动的若干组合情况

判定表组成

- 条件桩，左上角，一系列条件的集合
- 动作桩，左下角，一系列动作的集合
- 条件项，任何一个条件组合的特定取值和应该执行的动作
- 动作项，每个条件产生的结果集

每个条件项和动作项组成的一列称为规则，应该针对每一个规则设计一条测试用例。

判定表驱动法

根据需求，设计判定表，进而导出测试用例的过程。

在所有黑盒测试方法中，基于判定表的测试是最为严格、最具有逻辑性的测试方法

可以设计出完整的测试用例集合

判定表类型

- 有限条目判定表： 所有条件都是二值条件（真/假）
- 扩展条目判定表：条件可以有多个值
    - 在画判定表的时候可以把最初的有限条目判定表

判定表驱动法步骤

1. 列出所有的条件桩和动作桩
2. 确定规则的个数：对条件桩运用乘法原则
3. 填入条件项
4. 填入动作项
5. 简化判定表，合并相似规则
    - 规则合并

        - 动作项相同，条件项只有一项不同，可以将该项合并，说明执行的动作与该条件的取值无关，称为无关条件
    - 规则包含
        - 无关条件项 `-` 在逻辑上又可包含其他条件项目取值，具有相同动作的规则还可进一步合并。

适用情况

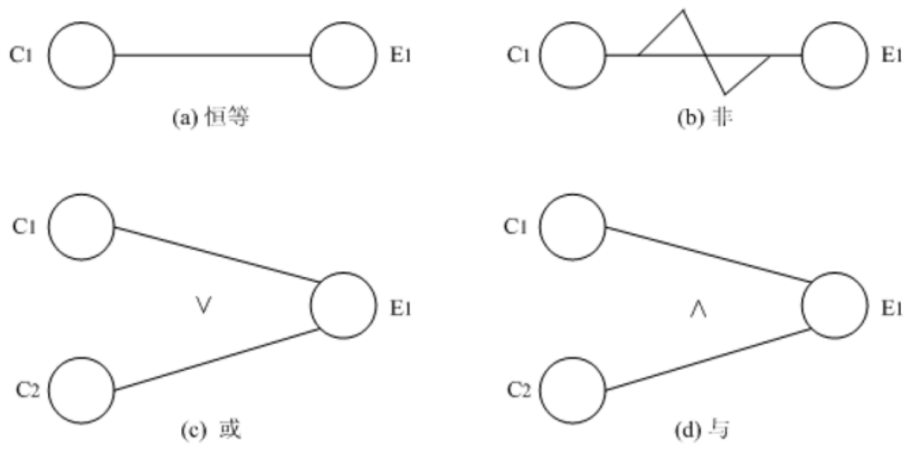
1. 需求规格说明中已给出判定表，或很容易转换成判定表。
2. 条件的排列顺序不影响执行哪些操作。
3. 规则的排列顺序不影响执行哪些操作。
4. 如果某一规则要执行多个操作，这些操作的执行顺序无关紧要。

1.5. 因果图法

概念：用图解的方式分析输入条件的组合，设计测试用例的方法。

关系符号

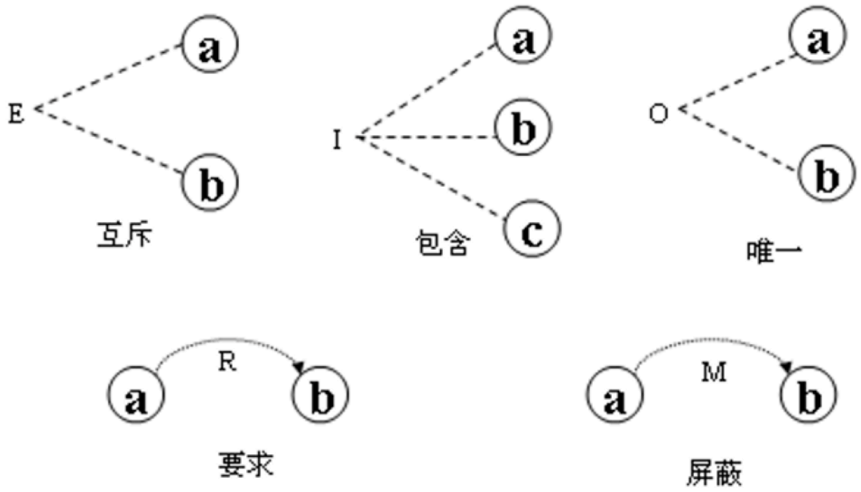
- 因果关系
- 恒等、非、与、或



因果图的基本图形符号

约束符号

- 原因与原因之间，结果与结果之间的关系



- E：互斥，不能同时为1，可能都不成立，但最多一个成立
- I：包含，至少有一个成立，或都不成立
- O：唯一，有且只有一个成立
- R：要求，A为1，B必须为1
- 以上四种是原因与原因之间的约束，下面还有一种是结果与结果之间的约束
- M：屏蔽，A为1，B一定为0，B被屏蔽了

使用场景

- 需求中的条件相互依赖
- 相互制约
- 原因与结果比较明显
- 条件动作关系不明
- 判定表不是很好画，借助因果图去画判定表。

步骤

1. 分析需求，找到原因和结果
2. 找到因果之间的关系
3. 画因果图，画约束关系
4. 转化为判定表
5. 设计测试用例

1.6. 场景法

概念

从一个流程开始，通过遍历所有路径来设计测试用例的过程。经过遍历所有可能的基本流和备选流来完成整个场景

- 基本流：也叫有效流或正确流，模拟用户正确的业务操作流程，只有一个起点和终点
- 备选流：也叫无效流或错误流，模拟用户错误的业务操作流程

步骤

1. 画流程图
2. 描述基本流、备选流
3. 构造用例场景列表，覆盖所有经过基本流、备选流的路径
4. 设计测试用例矩阵，覆盖全部场景
5. 确定测试数据

测试思想

根据《需求规格说明书》描述的包含时间流信息来构造场景，并设计相应的测试用例，使每个场景至少发生一次

为什么使用场景法

现在的系统基本上都是由事件来触发控制流程的。每个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形成事件流。

元素符号:

- V 有效的
- I 无效的
- N/A 无关

1.7. 错误推测法

- 凭借个人经验和直觉，对软件中可能存在的问题设计测试用例的过程。
- 列举出程序中所有可能的错误和容易发生错误的特殊情况，根据他们选择测试用例

---

2. 白盒测试技术

需要完全了解程序结构和处理过程,按照程序内部逻辑测试程序,检验程序中每条通路是否按预定要求正确工作。也被称为结构测试或逻辑驱动测试

定义

使测试充分地覆盖软件系统的内部结构，并以软件结构中的某些元素是否都已得到测试为准则来判断测试的充分性。

2.1. 静态白盒法

在不执行程序的条件下，对文档等材料进行评审

方法

- 同行评审：编写或设计的程序员间进行
- 走查：程序员向其它程序员作陈述
- 评审：最正式的审查类型

查找的问题

- 数据引用错误
- 数据声明错误
- 即计算错误
- 比较错误
- 控制流程错误
- 参数错误
- 输入/输出错误

2.2. 侵入式法

为了验证程序运行是否正确，在源码中注入一些语句进行验证。

- 程序插桩：
- 断言测试：
- 缺陷种植
    - 往原来正确的代码中放入bug，假设放入100个bug。再去测，看能发现多少个缺陷，看缺陷的发现率如何？

比如种 100 个 bug，发现 50 个bug 发现率 50%

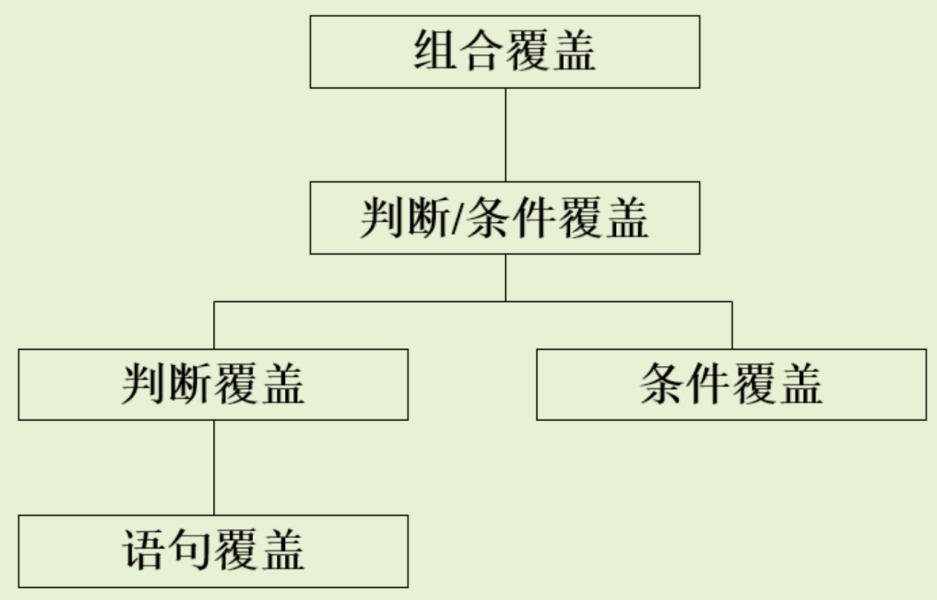
假设把当前种的缺陷除去，发现系统真正的缺陷是 200 个，预估系统有 400 个 bug

2.3. 逻辑覆盖法

把程序内部结构的各个元素进行覆盖，去设计测试用例的方法。

逻辑覆盖法分类

- 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。
- 判定覆盖：又叫分支覆盖，判定覆盖比语句覆盖的标准稍强一些，指通过设计足够的测试用例，使程序中每一个判定分别取一次真值和假值
- 条件覆盖：对于每个判定的每个条件，都应该取一次真值和假值。
- 判定/条件覆盖
    - 判定中的每个条件的每个结果至少出现一次
    - 每个判定本身的所有可能结果都出现一次
    - 同时包含了判定覆盖和条件覆盖
- 条件组合覆盖：使得判定中条件结果的所有可能组合至少出现一次
    - 包含了前面四种
- 路径覆盖：覆盖被测程序所有可能的路径，并未考虑条件的组合



步骤

1. 根据源码，画出流程图；
2. 依据不同的覆盖目标，编写测试用例

2.4. 基本路径法

路径测试

- 从流程图上程序的一次执行对应于从入口到出口的一条路径，针对路径的测试即为路径测试
- 如果程序的每一个独立路径都被测试过

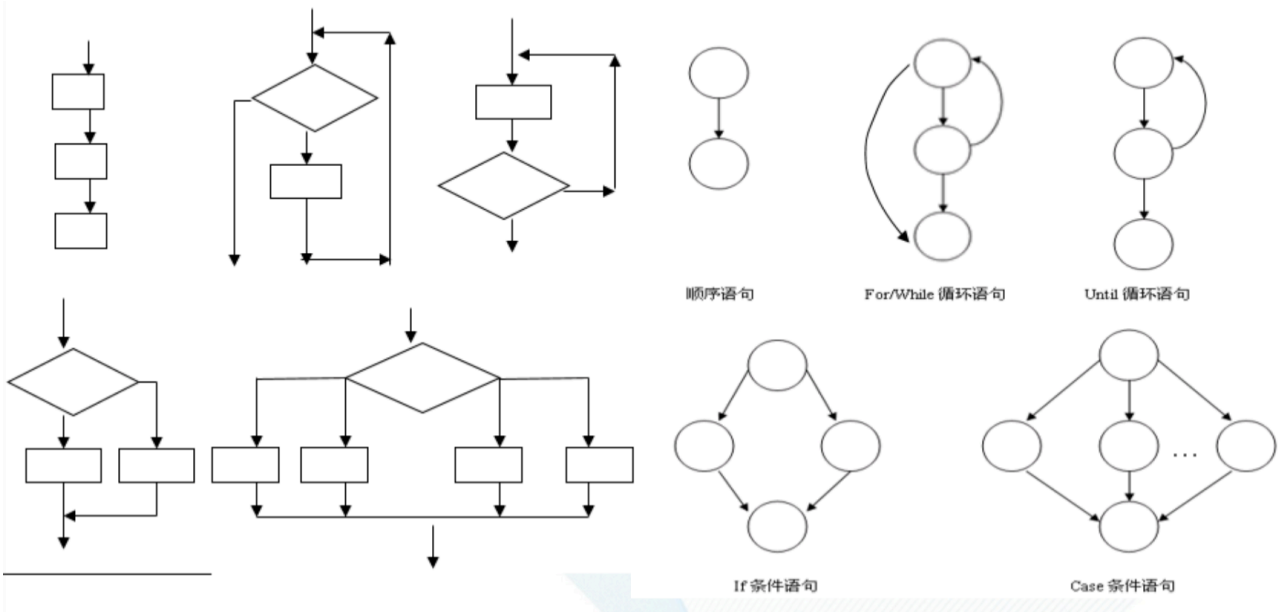
概念

正常情况应该把程序的每一条路径都测试一遍。对于复杂度高的循环多的程序，不能测到所有的路径，找到独立路径，把独立路径测试一遍，就认为测试通过。找独立路径进行测试的方法，称为基本路径法。

- 顺序结构
- `while / for` 循环结构
- `do {} while` 结构（先执行后判定）
- `if else` 结构

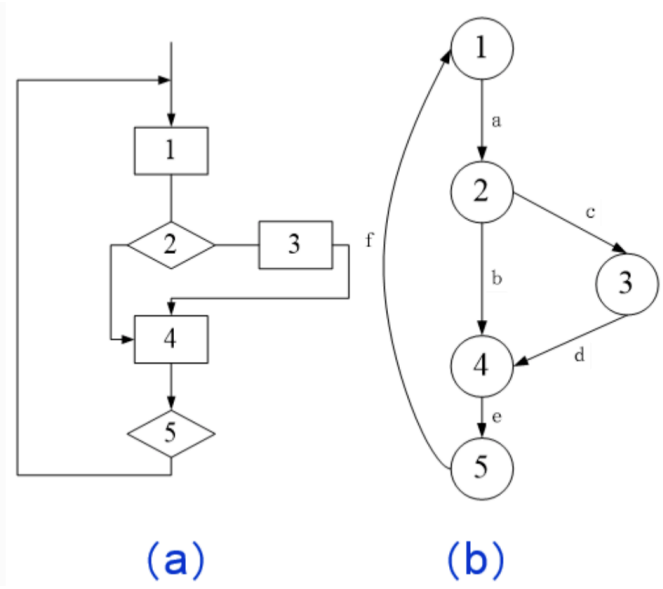
控制流图

- 流程图的简化，由边和节点组成。（节点和控制流线）
- 依据控制流图，计算环形复杂度，环形复杂度的值，就是独立路径的条数。
- 目的是找到独立路径的个数，方便后面设计测试用例
- 包含条件的节点叫判定节点，也叫谓词节点，由判定节点发出的边必须终止于某一个节点，由边和节点所限定的范围被称为区域。



图矩阵

- 方形矩阵，邻接矩阵形式，边表示连接



	1	2	3	4	5
1		a			
2			c	b	
3				d	
4					e
5	f				

测试步骤

1. 根据源代码或者流程图，画出控制流图；
2. 分析控制流图，计算环形复杂度（独立路径条数）

- 环形复杂度的作用帮助确定独立路径条数，验证后续找到的独立路径是否全面

3. 根据独立路径条数，找到相应的独立路径集
4. 根据路径，找到合适的输入数据，编写测试用例。

环形复杂度计算

控制流图中区域的数量对应环形复杂度。

1) $V(G) = E - N + 2$

- E 表示控制流图的边，N 表示控制流图的节点

2) $V(G) = P + 1$

- P 表示判定节点的个数

独立路径

- 独立路径至少用到了之前的路径没有用到的边或一条新的通路
- 独立路径条数等于环形复杂度
- 程序基本路径集指由若干条独立路径组成的集合，数量由环形复杂度确定

确定基本路径集

1. 随便选一条路径作为基线路径：
2. 沿着基线路径从后往前推，遇到判定节点，路径就翻转即换一个判断路径方向，找到一条新的路径，把新路径当做基线路径，重复此步骤直到所有判定节点都被反转

2.5. Junit

拿到一个代码：TestProject 导入 Eclipse 进行白盒测试

搭建 Junit 测试环境

- `@Test` 修饰的就是测试方法
- `@Before` 所有测试方法执行前执行
- `@After` 所有测试方法执行后执行
- `@BeforeClass` 最早执行的
- `@AfterClass` 最晚执行