CHAPTER 3

第 3 章

软件测试方法

通过对软件缺陷的产生、分类、构成所做的讨论，更容易理解软件测试的目的，软件测试是为了更快、更早地将软件产品或软件系统中所存在的各种问题找出来，并促使系统分析人员、设计人员和程序员尽快地解决这些问题，最终及时地向客户提供高质量的软件产品。要做到这一点，确保在有限的时间内找出系统中绝大部分的软件缺陷，必须依赖有效的软件测试方法。在第 2 章对白盒测试和黑盒测试、静态测试和动态测试、主动测试和被动测试等了解之后，有必要了解具体的软件测试方法。这些具体的方法有助于实现测试(用例)的设计，能更有效地完成测试的执行，达到设定的测试目标。

在讨论具体的软件测试方法时，可以从不同层次、不同维度或角度去看。从高层次看，测试方法体现了方法论或测试流派，是作为一个方法类别出现的，如：

(1) 基于直觉和经验的方法；

(2) 基于输入域的方法；

(3) 基于代码的方法；

(4) 基于故障模式的测试方法；

(5) 基于模型的测试方法；

(6) 基于使用的方法；

(7) 基于需求验证或标准验证的测试方法；

(8) 基于逻辑分析的测试方法；

(9) 基于上下文驱动的测试方法；

(10) 基于风险的测试方法。

还有一些测试方法，依赖于软件过程模式、软件开发方法或应用领域，例如：

(1) 面向对象的软件采用面向对象的测试方法；

(2) 面向服务架构(Service-Oriented Architecture，SOA)会采用 SOA 的测试方法；

(3) Web 应用测试方法；

(4) 移动 App 应用测试方法；

(5) 针对嵌入式应用的嵌入式测试方法(技术)；

(6) 敏捷开发中会采用更贴近敏捷的测试方法(技术实践)。

不过,这些特定应用领域的测试方法,不能算真正的测试方法,而是测试技术,是前面(1)～(10)那些测试方法及其综合的运用。而人们经常说的等价类划分方法、边界值分析方法、正交试验方法等属于具体的、更低层次的测试方法,有特定的应用场景。甚至有些标准或权威材料把它们归为测试技术,最终被纳入上述(1)～(10)那些测试方法类别之中。

3.1 基于直觉和经验的方法

基于直觉和经验的测试方法,不是严格意义上的科学测试方法,带有一定的随机性,测试结果不够可靠,甚至可以看作是没有办法的办法。但是,软件测试具有社会性,呈现一定的不确定性,这时,人的直觉和经验又往往能发挥很好的作用。例如,产品易用性测试、用户体验的检验,虽然有一定的规律可循、要遵守某些原则,但很难靠一种明确的科学方法来完成测试,而是比较依赖于测试人的直觉和经验,如下面要介绍的 ad-hoc 测试方法,具有一定的探索性。业界比较流行的"探索式测试(Exploratory Testing)"被认为是一种测试方式,而不是一种方法,因为在探索式测试中可以采用各种各样的测试方法。在 SWEBOK 3.0 中,错误猜测法被归为基于故障模式的测试方法,这也是可以的,但更适合归为基于直觉和经验的测试方法。

3.1.1 Ad-hoc 测试方法和 ALAC 测试

自由测试(Ad-hoc Testing)强调测试人员根据自己的经验,不受测试用例的束缚,放开思路、灵活地进行各种测试。这里所说的经验,包含下列几个方面的经验。

(1) 软件开发(如系统设计、编程等)的经验和知识;

(2) 与失效和缺陷打交道的经验(发现和处理缺陷的经验和知识);

(3) 对被测软件系统的经验和知识;

(4) 软件系统涉及的业务知识;

(5) 其他方面的经验,包括心理学、社会学等。

自由测试可以作为严格意义上的测试方法的一种补充手段,完善软件测试用例,无拘无束、思维活跃,能发现一些隐藏比较深的缺陷,有时可以达到出人意料的效果。有时测试人员,在熟悉产品的新功能特性时,也可以进行有测试倾向的操作,发现问题,获得一箭双雕的效果。

ALAC,是 Act-like-a-customer(像客户那样做)的简写,是一种基于客户使用产品的知识开发出来的测试方法,它的出发点是著名的 Pareto 80/20 规律,用到软件测试中,可以概括为以下两条。

(1) 即一个软件产品或系统中全部功能的 20%是常用的功能,用户的 80%时间都在使用这 20%功能;而软件产品或系统中剩下的 80%不是常用的功能,用户使用得比较少,只有 20%时间在使用剩下的 80%功能。

(2) 测试发现的所有错误的 80%很可能集中在 20%的程序模块中,另外 20%的错误很可能集中在 80%的程序模块中。

ALAC 测试就是基于这样一个思想,如图 3-1 所示,考虑复杂的软件产品肯定存在许多错误,而测试的时间是有限的,然后像客户那样做,对常用的功能进行测试。ALAC 测试方法适合一些的特别场合,如产品只是一个演示版、开发预算很低、没有足够时间进行测试、整个开发计划日程表很紧,其最大的益处是降低测试成本、缩短测试时间,缺陷查找和改正将针对那些

客户最容易遇到的错误。从这个角度看,ALAC 测试和基于风险的测试方法接近。

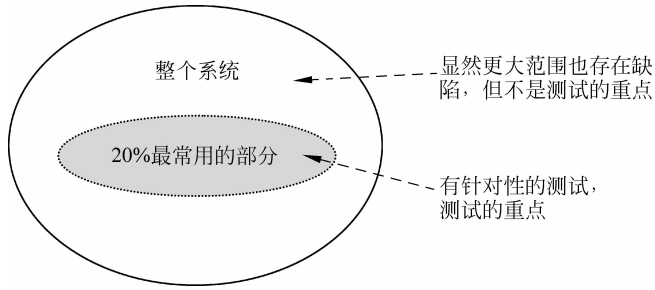

图 3-1 ALAC 测试方法的原理示意图

3.1.2 错误推测法

有经验的测试人员往往可以根据自己的工作经验和直觉推测出程序可能存在的错误,从而有针对性地进行测试,这就是错误推测法。错误推测法是测试者根据经验、知识和直觉来发现软件错误,来推测程序中可能存在的各种错误,从而有针对性地进行测试。“只可意会,不能言传”,就是表明这样一个道理。

错误推测法基于这样一个思想——某处发现了缺陷,则可能会隐藏更多的缺陷。在实际操作中,列出所有可能出现的错误和容易发现错误的地方,然后依据测试者经验做出选择。

【示例一】 上个版本发现的缺陷也许对我们当前版本测试有所启发,进行类似的探索新测试,说不准,可以发现一些严重的缺陷。

【示例二】 等价类划分法和边界值分析法通过选择有代表性的测试数据来暴露程序错误,但不同类型、不同特点的程序还存在其他一些特殊的、容易出错的情况,例如,一些特殊字符(如 * % / \ # @ $ ^ . | 等)被输入到系统后产生例外情况。有时,将多个边界值组合起来进行测试,可能使程序出错。

【示例三】 客户端在正常连接时一般没问题,可以试试断掉连接,让它重新连接看看是否出现系统崩溃的问题,而且可以不断调整失去连接的时间,或者尝试不同的连接次数等,以发现一些例外。

【示例四】 就程序中容易出现的问题,例如空指针、内存没有及时释放、session 失效或 JavaScript 字符转义等,设想各种情况,能否引起上述问题的发生,从而设计出一些特别的测试用例来发现缺陷。

错误推测法能充分发挥人的直觉和经验,在一个测试小组中集思广益,方便实用,特别是在软件测试基础较差的情况下,很好地组织测试小组进行错误猜测,是一种有效的测试方法。但错误推测法不是一个系统的测试方法,所以只能用作辅助手段,即先用上述方法设计测试用例,在没有别的办法可用情况下,再采用错误推测法,补充一些例子来进行一些额外的测试。优点是测试者能够快速且容易地切入,并能够体会到程序的易用与否;缺点是难以知道测试的覆盖率,可能丢失大量未知的区域,并且这种测试行为带有主观性且难以复制。

3.2 基于输入域的方法

软件系统从本质上看，就是对输入数据的处理，转化成所期望的结果，所以通过数据的验证来验证系统的功能性是很自然的思路。从被测试的对象看，无论是整个系统，还是一个模块、一个函数，都有数据输入或参数调用，通过对不同数据的输入，检查其输出的数据以判断测试是否通过的方法，都归为基于输入域的方法。等价类划分法、边界值分析法都是典型的基于输入域的方法，而决策表、因果图、两两组合正交、正交实验法等也可归为输入域验证方法，只是多变量组合数据的测试，不是单一变量数据的测试。而在决策表、因果图等方法中采用了表格、符号等方式来定义问题和分析问题，可以看作是基于模型的方法，但在本教材中，把决策表、两两组合正交、正交实验法等归为组合方法，而只把因果图、功能图等方法归为基于模型的方法。

3.2.1 等价类划分法

数据测试是功能测试的主要内容，或者说功能测试最主要的手段之一就是借助数据的输入输出来判断功能能否正常运行。在进行数据输入测试时，如果需要证明数据输入不会引起功能上的错误、或者其输出结果在各种输入条件下都是正确的，就需要将可输入数据域内的值完全尝试一遍（即穷举法），但实际是不现实的。

假如一个程序 P 有输入量 I1 和 I2 及输出量 O，在字长为 32 位的计算机上运行。如果 I1 和 I2 均取整数，则测试数据的最大可能数目为：$2^{32}\times 2^{32}=2^{64}$。

如果测试程序 P，采用穷举法力图无遗漏地发现程序中的所有错误，且假定运行一组（I1，I2）数据需 1ms，一天工作 24h，一年工作 365 天，则 2^{64} 组测试数据需 5 亿年。从而穷举的黑盒测试通常是不能实现的。因此只能选取少量有代表性的输入数据，以期用较小的测试代价暴露出较多的软件缺陷。

为了解决这个问题，人们就设想是否可以用一组有限的数据去代表近似无限的数据，这就是“等价类划分”方法的基本思想。等价类划分法就是解决如何选择适当的数据子集来代表整个数据集的问题，通过降低测试的数目去实现“合理的”覆盖，覆盖了更多的可能数据，以发现更多的软件缺陷。等价类划分法基于对输入或输出情况的评估，然后划分成两个或更多子集来进行测试的一种方法，即它将所有可能的输入数据（有效的或无效的）划分成若干个等价类，从每个等价类中选择一定的代表值进行测试。等价类划分法是黑盒测试用例设计中一种重要的、常用的设计方法，将漫无边际的随机测试变为具有针对性的测试，极大地提高了测试效率。

等价类是指某个输入域的一个特定的子集合，在该子集合中各个输入数据对于揭露程序中的错误都是等效的。也就是说，如果用这个等价类中的代表值作为测试用例未发现程序错误，那么该类中其他的数据（测试用例）也不会发现程序的错误。这样，对于表征该类的某个特定的数据输入将能代表整个子集合的输入，即测试某等价类的代表值就等效于对于这一类其他值的测试。举个例子，设计这样的测试用例，来实现一个对所有的实数进行开方运算的程序的测试，这时候需要将所有的实数（输入域）进行划分，可以分成正实数、负实数和 0，使用 +1.4444 来代表正实数，用 −2.345 来代表负实数，输入的等价类就可以使用 +1.4444，−2.345 和 0 来表示。

在确定输入数据的等价类时，常要分析输出数据的等价类，以便根据输出数据的等价类导

出对应的输入数据等价类。这样就在等价类划分过程中,一般经过两个过程——分类和抽象。

(1) 分类,即将输入域按照具有相同特性或者类似功能进行分类。

(2) 抽象,即在各个子类中去抽象出相同特性并用实例来表征这个特性。

等价类划分法的优点是基于相对较少的测试用例,就能够进行完整覆盖,很大程度上减少了重复性;缺点是缺乏特殊用例的考虑,同时需要深入的系统知识,才能选择有效的数据。

1. 有效等价类和无效等价类

在进行等价类划分的过程中,不但要考虑有效等价类划分,同时需要考虑无效的等价类划分。有效等价类和无效等价类定义如下。

(1) 有效等价类是指输入完全满足程序输入的规格说明、有意义的输入数据所构成的集合,利用有效等价类可以检验程序是否满足规格说明所规定的功能和性能。

(2) 无效等价类和有效等价类相反,即不满足程序输入要求或者无效的输入数据构成的集合。使用无效等价类,可以测试程序/系统的容错性——对异常输入情况的处理。

在程序设计中,不但要保证所有有效的数据输入能产生正确的输出,同时需要保障在输入错误或者空输入的时候能有异常保护,这样的测试才能保证软件的可靠性。

在使用等价类划分法时,设计一个测试用例,使其尽可能多地覆盖尚未被覆盖的有效等价类,重复这个过程,直至所有的有效等价类都被覆盖,即分割有效等价类直到最小。对无效等价类,进行同样的过程,设计若干个测试用例,覆盖无效等价类中的各个类。

2. 不同情形的处理

(1) 在输入条件规定了取值范围或者个数的前提下,则可以确定一个有效等价类和两个无效等价类。例如,程序输入条件为满足小于 100 大于 10 的整数 x,则有效等价类为 $10<x<100$,两个无效等价类为 $x<10$ 和 $x>100$。

(2) 在输入条件规定了输入值的集合或者规定了“必须如何”的条件下,可以确定一个有效等价类和一个无效等价类。例如,程序输入条件为 $x=10$,则有效等价类为 $x=10$,无效等价类为 $x\neq 10$。

(3) 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。例如,程序输入条件为 BOOL $x=$true,则有效等价类为 $x=$true,无效等价类为 $x=$false。

(4) 在规定了一组输入数据(包括 n 个输入值),并且程序要对每一个输入值进行分别进行处理的情况下,可确定 n 个有效等价类和一个无效等价类。例如,程序输入条件为 x 取值于一个固定的枚举类型{1,3,7,10,15},则有效等价类为 $x\in\{1,3,7,10,15\}$,而程序中对这 5 个数值分别进行了处理,对于任何其他的数值使用默认处理方式,此时无效等价类为 $x\notin\{1,3,7,10,15\}$的集合。

(5) 在规定了输入数据必须遵守的规则的情况下,可确定一个有效等价类和若干个无效等价类。例如,输入是页面上用户输入有效 E-mail 地址的规则,必须满足几个条件,含有@,@后面格式为 x. x,E-mail 地址不带有特殊符号”,#,‘,&,有效等价类就是满足所有条件的输入的集合,无效等价类就是不满足其中任何一个条件或者所有条件的输入的集合。

(6) 在确定已知的等价类中各元素在程序处理中的方式不同的情况下,则应再将该等价类进一步划分为更小的等价类。

3. 示例

有一报表处理系统,要求用户输入处理报表的日期。假设日期限制在 2000 年 1 月至

2020年12月，即系统只能对该段时期内的报表进行处理。如果用户输入的日期不在这个范围内，则显示错误信息。并且此系统规定日期由年月的6位数字组成，前4位代表年，后两位代表月。则检查日期时，可用表3-1进行等价类划分和编号。

表3-1 等价类划分的一个实例

输入	有效等价类	无效等价类
报表日期	① 6位数字字符	② 有非数字字符 ③ 少于6个数字字符 ④ 多于6个数字字符
年份范围	⑤ 在2000～2020之间	⑥ 小于2000 ⑦ 大于2020
月份范围	⑧ 在1～12之间	⑨ 等于0 ⑩ 大于12

在进行功能测试时，只要对有效等价类和无效等价类测试进行测试，覆盖①、⑤、⑧三个有效等价类测试，只要用一个值201006即可；对无效等价类的测试则要分别输入7个非法数据，如200a0b、20102、1012012、198802、203011、200000、202013。合起来只要完成8个数据的输入就可以了。如果不用等价类划分法，其测试的输入值则高达几百个，可见等价类划分法提高了测试效率。

3.2.2 边界值分析法

实践证明，程序往往在输入输出的边界值情况下发生错误。边界包括输入等价类和输出等价类的大小边界，检查边界情况的测试用例是比较高效的，可以查出更多的错误。如上面介绍的处理报表日期的例子，等价类划分法就忽略了几个边界条件，如200001(边界有效最小值)、202012(边界有效最大值)以及边界无效值199901、199912、202101、202112等，而程序往往在这些地方容易出错。这就要求对输入的条件进行分析并找出其中的边界值条件，通过对这些边界值的测试来发现更多的错误。

边界值分析法就是在某个输入输出变量范围的边界上，验证系统功能是否正常运行的测试方法。因为错误最容易发生在边界值附近，所以边界值分析法对于多变量函数的测试很有效，尤其对于像C/C++数据类型要求不是很严格的语言更能发挥作用。缺点是对布尔值或逻辑变量无效，也不能很好地测试不同的输入组合。边界值分析法常被看作是等价类划分法的一种补充，两者结合起来使用更有效。

边界值分析法要取决于变量的范围和范围的类型，确认所有输入的边界条件或临界值，然后选择这些边界条件、临界值及其附近的值来进行相关功能的测试。边界值分析法处理技巧如下。

(1) 如果输入条件规定了值的范围，则取刚刚达到这个范围的边界值(如上述200001、202012)，以及刚刚超过这个范围边界的值(如上述199912、202101)；

(2) 如果输入条件规定了值的个数，则用最大个数、最小个数、比最大个数多1个、比最小个数少1个的数等作为测试数据；

(3) 根据规格说明的每一个输出条件，分别使用以上两个规则；

(4) 如果程序的规格说明给出的输入域或输出域是有序集合(如有序表、顺序文件等)，则应选取集合的第一个和最后一个元素作为测试数据。

在边界值分析法中,最重要的工作是确定边界值域。一般情况下,先确定输入和输出的边界,然后根据边界条件进行等价类的划分。这里给出一个排序程序的边界值分析的例子,其边界条件如下。

(1) 排序序列为空;

(2) 排序序列仅有一个数据;

(3) 排序序列为最长序列;

(4) 排序序列已经按要求排好序;

(5) 排序列的顺序与要求的顺序恰好相反;

(6) 排序序列中的所有数据全部相等。

上面提到的例子是常用的数组边界检查时遇到的。通常情况下,软件测试所包含的边界检验有几种类型:数字、字符、位置、质量、大小、速度、方位、尺寸、空间等,而相应的边界值假定为最大/最小、首位/末尾、上/下、最快/最慢、最高/最低、最短/最长、空/满等情况,这需要对用户的输入以及被测应用软件本身的特性进行详细的分析,才能够识别出特定的边界值条件。另外,还需要选取正好等于、刚刚大于和刚刚小于边界值的数据作为测试数据,如表 3-2 所示。

表 3-2 边界值附近的数据确定的几种方法

项	边界值	测试用例的设计思路
字符	起始－1 个字符/结束＋1 个字符	假设一个文本输入区域要求允许输入 1～255 个字符,输入 1 个和 255 个字符作为有效等价类;输入 0 个和 256 个字符作为无效等价类,这几个数值都属于边界条件值
数值	开始位－1/结束位＋1	例如,数据的输入域为 1～999,其最小值为 1,而最大值为 999,则 0、1000 刚好在边界值附近
方向	刚刚超过/刚刚低于	
空间	小于空余空间一点/大于满空间一点	例如,测试数据存储时,使用比最小剩余空间大一点(几 KB)的文件作为最大值检验的边界条件

1. 数值的边界值检验

计算机是基于二进制进行工作的,因此,软件的任何数值运算都有一定的范围限制,如表 3-3 所示。

表 3-3 各类二进制数值的边界

项	范 围 或 值
位(b)	0 或 1
字节(B)	0～255
字(Word)	0～65 535(单字)或 0～4 294 967 295(双字)
千(K)	1024
兆(M)	1 048 576
吉(G)	1 073 741 824
太(T)	1 099 511 627 776

这样,在数值的边界值条件检验中,就可以参考这个表进行。如对字节进行检验,边界值条件可以设置成 254、255 和 256。

2. 字符的边界值检验

在计算机软件中，字符也是很重要的表示元素，其中 ASCII 和 Unicode 是常见的编码方式，表 3-4 列出了一些简单的 ASCII 码对应表。

表 3-4 字符和 ASCII 码值的对应关系

字符	ASCII 码值	字符	ASCII 码值
空(NULL)	0	A	65
空格(SPACE)	32	B	66
斜杠(/)	47	Y	89
0	48	Z	90
9	57	左中括号[	91
冒号(:)	58	反斜杠(/)	92
分号(;)	59	右中括号]	93
<	60	单引号(‘)	96
=	61	a	97
>	62	b	98
?	63	y	121
@	64	z	122

在文本输入或者文本转换的测试过程中，需要非常清晰地了解 ASCII 码的一些基本对应关系，例如，小写字母 a 和大写字母 A 在表中的对应是不同的，而 0～9 的边界字符则为“/”、“:”，这些也必须被考虑到数据区域划分的过程中，从而根据这些定义等价有效类来设计测试用例。

3. 其他边界值检验

如默认值、空值、空格、未输入值、零、无效数据、不正确数据和干扰(垃圾)数据等。

3.3 基于组合及其优化的方法

等价分类法和边界分析法用于单因素、单变量的数据分析，但多因素或多变量的输入情况就不一样，这些因素之间相互地组合。虽然各种输入条件可能出错的情况已经被考虑到了，但多个输入情况组合起来可能出错的情况却被忽视了。检验各种输入条件的组合并非一件很容易的事情，因为即使将所有的输入条件划分成等价类，它们之间的组合情况也相当多，因此，需要考虑采用一种适合于多种条件的组合，相应地产生多个动作(结果)的方法来进行测试用例的设计，这就需要组合分析。

组合分析是一种基于每对参数组合的测试技术，主要考虑参数之间的影响是主要的错误来源和大多数的错误起源于简单的参数组合。优点是低成本实现、低成本维护、易于自动化、易于用较少的测试案例发现更多的错误和用户可以自定义限制；缺点是经常需要专家领域知识、不能测试所有可能的组合和不能测试复杂的交互。

3.3.1 判定表方法

对于多因素输入和输出，如果关系简单，根据某一个输入组合就能直接判断输出结果，而且每个输入条件或输出结果都可以用“成立”或“不成立”来表示，即输入条件和输出条件只有

1 和 0 两个取值,这时就采用判定表方法来设计组合(测试用例)。判定表方法是借助表格方式完成对输入条件的组合设计,以达到完全组合覆盖的测试效果。一个判定表由“条件和活动(条件作为输入、活动作为输出)”两部分组成,也就是列出了一个测试活动执行所需的条件组合,所有可能的条件组合定义了一系列的选择,而测试活动需要考虑每一个选择。例如,打印机是否能打印出来正确的内容,有多个因素影响,包括驱动程序、纸张、墨粉等。判定表方法就是对多个条件的组合进行分析,从而设计测试用例来覆盖各种组合。判定表是从输入条件的完全组合来满足测试的覆盖率要求,具有很严格的逻辑性,所以基于判定表的测试用例设计方法是最严格的组合设计方法之一,其测试用例具有良好的完整性。

在了解如何制定判定表之前,先要了解 5 个概念——条件桩、动作桩、条件项、动作项和规则。

(1) 条件桩:列出问题的所有条件,如上述三个条件——驱动程序、纸张、墨粉等。

(2) 动作桩:列出可能针对问题所采取的操作,如打印正确内容、打印错误内容、不打印等。

(3) 条件项:针对所列条件的具体赋值,即每个条件可以取真值和假值。

(4) 动作项:列出在条件项(各种取值)组合情况下应该采取的动作。

(5) 规则:任何一个条件组合的特定取值及其相应要执行的操作。在判定表中贯穿条件项和动作项的一列就是一条规则。

判定表制定一般经过下面 4 个步骤。

(1) 列出所有的条件桩和动作桩;

(2) 填入条件项;

(3) 填入动作项,制定初始判定表;

(4) 简化、合并相似规则或者相同动作。

仍以上述“打印机打印文件”为例子来说明如何制定判定表。首先列出所有的条件桩和动作桩,为了简化问题,不考虑中途断电、卡纸等因素的影响,那么条件桩为:

(1) 驱动程序是否正确?

(2) 是否有纸张?

(3) 是否有墨粉?

而动作桩有两种:打印内容和不同的错误提示。而且假定:优先警告缺纸,然后警告没有墨粉,最后警告驱动程序不对。然后输入条件项,即上述每个条件的值分别取“是(Y)”和“否(N)”,可以简化表示为 1 和 0。根据条件项的组合,容易确定其活动,如表 3-5 所示。如果结果一样,某些因素取 1 或 0 没有影响,即以“—”表示,可以合并这两项,最终优化判定表如表 3-6 所示。根据表 3-6,就可以设计测试用例,每一列代表一条测试用例。

表 3-5 初始化的判定表

序号		1	2	3	4	5	6	7	8
条件	驱动程序是否正确	1	0	1	1	0	0	1	0
	是否有纸张	1	1	0	1	0	1	0	0
	是否有墨粉	1	1	1	0	1	0	0	0
动作	打印内容	1	0	0	0	0	0	0	0
	提示驱动程序不对	0	1	0	0	0	0	0	0
	提示没有纸张	0	0	1	0	1	0	1	1
	提示没有墨粉	0	0	0	1	0	1	0	0

表 3-6 优化的判定表

序 号		1	2	4/6	3/7/8
条件	驱动程序是否正确	1	0	—	—
	是否有纸张	1	1	1	0
	是否有墨粉	1	1	0	—
动作	打印内容	1	0	0	0
	提示驱动程序不对	0	1	0	0
	提示没有纸张	0	0	0	1
	提示没有墨粉	0	0	1	0

3.3.2 因果图法

因果图法(Cause-effect Diagram)借助图形,着重分析输入条件的各种组合,每种组合条件就是“因”,它必然有一个输出的结果,这就是“果”。因果图是一种形式化的图形语言,由自然语言写成的规范转换而成,这种形式语言实际上是一种使用简化记号表示数字逻辑图,不仅能发现输入、输出中的错误,还能指出程序规范中的不完全性和二义性。因果图法就是一种利用图解法分析输入的各种组合情况,有时还要依赖所生成的判定表。

由因果图法怎样生成测试用例呢? 如图 3-2 所示,经过以下 4 个步骤。

(1) 分析软件规格说明书中的输入输出条件并分析出等价类,将每个输入输出赋予一个标志符; 分析规格说明中的语义,通过这些语义来找出相对应的输入与输入之间,输入与输出之间关系。

(2) 将对应的输入输出之间,输入与输出之间的关系关联起来,并将其中不可能的组合情况标注成约束或者限制条件,形成因果图。

(3) 由因果图转化成判定表。

(4) 将判定表的每一列拿出来作为依据,设计测试用例。

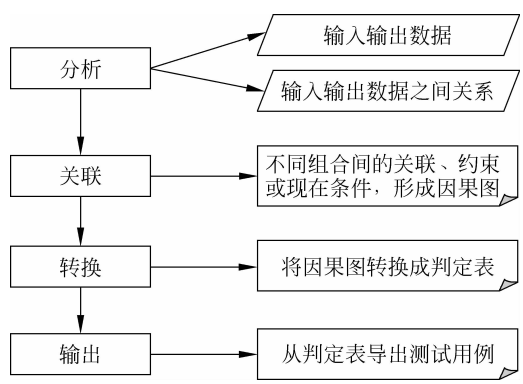

图 3-2 因果图法示例

例如,某个软件规格说明中包含以下的要求: 第一列字符必须是 A 或 B,第二个字符必须是一个数字,在此情况下进行文件的修改; 但如果第一列字符不正确,则输出信息 L; 如果第二列字符不是数字,则给出信息 M。采用因果图方法进行分析,可根据表 3-7 获得图 3-3 的各种组合,其中 ∧ 表示“或”、∨ 表示“与”、∽表示“非”的关系。

表 3-7 因果关系表

编号	原因(Cause)	编号	结果(Effect)
C1	第一列字符是 A	E1	修改文件
C2	第一列字符是 B	E2	给出信息 L
C3	第二列字符是一个数字	E3	给出信息 M
11	中间原因		

根据图 3-3 可以制定一张判定表，三个因素共有 8 种组合，考虑到 C1(首字符是 A)成立时，C2(首字符是 B)就不能成立，就变成 6 种组合，如表 3-8 所示。可以根据判定表来设计测试用例，每一列就代表一个测试用例，共设计 6 个测试用例。但实际上，还可以进一步优化，由于第二个字母不是数字时，第一个字母不管是 A 或是 B，或 A、B 都不是，结果都一样，即 E3 成立。所以表 3-8 可进一步优化为 4 种组合，即 4 个测试用例，如表 3-9 所示。

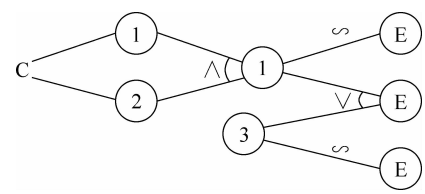

图 3-3 因果图表示

表 3-8 上述例子的判定表

序号		1	2	3	4	5	6
原因	C1	1	0	0	1	0	0
	C2	0	1	0	0	1	0
	C3	1	1	1	0	0	0
结果	E1	1	1	0	0	0	0
	E2	0	0	1	0	0	0
	E3	0	0	0	1	1	1

表 3-9 优化后的判定表

序号		1	2	3	4/5/6
原因	C1	1	0	0	—
	C2	0	1	0	—
	C3	1	1	1	0
结果	E1	1	1	0	0
	E2	0	0	1	0
	E3	0	0	0	1
用例		首字符为 A，第 2 个字符为数字	首字符为 B，第 2 个字符为数字	首字符为 x，第 2 个字符为数字	首字符为 A 或 B 或 x，第 2 个字符不是数字

3.3.3 Pair-wise 方法

在实际的软件项目中，作为输入条件的原因非常多，每个条件不只有“是”和“否”两个值，而是有多个取值。这时决策表已无能为力，需要借助其他方法实现。如果输入条件多，而每个条件又有多个取值，那么这个组合数就是一个非常大的数字，如果要执行覆盖全部组合测试，其测试工作量也非常大，但测试的时间和人力资源是有限的，如果不优化组合，测试任务可能就完成不了。为了有效地、合理地减少输入条件的组合数，极大地降低工作量，可以利用

Pair-wise 方法、正交实验设计方法等来简化问题，大大减少组合数。

Pair-wise 方法，也称为“成对组合测试”、“两两组合测试”，即将众多因素的值两两组合起来而大大减少测试用例组合。这种方法是由 Mandl 于 1985 年在测试 Ada 编译程序时提出的。后来一些研究显示，通过应用成对覆盖测试技术，其模块覆盖率为 93.5%，判断覆盖率为 83%，满足测试的基本要求，具有经济有效的特点。虽然也可以采用三三组合、四四组合来达到更高的覆盖率，但其组合数也增长很快，增加了测试工作量。所以，对一般应用系统，采用两两组合即可基本满足测试的要求。

下面举一个例子说明这种方法的有效性。例如，在线购物网站有多种条件影响操作界面或操作功能，主要有以下几方面。

(1) 登录方式(LOGIN)：未登录、第一次登录、正常登录。

(2) 会员状态(MEMBERSHIP)：非会员、会员、VIP 会员、雇员。

(3) 折扣(DISCOUNT)：没有、假日 95 折、会员 9 折、VIP 会员 8 折。

(4) 物流方式(SHIP)：标准、快递、加急。

如果完全组合，其组合数是 3×4×4×3=144，但如果采用两两组合，其组合数只有 17 项，如表 3-10 所示，工作量减少了近 88%。组合数越多，则效果越显著，有一个基于不同测试环境的兼容性测试(6 个因素，每个因素取值 2～8 项不等)，其组合有 4704 个，但其两两组合数只有 61 个。如果靠手工方式生成组合还是比较麻烦，最好的方式还是靠工具自动生成组合，快捷有效，不容易出错。有许多工具可以选择，最方便的工具是微软的 PICT(见 http://www.pairwise.org/tools.asp)，而且还可以为不同条件之间设定约束关系，进一步优化或减少组合。对于上述这个例子，如果需要，可以加入下面这些约束条件：

IF [LOGIN]="未登录" THEN [MEMBERSHIP]="非会员";

IF [LOGIN]="第一次登录" THEN [MEMBERSHIP] <> "VIP 会员";

IF [MEMBERSHIP]="会员" THEN [DISCOUNT]="会员 9 折";

IF [MEMBERSHIP]="VIP 会员" THEN [DISCOUNT]="VIP 会员 8 折";

表 3-10 Pair-wise 方法的示例

序号	登录方式	会员状态	折扣	物流
1	未登录	雇员	假日 95 折	快递
2	正常登录	会员	假日 95 折	加急
3	第一次登录	VIP 会员	没有	标准
4	未登录	非会员	VIP 会员 8 折	标准
5	正常登录	非会员	没有	快递
6	未登录	雇员	没有	加急
7	第一次登录	非会员	假日 95 折	加急
8	第一次登录	VIP 会员	会员 9 折	快递
9	未登录	会员	没有	标准
10	正常登录	雇员	VIP 会员 8 折	标准
11	未登录	VIP 会员	VIP 会员 8 折	加急
12	未登录	雇员	会员 9 折	标准
13	正常登录	VIP 会员	假日 95 折	标准
14	正常登录	非会员	会员 9 折	加急
15	第一次登录	雇员	VIP 会员 8 折	快递
16	第一次登录	会员	会员 9 折	快递
17	正常登录	会员	VIP 会员 8 折	标准

3.3.4 正交试验法

解决组合数非常大的问题,除了 Pair-wise 方法之外,另一有效方法就是正交实验设计方法(Orthogonal Test Design Method,OTDM)。正交实验设计方法是依据 Galois 理论,从大量的(实验)数据(测试例)中挑选适量的、有代表性的点(条件组合),从而合理地安排实验(测试)的一种科学实验设计方法。

1. 确定影响功能的因子与状态

首先要根据被测试软件的规格说明书,确定影响某个相对独立的功能实现的操作对象和外部因素,并把这些影响测试结果的条件因素作为因子(Factors),而把各个因子的取值作为状态,状态数称为水平数(Levels)。即确定:

(1) 有哪些因素(变量)? 其因子数是多少?

(2) 每个因素有哪些取值? 其水平数是多少?

对因子与状态的选择可按其重要程度分别加权。可根据各个因子及状态的作用大小,出现频率的大小以及测试的需要,确定权值的大小。

2. 选择一个合适的正交表

根据因子数和最大水平数、最小水平数,选择一个测试(Run)次数最少的、最适合的正交表。正交表是正交试验设计的基本工具,它是通过运用数学理论在拉丁方和正交拉丁方的基础上构造而成的规格化表格,可以参考 http://www.math.hkbu.edu.hk/UniformDesign 或 http://www.research.att.com/~njas。一般用 L 代表正交表,常用的有 $L_8(2^7)$、$L_9(3^4)$、$L_{16}(4^5)$等。例如,$L_8(2^7)$中的 7 为因子数(即正交表的列数),2 为因子的水平数,8 为测试次数(即正交表的行数)。现在,还可以使用相应工具软件(如正交设计助手 II)来帮助决策和应用。

3. 利用正交表构造测试数据集

(1) 把变量的值映射到表中,为剩下的水平数选取值。

(2) 把每一行的各因素水平的组合作为一个测试用例。再增加一些没有生成的但可疑的测试用例。

在使用正交表来设计测试用例时,需要考虑不同的情况,例如,因子数和水平数相符、水平数不相符等。

① 如果因子数不同,可以采用包含的方法。在正交表公式中找到包含该情况的公式,如果有 N 个符合条件的公式,那么选取行数最少的公式。

② 如果水平数不等,采用包含和组合的方法选取合适的正交表公式。

4. 示例

在企业信息系统中,员工信息查询功能是常见的。例如,设有三个独立的查询条件,以获得特定员工的个人信息。

(1) 员工号(ID)。

(2) 员工姓名(Name)。

(3) 员工邮件地址(Mail Address)。

即有三个因子,每个因素可以填,也可以不填(空),即水平数为 2。根据因子数是 3、水平数是 2 和行数取最小值,所以选择 $L_4(2^3)$。这样就可以构造正交表,如图 3-4 左边所示。根据左边的结果,很容易得到所需的测试用例,如图 3-4 右边所示,这样基本测试用例设计完成。如果考虑一些特殊情况,再增加一个测试用例,即三项内容都为空,直接单击“查询”功能,进行查询。

		因子数		
水平数		1	2	3
	1	1	1	1
	2	1	0	0
	3	0	1	0
	4	0	0	1

		查询条件		
测试用例		员工号	姓名	邮件地址
	1	填	填	填
	2	填	空	空
	3	空	填	空
	4	空	空	填

图 3-4 正交表构建并转化为测试用例

从中可以看出,如果按每个因素两个水平数来考虑,需要 8 个测试用例,而通过正交实验法来设计测试用例只有 5 个,有效地减少了测试用例数,而测试效果是非常接近的,即用最小的测试用例集合去获取最大的测试覆盖率。对于因子数、水平数较高的情况下,测试组合数会很多,正交实验法的优势更能体现出来,可以大大降低测试用例数,降低工作量。

3.4 基于逻辑覆盖的方法

在进行单元测试时,特别是针对程序函数进行测试时,会优先考虑代码行的覆盖,一般认为这是最基本的,例如,在衡量开发的单元测试工作时,常常会设定一个目标就是代码行覆盖率要超过 80%或 100%。要做到代码行的覆盖,就是要做到代码结构的分支覆盖。如果再进一步,就是要检验构成分支判断的各个条件及其组合,即条件覆盖和条件组合覆盖。基本路径覆盖一般不归为逻辑覆盖,但从它们密切的关系看,可以统一为逻辑覆盖。逻辑覆盖不局限于代码这个层次,可以扩展到业务流程图、数据流图等,让测试覆盖需求层次的业务逻辑,这可能更为重要。

3.4.1 判定覆盖

判定覆盖法的基本思想是设计若干用例,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,即判断真假值均曾被满足。一个判定往往代表着程序的一个分支,所以判定覆盖也被称为分支覆盖。假如给出如下示例程序,绘制成如图 3-5(a)所示的程序流程图,为了便于表达问题,程序流程图(a)可以简化为流程图(b),其中:

条件 M={a>0 and b>0}

条件 N={a>1 or c>1}

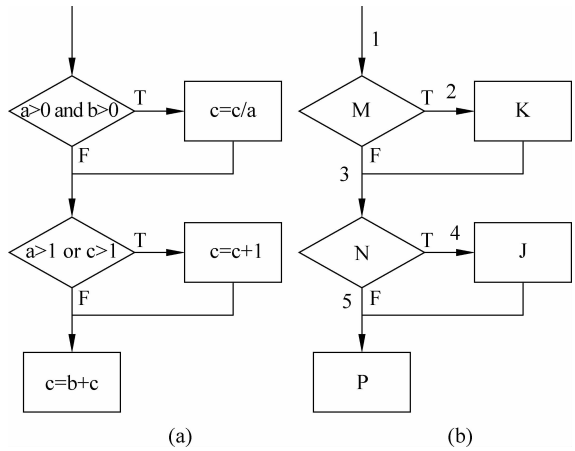

图 3-5 程序流程图

示例程序源码

```
Dim a,b As Integer
Dim c As Double
If (a > 0 AND b > 0) Then
    c = c/ a
End If
If (a > 1 OR c > 1) Then
    c = c + 1
End If
c = b + c
```

按照判定覆盖的基本思路,就是要设计相应的测试用例(为变量 a、b、c 赋予特定的值),可以使判定 M、N 分别为真和假,从而达到判定覆盖。例如:

(1) 令 a=2,b=1,c=6,覆盖 M=.T.且 N=.T.。

(2) 令 a=-2,b=1,c=6,覆盖 M=.F.且 N=.F.。

通过这两个测试用例,就达到判定覆盖的要求。如果满足了判定覆盖,也就满足了语句覆盖,但是,如果只测试了上面两个测试用例,这时程序中错将 and 写成 or、或错将 or 写成 and,上述两个测试用例是无法发现这类问题的。从逻辑关系来看,如表 3-11 所示,(1)和(4)逻辑关系判断的结果一样,如果是通过(1)和(4)组合完成分支覆盖,不能发现 and 和 or 互换的问题,AND 关系需要采用(1)和(2)组合或(1)和(3)组合,而 OR 关系需要采用(2)和(4)组合或(3)和(4)组合,才能避免这样的问题。

表 3-11 逻辑运算的各种组合

AND 关系	OR 关系
(1).T. and .T.→.T.	(1).T. or .T.→.T.
(2).T. and .F.→.F.	(2).T. or .F.→.T.
(3).F. and .T.→.F.	(3).F. or .T.→.T.
(4).F. and .F.→.F.	(4).F. or .F.→.F.

3.4.2 条件覆盖

条件覆盖的基本思想是设计若干测试用例，执行被测程序以后，要使每个判断中每个条件的可能取值至少满足一次。对于第一个判定条件 M，可分解成两个条件。

(1) 条件 a>0：取真(TRUE)时为 T1，取假(FALSE)时为 F1。

(2) 条件 b>0：取真(TRUE)时为 T2，取假(FALSE)时为 F2。

对于第二个判定条件 N，则分解成：

(1) 条件 a>1：取真(TRUE)时为 T3，取假(FALSE)时为 F3。

(2) 条件 c>1：取真(TRUE)时为 T4，取假(FALSE)时为 F4。

根据条件覆盖的基本思想，要分别让各个条件能取“.T.”或“.F.”来设计相应的测试用例，最优化的测试用例，就是 3.4.1 节所讨论的，满足表 3-11 中的(1)和(4)组合、或者(2)和(3)组合就能做到条件覆盖。如表 3-12 所示，就是选了(2)和(3)组合，覆盖了 4 个条件，但没有满足 3.4.1 节的判定覆盖要求，即判定条件 M 和 N 的“真、假”没有至少被执行一次，而是 M 取假两次、N 取真两次，也就不能保证代码行被覆盖，这也说明条件覆盖的测试不一定比代码行覆盖、判定覆盖好。

表 3-12 条件覆盖的测试用例

测试用例	取值条件	具体取值条件
输入：a=2,b=-1,c=-2 输出：a=2,b=-1,c=-2	T1,F2,T3,F4	a>0,b≤0,a>1,c≤1
输入：a=-1,b=2,c=3 输出：a=-1,b=2,c=6	F1,T2,F3,T4	a≤0,b>0,a≤1,c>1

在 3.4.1 节判定覆盖测试中，如果对 AND 覆盖测试选择(1)和(2)、对“OR”覆盖测试选择(2)和(4)，那么条件则没有做到全覆盖，这也说明达到判定覆盖的要求，也不能保证条件覆盖，即判定覆盖也不比条件覆盖强。为了进行更充分的测试，必须引入判定-条件覆盖。

3.4.3 判定-条件覆盖

判定-条件覆盖实际上是将前两种方法结合起来的一种设计方法，它是判定和条件覆盖设计方法的交集，即设计足够的测试用例，使得判断条件中的所有条件可能取值至少执行一次，同时，所有判断的可能结果至少执行一次。按照这种思想，在前面的例子中应该至少保证判定条件 M 和 N 取真和取假各一次，同时要保证 8 个条件取值(T1,F1,T2,F2,…,F4)也至少被执行一次。根据前面的讨论，实际上，在 3.4.1 节就已经知道按照表 3-11 中的(1)和(4)的组合能解决这个问题，按照表 3-13 可完成测试用例的设计。即使做到判定条件覆盖，3.4.1 节中谈到的问题：AND 和 OR 互换问题是无法被监测的，所以测试仍不够充分，需要引入条件组合覆盖。

表 3-13 判定-条件覆盖的测试用例

测试用例	取值条件	具体取值条件	判定条件
输入：a=2,b=1,c=6 输出：a=2,b=1,c=5	T1,T2,T3,T4	a>0,b>0,a>1,c>1	M=.T. N=.T.
输入：a=-1,b=-2,c=-3 输出：a=-1,b=-2,c=-5	F1,F2,F3,F4	a≤0,b≤0,a≤1,c≤1	M=.F. N=.F.

3.4.4 条件组合覆盖

条件组合覆盖的基本思想是设计足够的测试用例,使得判断中每个条件的所有可能至少出现一次,并且每个判断本身的判定结果也至少出现一次。它与条件覆盖的差别是它不是简单地要求每个条件都出现“真”与“假”两种结果,而是要求让这些结果的所有可能组合都至少出现一次。

按照条件组合覆盖的基本思想,对于前面的例子,设计组合条件如表 3-14 所示。

表 3-14 存在的 8 种组合条件示例

组合编号	覆盖条件取值	判定条件取值	判定-条件组合
1	T1,T2	M=.T.	a>0,b>0,M 取真
2	T1,F2	M=.F.	a>0,b≤0,M 取假
3	F1,T2	M=.F.	a≤0,b>0,M 取假
4	F1,F2	M=.F.	a≤0,b≤0,M 取假
5	T3,T4	N=.T.	a>1,c>1,N 取真
6	T3,F4	N=.T.	a>1,c≤1,N 取真
7	F3,T4	N=.T.	A≤1,c>1,N 取真
8	F3,F4	N=.F.	A≤1,c≤1,N 取假

针对 8 种组合条件,再来设计能覆盖所有这些组合的测试用例,如表 3-15 所示。

表 3-15 条件组合覆盖的测试用例

测试用例	覆盖条件	覆盖路径	覆盖组合
输入:a=2,b=1,c=6 输出:a=2,b=1,c=5	T1,T2,T3,T4	P1(1-2-4)	1,5
输入:a=2,b=-1,c=-2 输出:a=2,b=-1,c=-2	T1,F2,T3,F4	P3(1-3-4)	2,6
输入:a=-1,b=2,c=3 输出:a=-1,b=2,c=6	F1,T2,F3,T4	P3(1-3-4)	3,7
输入:a=-1,b=-2,c=-3 输出:a=-1,b=-2,c=-5	F1,F2,F3,F4	P4(1-3-5)	4,8

在表 3-15 中引入了路径的概念,即程序执行经过的程序流程图的轨迹。由流程图 3-5(b)可以知道,该程序模块有 4 条不同的路径:

P1:(1-2-4)即 M=.T.且 N=.T.
P2:(1-2-5)即 M=.T.且 N=.F.
P3:(1-3-4)即 M=.F.且 N=.T.
P4:(1-3-5)即 M=.F.且 N=.F.

这样根据每个测试用例所经历的程序代码,就能确定其覆盖的路径 P1、P3 和 P4,但 P2(1-2-5)没有被覆盖。这也说明对于最强的逻辑覆盖——条件组合覆盖,其测试也不是非常充分的,所以,更充分的测试,不仅要求覆盖各个条件和各个判定,而且还能覆盖基本路径。即使这样,也不够充分,还需要考虑输入数据域、数据流控制。例如,在上面的代码中,对 a、b、c 取值则要考虑边界条件,而且要考虑对测试结果产生的影响,例如,从边界条件看,b 需要取 0、-0.1 和 0.1,但从最后语句(c=c+b)来看,b 最好不取零,从而能够观察 b 值对 c 值

的影响。

但如果进一步仔细分析，可能会觉得条件组合测试有些浪费，例如，选择表 3-14 第 3、4 行来分析，固定第一个条件(因为都是 F1)，改变第 2 个条件(即取不同的值 T2、F2)，其结果不受影响，这样的测试就没有意义。反过来，像表 3-14 第 1、2 行，固定第一个条件(因为都是 T1)，改变第 2 个条件(即取不同的值 T2、F2)，其结果不一样，这样的测试才有价值。同样，第 5、6 行放在一起测试没意义，而 7、8 行放在一起测试有价值。概括起来，对表 3-11 进行优化，每一逻辑运算(AND 或 OR)其必要的测试包含三组，如表 3-16 所示。这就引出修改的条件/判定覆盖(Modified Condition/Decision Coverage，MC/DC)，MC/DC 测试覆盖要求如下。

(1) 每一个判断的所有可能结果都出现过；

(2) 每一个判断中所有条件的取值都出现过；

(3) 每一个进入点及结束点都执行过；

(4) 判断中每一个条件都可以独立地影响判断的结果。

航空软件质量标准 DO-178C 中指定会影响飞机起飞及降落安全性的软件(A 等级软件)需满足 MC/DC。

表 3-16 必要的测试条件组合

AND 关系	OR 关系
(1) . T. and . T. →. T.	(1) . T. or . F. →. T.
(2) . T. and . F. →. F.	(2) . F. or . T. →. T.
(3) . F. and . T. →. F.	(3) . F. or . F. →. F.

3.4.5 基本路径覆盖

顾名思义，基本路径覆盖就是设计所有的测试用例，来覆盖程序中的所有可能的、独立的执行路径。根据 3.4.4 节讨论，也就是调整表 3-15 中第 2、3 个测试用例，使测试不仅覆盖路径 P3(1-3-4)，而且能够覆盖路径 P2(1-2-5)，这样就可以完全覆盖路径 P1、P2、P3 和 P4。例如，调整第 2 个测试用例后，就能覆盖 P2(1-2-5)，如表 3-17 所示，但不能保证覆盖所有的条件组合(如组合 2、6)。

表 3-17 基本路径覆盖的测试用例

测试用例	覆盖路径	覆盖条件	覆盖组合
输入：a=2，b=1，c=6 输出：a=2，b=1，c=5	P1(1-2-4)	T1，T2，T3，T4	1，5
输入：a=1，b=1，c=－3 输出：a=1，b=1，c=－2	P2(1-2-5)	T1，T2，F3，F4	1，8
输入：a=－1，b=2，c=3 输出：a=－1，b=2，c=6	P3(1-3-4)	F1，T2，F3，T4	3，7
输入：a=－1，b=－2，c=－3 输出：a=－1，b=－2，c=－5	P4(1-3-5)	F1，F2，F3，F4	4，8

通过前面的例子可以看到，采用其中任何一种方法都不能完全覆盖所有的测试用例，因此，在实际的测试用例设计过程中，可以根据需要和不同的测试用例设计特征，将不同的设计方法组合起来，交叉使用，以达到最高的覆盖率。

采用条件组合和路径覆盖两种方法的结合来重新设计测试用例，如表 3-18 所示，也就是在表 3-15 或表 3-17 基础上增加一个用例，通过共 5 个测试用例就能覆盖各种情况，包括条件、判定、条件组合、路径等，使程序得到完全的测试。

表 3-18　完全覆盖的测试用例

测试用例	覆盖路径	覆盖条件	覆盖组合
输入：a=2,b=1,c=6 输出：a=2,b=1,c=5	P1(1-2-4)	T1,T2,T3,T4	1,5
输入：a=1,b=1,c=−3 输出：a=1,b=1,c=−2	P2(1-2-5)	T1,T2,F3,F4	1,8
输入：a=2,b=−1,c=−2 输出：a=2,b=−1,c=−2	P3(1-3-4)	T1,F2,T3,F4	2,6
输入：a=−1,b=2,c=3 输出：a=−1,b=2,c=6	P3(1-3-4)	F1,T2,F3,T4	3,7
输入：a=−1,b=−2,c=−3 输出：a=−1,b=−2,c=−5	P4(1-3-5)	F1,F2,F3,F4	4,8

基本路径覆盖的前提就是知道有多少条基本路径，对于简单程序，通过直接观察就能掌握，但对于复杂的应用程序就很难了。基本路径测试法是在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例的方法。设计出的测试用例要保证被测试程序的每个可执行语句至少被执行一次。基本路径测试法通过以下几个基本步骤来实现。

(1) 程序的流程图。程序流程控制图是描述程序控制流的一种图示方法，可以用如图 3-6 所示基本图元(顺序、分支、循环等)来描述任何程序结构。图 3-5 可以转化为如图 3-7 所示的程序流程图。

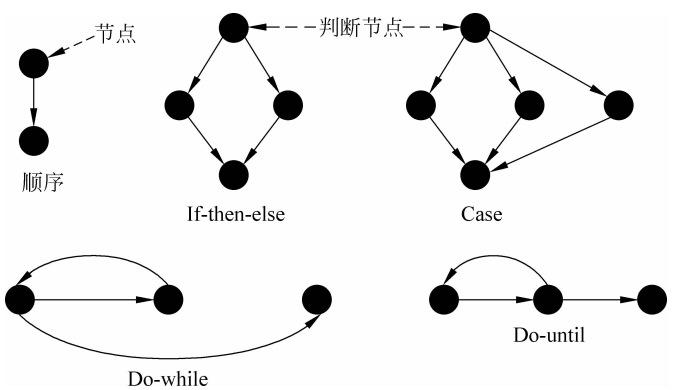

图 3-6　程序流程的基本图元

(2) 计算程序环路复杂度。通过对程序的控制流程图的分析和判断来计算模块复杂性，从程序的环路复杂性可导出程序基本路径集合中的独立路径条数。环路复杂性可以用 $V(G)$ 来表示，其计算方法有：

① $V(G)$=区域数目。区域是由边界和节点包围起来的形状所构成，计算区域时应包括图的外部区域，将其作为一个区域。图 3-7 的区域数目是 3，也就是有三条基本路径。

② $V(G)$=边界数目−节点数目+2。按此计算，图 3-7 结果也是 3(即 6−5+2)。

③ $V(G)$=判断节点数目+1。如图 3-7 中,判断节点有 A、C,则 $V(G)=2+1=3$。

(3) 确定基本路径。通过程序流程图的基本路径来导出基本的程序路径的集合。通过上面的分析和计算,知道如图 3-7 所示程序有三条基本路径,下面给出一组基本路径。在一个基本路径集合里,每条路径是唯一的。但基本路径组(集合)不是唯一的,还可以给出另外两组基本路径。

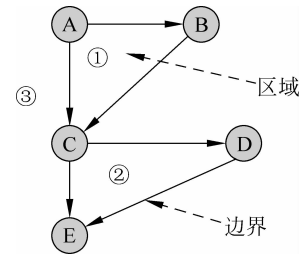

图 3-7 程序流程图示例

A—C—E

A—B—C—E

A—B—C—D—E

(4) 准备测试用例,确保基本路径组中的每一条路径被执行一次。

① a=—1,b=—2,c=—3 可以覆盖路径 A—C—E。

② a=1,b=1,c=—3 可以覆盖路径 A—B—C—E。

③ a=2,b=1,c=6 可以覆盖路径 A—B—C—D—E。

3.5 基于缺陷模式的测试

如果过去我们犯了不少错误(即产生软件缺陷),自然会想从中吸取教训,对过去所发现的各种具体缺陷进行归纳整理,抽象出共性,生成缺陷模式,然后基于这种模式去预防问题,也可以用这种模式来检查被测试对象,看是否有相互匹配的问题。在软件测试中,如果知道其缺陷模式,就可以根据缺陷模式进行匹配,然后发现类似的问题,这就是基于缺陷模式的测试(Defect-Pattern-Based Testing,DPBT)。错误猜测法主要是根据测试人自己的经验,按照常见的问题进行探测性的测试,一般属于手工测试。如果将常见的缺陷模式固化到测试工具中,然后就可通过工具进行静态分析以完成测试。如果是针对代码缺陷模式的测试工具,就可以用工具对代码进行扫描以完成测试,许多静态测试工具,如 FindBugs、flawfinder、Klocwork Insight、Fortify Static Code Analyzer 等,都是基于缺陷模式实现的。

如果检测算法是完全的,则能够从软件中排除该类模型。例如,在内存为 1GB、CPU 为 1.8GHz 的 PC 上,FindBugs 对 J2SE 中的 rt.jar 分析,该程序包有 13 083 个类,约 40MB 大小,所耗时间只需 45min。基于缺陷模式的测试方法具有测试效率高、对逻辑复杂故障测试效果好等特点,并且比较容易实现自动化测试。采用工具进行检验,能够发现其他测试方法所不能发现的软件故障和安全隐患,而且只要建立规则而不需要开发测试脚本,测试的投入低、产出高,应用效果良好,但也要看到其不足之处。

(1) 误报问题。误报问题是基于缺陷模式的软件测试技术的一个共性问题。通常基于缺陷模式的软件测试技术都属于静态分析技术,由于某些故障的确定需要动态地执行信息,因此对于基于静态分析的工具来说,误报问题是不可避免的。故障的最后确认一般需要人工分析,如果误报过多,确认的工作量就可能过大而无法忍受。为此,将动态测试与静态测试有机结合来解决误报问题。

(2) 漏报问题。漏报问题主要由缺陷模式定义和模式检测算法引起。目前对于软件缺陷模式没有一个规范的、统一的和形式化的定义。不同的故障查找工具都给出自己所能检测的缺陷模式的定义,但是这些定义很多都是一些自然语言的描述,有的甚至只是给出一个简单的

例子进行说明。同时,针对具体的软件缺陷模式,不同的检测工具一般设计自己的检测算法,从检测的效率和实现的复杂性上考虑,不同的算法给出不同的假设以降低计算复杂性,这导致对于相同的模型,用不同的工具进行检测得到的缺陷结果集合很大不同。参考文献 11 使用一些常用的 Java 程序故障自动分析工具对同一个软件进行测试,发现不同的工具得到的检测结果集(故障集)差别较大。

(3) 模式机理。由于编程过程中,程序员具有较强的个体性,因此缺陷模式是多种多样的。通常软件中的故障主要来源于程序员,如错误的理解造成的、二义性造成的、疏忽造成的和遗漏造成的。

3.5.1 常见的缺陷模式

大量的测试数据统计分析可以发现有些软件缺陷是具有共性的,所以总结了常见的一些缺陷模式,例如,内存泄漏缺陷模式、非法指针引用缺陷模式。其实,导致程序员犯错的因素很多,有程序员本身的编程水平、习惯以及所属团队软件工程管理水平等,编程语言及其相关类库的有些难以理解的特性也是一个比较重要的原因。

(1) 故障模式。即常见的软件故障,如内存泄漏、使用空指针、数组越界、非法计算、使用未初始化变量、不完备的构造函数以及操作符异常等。

(2) 安全漏洞模式。即为他人攻击系统提供可能,而一旦攻击者得手,系统就可能发生瘫痪,所造成的危害可能更大,因此,此类漏洞应当尽量避免,如缓冲区溢出漏洞模式、被感染数据漏洞模式、竞争条件漏洞模式以及风险操作随机数漏洞模式等。

(3) 差性能模式。该模式在软件动态运行时效率比较低下,因此建议采用更高效的代码来完成同样的功能。这类模式主要包括调用了不必要的基本类型包装类的构造方法、空字符串的比较、复制字符串、未声明为 static 的内部类、参数为常数的数学方法、创建不必要的对象以及声明未使用的属性及方法等。

(4) 并发缺陷模式。该模式主要针对程序员对多线程、Java 虚拟机的工作机制不了解引起的问题,以及由于线程启动的任意性和不确定性使用户无法确定所编写的代码具体何时执行而导致对公共区域的错误使用,如死锁等。

(5) 不良习惯模式。该模式主要是由于程序员编写代码的不好习惯造成的一些错误。包括文件的空输入、垃圾回收的问题,以及类、方法和域的命名问题,方法调用,对象序列化,域初始化,参数传递和代码安全性问题等。

(6) 代码国际化模式。该模式主要是在语言进行国际化的过程中,可能造成本地设置和程序需求不符的情况,造成匹配错误。

(7) 易诱骗代码模式。该模式主要指代码中容易引起歧义的、迷惑人的编写方式。比如无意义的比较,永远是真值的判断,条件分支使用相同的代码,声明了却未使用的域等,即那些混淆视听,无法正常判断程序的真正意图的代码。

3.5.2 DPBT 的测试过程

测试过程从源代码输入开始,经历预编译、词法分析、语法分析与语义处理、抽象语法树生成、控制流图生成和 IP(Illegal Pattern,非法模式)扫描等几个步骤,最后自动生成 IP 报表。

(1) 预处理。由于源程序中存在宏定义、文件包含和条件编译等预处理命令,因此在进行词法分析前必须进行预编译,将宏进行展开,这样有利于变量的查找。

(2) 词法分析(Lexical Analysis)。将预编译阶段产生的中间代码进行分解,形成各种符号表,为语法分析做准备。符号表的结构主要有标识符表、类型表、关键字表、常数表、运算符表和分界符表。

(3) 语法分析(Parsing)和语义处理(SemanticAnalysis)。这一步主要是将输入字符串识别为单词符号流,并按照标准的语法规则,对源程序做进一步分析,区分出变量定义、赋值语句、函数等。语法分析的结果是生成语法树,并提供对外的接口。此外,通过语法树可以生成程序的控制流图和变量的定义使用链,为下一步的故障查找做准备。

(4) 抽象语法树生成。语法分析和语义处理之后生成抽象语法树,源程序中的所有语句都作为抽象语法树的节点。抽象语法树是后续操作的基础,含有后续处理所需的各种信息,如语句类型、变量名称及类型等。

(5) 控制流图生成。在抽象语法树的基础上生成控制流图,描绘程序结构。生成的控制流图必须反映源程序的结构。

(6) IP 扫描。IP 扫描是测试过程的关键步骤,首先定义测试模型,然后在控制流图上遍历对测试模型进行匹配,从而生成 IP 报表。工具所能检测的故障的集合取决于定义的测试模型集合。

(7) 人工确认。由于误报的存在,因此需要对生成的 IP 进行人工确认。

3.6 基于模型的测试

模型是对系统的抽象,是对被测系统预期行为动作的抽象描述,实际上就是用语言把一个系统的行为描述出来,定义系统的各种状态及其之间的转换关系,例如随机模型、贝叶斯图解模型、有限状态模型等。基于模型的测试(Model-Based Testing,MBT)是利用模型来生成相应的测试用例,然后根据实际结果和原先预想的结果的差异来测试系统,如图 3-8 所示。基于模型的测试,先是从概念上形成模型,然后试图用数学的方法来描述这个模型,逐渐实现这个模型,形成仿真模型,完成所需的测试。基于模型的测试,往往不是直接针对被测系统(System Under Test,SUT)进行,而是根据算法、规则,针对源代码进行检测。

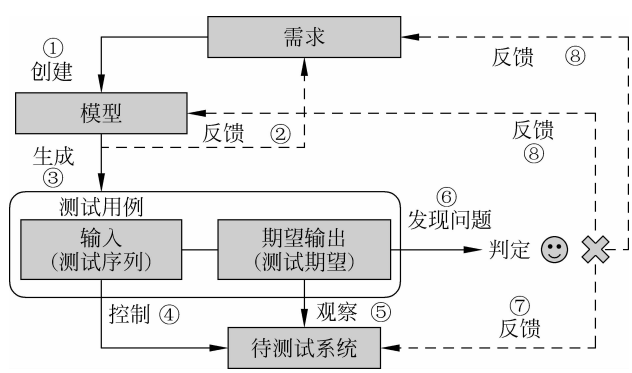

图 3-8 基于模型测试的示意图

基于模型的软件测试技术不能替代已有的其他测试技术,而是对其他测试技术一个有力的补充。基于模型的软件测试技术已应用于通信协议测试、API 测试等,微软研究院用 C# 开发了相应的工具——Spec Explorer,如图 3-9 所示,它还能与 Visual Studio 集成在一起。

官方网站：http://msdn.microsoft.com/en-us/devlabs/ee692301.aspx

Spec Explorer 团队博客：http://blogs.msdn.com/sechina/

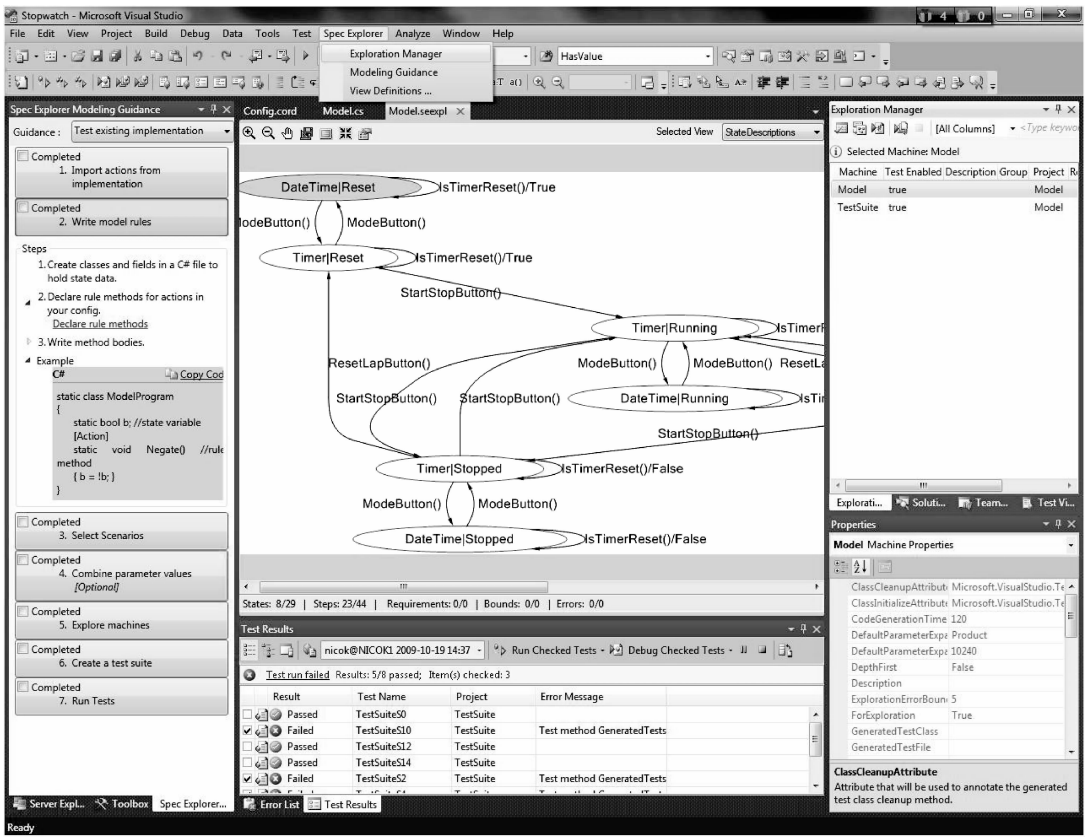

图 3-9　微软的测试建模工具 Spec Explorer 主要界面

3.6.1　功能图法

一个程序的功能通常由静态说明和动态说明组成，动态说明描述了输入数据的次序或者转换的次序；静态说明描述了输入条件和输出条件之间的对应关系。对于比较复杂的程序，由于大量的组合情况的存在，如果仅使用静态说明来组织测试往往是不够的，必须通过动态说明来补充测试。功能图法就是因此而产生的一种测试用例设计方法。

功能图法就是使用功能图形式化地表示程序的功能说明，并机械地生成功能图的测试用例。功能图模型由状态迁移图和逻辑功能模型组成。

(1) 状态迁移图用于表示输入数据序列以及相应的输出数据，由输入和当前的状态决定输出数据和后续状态。

(2) 逻辑功能模型用于表示在状态输入条件和输出条件之间的对应关系。逻辑功能模型只适合于描述静态说明，输出数据仅由输入数据决定。

测试用例需要覆盖一系列的系统状态，并依靠输入输出数据满足的一对条件来触发每个状态的发生。举个例子来说明，假设进行 Windows 的屏幕保护程序测试(有密码保护功能)，图 3-10、图 3-11 和表 3-19 分别呈现了程序流程图、状态迁移图以及对应的逻辑功能表。

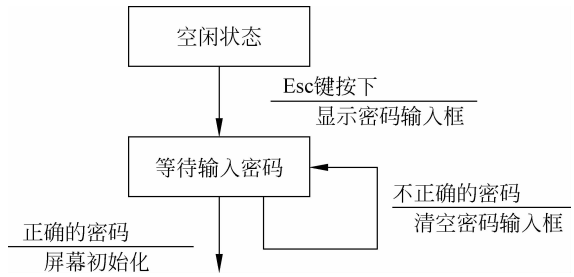

图 3-10 程序流程图

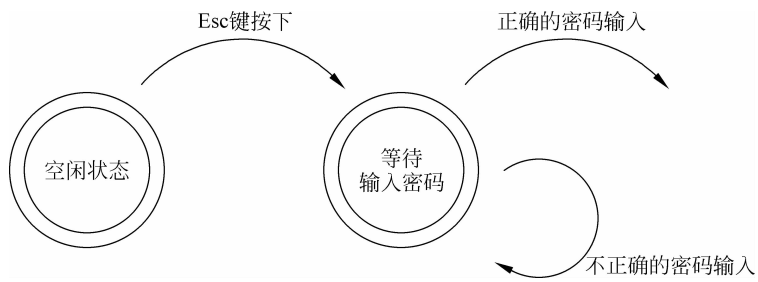

图 3-11 状态迁移图

表 3-19 逻辑功能表

输入	Esc 键按下	I1
	其他键按下	I2
	正确的密码输入	I3
	错误的密码输入	I4
输出	显示密码输入框	O1
	密码错误提示信息	O2
状态	空闲状态	S1
	等待输入密码	S2
	返回空闲状态	S3
	初始化屏幕	S4

接下来,需要利用功能图来生成测试用例,从逻辑功能表中,可以根据所有的输入输出以及状态来生成所需要的节点和路径,形成实现功能图的基本路径组合。这样,就可以使用3.4.5节介绍的基本路径覆盖法来设计测试用例。

3.6.2 模糊测试方法

模糊测试(Fuzz testing)方法,简单地说,就是通过一个自动产生数据的模板或框架(称为模糊器)来构造或自动产生大量的、具有一定随机性的数据作为系统的输入,从而检验系统在各种数据情况下是否会出现问题。例如,在键盘或鼠标大量随机输入的情况下,早期的Windows NT 4.0有21%的程序会崩溃,还有24%的程序会挂起。

模糊测试方法在1989年由威斯康星州的麦迪逊大学的Barton Miller教授提出,他的实验内容是开发一个基本的命令行模糊器以测试UNIX程序。这个模糊器可以用随机数据来"轰炸"这些测试程序直至其崩溃。早期的模糊测试工具是1991年发布的crashme,其主要功

能是让 UNIX 系统去执行随机机器指令以测试这些系统的健壮性。这方面的论文从 1990 年开始陆续发表,可以参考网站 http://www.cs.wisc.edu/~bart/fuzz/fuzz.html,但以前应用不多,而当互联网应用越来越广泛时,软件系统的安全性成为人们关注的焦点,模糊测试方法又重新得到重视。

模糊测试方法可以模拟黑客来对系统发动攻击测试,在安全性测试上发挥作用之外,还可以用于对服务器的容错性测试。模糊测试方法缺乏严密的逻辑,不去推导哪个数据会造成系统破坏,而是设定一些基本框架,在这个框架内产生尽可能多的杂乱数据进行测试,发现一些意想不到的系统缺陷。由于要产生大量数据,模糊测试方法一般不能通过手工测试,而是通过工具来自动执行。模糊测试工具的工作过程比较简单,即经过下列 4 个步骤。

(1) 测试工具通过随机或是半随机的方式生成大量数据;

(2) 测试工具将生成的数据发送给被测试的系统(输入);

(3) 测试工具检测被测系统的状态(如是否能够响应,响应是否正确等);

(4) 根据被测系统的状态判断是否存在潜在的安全漏洞。

模糊测试的工具有 SPIKE、Sulley、COMRaider、iDbg、WebFuzz、ProtoFuzz、DFUZ(www.genexx.org/dfuz)等,更多的工具请参考 www.fuzzing.org。

模糊测试工具的核心就是所构造的模糊器。模糊器一般分为以下两种。

(1) 基于变异(Mutation-based)的模糊器;

(2) 基于生成(Generation-based)的模糊器。

例如,SPI 模糊器是一个简单但设计很精巧的图形化 Web 应用模糊器,它向用户提供了对模糊测试所使用的原始 HTTP 请求的完整的控制。工具可以抓取到客户端和服务器之间的通信数据,根据这些数据分析出客户端与服务器之间的通信协议,然后根据协议的定义,自动填充可变字段的内容,实现数据的变异,然后再向服务器发送这些经过变异的数据,尝试找到可能的漏洞。模糊测试工具还可以作为攻击服务器的武器,例如:

(1) 发送巨量的随机数据,来进行服务器的攻击测试,可能会导致网站拒绝服务。

(2) 通过大量随机测试,可能会实现 HTTP 报文注入,获得服务器的权限,或导致服务器的 HTTP 服务不可用,可参考 http://sourceforge.net/projects/taof/。

可以用下面两个示例来进一步解释模糊测试方法。

【示例一】 第一个例子是微软的字处理软件 Word。如果要测试 Word 的容错性,是不是需要考虑创建各种不同的文件来验证? 例如,创建几十万个 Word 文档,而且它们的文件名、大小和内容都不相同,是随机的,当然,这些文档不能让 Word 自身创建,而是通过开发一个工具,将随机的二进制数据源输送到某个文件中来生成测试文档。也可以准备一份 Word 文件,用随机数据替换该文件中的一部分内容,生成新的测试文件。也可以将整个文件打乱,而不是仅替换其中的一部分。总之,产生大量的、包含随机数据的文件来对 Word 文件分析器进行测试,其测试结果会出现以下三种情况。

(1) 对于大多数测试文件,会出现“文件格式无法识别”或“文件已破坏而无法打开”等错误提示。

(2) 少数文件将会正常打开,在几十万个文档中可能包含某些可识别控件和可打印字符的组合。

(3) Word 可能会出错,可能有几个文件包含文件分析程序没有预见到的数据。

第三种情况正是模糊测试方法的价值,发现了 Word 中的问题。

【示例二】 网络数据传输过程中，数据包可能会丢失，源数据包是正确的，而服务器收到的数据包可能是不正确的数据包，这时服务器是否能够抛弃非法数据包，就成为服务器稳定性的重要能力之一。在网络环境中，有很多的网络协议，包括 TCP/IP、UDP、HTTP、LDAP、FTP、SIP、DHCP、DNS、SMB、SMTP 等。每个协议实现都包括一个针对来源于网络的数据包的分析程序，而每个分析程序都可能需要一些复杂的处理逻辑，往往会存在许多边界情况，从而使之难以验证。这时候，不得不借助模糊测试方法。

例如，需要测试一个 HTTP 客户端(浏览器)，可以让客户端发送由纯随机数据构成的模糊化请求。如果这种方式效果不明显，可以配置模糊测试参数或框架，使用已知有效数据、故意错误数据和随机数据的组合，而不是用大量纯随机数据来测试该客户端，以达到更好的测试效果。

在模糊化的过程中，测试数据会随着对可疑行为的进一步了解而不断完善。例如，HTTP 客户端发出的请求最初包含随机数据，随后可能会增加各种已知的有效数据或错误数据来进行更深入的验证。

3.7 形式化测试方法

在软件需求定义中，当采用自然语言来描述时，其语义不够清晰，容易存在歧义性。需求分析的结果也往往依赖于参与者的经验和理解，没有严格量化的标准。在需求之后的设计、实施活动会受到影响，对功能特性的验证缺乏客观、定量的依据，具有不确定性，测试覆盖率的度量不可靠等问题。这些问题，对于一般的应用软件可以被接受，但是对于一些非常关键的软件应用系统，如核电站控制软件、航天器的控制系统和导弹防御系统等，这些问题必须得到解决。

为了解决基于自然语言的设计和描述所带来的问题，人们提出了形式化方法。形式化方法的基础是数学和逻辑学，通过严格的数字逻辑和形式语言来完成软件(需求、设计规格等)定义，其结果语义清晰、无歧义，然后可以通过相应的工具实施自动化分析、编码和验证。UML(Unified Modeling Language，统一建模语言)可以看作是一种半形式化的方法，虽然不能完全量化地描述软件，但已具有了较清晰定义的形式和部分的语义定义，并有相应的工具可以帮助自动生成代码、测试用例等。

3.7.1 形式化方法

形式化方法实际上就是基于数学的方法来描述目标软件系统属性的一种技术。不同的形式化方法的数学基础是不同的，例如：

(1) VDM 就是基于一阶谓词逻辑和已建立的抽象数据类型来描述每个运算或函数的功能，从而形成一种功能构造性规格说明技术；

(2) 形式规格说明语言 Z 则是基于一阶谓词和集合论的数学基础，利用(状态/操作)模式和模式演算对目标软件系统的结构和行为特征进行抽象描述；

(3) 基于时态逻辑的形式化方法着重描述并发系统中状态迁移序列，用于刻画并发系统所需验证的性质。

形式化方法主要通过形式化规范语言(Formal Specification Language，FSL)来完成需求定义、设计、编程和测试的描述，而且这种描述是通过数学方法实现的，具有精确语义，所以可

以保证描述的一致性和完备性等。可以这么说,凡是采用严格的数学语言、具有精确的数学语义的方法,都称为形式化方法。形式化规范说明语言,一般由以下三个主要的成分构成。

(1) 语法,定义用于表示规约的特定符号;

(2) 语义,帮助定义用于描述系统的对象及其属性;

(3) 一组关系,定义如何确定某个对象是否满足规约的规则。

形式化方法并不是解决软件开发问题的万能灵药。它也有缺点,也不能保证不出现错误。例如,在非形式化的客观需求与形式化规范之间的关系,难以很好地处理。从理论上看,通过对形式化规范进行深入分析,证明所需性质,但实际证明比较复杂,需要强有力的支持形式化方法的工具仍然比较缺乏。

形式化方法的更大作用体现在软件规格和验证之上,这包括软件系统的精确建模和软件规格特性的具体描述,即可以看作是面向模型的形式化方法和面向属性的形式化方法。如果进一步进行分类,形式化方法可以分为以下几类。

(1) 基于模型的方法,通过明确定义状态和操作来建立一个系统模型,使系统从一个状态转换到另一个状态。用这种方法虽可以表示非功能性需求(诸如时间需求),但不能很好地表示并发性。基于这种方法的语言有 Z 语言、B 语言等。

(2) 代数方法,通过联系不同操作间的行为关系而给出操作的隐式定义,而不定义状态。同样,它也未给出并发的显式表示。这类方法的形式化语言有 OBJ(http://cseweb.ucsd.edu/~goguen/sys/obj.html)、CLEAR、语义语言(Action Semantic Language,ASL)、ACT 等。

(3) 过程代数方法,给出并发过程的一个显式模型,并通过限制所有容许的、可观察的过程间通信来表示系统行为,如通信顺序过程(Communicating Sequential Processes,CSP)、通信系统演算(Calculus of Communication Systems,CCS)、通信过程代数(Algebra of Communicating Process,ACP)、时序排序规约语言(Language of Temporal Ordering Specification,LOTOS)、计时 CSP(TCSP)、通信系统计时可能性演算(TPCCS)等形式化语言。

(4) 基于逻辑的方法,用逻辑描述系统预期的性能,包括底层规约、时序和可能性行为,并采用与所选逻辑相关的公理系统证明系统具有预期的性能。它还可采用具体的编程构造扩充逻辑从而得到一种广谱的形式化方法,通过保持正确性的细化步骤集来开发系统。如区间时序逻辑(IntervalTemporalLogic,ITL)、Hoare 逻辑、模态逻辑、时序逻辑、时序代理模型(Temporal Agent Model,TAM)、命题线性时序逻辑(Propositional Linear TemporalLogic,PLTL)、实时时序逻辑(Real Time TemporalLogic,RTTL)、计算树逻辑(Computation Tree Logic,CTL)等。

(5) 基于网络的方法,根据网络中的数据流显式地给出系统的并发模型,包括数据在网中从一个节点流向另一个节点的条件。采用具有形式语义的图形语言,具有易理解性,如 Petri 网、谓词变换网等。

3.7.2 形式化验证

形式化验证,就是根据某些形式规范或属性,使用数学方法(形式逻辑方法)证明其正确性或非正确性。形式化验证首先被用于生成软件规格说明书,然后将其作为软件开发的基础和软件测试验证的依据。因为它是基于一种严格定义的规范语言来描述软件产品,这样可以借助相应的工具来完成软件产品的验证。对形式化规范进行分析和推理,研究它的各种静态和动态性质,验证是否一致、是否完整,从而找出所存在的错误和缺陷。

传统的验证方法包括模拟和测试，都是通过实验的方法对系统进行查错。模拟和测试分别在系统抽象模型和实际系统上进行，其一般的方法是在系统的某点给予输入，观察在另一点的输出，要完成大量的数据输入和输出结果的检查，而且由于实验所能涵盖的系统行为有限，很难找出所有潜在的错误。因此，早期的形式验证主要研究如何使用数学方法，严格证明一个程序的正确性(即程序验证)。

软件测试无法证明系统不存在缺陷，也不能证明它符合一定的属性。只有形式化验证过程可以证明一个系统不存在某个缺陷或证明一个系统符合某个属性。但是，还是无法证明某个系统没有缺陷，这是因为不能形式化地定义“没有缺陷”。所以，我们能做的就是证明一个系统不存在可以想得到的缺陷，以及验证满足系统质量要求的属性。

目前关于形式化验证方法的研究主要集中在信念逻辑、代数方法、模型检测等方面，例如：

(1) 采用有限状态机(Finite State Machine，FSM)或扩展有限状态机(Enhance Finite State Machine，EFSM)进行模型检验。

(2) 采用 SPIN(http://spinroot.com)和线性时态语言验证其相关属性。

(3) UML 语义转换形式化验证。

(4) 标准 RBAC 模型，包括 4 个部件模型：基本模型 RBAC0(Core RBAC)、角色分级模型 RBAC1(Hierarchal RBAC)、角色限制模型 RBAC2(Constraint RBAC)和统一模型 RBAC3(Combines RBAC)。

(5) 扩展的 RBAC(Role-Based Access Control，基于角色的存取控制)模型和基于粒计算(Granular Computing)的 RBAC 模型(G-RBACModel)。

(6) 符号模型检验(Symbolic Model Checking)，将问题形式化成为一种特定的符号表示，然后诉诸于某种特定的问题求解方法，如 BDD (Binary Decision Diagram)、SAT(可满足性问题的求解器)、ATPG(Automatic Test Pattern Generation，自动测试模式发生器)、定理证明器等。

(7) BAN(Burrows-Abadi-Needham)逻辑模型，用于安全协议的验证。

下面着重讨论基于模型的软件测试和扩展有限状态机方法等。

3.7.3 扩展有限状态机方法

有限状态机是一种用来进行对象行为建模的工具，其作用主要是描述对象在它的生命周期内所经历的状态序列，以及如何响应来自外界的各种事件。在面向对象的软件系统中，一个对象无论多么简单或者多么复杂，都必然会经历一个从开始创建到最终消亡的完整过程，这通常被称为对象的生命周期。一般说来，对象在其生命期内是不可能完全孤立的，它必须通过发送消息来影响其他对象，或者通过接收消息来改变自身。许多实用的软件系统都必须维护一两个非常关键的对象，它们通常具有非常复杂的状态转换关系，而且需要对来自外部的各种异步事件进行响应。例如，在 VoIP 电话系统中，电话类(Telephone)的实例必须能够响应来自对方的随机呼叫，来自用户的按键事件，以及来自网络的信令等。在处理这些消息时，类 Telephone 所要采取的行为完全依赖于它当前所处的状态，因而此时使用状态机就将是一个不错的选择。

有限状态机(Finite State Machine，FSM)模型包含 5 个元素，即输入符号、输出符号、状态集合、状态转移函数和输出函数，而扩展有限状态机(Extended Finite State Machine，EFSM)模型是在 FSM 模型基础上增加了动作和转移条件，以处理系统的数据流问题，而 FSM 模型

只能处理系统的控制流问题。所以,EFSM 模型包含 6 个元素,增加了一个初始状态,并将 FSM 模型中的“状态转换函数和输出函数”变为“变量集合和转移集合”,如图 3-12 所示。基于 FSM/ EFSM 模型,自动化编程和测试的研究和实践越来越多。

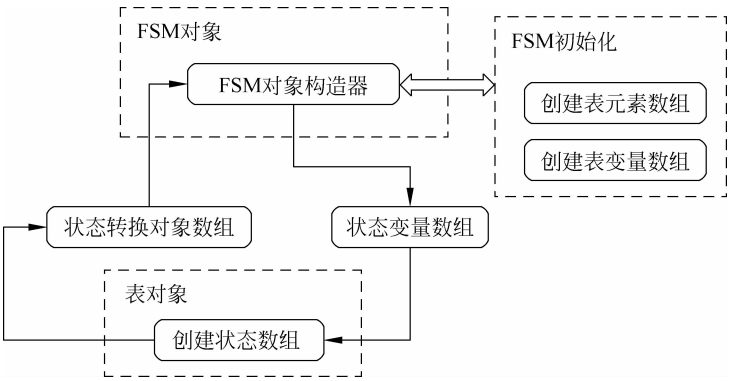

图 3-12 EFSM 模型示意图

对于 FSM 模型应用很多,最典型的一个例子就是电梯控制程序。电梯可以看作由两部分——电梯门和轿箱组成。实际上,电梯门有两种基本状态——开和闭,但如果更细致地分析,就可以增加两种状态——正在打开和正在关闭。因为在电梯正在关闭的过程中,电梯还是可以接收指令,转为“正在打开”状态。所以,电梯门的控制可以通过 FSM 来描述,相对简单,如图 3-13 所示。实际的控制系统必须统一控制,将电梯门和轿箱作为一个整体考虑,其 FSM 描述如图 3-14 所示。

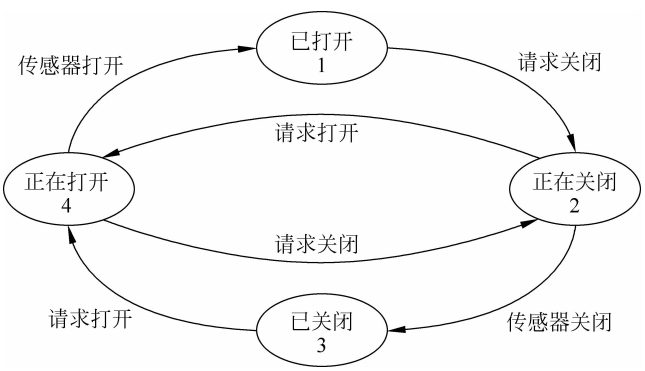

图 3-13 电梯门控制的 FSM 示意图

基于 EFSM 测试的输入应该包含两个部分:测试输入序列及其包含的变量值(输入数据)。手工选取这些测试数据的工作十分烦琐,一般需要采用自动选取的方法,如聚类方法、二叉树遍历算法和分段梯度最优下降算法等,从而极大地提高实际测试工作的效率。

为实用的软件系统编写状态机并不是一件轻松的事情,特别是当状态机本身比较复杂的时候尤其如此,需要投入大量的时间与精力才能描述状态机中的各种状态,所以不得不尝试开发一些工具来自动生成有限状态机的框架代码,例如,基于 Linux 的有限状态机建模工具 FSME(Finite State Machine Editor),如图 3-15 所示。FSME 能够让用户通过图形化的方式来对程序中所需要的状态机进行建模,并且还能够自动生成用 C++或者 Python 实现的状态机框架代码。

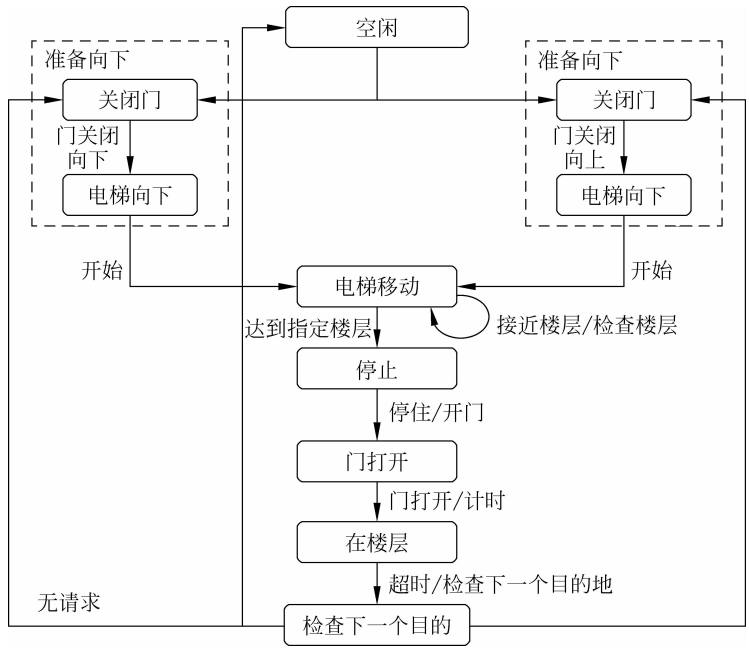

图 3-14 电梯控制系统的 FSM 示意图

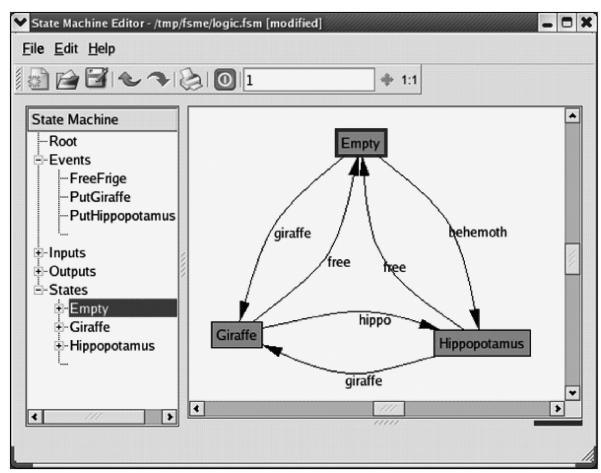

图 3-15 有限状态机建模工具 FSME 界面示意图

小结

本章介绍了各种测试方法，从基于直觉和经验的测试方法、基于输入域的测试方法、基于组合及其优化的方法，到基于逻辑覆盖的方法、基于缺陷模式的方法、基于模型的方法和形式化方法等。对测试方法可能有不同的划分。例如，之前人们习惯于把测试方法分为两类：白盒测试方法和黑盒测试方法，这样划分比较粗糙，也容易限制测试人员的思维。为了更好地展示测试思路，就需要找准测试的切入点，也就是明确如何找到被测试的系统或单元的突破口，从而更全面地操作被测试对象，对被测试对象施加影响，从而评估被测试对象的行为表现或输

出结果。要做到这点,主要依赖数据流和控制流等的分析。

(1) 数据流分析,包括输入输出空间的分析,如等价类划分、边界值分析;如果输入空间是由多个变量或多个参数等构成,就需要考虑组合问题,即判定表、Pairwise 方法、正交实验法。

(2) 控制流分析,就是对程序或软件的状态转换、运行路径进行分析,自然就有有限状态机、功能图、条件覆盖、分支覆盖(判定覆盖)和基本路径覆盖。

无论采用哪种方法,最终需要对测试覆盖率进行分析,以评估测试的充分性。这种覆盖率分析,也主要是从数据覆盖、运行路径是否被覆盖进行分析,从这个角度,也可以帮助我们更好地理解测试方法。

在进行数据流或控制流分析时,如果问题复杂,就需要借助建模的技术帮助实现,包括有限状态机、因果图、模糊测试等方法。决策表、功能图等也可以归为建模技术,实际上,一个方法可以归为不同的两个或三个类别。当上面这些方法都不适用时,或是作为上述方法的一种补充,就有了基于直觉和经验的测试方法、基于缺陷模式的方法。

这里介绍了大部分的测试方法,但测试方法不局限于这些,还有其他一些方法,例如,基于用户场景的测试方法、业务端到端的测试方法(基于业务流程路径的验证方法)、基于需求直接验证的方法等。不同的测试方法有各自的出发点,其侧重点不一样,有其特定的应用范围。例如,基于逻辑覆盖的方法主要用于单元测试或者系统业务流端到端的测试,而基于输入域的测试适合对数据进行测试,基于组合的方法可以应用于系统兼容性测试,模糊测试方法应用于容错性测试和安全性测试,形式化方法则用于高可靠性的关键软件系统的测试。

SWEBOK 3.0 作为软件工程专业的知识体系,成为软件工程教学的重要参照体系,有必要分析一下这里所介绍的方法是否覆盖了 SWEBOK 3.0 所要求的各种方法,如表 3-20 所示。

表 3-20 SWEBOK 3.0 测试方法

SWEBOK 3.0 测试方法分类	SWEBOK 对应的具体测试方法	本教材对应的分类及方法
基于直觉和经验的方法	Ad-hoc 测试方法、探索式测试	基于直觉和经验的方法,增加“错误猜测法”,但“探索式测试”不被认为是一种方法
基于输入域的方法 IDBT	等价类、边界值、两两组合(Pairwise)、随机测试	基于输入域的方法 基于组合及其优化的方法(决策表、应图、两两组合、正交实验法)
基于代码的方法 CBT	基于控制流的标准、基于数据流的标准、CBT 参考模型	基于逻辑覆盖的方法,如判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖、基本路径覆盖
基于故障的方法 FBT	故障模型、错误猜测法、变异测试	基于缺陷模式的方法,如常见缺陷模式、模糊测试方法
UBT	操作配置、用户观察启发	基于场景的方法
MBT	决策表、有限状态机、形式化验证、TTCN3、工作流模型	基于模型的方法 形式化方法
TBNA	OOS、Web、Real-time、SOA、Embedded、Safe-critical	应用领域,不能算是测试方法,而是如何结合相应的软件技术完成其应用领域的测试

思考题

1. 在逻辑覆盖方法中，语句覆盖、判定覆盖、条件覆盖和基本路径覆盖，哪一种覆盖率高？为什么？

2. 针对“邮件地址”输入域进行验证，通过等价类划分法设计相应的测试用例，包括尽可能多的无效等价类。

3. 综合运用边界值方法和等价类划分法设计相应的测试用例：输入三个整数作为边，分别满足一般三角形、等腰三角形和等边三角形。

4. 根据如图 3-16 所示程序流程图，分别用最少的测试用例完成语句覆盖、判定覆盖、条件覆盖和路径覆盖的测试设计。

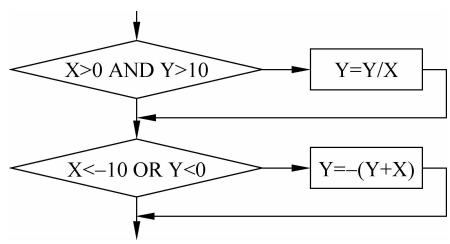

图 3-16 思考题 4 程序流程图

5. 针对下列可能存在的程序结构设计测试用例。

(1) 程序要求：10 个铅球中有一个假球(比其他铅球的重量要轻)，用天平三次秤出假球。

(2) 程序设计思路：第一次使用天平分别秤 5 个球，判断轻的一边有假球；拿出轻的 5 个球，拿出其中 4 个秤，两边分别放两个球；如果两边同重，则剩下的球为假球；若两边不同重，拿出轻的两个球秤第三次，轻的为假球。

6. 结合边界值分析法和等价类划分法，针对不同月薪需要缴纳不同的个人所得税计算程序，来设计充分的测试用例。设个人所得税的起征点 3500 元，税率见表 3-21。

表 3-21 税率计算表

应纳税所得额(减去起征点 3500 元后的结果)	税率/%
不超过 500 元	5
超过 500～2000 元	10
超过 2000～5000 元	15
超过 5000～20 000 元	20
超过 20 000～40 000 元	25
超过 40 000～60 000 元	30
超过 60 000～80 000 元	35
超过 80 000～100 000 元	40
超过 100 000 元	45

7. 年、月、日分别由 Y、M 和 D 来存储相应的值，现在要测试 NextData(Y,M,D)函数，用判定表方法来设计相应的测试用例。

8. 针对下列因素，使用 PICT 工具完成 Pairwise 的组合测试，如果存在约束条件，需要添

加后进行计算。

(1) 驾驶记录：过去5年内没有违规，过去3年内没有违规，过去3年内违规小于3次，过去3年内违规3次或3次以上，过去1年内违规3次或3次以上。

(2) 汽车型号：一般国产车，高档国产车(≥20万)，进口车，高档进口车(≥100万)。

(3) 使用汽车的方式：出租车，商务车，私家车。

(4) 所住的地区：城市中心地带，市区，郊区，农村。

(5) 受保的项目：全保，自由组合，最基本保险。

(6) 司机的驾龄：≤1年，≤3年，≤5年，≤10年，>10年。

(7) 保险方式：首次参保，第二次参保，连续受保(≥3次)。

9. 通过扩展有限状态机来描述表示堆栈算法，然后转化为状态树，然后设计测试用例覆盖独立的树根到树叶的路径。