第3章 软件测试方法

软件测试根据测试过程是否在实际应用环境中运行，可分为静态测试和动态测试。静态测试技术是软件测试中的重要手段之一，其测试对象可以是需求文件、设计文件或者源程序等，适用于新开发的和重用的代码。通常情况下，静态测试是在代码完成并且成功通过编译或汇编后进行的。动态测试是指通过运行程序来发现错误，通过观察代码运行过程来获取系统的各方面信息，并从中发现缺陷。所有的黑盒测试和绝大多数的白盒测试都可以算作动态测试。

本章要点

- 静态测试的内容及方法
- 动态测试的内容及方法
- 主动测试与被动测试
- 白盒测试的内容及方法
- 黑盒测试的内容及方法
- 白盒测试和黑盒测试的比较

3.1 静 态 测 试

静态测试是不执行被分析的程序测试方法，通过对源代码进行研读、找出其中的错误或可疑之处，收集度量数据。静态测试包括对软件需求和设计规格说明书的评审、对程序代码的复审等。静态测试的查错和分析功能是其他方法所不能替代的，对于发现逻辑设计和编码错误具有重要价值。

静态测试的内容包括代码检查、静态结构分析等。它可以采用人工进行，充分发挥人的逻辑思维优势，也可以借助软件工具自动进行，提升工作效率。

3.1.1 代码检查

代码检查是发现逻辑设计和编码缺陷的重要技术。它包括桌面检查、代码审查和走查等方法。代码检查应该在编译和动态测试之前进行。在检查前，应准备好需求描述文档、程序设计文档、程序的源代码清单、代码编写标准和代码错误检查表等。

在实际应用中，代码检查具有快速发现缺陷的能力，通常能够揭示 30%～70%的逻辑设计和编码缺陷，以及其他问题。它有助于识别问题本身，而不仅是问题的表现。但代码检查非常耗费时间，并且需要经验和知识的积累。

代码检查可以手动进行，这依赖团队成员的逻辑思维和经验。此外，还可以使用测试软

件进行自动化测试,以提高测试效率和降低劳动强度。

1. 桌面检查

桌面检查是一种由开发者独立进行的代码检查方法,通常在个人工作环境中进行。开发者通过仔细阅读代码并自行发现潜在问题,然后进行修正或改进。

桌面检查通常涉及对代码的逐行检查,包括变量、函数、逻辑结构等方面。开发者需要对代码进行细致的分析,以确保代码的质量和可读性。

2. 代码审查

代码审查是通过人工审查软件代码的方法,检查其中的错误或潜在问题。审查可以由一个或多个开发者进行,他们对代码进行仔细检查,以发现潜在的问题并提出改进建议。

通常,代码审查分为两种方式:主持式审查和非主持式审查。主持式审查由一名主持者引导,参与者对代码进行讨论和审查。非主持式审查则由开发者自行对代码进行检查,然后提交审查结果。

3. 走查

走查是一种面对面的代码检查方法,由项目团队成员集体参与。在走查会议上,开发者将代码展示给其他团队成员,共同讨论代码中的问题和改进点。

走查会议通常由项目负责人主持,开发者逐行展示代码,并与团队成员一起讨论。团队成员可以提出问题、建议改进,并共同寻找解决方案。

4. 代码检查常用检查项

通过代码检查,可以获得软件组成的重要基本信息,如变量标识符、过程标识符、常量等,具体包括:

- 标号交叉引用表。列出各模块出现的全部标号,并标注其属性,如已说明、未说明、已使用、未使用等。同时包括在模块以外的全局标号、计算标号等。
- 变量交叉引用表。列出各变量的属性,如已说明、未说明、隐式说明、类型及使用情况等。进一步区分变量是否出现在赋值语句的右边,是否属于普通变量、全局变量或特权变量等。
- 子程序、宏和函数表。列出各子程序、宏和函数的属性,包括已定义、未定义和定义类型、参数表、输入参数个数、顺序、类型、输出参数个数、顺序、类型、已引用、未引用、引用次数等。
- 等价表。列出等价语句或等值语句中出现的全部变量和符号。
- 常数表。列出全部数字常数和字符常数,并指出它们在哪些语句中首先被定义。

在进行代码检查时,需要关注以下检查项目。

- 检查变量的交叉引用表。重点检查未说明变量和违反类型规定的变量,并逐行检查变量的引用、使用序列、临时变量在某条路径上的重写情况,以及局部变量、全局变量与特权变量的使用情况。
- 检查标号的交叉引用表。验证所有标号的正确性,检查标号的命名是否正确,以及转向指定位置的标号是否正确。
- 检查子程序、宏和函数表。验证每次调用和所调用位置是否正确,确定每次调用的子程序、宏和函数是否存在,检验调用序列中调用方式与参数顺序、个数、类型上的一致性。

- 等价性检查。检查所有等价变量类型的一致性，解释所包含的类型差异。
- 标准检查。用标准检查工具软件或手工检查程序中违反标准的问题。
- 风格检查。检查程序在设计风格上是否合适。
- 比较控制流。比较程序员设计的控制流图和实际控制流图，寻找和解释每个差异，并对文档进行修改和错误修正。
- 选择、激活路径检查。在程序员设计的控制流图上选择路径，再到实际控制流图上激活这条路径，以验证路径的正确性。
- 对照程序说明检查。阅读程序源代码，逐行进行分析思考，比较实际的代码和期望的代码，以发现程序中的错误和问题。
- 充分文档检查。通过编写文档，对文档进行检查和测试，这有助于测试人员发现更多的错误。同时，管理层可以通过审核这些文档来获取关于模块质量、完整性、采用的测试方法以及开发人员专业能力等重要信息。

3.1.2 静态结构分析

静态结构分析是利用图形化形式展示程序内部结构的方法，帮助测试人员分析程序的结构。这种方法主要用于白盒测试，提供了测试人员深入理解代码内部结构的手段。研究表明，程序员约有 38％的时间用于理解软件系统，而静态结构分析则通过图形化表达程序结构，提高了对代码的理解效率。

在静态结构分析中，测试人员利用各种图表来表示程序的结构特征，如系统结构、数据接口、内部控制逻辑等，并生成函数调用关系图、模块控制流图、内部文件调用关系图、子程序表、宏和函数参数表等。

常用的关系图主要有函数调用关系图和模块控制流图。

- 函数调用关系图列出了系统中所有的函数，并用连线表示它们之间的调用关系。通过这个图表，可以清晰地了解函数之间的调用关系，检查调用频率、函数之间的依赖关系，以及是否存在孤立的函数没有被调用等。这有助于发现系统结构中的缺陷和优化调用关系。
- 模块控制流图是由许多节点和连接节点的边组成的图形，其中每个节点代表一条或多条语句，边表示控制流向。这种图形化表达方式直观地展示了函数的内部结构和控制流程。通过分析模块控制流图，测试人员可以快速发现代码中的错误和缺陷，了解程序的执行路径和逻辑结构。

模块控制流图图元符号如图 3-1 所示。

用模块控制流图表示的基本程序结构如图 3-2 所示。

静态结构分析有以下几种形式。

1. 类型和单元分析

通过扩展程序设计语言中的数据类型，如数组下标类型和循环语句中的计数器类型等，以强化数据类型的检查。这样的扩展可以使得程序在静态预处理阶段就能够发现程序中的类型错误。

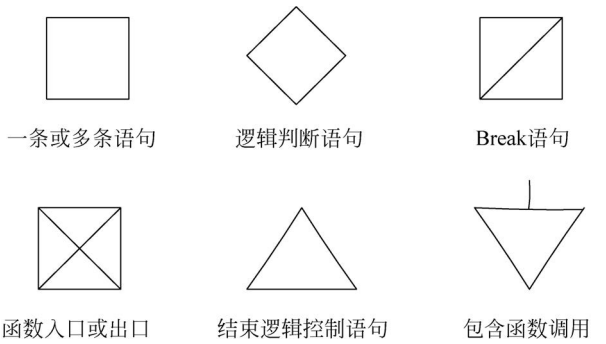

图 3-1　模块控制流图图元符号

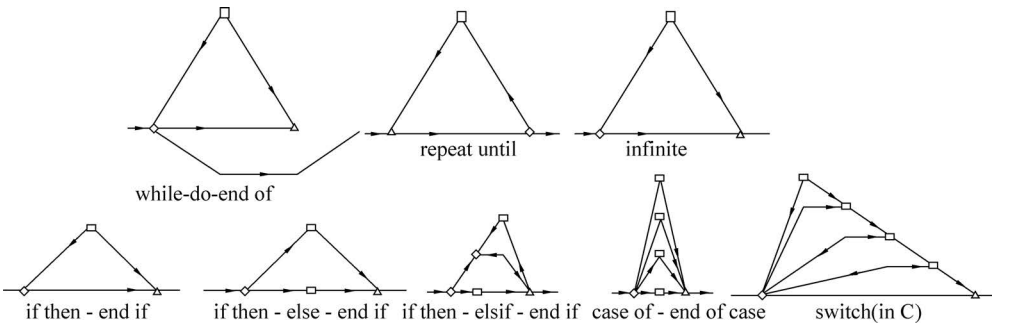

图 3-2　模块控制流图的基本结构

2. 引用分析

在静态错误分析中,最广泛使用的技术就是发现引用异常。它主要用于发现在程序控制路径中变量在赋值之前被引用,或者在赋值之后未被引用的情况。引用异常检测需要沿着程序的每一条路径进行检查,通常采用深度优先或类似的遍历算法,建立定义表和未引用表,以便检测和记录异常引用的变量。

3. 表达式分析

通过对程序中的表达式进行分析,发现并纠正表达式中的错误,如不正确使用括号、数组下标越界、除数为零等。此外,还包括对浮点数计算误差的检查,因为使用二进制数表示浮点数可能导致计算结果与预期不符。

4. 接口分析

接口分析主要用于检查模块之间接口的一致性以及外部数据库之间接口的一致性。这种分析检查实参与形参在类型、数量、维数、顺序上的一致性,以及全局变量和公共数据区在使用上的一致性。

以“猜数字游戏”案例项目为例,该项目是一个简单的命令行交互式游戏,在这个游戏中,计算机生成一个随机数,玩家需要在有限次数内猜测这个数。游戏程序会根据玩家猜测提供反馈,告诉玩家猜测的数字是太高、太低还是正确。该游戏的核心功能包括生成随机数、接收玩家输入、验证输入的有效性、比较输入的数字与目标数字,以及控制游戏流程。导出其函数调用关系图如图 3-3 所示。

从该函数调用关系图(如图 3-3 所示)中,可以得到以下信息。

- 函数之间的调用关系符合设计规格说明书中的要求。

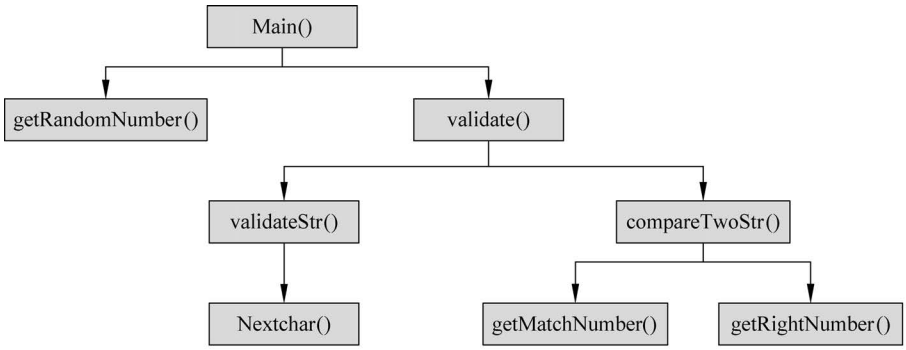

图 3-3　函数调用关系图

- 不存在递归调用。
- 调用层次最深为 4 层。
- 不存在独立的和没有被调用的函数。
- 比较重要的函数有 validate()、compareTwoStr()、validateStr()等。

也可以对每个函数使用模块控制流图，考查每个函数的结构是否合理、是否存在错误等。

3.2　动 态 测 试

动态测试是软件测试中最常用的方法之一，通过运行程序来发现错误，并观察代码的执行过程以获取系统行为、变量实时结果、内存、堆栈、线程以及测试覆盖率等信息，以判断系统是否存在问题或发现缺陷。动态测试分为主动测试和被动测试两种方法。

3.2.1　主动测试

主动测试是一种常见的测试方法，测试人员通过向被测试对象发送请求或者利用数据、事件来驱动被测试对象的行为，从而验证其反应或输出结果。在主动测试中，测试人员与被测试对象之间存在直接的相互作用，测试人员完全控制被测试对象处于测试状态，而不是实际工作状态。这种方法通常用于测试环境中，如图 3-4(a)所示。

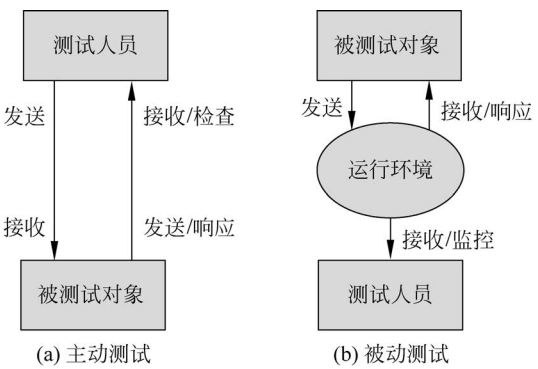

图 3-4　主动测试与被动测试

3.2.2 被动测试

被动测试适用于产品的在线测试,特别适用于性能测试和在线监控。在被动测试中,软件产品在实际运行环境中运行,测试人员被动地监控产品的运行,通过一定的机制来获取系统运行的数据,包括输入、输出数据,如图3-4(b)所示。这种方法常用于嵌入式系统测试和大规模复杂系统的性能测试。相比主动测试,被动测试不需要测试人员设计测试用例,而是通过建立监控程序获取系统运行的各种数据,但数据的完整性可能无法保证。

3.3 白盒测试

白盒测试,又称为玻璃盒测试、结构化测试或逻辑驱动测试,专注于深入探究软件产品内部细节和逻辑结构的测试方法。它将被测程序视为一个透明的盒子。白盒测试利用系统构件层设计的一部分来定义控制结构,从而生成测试用例。这需要对系统内部结构和工作原理有清晰的理解。白盒测试的准备时间较长,如果要完成覆盖全部程序语句、分支的测试,一般要花费比编程更长的时间。白盒测试对技术的要求较高,测试成本也比较大。

白盒测试的测试方法有程序插桩法、逻辑覆盖法和基本路径法等。

3.3.1 程序插桩法

在程序调试过程中,常常需要在代码中插入特定的打印语句,以输出相关信息,帮助了解程序的动态特性,如程序的执行路径或特定变量在特定时刻的取值。这一思想逐渐演化为程序插桩技术,在软件动态测试中作为基本的测试手段得到了广泛应用。

简而言之,程序插桩技术是通过向被测程序中插入特定操作来实现测试目的的方法。通过向源程序中添加一些语句,可以检测程序语句的执行情况、变量的状态变化等。例如,通过程序插桩技术,可以了解一个程序在某次运行中所有可执行语句的覆盖情况,或者语句的实际执行次数。

通过插入的语句获取程序执行时的动态信息,类似于在机器的特定部位安装记录仪表来了解其动态特性。这些插入的语句可被称为“探测器”,用以实现探测和监控的功能。通过在程序的特定部位插入记录动态特性的语句,最终捕获程序执行过程中的重要事件和历史信息。

在设计插桩程序时需要考虑的问题如下。

- 探测信息。需要明确要探测哪些信息,例如,变量值的变化、函数的调用、条件分支的执行、异常的抛出等。
- 设置探测点。在程序的哪些部分设置探测点,通常与测试目标的执行流程一致,如程序块的首个可执行语句之前、循环语句处、条件语句分支处、输入/输出语句后等。
- 探测点数量。需要根据具体情况和程序结构考虑设置多少个探测点,原则上应该设置最少的探测点,但涉及复杂的控制结构时需进行具体分析。

在应用程序插桩技术时,常在程序的特定部分插入用来判断变量特性的语句,称为断言语句。这些语句是程序正确性证明的基本步骤,虽不是严格证明,但在实践中非常实用。一些编译系统支持使用表达式形式的断言语句来进行验证。

3.3.2 逻辑覆盖法

逻辑覆盖法也是一类常用的白盒动态测试方法，其基本原理是利用程序内部逻辑结构，通过遍历不同的程序路径来实现对程序测试的覆盖。逻辑覆盖法要求测试人员对程序的逻辑结构有清晰的了解。

逻辑覆盖法涵盖了一系列测试过程，逐渐进行越来越完整的通路测试。根据覆盖的详尽程度，可以分为以下几种覆盖准则：语句覆盖、判定覆盖、条件覆盖、条件判定覆盖、条件组合覆盖和路径覆盖。

1. 语句覆盖

语句覆盖是保证代码中的每条语句都至少执行一遍。语句覆盖可以很直观地从源代码推导测试用例，无须详细分析每条判定表达式。但不能检查判断逻辑是否有问题。尽管是最基本的覆盖准则，但语句覆盖仍然是必要的。语句覆盖可以通过测试覆盖率工具，如TrueCoverage、PureCoverage 等来检查。

语句覆盖率是已执行的语句数与总语句数的比例。高质量软件要求语句覆盖率达到100%，而对于一般软件，也应该保持在 90%以上，以确保测试的充分性。

考虑以下程序：

```
if(A > 1 and B = 0)
X = X/A;
if(A = 2 or X > 1)
X = X + 1;
```

这段代码的程序流程图如图 3-5 所示。

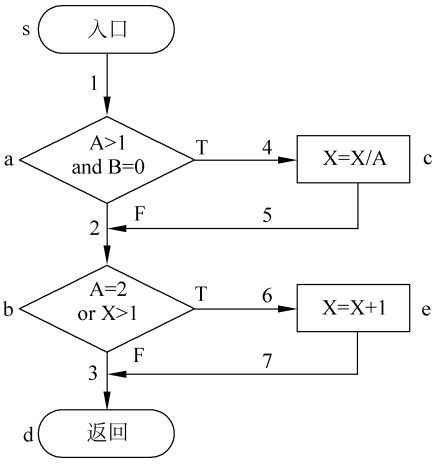

图 3-5　被测试模块的流程图

为了使每个语句都执行一次，程序的执行路径应该是 sacbed。为此，只需要输入下面的测试数据(实际上 X 可以是任意实数)。

A＝2，B＝0，X＝4

语句覆盖对程序的逻辑覆盖很少。在上面的例子中，两个判定条件都只测试了条件为真的情况，忽略了条件为假时的处理错误。此外，语句覆盖也未考虑到判定表达式中每个条

件取不同值时的情况。在上面的例子中,为了执行 sacbed 路径,只需两个判定表达式(A>1) and(B=0)和(A=2)or(X>1)都取真值,因此使用上述一组测试数据就够了。但是,如果程序中把第1个判定表达式中的逻辑运算符"and"错写成"or",或把第2个判定表达式中的条件"X>1"误写成"X<1",使用上面的测试数据就不能查出这些错误。

综上所述,可以看出语句覆盖是很弱的逻辑覆盖标准。为了更充分地测试程序,可以采用下述覆盖准则。

2. 判定覆盖

判定覆盖又称判断覆盖、分支覆盖,是比语句覆盖稍强的覆盖标准。它要求设计足够的测试用例,以确保每个判断都获得每一种可能的结果至少一次。虽然判定覆盖在单元测试中很常用,但是判定覆盖准则依然不够严格。

对于上面的例子来说,能够分别覆盖路径 sacbed 和 sabd 的两组测试数据,或者可以分别覆盖路径 sacbd 和 sabed 的两组测试数据,都满足判定覆盖标准。例如,用下面两组测试数据就可做到完全的判定覆盖。

- A=3,B=0,X=3 (覆盖 sacbd)
- A=2,B=1,X=1 (覆盖 sabed)

判定覆盖比语句覆盖强,但是对程序逻辑的覆盖程度仍然不高,如上面的测试数据只覆盖了程序全部路径的一半。

3. 条件覆盖

条件覆盖是指程序中每个判断中的每个条件的所有可能的取值至少要都执行一次。它是一种对控制流更敏感的覆盖标准,独立地测量每一个子表达式的执行情况。尽管条件覆盖提供了更细粒度的覆盖度量,但它并不能完全取代判定覆盖。

考虑以下程序:

```
if (a && b) {
      return true;
}else {
      return false;
}
```

通过以下两个测试用例可以得到100%的条件覆盖率。

- a=true,b=false
- a=false,b=true

但上述测试用例条件都不会使 if 的逻辑运算式成立,因此不符合判定覆盖的条件。

4. 条件判定覆盖

条件判定覆盖是一种综合了判定覆盖和条件覆盖的逻辑覆盖标准。它要求设计足够多的测试用例,以确保每个判定中的每个条件的所有可能取值都至少出现一次,并且每个判定的各种可能结果也至少出现一次。条件判定覆盖综合了判定覆盖和条件覆盖的优点,但它仍然存在一些局限性。例如,它没有考虑单个判断对整体结果的影响,可能会导致某些逻辑错误未被发现。

下面以一段代码为例,说明条件判定覆盖的测试用例的设计过程:

```
int x, y;
double z;
if(x > 0 && y > 0)
        z = z / x;
if(x > 1 || z > 1)
        z = z + 1;
z = y + z;
```

对其设计测试用例的第一步就是绘制出它的程序流程图，如图 3-6 所示。

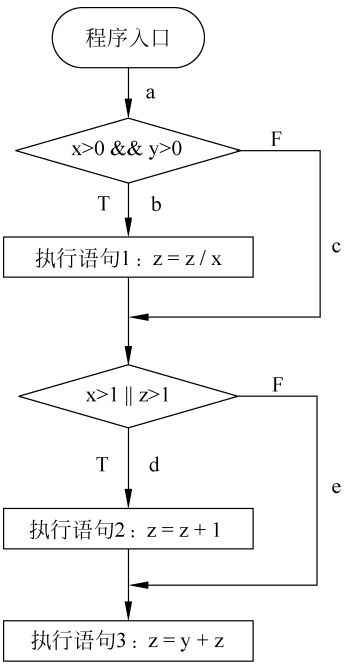

图 3-6　程序流程图

条件判定覆盖的测试用例设计可通过取条件覆盖和判定覆盖的用例的并集来实现。

条件覆盖的核心思想是确保每个判断的所有逻辑条件的每种可能取值至少执行一次。

对于判断语句 x＞0 && y＞0：

条件 x＞0 取真为 T1，取假为－T1。

条件 y＞0 取真为 T2，取假为－T2。

对于判断语句 x＞1 ‖ z＞1：

条件 x＞1 取真为 T3，取假为－T3。

条件 z＞1 取真为 T4，取假为－T4。

设计测试用例如表 3-1 所示。

表 3-1　条件覆盖的测试用例

输　　入	通 过 路 径	条 件 取 值
x＝7，y＝1，z＝3	a-b-d	T1，T2，T3，－T4
x＝－1，y＝－3，z＝0	a-c-e	－T1，－T2，－T3，－T4

判定覆盖的目标是确保每个判断的取真分支和取假分支至少执行一次。

对于判断语句 x＞0 && y＞0：取真为 M，取假为－M。

对于判断语句 x>1 ‖ z>1：取真为 N,取假为－N。

设计测试用例如表 3-2 所示。

表 3-2　判定覆盖的测试用例

输　　入	通 过 路 径	判 定 取 值
x=7,y=1,z=3	a-b-d	M,N
x=－1,y=－3,z=0	a-c-e	－M,－N

综合表 3-1 和表 3-2,条件判定覆盖的测试用例如表 3-3 所示。

表 3-3　条件判定覆盖的测试用例

输　　入	通 过 路 径
x=7,y=1,z=3	a-b-d
x=－1,y=－3,z=0	a-c-e

5. 条件组合覆盖

条件组合覆盖也称为多条件覆盖,要求设计足够的测试用例,以涵盖每个判断条件的所有可能组合。多条件覆盖需要的测试用例是用一个条件的逻辑操作符的真值表来确定的。这种测试方法是一种彻底的测试方法,可以确保对程序逻辑的全面覆盖。

然而,尽管条件组合覆盖是一种全面的测试方法,但它依然存在一些缺点。

- 它可能需要非常冗长乏味的测试用例来确定最小设置,尤其是在处理一些非常复杂的布尔表达式时。
- 对于相似复杂性的条件,需要大量的变化,这可能增加测试用例设计的难度。
- 存在路径遗漏的风险,即使覆盖了各种条件组合,也可能会遗漏某些程序路径。

以上一段代码为例,给出其测试用例的设计过程。

对各判断语句的逻辑条件的取值组合标记如下。

(1) x>0,y>0,记作 T1,T2,条件组合取值 M。

(2) x>0,y≤0,记作 T1,－T2,条件组合取值－M。

(3) x≤0,y>0,记作－T1,T2,条件组合取值－M。

(4) x≤0,y≤0,记作－T1,－T2,条件组合取值－M。

(5) x>1,z>1,记作 T3,T4,条件组合取值 N。

(6) x>1,z≤1,记作 T3,－T4,条件组合取值 N。

(7) x≤1,z>1,记作－T3,T4,条件组合取值 N。

(8) x≤1,z≤1,记作－T3,－T4,条件组合取值－N。

设计测试用例如表 3-4 所示。

表 3-4　多条件覆盖的测试用例

输　　入	通 过 路 径	条 件 取 值	覆盖组合号
x=1,y=3,z=2	a-b-d	T1,T2,－T3,T4	(1),(7)
x=2,y=0,z=8	a-c-d	T1,－T2,T3,T4	(2),(5)
x=－1,y=1,z=1	a-c-e	－T1,T2,－T3,－T4	(3),(8)
x=－2,y=－3,z=0	a-c-e	－T1,－T2,－T3,－T4	(4),(8)
x=5,y=9,z=0	a-b-d	T1,T2,T3,－T4	(1),(6)

6. 路径覆盖

路径覆盖是指测试用例中实际执行的路径数量与被测试模块所有可能执行路径的比率。在路径覆盖中，只需要考虑所有可能的执行路径，对于不可能执行的路径是不考虑的。而且对于一些大型程序，其包含的路径总量是非常庞大的，如果要把所有路径都找出来去覆盖也是不现实的。因此可以借助以下一些方法来寻找程序中的路径。

1）单个判断语句的路径计算

对于单个判断语句，路径分为判断条件为真和判断条件为假两条路径。在不考虑判断分支中的路径分支时，路径数与判断分支数相等。

2）单个循环语句中的路径计算

在循环语句中，通常循环的每次迭代都可以看作一条路径，但这样计算出的路径的测试工作量太大，所以通过以下方式简化循环中的路径计算。

（1）如果循环中的条件始终满足，循环内的语句必定执行，整个循环可视为一条路径。

如以下循环语句：

```
int i, sum;
for(i = 1;i < = 100;i++)
{
        sum += i;
}
```

循环中的条件一定满足，循环语句 sum＋＝i 一定会被执行，直到 i＝101 时循环结束。在这种情况下，整个循环语句可能的执行路径可看作只有一条。

（2）如果循环中的条件不一定满足，循环内的语句可能被执行，也可能不被执行，循环可视为两条路径：一条循环执行，另一条循环不执行。

如以下循环语句：

```
int sum = 0;
while  (x < 100) {
       sum += x;
       x++;
}
```

在上面的循环中，当 x≥100 时，循环不执行，因此可能的执行路径有两条。

（3）当循环可能被中断时，每次迭代都有可能产生新的路径，路径数量等于输入参数的等价类的所有组合数量。

如以下循环语句：

```
int GetSum(int maxnum){
      int sum = 0;
      for(int i = 0;i < 50;i++){
            sum += i;
            if(sum > = maxnum){
                  break;
            }
      }
```

```
        return sum;
    }
```


在上面这段程序中,由于参数 maxnum 不同,循环的每次迭代都有两种可能,一种是满足判断条件,另一种是不满足判断条件,所以可以认为路径的数量是 51 条,其中 50 条是 for 循环中的 if 判断条件都满足的情况,另外 1 条是 for 循环中 if 判断条件没有被满足,break 语句没有被执行的情况。实际上,程序中可能执行的路径总数不多于输入参数的等价类的所有组合数量,上例中 GetSum()函数的输入参数实际上有 51 个等价类,可能的执行路径数量也是 51 条。

(4) 有嵌套判断或循环时的路径计算,对于有嵌套判断或循环的程序,一般先从内层开始计算路径,然后通过相乘得到总的路径数。

下面是有嵌套判断语句的例子。

```
int GetMaxnum(int a,int b,int c){
        if(a>=b){
                if(a>=c)
                        return a;
                else
                        return c;
        }else{
                if(b>=c)
                        return b;
                else
                        return c;
        }
}
```

对以上程序的路径统计,先计算出第一个 if 内的路径为 2 条,再计算出 else 中的路径为 2 条,因此相乘得到总路径为 4 条。

3.3.3 基本路径法

基本路径法是一种基于程序控制流图的技术,通过分析控制结构的复杂性,导出基本可执行的路径集合,从而设计测试用例。这种方法旨在确保测试用例能够覆盖程序的每个可执行语句至少一次。在基本路径法中,需要使用程序的控制流图进行可视化表达。

控制流图是一种图形化描述程序控制流的方法,圆圈称为控制流图的一个结点,表示一个或多个无分支的语句或源程序语句;箭头称为边或连接,代表控制流。在将程序流程图简化成控制流图时,应注意以下几点。

- 在选择或多分支结构中,分支的汇聚处应有一个汇聚结点。
- 边和结点圈定的区域叫作区域,当对区域计数时,图形外的区域也应记为一个区域。

控制流图表示如图 3-7 所示。

环路复杂度是一种度量程序模块或函数复杂度的指标。它通过计算程序中基本的独立路径数量来量化程序逻辑的复杂性。环路复杂度的核心目的是提供一个测试用例数量的上限,以确保每个语句至少被执行一次。独立路径是指那些在执行过程中不会重复访问之前已访问过的路径。计算环路复杂度通常有以下三种方法。

- 将流图中区域的数量作为环路复杂度的度量。

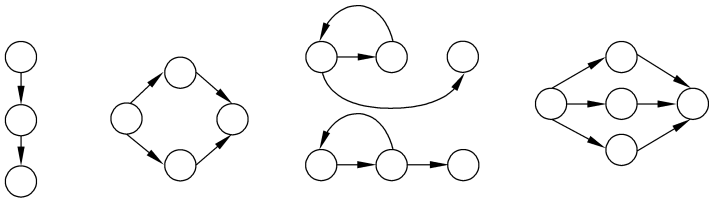

图 3-7　控制流图表示

- 给定流图 G 的环路复杂度 $V(G)$，定义为 $V(G)=E-N+2$，其中，E 是流图中边的数量，N 是流图中结点的数量。
- 给定流图 G 的环路复杂度 $V(G)$，定义为 $V(G)=P+1$，其中，P 是流图 G 中判定结点的数量。

基本路径测试法适用于模块的详细设计和源代码级别的测试。其步骤如下。

- 以详细设计或源代码为基础，导出程序的控制流图。
- 计算得出控制流图 G 的环路复杂度 $V(G)$。
- 确定独立路径的基本集。
- 生成测试用例，确保基本路径集中的每条路径至少被执行一次。

测试用例执行后与预期结果进行比较，确保所有可执行语句至少被执行一次。然而需要注意的是，一些独立路径可能与程序的正常控制流相关联，这些路径的测试可能会包含在其他测试路径中。

以下是具体实例的详细解释，以便理解基本路径测试法的细节。

对于下面的程序，假设输入的取值范围是 1000＜year＜2001，使用基本路径测试法为变量 year 设计测试用例，以满足基本路径覆盖的要求。

```
int IsLeap(int year)
{
       if (year % 4 == 0)
       {
              if ( year % 100 == 0)
              {
                     if ( year % 400 == 0)
                     leap = 1;
                     else
                     leap = 0;
              }
              else
                     leap = 1;
       }
       else
           leap = 0;
       return leap;
}
```

根据源代码绘制程序的控制流图如图 3-8 所示。

通过控制流图，计算环路复杂度 $V(G)$＝区域数＝4。

线性无关的路径集为

(1) 1-3-8。

(2) 1-2-5-8。

(3) 1-2-4-7-8。

(4) 1-2-4-6-8。

设计测试用例如下。

路径(1)：输入数据 year=1999,预期结果 leap=0。

路径(2)：输入数据 year=1996,预期结果 leap=1。

路径(3)：输入数据 year=1800,预期结果 leap=0。

路径(4)：输入数据 year=1600,预期结果 leap=1。

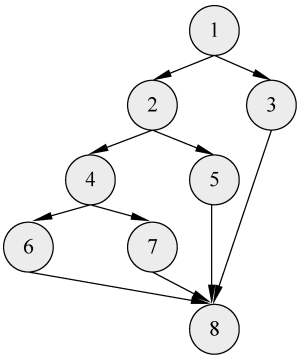

图 3-8　控制流图

3.4　黑盒测试

黑盒测试又叫功能测试,它主要关注被测软件功能的实现,而不是其内部逻辑。在黑盒测试中,被测对象的内部结构、运作情况对测试人员是不可见的,测试人员把被测试的软件系统看成一个黑盒子,并不需要关心盒子的内部结构和内部特性,而只关注软件产品的输入数据和输出结果,从而检查软件产品是否符合它的功能说明。黑盒测试有多种方法,如场景法、等价类划分、边界值分析、因果图法、决策表法等。

3.4.1　黑盒测试方法

1. 等价类划分法

根据软件测试原则,要做到穷举测试通常是不可能的,事实上也是不必要的。因此需要采用有效的测试用例选择方法。

等价类划分法是一种典型的黑盒测试方法,用这种方法设计测试用例可以不用考虑程序的内部结构,基于对程序的要求和说明,即需求说明书来选择测试用例。

等价类划分法的核心思想是确保测试用例的代表性,以覆盖各种可能情况。这是为了在测试中尽可能发现潜在的错误,而不是仅发现与一个特定测试用例相关的错误。

在划分等价类时,有如下一些规则应该遵循。

(1) 如果输入条件规定了取值范围或个数,则可确定一个有效等价类和两个无效等价类。例如,输入值是选课人数,取值为0～100,那么有效等价类是：0≤学生人数≤100。无效等价类是：学生人数<0；学生人数>100。

(2) 如果输入条件规定了输入值的集合或是规定了“必须如何”的条件,则可确定一个有效等价类和一个无效等价类。例如,输入值是日期类型的数据,那么有效等价类是日期类型的数据；无效等价类是非日期类型的数据。

(3) 如果输入是布尔表达式,可以分为一个有效等价类和一个无效等价类,如要求密码非空,则有效等价类为非空密码,无效等价类为空密码。

(4) 如果输入条件是一组值,且程序对不同的值有不同的处理方式,则每个允许的输入值对应一个有效等价类,所有不允许的输入值的集合为一个无效等价类。例如,输入条件“职称”的值是初级、中级或高级,那么有效等价类应该有三个：初级、中级、高级。无效等价

类有一个：其他任何职称。

(5) 如果规定了输入数据必须遵循的规则，则可以划分出一个有效的等价类(符合规则)和若干个无效的等价类(从不同的角度违反规则)。

设计测试用例的步骤可以归结为以下三步。

(1) 对每个输入和外部条件进行等价类划分，画出等价类表，并对每个等价类进行编号。

(2) 设计一个测试用例，使其尽可能多地覆盖有效等价类，重复这一步，直到所有的有效等价类被覆盖。

(3) 为每一个无效等价类设计一个测试用例，确保覆盖不同的无效情况。

下面将以一个测试 NextDate 函数的具体实例为出发点，讲解使用等价类划分法设计测试用例的过程。该函数接收三个输入参数(年、月、日)，函数返回输入日期后面一天的日期：1≤月份≤12，1≤日期≤31，1812 ≤年≤2012。给出等价类划分表并设计测试用例。

划分等价类，得到等价类划分表，如表 3-5 所示。

表 3-5　等价类划分表

输入及外部条件	有效等价类	等价类编号	无效等价类	等价类编号
日期的类型	数字字符	1	非数字字符	8
年	1812～2012	2	小于 1812	9
			大于 2012	10
月	1～12	3	小于 1	11
			大于 12	12
非闰年的 2 月	日取值为 1～28	4	日小于 1	13
			日大于 28	14
闰年的 2 月	日取值为 1～29	5	日小于 1	15
			日大于 29	16
月份为 1 月、3 月、5 月、7 月、8 月、10 月、12 月	日取值为 1～31	6	日小于 1	17
			日大于 31	18
月份为 4 月、6 月、9 月、11 月	日取值为 1～30	7	日小于 1	19
			日大于 30	20

为有效等价类设计测试用例，如表 3-6 所示。

表 3-6　有效等价类的测试用例

序号	输入数据			预期输出			覆盖范围(等价类编号)
	年	月	日	年	月	日	
1	2003	3	15	2003	3	16	1,2,3,6
2	2004	2	13	2004	2	14	1,2,3,5
3	1999	2	3	1999	2	4	1,2,3,4
4	1970	9	29	1970	9	30	1,2,3,7

为无效的等价类设计测试用例，如表 3-7 所示。

表 3-7　无效等价类的测试用例

序号	输入数据			预期结果	覆盖范围（等价类编号）
	年	月	日		
1	xy	5	9	输入无效	8
2	1700	4	8	输入无效	9
3	2300	11	1	输入无效	10
4	2005	0	11	输入无效	11
5	2009	14	25	输入无效	12
6	1989	2	−1	输入无效	13
7	1977	2	30	输入无效	14
8	2000	2	−2	输入无效	15
9	2008	2	34	输入无效	16
10	1956	10	0	输入无效	17
11	1974	8	78	输入无效	18
12	2007	9	−3	输入无效	19
13	1866	12	35	输入无效	20

通过案例可以了解，等价类划分法是一种有效的黑盒测试方法，能够设计出能够充分覆盖程序功能的测试用例，同时避免了测试用例的冗余。然而，要使用等价类划分法，测试人员需要对程序规格说明书进行深入理解，并且合理地划分等价类。有时，规格说明书可能没有明确定义对于无效输入的预期输出，这可能导致测试人员需要花费大量时间来确定这些测试用例的预期输出。因此，这也被认为是等价类划分法的一个局限性。

2. 边界值分析法

人们从长期的测试工作经验中得知，大量的错误往往发生在输入和输出范围的边界上，而不是范围的内部。因此，针对边界情况设计测试用例，能够更有效地发现错误。

边界值分析法是一种补充等价类划分法的黑盒测试方法，它不是简单选择等价类中的任意元素，而是选择等价类边界的测试用例。实践证明，这些测试用例往往能取得很好的测试效果。边界值分析法不仅适用于输入范围的边界，还适用于输出范围的边界。

用边界值分析法设计测试用例时应遵守以下几条原则。

- 如果输入条件规定了取值范围，测试用例应包括该范围的边界内及刚刚超范围的边界外的值。
- 若规定了值的个数，应分别以最大、最小个数和稍小于最小和稍大于最大个数作为测试用例。
- 针对每个输出条件，也使用上述原则。
- 如果程序规格说明书中提到的输入或输出范围是有序的集合，应注意选取有序集的第一个和最后一个元素作为测试用例。
- 分析规格说明，找出其他的可能边界条件。

边界值分析法利用输入变量的最小值、略大于最小值、输入范围内的任意值、略小于最大值、最大值设计测试用例。如图 3-9 所示，对于 n 个变量，使除一个以外的所有变量都取正常值，使剩余的变量取上述 5 个值，对每个变量都重复进行。一个 n 变量的函数的边界值有 $4n+1$ 个测试用例。

健壮性测试是边界值分析的一种简单扩展，除了使用 5 个边界值分析取值，还要采用一个略小于最小值和一个略大于最大值的取值。健壮性测试更关注例外情况如何处理，如图 3-10 所示。1 个 n 变量函数的健壮性边界值有 $6n+1$ 个测试用例。

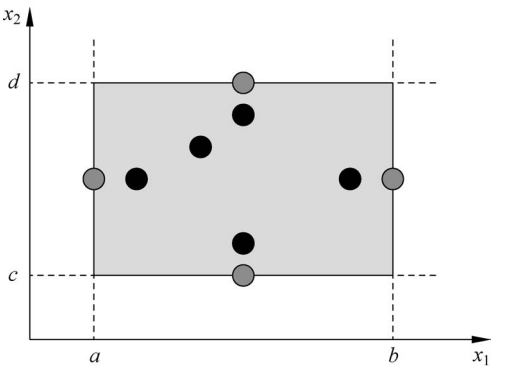

图 3-9　边界值分析测试用例示意图

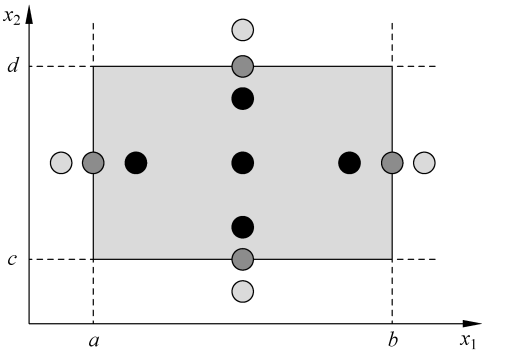

图 3-10　健壮性边界值测试用例示意图

如果对每一个变量，首先取边界值的 5 个取值作为集合，然后对这些集合进行笛卡儿积运算生成测试用例，则称为最坏情况测试。1 个 n 变量函数的最坏情况测试有 5^n 个测试用例。同理，可以进行健壮性最坏情况测试。1 个 n 变量函数的健壮最坏情况测试有 7^n 个测试用例。可以看到，测试的完全度与测试复杂度是成正比的，在实际测试中选择哪种边界值分析法要根据项目具体需要确定。

除了常规的边界条件，在软件内部可能还存在次边界条件。这些条件在用户界面上不明显，但仍然需要测试。次边界条件的发现和确定通常需要测试人员对软件的工作方式和领域知识有一定的了解。

仍以 NextDate 函数为例，用边界值分析法再补充如表 3-8 所示的测试用例。

表 3-8　边界值分析法设计的测试用例

序号	边　界　值	输入数据			预期输出		
		年	月	日	年	月	日
1	使年刚好等于最小值	1812	3	15	1812	3	16
2	使年刚好等于最大值	2012	3	15	2012	3	16
3	使年刚刚小于最小值	1811	3	15	输入无效		
4	使年刚刚大于最大值	2023	3	15	输入无效		
5	使月刚好等于最小值	2000	1	15	2000	1	16
6	使月刚好等于最大值	2000	12	15	2000	12	16
7	使月刚刚小于最小值	2000	0	15	输入无效		
8	使月刚刚大于最大值	2000	13	15	输入无效		
9	使闰年的 2 月的日刚好等于最小值	2000	2	1	2000	2	2
10	使闰年的 2 月的日刚好等于最大值	2000	2	29	2000	3	1
11	使闰年的 2 月的日刚刚小于最小值	2000	2	0	输入无效		
12	使闰年的 2 月的日刚刚大于最大值	2000	2	30	输入无效		
13	使非闰年的 2 月的日刚好等于最小值	2001	2	1	2001	2	2
14	使非闰年的 2 月的日刚好等于最大值	2001	2	28	2001	3	1

续表

序号	边界值	输入数据			预期输出		
		年	月	日	年	月	日
15	使非闰年的2月的日刚刚小于最小值	2001	2	0	输入无效		
16	使非闰年的2月的日刚刚大于最大值	2001	2	29	输入无效		
17	使1月、3月、5月、7月、8月、10月、12月的日刚好等于最小值	2001	10	1	2001	10	2
18	使1月、3月、5月、7月、8月、10月、12月的日刚好等于最大值	2001	10	31	2001	11	1
19	使1月、3月、5月、7月、8月、10月、12月的日刚刚小于最小值	2001	10	0	输入无效		
20	使1月、3月、5月、7月、8月、10月、12月的日刚刚大于最大值	2001	10	32	输入无效		
21	使4月、6月、9月、11月的日刚好等于最小值	2001	6	1	2001	6	2
22	使4月、6月、9月、11月的日刚好等于最大值	2001	6	30	2001	7	1
23	使4月、6月、9月、11月的日刚刚小于最小值	2001	6	0	输入无效		
24	使4月、6月、9月、11月的日刚刚大于最大值	2001	6	31	输入无效		

3. 因果图法

等价类划分法和边界值分析法都主要考虑的是输入条件,而没有考虑输入条件的各种组合以及各个输入条件之间的相互制约关系。然而,如果在测试时考虑到输入条件的所有组合方式,可能其本身非常大甚至是一个天文数字。因此,我们需要一种方法来有效描述和考虑不同输入条件的组合,并据此设计相应的测试用例。这就需要利用因果图法。

因果图法是一种黑盒测试方法,用于从自然语言的程序规格说明书中识别输入条件与输出、程序状态的因果关系。因果图可以帮助测试人员高效选择测试用例,并指出规格说明书中存在的问题。

在因果图中,用C表示原因,E表示结果,各节点表示状态,取值0表示某状态不出现,取值1表示某状态出现。因果图包括恒等、非、或、与等关系符号,如图3-11所示。

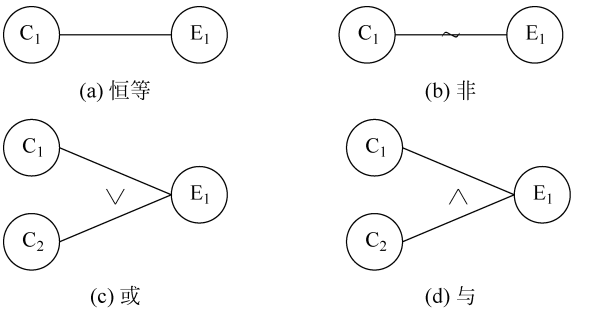

图3-11　因果图基本符号

- 恒等：若原因出现则结果出现，若原因不出现则结果不出现。
- 非(～)：若原因出现则结果不出现，若原因不出现则结果反而出现。
- 或(∨)：若几个原因中有一个出现则结果出现，若几个原因都不出现则结果不出现。
- 与(∧)：若几个原因都出现结果才出现，若其中一个原因不出现则结果不出现。

为了表示原因与原因之间、结果与结果之间可能存在的约束关系，在因果图中可以附加一些表示约束条件的符号，如图 3-12 所示。

- E 约束(互斥)：表示 a 和 b 两个原因不会同时成立，最多有一个可以成立。
- I 约束(包含)：表示 a 和 b 两个原因至少有一个必须成立。
- O 约束(唯一)：表示 a 和 b 两个条件必须有且仅有一个成立。
- R 约束(要求)：表示 a 出现时，b 也必须出现。
- M 约束(强制)：表示 a 是 1 时，b 必须为 0。

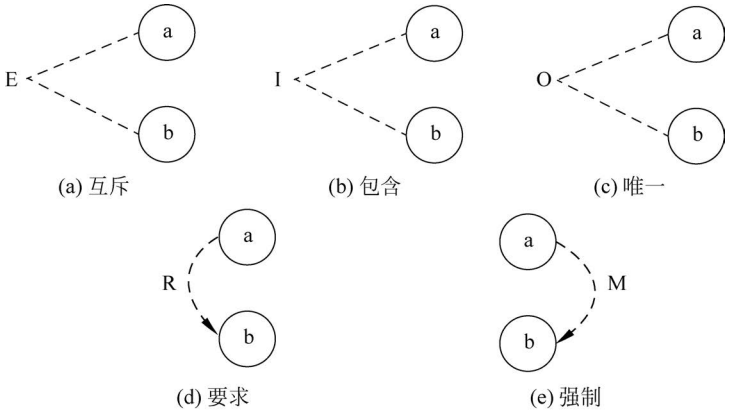

图 3-12　因果图约束符号

因果图法设计测试用例的步骤如下。

- 分析程序规格说明书的描述中，确定哪些是原因，哪些是结果。原因通常是输入条件或输入条件的等价类，结果通常是输出条件。
- 分析程序规格说明书中描述的语义内容，并将其表示成连接各个原因与各个结果的因果图。
- 考虑语法或环境的限制，标明因果图上的约束条件，以确保图的合理性。
- 把因果图转换为决策表。
- 根据决策表设计测试用例。

后面两个步骤中提到的决策表，将在下面进行详细介绍。如果项目在设计阶段已存在决策表，则可以直接使用而不必再画因果图。

下面以一个自动饮料售货机软件为例，展示因果图分析方法。该自动饮料售货机软件的规格说明如下。

有一个处理单价为 1 元 5 角的盒装饮料的自动售货机软件。若投入 1 元 5 角硬币，按下“可乐”、“雪碧”或“红茶”按钮，相应的饮料就送出来。若投入的是 2 元硬币，则在送出饮料的同时退还 5 角硬币。

首先从软件规格说明中分析原因、结果以及中间状态。分析结果如表 3-9 所示。

表 3-9　自动饮料售货机软件分析结果

原因	C1：投入 1 元 5 角硬币 C2：投入 2 元硬币 C3：按“可乐”按钮 C4：按“雪碧”按钮 C5：按“红茶”按钮
中间状态	11：已投币 12：已按钮
结果	E1：退还 5 角硬币 E2：送出“可乐” E3：送出“雪碧” E4：送出“红茶”

根据表 3-9 中的原因与结果，结合软件规格说明，连接成如图 3-13 所示因果图。

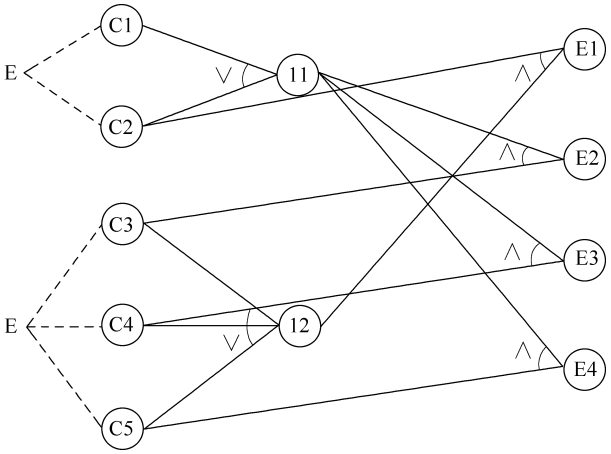

图 3-13　自动饮料售货机软件因果图

4. 决策表法

决策表法是用来分析和表达多逻辑条件下执行不同操作情况的工具，在一些数据处理问题中非常有用。决策表通常由 4 部分组成，如图 3-14 所示。

- 条件桩：列出问题的所有条件。
- 条件项：列出所列条件下的取值，在所有可能情况下的真假值。
- 动作桩：列出问题规定可能采取的动作。
- 动作项：列出在条件项的各种取值情况下应采取的动作。

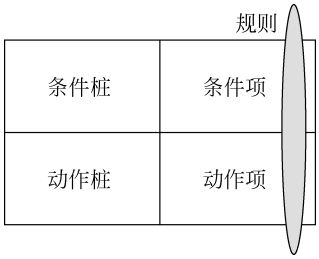

图 3-14　决策表组成

在决策表中，一条规则贯穿条件项和动作项的一列。如果两条或多条规则具有相同的动作，并且其条件项之间存在着相似的关系，可以进行规则合并。

决策表的建立应当根据软件规格说明书，分为以下几个步骤。

- 确定规则个数。
- 列出所有条件桩和动作桩。
- 填入条件项。
- 填入动作项，制定初始决策表。
- 简化，合并相似规则或者相同动作。

在简化并得到最终决策表后，只需选择适当的输入，使决策表每一列的输入条件得到满足，即可生成测试用例。

将前面得到的自动饮料售货机软件因果图转换为决策表，如表 3-10 所示。

表 3-10　自动饮料售货机软件决策表

		1	2	3	4	5	6	7	8	9	10	11
条件	C1：投入 1 元 5 角硬币	1	1	1	1	0	0	0	0	0	0	0
	C2：投入 2 元硬币	0	0	0	0	1	1	1	1	0	0	0
	C3：按“可乐”按钮	1	0	0	0	1	0	0	0	1	0	0
	C4：按“雪碧”按钮	0	1	0	0	0	1	0	0	0	1	0
	C5：按“红茶”按钮	0	0	1	0	0	0	1	0	0	0	1
中间状态	11：已投币	1	1	1	1	1	1	1	1	0	0	0
	12：已按钮	1	1	1	0	1	1	1	0	1	1	1
动作	E1：退还 5 角硬币	0	0	0	0	1	1	1	0	0	0	0
	E2：送出“可乐”按钮	1	0	0	0	1	0	0	0	0	0	0
	E3：送出“雪碧”按钮	0	1	0	0	0	1	0	0	0	0	0
	E4：送出“红茶”按钮	0	0	1	0	0	0	1	0	0	0	0

可以根据上述决策表设计测试用例，验证适当的输入组合是否能得到正确的输出。特别是在本案例中，利用决策表法能够清晰地验证自动饮料售货机软件的功能完备性。

5. 正交实验法

正交实验法是一种有效、合理地减少测试工时与费用的方法，适用于软件测试中测试用例的设计。其核心思想是从大量的可能性中挑选出适量的、有代表性的测试点，以合理安排测试的设计。

日本著名的统计学家田口玄一将正交实验选择的水平组合列成表格，称为正交表。例如，做一个三因素三水平的实验，按全面实验要求，须进行 $3^3=27$ 种组合的实验，且尚未考虑每一组合的重复数。若按 $L_9(3^3)$ 正交表安排实验，只需实验 9 次，按 $L_{18}(3^7)$ 正交表也只需进行 18 次实验，显然大大减少了工作量。因而正交实验设计在很多领域的研究中已经得到广泛应用。

正交表的形式为 $L_{行数}(水平数^{因素数})$。其中，行数表示正交表中的行的个数，即实验的次数，也是通过正交实验法设计的测试用例的个数。因素数是正交表中列的个数，即要测试的功能点。水平数是任何单个因素能够取得的值的最大个数。正交表中包含的值为从 0 到“水平数－1”或从 1 到“水平数”，即要测试功能点的输入条件。

正交表具有以下两个性质。

（1）每一列中，不同的数字出现的次数相等。例如，在两水平正交表中，任何一列都有数码“1”与“2”，且任何一列中它们出现的次数是相等的；如在三水平正交表中，任何一列都有“1”“2”“3”，且在任一列的出现数均相等。

(2) 任意两列中数字的排列方式齐全而且均衡。例如,在两水平正交表中,任何两列(同一横行内)有序对共有4种:(1,1)、(1,2)、(2,1)、(2,2)。每种对数出现次数相等。在三水平情况下,任何两列(同一横行内)有序对共有9种:(1,1)、(1,2)、(1,3)、(2,1)、(2,2)、(2,3)、(3,1)、(3,2)、(3,3),且每对出现数也均相等。

下面以用户注册功能为例,展示正交实验法设计测试用例的方法。该用户注册页面中有7个输入框,分别是用户名、密码、确认密码、真实姓名、地址、手机号、电子邮箱。假设每个输入框只有填与不填两种状态,则可以设计 $L_8(2^7)$ 的正交表,如表3-11所示。其中,因素 C_1~C_7 分别表示上述7个输入框。表中"1"表示该输入框填写,"0"表示不填。

根据表3-11可以得到8个测试用例,读者可以根据各因素代表的输入框含义自己生成测试用例。

表3-11 $L_8(2^7)$正交表

行号	因素						
	C_1	C_2	C_3	C_4	C_5	C_6	C_7
1	1	1	1	1	1	1	1
2	1	1	1	0	0	0	0
3	1	0	0	1	1	0	0
4	1	0	0	0	0	1	1
5	0	1	0	1	0	1	0
6	0	1	0	0	1	0	1
7	0	0	1	1	0	0	1
8	0	0	1	0	1	1	0

6. 场景法

现代软件通常采用事件触发来控制应用程序的流程,这些事件的触发形成了不同的场景,而它们的不同触发顺序和处理结果也构成了事件流。这种思想也可以应用到软件测试中,通过生动地描绘事件触发时的情形,有助于测试人员更好地执行测试用例,同时也增加了测试用例的可理解性和可执行性。

用例场景是通过描述覆盖从用例的起始点到终点的路径来确定的过程,这个流程包括所有的基本流和备选流,确保所有可能的情况都能在测试中被覆盖到。基本流是指经过用例的最简单路径,表示无任何差错,程序从开始执行到结束;备选流则是在某些特定条件下执行的分支流程,可以从基本流开始或另一个备选流开始,也可以终止用例。

应用场景法进行黑盒测试的步骤如下。

- 根据规格说明,描述出程序的基本流和各个备选流。
- 根据基本流和各个备选流生成不同的场景。
- 对每一个场景生成相应的测试用例。
- 对生成的所有测试用例进行复审,去掉多余的测试用例,对每一个测试用例确定测试数据。

以一个经典的ATM机为例,介绍使用场景法设计测试用例的过程。ATM机的取款流程的场景分析如图3-15所示,其中,灰色框构成的流程为基本流。

该程序用例场景如表3-12所示。

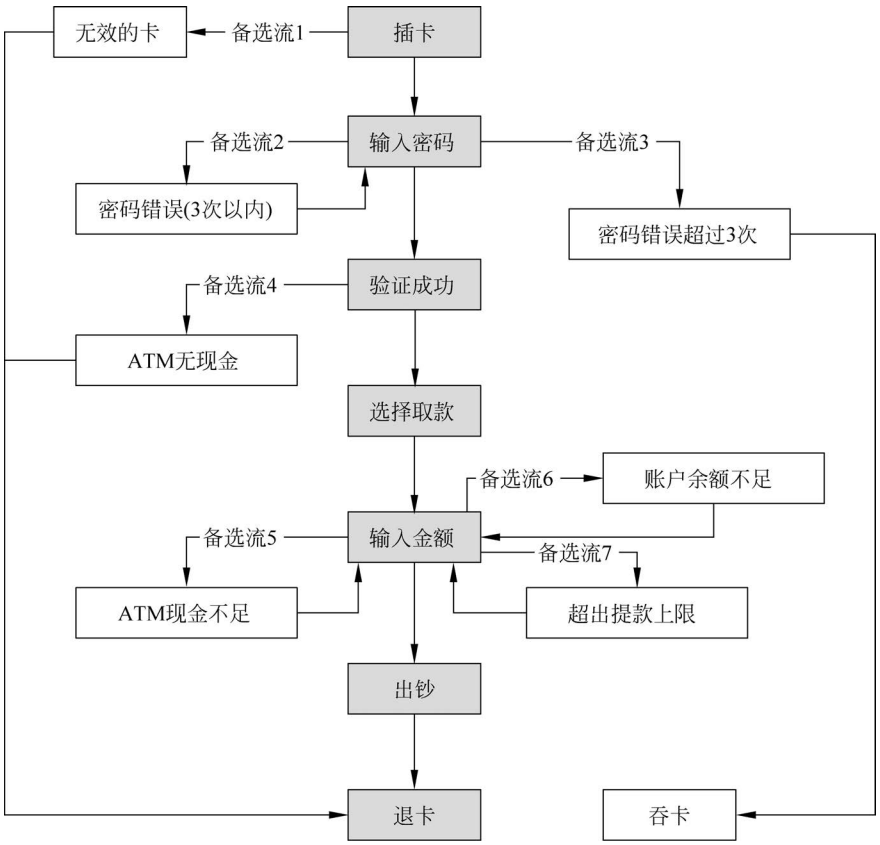

图 3-15　ATM 取款流程场景法分析图

表 3-12　用例场景

场景 1	成功提款	基本流	
场景 2	无效卡	基本流	备选流 1
场景 3	密码错误 3 次以内	基本流	备选流 2
场景 4	密码错误超过 3 次	基本流	备选流 3
场景 5	ATM 无现金	基本流	备选流 4
场景 6	ATM 现金不足	基本流	备选流 5
场景 7	账户余额不足	基本流	备选流 6
场景 8	超出提款上限	基本流	备选流 7

接下来设计用例覆盖每个用例场景，如表 3-13 所示。

表 3-13　场景法测试用例

用例号	场景	账　　户	密码	操　　作	预 期 结 果
1	场景 1	621226XXXXXXXXX3481	123456	插卡，取 500 元	成功取款 500 元
2	场景 2	—	—	插入一张无效卡	系统退卡，显示该卡无效
3	场景 3	621226XXXXXXXXX3481	123456	插卡，输入密码 111111	系统提示密码错误，请求重新输入
4	场景 4	621226XXXXXXXXX3481	123456	插卡，输入密码 111111 超过 3 次	系统提示密码输入错误超过 3 次，卡被吞掉

续表

用例号	场景	账　　户	密码	操　　作	预 期 结 果
5	场景5	621226XXXXXXXXX3481	123456	插卡,选择取款	系统提示ATM无现金,退卡
6	场景6	621226XXXXXXXXX3481	123456	插卡,取款2000元	系统提示现金不足,返回输入金额界面
7	场景7	621226XXXXXXXXX3481	123456	插卡,取款3000元	系统提示账户余额不足,返回输入金额界面
8	场景8	621226XXXXXXXXX3481	123456	插卡,取款3500元	系统提示超出取款上限(3000),返回输入金额界面

3.4.2 白盒测试和黑盒测试比较

白盒测试和黑盒测试是两种常见的软件测试方法,用于评估软件系统的质量和功能。它们在测试过程中的关注点和方法有所不同,表3-14给出了两种方法的基本对比。

表3-14　白盒测试和黑盒测试的对比

测 试 方 法	白 盒 测 试	黑 盒 测 试
涉及内容	考查程序逻辑结构和源代码	不涉及程序内部结构
生成测试用例	使用程序结构信息	使用软件规格说明书
适用范围	主要适用于单元测试和集成测试	从单元测试到系统验收测试
测试覆盖	尝试覆盖所有可能的逻辑路径	某些代码段得不到测试

白盒测试和黑盒测试各有其优缺点,无法完全取代对方。在实践中,通常会将它们结合使用,以达到更全面的测试覆盖和更高的测试效率。白盒测试能够深入分析代码,发现隐藏的错误,提供更高的测试覆盖率,验证软件的正确性。但是白盒测试通常比黑盒测试成本更高,需要在代码可用之前进行计划;需要更多的工作量,包括确定合适的测试数据和验证方法。相比之下,黑盒测试不需要了解内部结构,能更专注于功能和用户需求。黑盒测试可以通过设计测试用例来尽可能地覆盖代码段。但是,黑盒测试不能保证覆盖所有代码段,可能无法发现一些潜在的程序错误。

此外,灰盒测试是一种结合了白盒测试和黑盒测试要素的测试方法,同时考虑了用户端、特定的系统知识和操作环境。这种方法在评价应用软件的设计时考虑了系统组件的协同性作用。集成测试可以看作一种灰盒测试,重点验证系统的整体功能和性能,确保各组件之间的协同作用正常。

小　　结

本章主要介绍了静态测试和动态测试的定义、内容、分类及方法。

白盒测试关注软件产品的内部细节和逻辑结构,可以分为静态测试和动态测试。静态测试不需要执行程序,而是侧重于检查软件的描述与实现是否一致,以及是否存在潜在的冲突或歧义。动态测试需要执行程序,在模拟或真实环境下分析程序的行为,主要用于验证程序在运行时是否正确。本章介绍了白盒测试的常见方法,着重介绍了程序插桩技术、逻辑覆

盖法以及基本路径法，并且每种方法都以相关实例进行详细说明。

黑盒测试主要关注被测软件功能的实现，而不是其内部逻辑。本章重点介绍了几种常用的黑盒测试方法，以及相应的案例说明。

习　题　3

扫一扫

习题

扫一扫

自测题