第九章　软 件 测 试

编码完成后，就要对源程序进行测试。软件测试是一项“劳民伤财”的工作。统计表明，开发较大规模的软件，有 40%以上的精力是耗费在测试上的。即使富有经验的程序员，也难免在编码中发生错误；何况，有些错误在设计甚至分析阶段早已埋下了祸根。无论是早期潜伏下来的错误或编码中新引入的错误，若不及时排除，轻则降低软件的可靠性，重者可导致整个系统的失败。为了防患于未然，无论怎样强调测试的重要性，都不过分。

本章将首先介绍有关软件测试的基本概念，然后讨论常用的测试技术，以及不同层次的软件测试方法。主要从实用而不是从理论的角度来论述问题。一些与测试有密切关系的理论，如可靠性理论，程序正确性证明等，将在第十三章作简单介绍。

9.1 基 本 概 念

关于测试的目的，曾有过种种似是而非的说法。众多的术语和测试技术，也常令初学者眼花瞭乱。为此，本节将先就测试的目的、技术和步骤作一快速的巡视，以帮助读者在深入学习以前，对测试的概貌建立起一个初步的轮廓。详细的内容，从下节起再分节介绍。

1. 测试的目的与地位

在 G.J.Myers 的经典著作《软件测试技巧》一书中，给出了测试的定义。他认为，“程序测试是为了发现错误而执行程序的过程。”

E.W.Dijkstra 则说，“程序测试能证明错误的存在，但不能证明错误不存在。”

在这里，Myers 等人明确指出：测试的目的是发现程序中的错误；是为了证明程序有错，而不是证明程序无错。在软件开发过程中，分析、设计与编码等工作都是建设性的，唯独测试带有“破坏性”，因为它抱着“吹毛求疵”的目的，明确宣布要在程序中“找岔子”。Myers 还认为，这种吹毛求疵的态度至关重要。如果你为了证明程序无错而进行测试，错误就可能在你眼皮底下漏过；反之，只要你抱着证明程序有错的目的去测试，就会把大部分存在的错误都找出来。

根据 Myers 的定义，测试又是一个“（在计算机上）执行程序的过程”。分析和设计阶段都要对文档复审，源程序完成后，也要进行代码复审（code review）。这些复审对减少软件错误有重要的作用（参阅图 4.15 与图 4.16），但都不能代替在计算机上进行的测试。R.S.Pressman 认为，测试可视为分析、设计和编码 3 个阶段的“最终复审（ultimate review）”，可见其在软件质量保证中的重要地位。

另一个与测试密切相关的活动叫**纠错**（debugging）。测试的目的是发现错误，纠错则是为了确定错误的性质，并且加以纠正。在测试中发现源代码有错误，就进行纠错，错误改正后又继续测试。这种边测试边纠正的活动，常常借助于一种称为调试程序（debugging routine）的

专用工具，所以也有人把纠错称为调试。图 9.1 显示了测试和纠错时信息的流程。

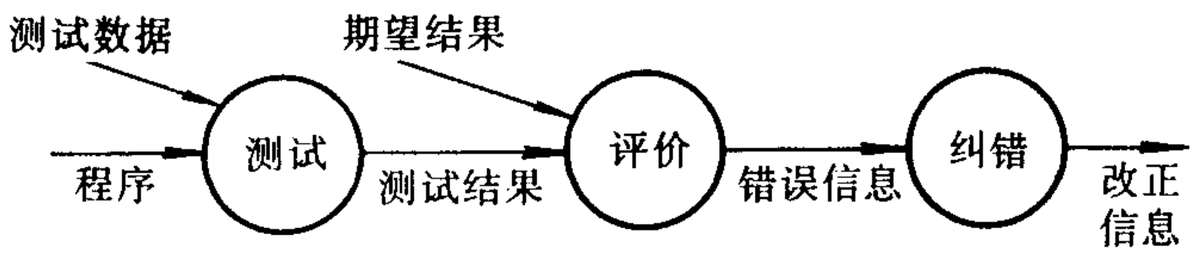


图 9.1　测试和纠错的信息流程

2. 测试的方法与技术

广义地说，程序测试不仅指在计算机上进行的测试（机器测试），还应包括用人工方式进行的代码复审（人工测试）。图 9.2 列出了这两类测试所采用的方法与技术。

- 程序测试
  - 人工测试（代码复审）
    - 代码会审（code inspection）
    - 走查（或排练，walk－through）
    - 办公桌检查（desk checking）
  - 机器测试（动态测试）
    - 黑盒测试技术（black－box testing）
    - 白盒测试技术（white－box testing）

图9.2　程序测试方法与技术

现将几个有关的问题简述如下：

(1) 机器测试与人工测试

程序通过编译后，要先经代码复审，然后再进行机器测试。机器测试是在设定的测试数据(test data) 上执行被测程序的过程，故又称为动态测试（dynamic testing)。代码复审采用人工方式进行，目的在于检查程序的静态结构，找出编译不能发现的错误。经验表明，组织良好的代码复审，可以发现程序中 30%到 70%的编码和逻辑设计错误，从而加快动态测试的进程，提高整个测试的效率。

根据 Myers 的研究，对于某些类型的错误，机器测试比人工测试有效；但对另一些类型的错误，人工寻找的效率往往比机器测试更高。而且，机器测试只能发现错误的症状，人工测试一旦发现了错误，同时就确定了错误的位置与性质。由此可见，人工测试并不是可有可无的、或为了节约计算机机时而采取的权宜措施。它是机器测试的准备，也是程序测试中不可缺少的环节。

(2) 白盒测试与黑盒测试

动态测试是一个包括①设计“测试用例”；②执行被测程序；和③分析执行结果并发现错误的过程。合理设计测试用例，是有效地完成测试的关键。按照在设计测试用例时是否涉及程序的内部结构，可以把动态测试区分为白盒测试和黑盒测试两类技术。

白盒测试时，测试者对被测程序的内部结构是清楚的。他从程序的逻辑结构入手，按照一定的原则（例如每条语句至少执行一次，或每条路径至少执行一次，等等）来设计测试用例，设定测试数据。由于被测程序的结构对测试者是透明的，所以有些书本又称这类测试为玻璃盒测试（glass-box testing）或结构测试（structural testing)。

黑盒测试的情况正好相反。此时，测试者把被测程序看成一个黑盒，完全用不着关心程序的内部结构。设计测试用例时，仅以程序的外部功能为根据。一方面检查程序能否完成一切应

做的事情，另一方面要考察它能否拒绝一切不应该做的事情。由于黑盒测试着重于检查程序的功能，所以也称为功能测试（functional testing）。

(3) 穷举测试与选择测试

能不能够通过动态测试，发现程序中的所有错误呢？人们自然地想到，应该让被测程序在一切可能的输入情况下执行一遍。这就是所谓的“穷举测试”（exhaustive testing）。

假定我们手中有一个 Pascal 的编译程序，要对它进行穷举测试。我们一方面要编出所有能够想象出来的合法的 Pascal 程序让它编译，另一方面又要编出一切不合法的 Pascal 程序，考察它能否指出程序的非法性质。显而易见，这两类（合法与不合法）程序的数量都是无穷量。可见用黑盒法来作穷举测试，实际上是不可能的。

白盒法的情况怎样呢？可仍用例子来说明。图 9.3 是一个小程序的程序图，含有 4 个选择分支和一个至多重复 20 次的循环。如果对这个程序进行穷举测试，就要把从 A 到 B 的各种路径全都测试一遍。不难看出，从 A 至 B 的不同路径共有（$5^1+5^2+\cdots+5^{19}+5^{20}$）$\approx 10^{14}$ 即 100 万亿条。进行如此规模的计算机测试，显然是不可想象的。

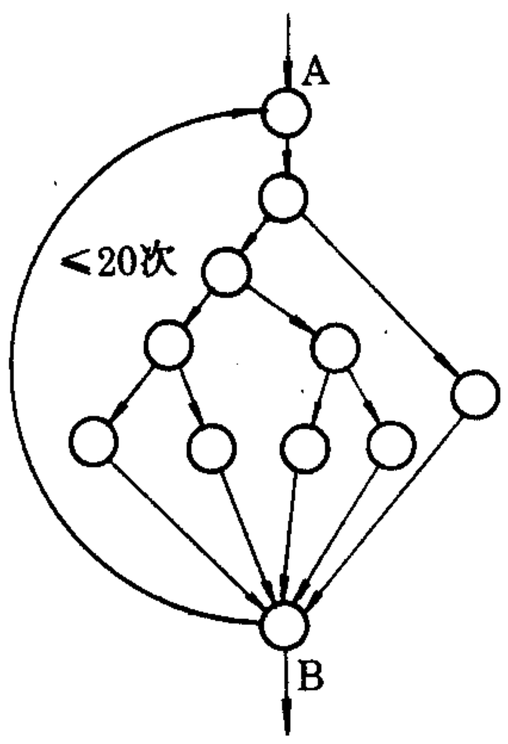


图 9.3 一个小程序的程序图

由此可见，穷举测试是不现实的，这就是如上所说的测试不能保证程序无错的原因。在实际测试中，无论白盒测试或黑盒测试，都只能选择一些有代表性的、典型的测试用例，进行有限的测试。通常称这种测试为选择测试（selective testing）。理想的选择测试，应该以最少的测试用例，发现最多的程序错误。这就是程序测试的经济学原则。

3. 软件测试的步骤

按照软件工程的观点，软件测试依次由以下 4 个层次的测试组成：

单元测试（unit testing）

综合测试（integration testing）

确认测试（validation testing）

系统测试（system testing）

其中单元测试在编码阶段完成，综合测试和确认测试均放在测试阶段完成。系统测试是指整个计算机系统（包括软件与硬件）的测试，可与系统的安装和验收结合进行。

图 9.4 显示了以上 4 级测试的信息流程。现有下列几点需要说明。

(1) 单元测试应该以模块为单位，并包括代码复审、动态测试等内容；

(2) 综合测试与确认测试应分别以软件的设计信息与需求信息为依据，确保在测试中分别达到上述信息所规定的要求；

(3) 确定测试用例时，单元测试可综合运用白盒与黑盒两类测试技术，其它测试主要采用黑盒测试技术；

(4) 各级测试均须在事先制订测试计划，事后写出测试报告；

(5) 根据心理学的观点，程序员对自己程序的错误容易视而不见，也不会千方百计寻找自己程序中的漏洞。因此，测试宜由独立的测试小组进行，避免由开发小组测试自己的程序。

测试一个大型程序所要求的创造力，有时会超过设计该程序所要求的创造力。所以测试小组的成员，应挑选有经验的优秀程序员来担任。

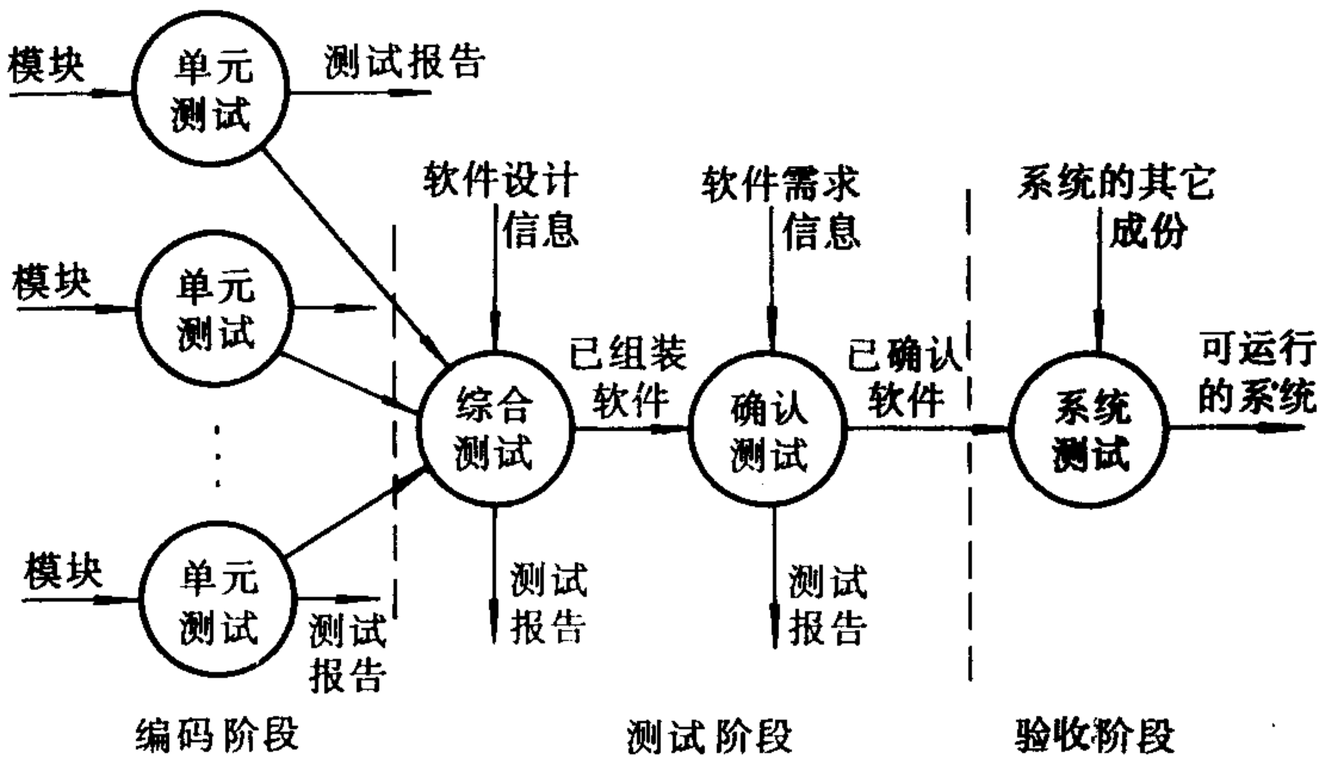


图 9.4　软件测试和步骤

9.2　代码的复审

代码复审在程序通过编译以后，动态测试开始之前进行。有些初学者以为代码通过了编译就问题不大，这是极大的误解。一些典型的调查表明，在大型软件存在的错误中，仅有小部分能在编译时间发现。图 9.2 中曾列出代码复审的 3 种方式，现分述如下。

1. 代码会审

代码会审以小组会的方式进行。会审小组一般由 3～4 人组成，包括组长 1 人，程序作者 1 人，其他程序员（或测试员）1～2 人。会前要先将程序清单分发给与会者，让他们熟悉要审的材料。开会时，程序作者逐句朗读和讲解程序，其他人则集中精力，捕捉程序在结构、功能、与编码风格等方面可能存在的问题。为了帮助大家发现问题，还可把复审的要点编成“错误检验表”(error checklist)，供与会者参考。表 9.1 列出了此类检验表的一般内容。

表 9.1　错误检验表的一般内容

数据引用错误	例如使用未赋过值的变量，或变量赋值后从不使用等
数据说明错误	例如对变量未作说明，变量类型与初始化的值不符等
数据计算错误	例如混合类型运算，用零作除数等
数据比较错误	例如比较运算符和逻辑运算符使用不当，企图在不同类型的变量间作比较等
控制流程错误	例如多作或少作了一次循环，在循环体中对循环变量重新定义等
接口错误	例如实参和形参的类型、顺序或数量不符，全程变量在各个模块中的定义不一致等
输入输出错误	例如忘记打开或关闭文件，I/O 出错处理不正确等

这种会审方式的主要优点是：①同行之间互相启发，更易于发现错误；②有利于互相学习交流经验。显然，只有当与会者都抱着正确的态度，会审在和谐合作的气氛中进行时，才能充分体现以上的优点。

为了确保会审能取得成效，组长应引导大家把精力集中于发现错误，把改正错误的工作放到会后去作。复审速度以每小时约 150 行程序为宜，每次会审的持续时间不要超过 2 小时，使与会者能始终保持充沛的精力。如果程序较长，可分成多次会审。

会后，应把查出的错误清单交给程序的作者，由作者改正后交回组长。如果错误很多，或有的错误要作重大的改正，应在改正之后安排重新会审。

2. 走查

与会审相似，走查也是以小组会的方式进行的。每一小组 3～5 人，每次持续 1～2 个小时。被审程序也要提前发给参加者，并要求他们在会前熟悉这些材料。

与会审的主要差别是，走查要求与会者扮演“计算机”的角色，用人工的方法来运行被审程序。因此，在会前至少要指定一人提出“测试用例”，开会时把这些测试数据“输入”被测程序，并在纸上跟踪监视程序的执行情况。让人代替机器沿着程序的逻辑“走”一遍，并从中“查”出错误，这就是“走查”这一名称的由来。

由于人工运行远比机器缓慢，走查时仅能使用少量简单的测试用例。实际上，与会者并非仅靠少量走查来达到对程序的复审，而是以走查为手段，随着走查的进程不断向程序作者提出有关的询问，从中发现程序的错误。经验表明，通过这些询问发现的错误往往比走查中直接暴露的错误还多。

走查会应有的气氛和对走查结果的处理与会审相似，这里就不多说了。

3. 办公桌检查

办公桌检查可以看成由一个人参加的代码复审。其内容可以是按照错误检验表来检查被审的程序，也可以仿照走查对程序进行人工运行。早期因程序规模较小，常采用这种方法。由于复审者和程序作者是同一个人，所以效果常不及前两种方式明显。除了一些规模很小的软件，现在已很少使用这种复审方式。

9.3 测试用例的设计

在此以前，已经多次提到过“测试用例”一词。一个测试用例，其实就是以发现错误为目的而精心设计的一组测试数据。测试一个程序，需要数量足够的一组测试用例。借用数据字典的表示方法，可以写成

$$\text{测试用例}=\{\text{输入数据}+\text{期望结果}\}$$

其中{ }表示重复。这个式子还表明，每一完整的测试用例不仅含有被测程序的输入数据，而且还包括用这组数据执行被测程序后预期的输出结果。每次测试，都要把实测的结果与期望结果作比较，若不相符，就表明程序可能存在错误。

设计测试用例是开始程序测试的第一步，也是有效地完成测试工作的关键。一个良好的测试用例，应该对发现错误具有较高的概率。表 9.2 列出了一些常用的设计测试用例的方法。下

面将分别作简要的说明。

表 9.2　测试用例的设计方法

黑 盒 方 法	白　　盒　　方　　法	
等价分类法	逻辑覆盖法	路径测试法
边界值分析法	语句覆盖	结点覆盖
错误猜测法	判定覆盖	边覆盖
因果图法*	条件覆盖	路径覆盖
	条件组合覆盖	

注:　有 * 者初学者可以不学

1. 黑盒测试方法

前已指出，黑盒测试以程序的功能作为测试依据。不言而喻，这里的功能将随被测程序的范围而异。所以无论选用以下的何种方法，第一步都要认真阅读被测代码（整个程序或某一模块）的功能说明。

(1) 等价分类法 (equivalence partitioning)

这种方法把被测程序的输入域划分为若干等价类，把漫无边际的随机测试变成有针对性的等价类测试。它的出发点是，每一个测试用例都代表了一类与它等价的其它例子。如果用这个例子未能发现程序的错误，则与它等价的其它例子一般也不会发现程序的错误。这样，测试人员就有可能使用少量“有代表性”的测试用例，来代替大量相类似的测试，从而大大减少总的测试次数。

设计等价类的测试用例一般分为两步进行，即①划分等价类并给出定义；②选择测试用例。选择的原则是：有效等价类的测试用例尽量公用，以期进一步减少测试次数；无效等价类必须每类一例，以防漏掉本来可能发现的错误。下面举一个简单的例子。

例 9.1　某城市的电话号码由 3 部分组成。这 3 个部分的名称与内容分别是

地区码:　空白或 3 位数字;

前　缀:　非′0′或′1′开头的 3 位数字;

后　缀:　4 位数字。

假定被测程序能接受一切符合上述规定的电话号码，拒绝所有不符合规定的号码，就可用等价分类法来设计它的测试用例。

第一步:　划分等价类。表 9.3 列出了划分的结果，包括 4 个有效等价类，11 个无效等价类。在每一等价类之后均加有编号，以便识别。

表 9.3　电话号码程序的等价类划分

输 入 条 件	有 效 等 价 类	无 效 等 价 类
地　区　码	空白①，3 位数字②	有非数字字符⑤，少于 3 位数字⑥，多于 3 位数字⑦，
前　　缀	从 200 到 999 之间的 3 位数字③，	有非数字字符⑧，起始位为′0′⑨，起始位为′1′⑩，少于 3 位数字⑪，多于 3 位数字⑫，
后　　缀	4 位数字④，	有非数字字符⑬，少于 4 位数字⑭，多于 4 位数字⑮

第二步：　确定测试用例。表中有 4 个有效等价类，可以公用以下两个测试用例：

测试数据	测试范围	期望结果
(　　) 276–2345	等价类①、③、④	有效
(635) 805–9321	等价类②、③、④	有效

对 11 个无效等价类，应选择 11 个测试用例。例如前 3 个无效等价类可能使用下列的 3 个测试用例：

测试数据	测试范围	期望结果
(20A)　123–4567	无效等价类⑤	无效
(33　)　234–5678	无效等价类⑥	无效
(7777)　345–6789	无效等价类⑦	无效

后 8 个无效等价类的测试用例留给读者做练习。这样，本例的 15 个等价类将至少需要 13 个测试用例。　　〔例 9.1 完〕

练习题

1. 为例 9.1 的无效等价类⑧～⑮各设计一个测试用例。
2. 为什么无效等价类必须每类一例？

划分等价类时，必须注意给出确切的定义。例如上例中的无效等价类⑤，初看起来似应定义为“含有既非空白又非数字的字符”，才能与等价类①、②同时对应。但如果这样定义，岂不是承认由空格和数字混合组成的地区码也有效了？这就与无效等价类⑥的定义直接矛盾。又如⑨、⑩两个等价类，也不应合并为“起始位为′0′或′1′”，因为如果合并，测试用例就可减为 1 个（以′0′起始或者以′1′起始），对程序的测试也就不够完整了。

(2) 边界值分析法 (boundary value analysis)

在等价分类法中，代表一个类的测试数据可以在这个类的允许值范围内任意选择。但如果把测试值选在等价类的边界上，往往有更好的效果，这就是边界值分析法的主要思想。

举例说，税法规定公民的个人收入调节税从超过 400 元开始征收。如果用一个程序来计算税款，则“收入≤400”可作为判别条件，满足条件者免税，否则要对超出 400 元的部分征税。选择测试用例时，固然可以用 300、500 两个测试数据分别代表免税和征税两个等价类，但如果程序中将判别条件误写为“收入＜400”，这两个数据都不能发现错误。假如将边界值 400 用作测试数据，这个错误就无法藏身了。在大多数情况下，边界值及其邻近的数据都属于敏感区，容易暴露程序的错误。

边界值分析也适用于考察程序的输出值边界。例如，如果某工资程序规定，当职工的扣款金额超过当月工资的一半时，其超出部分应留待次月扣除。如果按边界值分析法选择此时的测试用例，就应有意识地让扣款达到和超过工资额的半数，分别观察被测程序计算的当月实发工资有何变化。

(3) 错误猜测法 (error guessing)

所谓猜错，就是猜测被测程序中哪些地方容易出错，并据此设计测试用例。如果说等价法和边界值法均有线索可寻，则猜错法将更多地依赖于测试人员的直觉与经验。所以在通常情况

下，这种方法仅用作辅助手段，即首先用其它方法设计测试用例，再用猜错法补充一些例子。

例如，当对一个排序程序进行测试时，可先用边界值分析法设计以下的测试用例:

①输入表为空表;

②输入表中仅有一个数据;

③输入表为满表。

然后再用猜错法补充以下的用例:

④输入表已经排好了序;

⑤输入表的排序恰与所要求的顺序相反（例如程序功能为由小到大排序，输入表为由大到小排序);

⑥输入表中的所有数据全都相同;

等等。

*(4) 因果图法 (cause–effect graphing)

因果图法是借助图形来设计测试用例的一种系统方法，特别适用于被测程序具有多种输入条件，程序的输出又依赖于输入条件的各种组合的情况。

因果图是一种简化了的逻辑图，能直观地表明程序输入条件（原因）和输出动作（结果）之间的相互关系。为便于叙述，下面结合一个实例来说明使用因果图法产生测试用例的步骤。

例 9.2 某电力公司有 A、B、C、D 共 4 类收费标准，并规定，居民用电每月 100 度以下按 A 类收费，100 度及以上按 B 类收费。动力用电以每月 1 万度为分界。非高峰用电不足 1 万度按 B 类收费，达到 1 万度按 C 类收费。高峰用电万度以下为 C 类，达到或超过万度为 D 类。试用因果图法为该公司的电费计算程序设计一组测试用例。

以下列出产生设计用例的 4 点步骤:

(1) 列出程序的输入条件（因）和输出动作（果），如图 9.5 所示。

输入条件	输出动作
1. 居民用电	A. A 类计费
2. 动力用电	B. B 类计费
3. ＜100 度／月	C. C 类计费
4. ＜10,000 度／月	D. D 类计费
5. 高峰用电	

图 9.5 输入输出表

(2) 用因果图表明输入和输出之间的逻辑关系，见图 9.6。

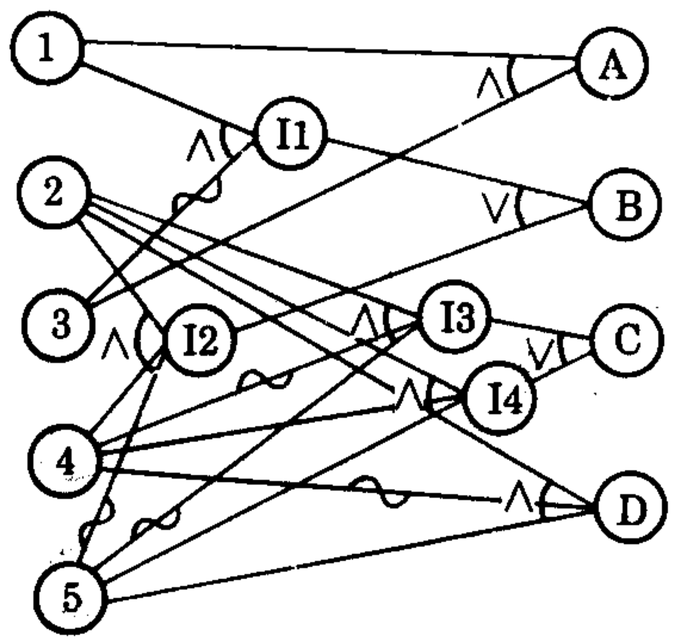
(3) 把因果图转换为判定表。这一步的具体作法是:

——选择一个输出动作，使处于“1”状态;

——在因果图上从后向前回溯，找出使此动作为“1”的各种输入条件组合;

——将每一个输入条件组合填入判定表中的一列，同时填入在此组合情况下各个输出动作的状态;

——选择下一个输出动作，重复以上 3 步，直至最后一个输出动作做完为止。



说　明

A：　逻辑与

V：　逻辑或

～：　逻辑非

1，2，3，4，5：因结点

A，B，C，D：果结点

I 1，I 2，I 3，I 4：中间结点

图 9.6　因果图

图 9.7 示出了本例得出的判定表。表中的因结点就是输入条件，果结点就是输出动作。

规　则		1	2	3	4	5	6
因结点	1	1	1	0	0	0	0
	2	0	0	1	1	1	1
	3	1	0				
	4			1	0	1	0
	5			0	0	1	1
中间结点	I 1		1				
	I 2			1			
	I 3				1		
	I 4					1	
果结点	A	1	0	0	0	0	0
	B	0	1	1	0	0	0
	C	0	0	0	1	1	0
	D	0	0	0	0	0	1

图 9.7　从因果图导出的判定表

(4) 为判定表中的每一列（或规则）设计一个测试用例，如图 9.8。

输入数据	预期结果	输入数据	预期结果
①居民电，90 度／月	A	④动力电，非高峰，1.2 万度／月	C
②居民电，110 度／月	B	⑤动力电，高峰，0.9 万度／月	C
③动力电，非高峰，8000 度／月	B	⑥动力电，高峰，1.1 万度／月	D

图 9.8　一组可能的测试用例

面向商业的环境将会增加产量。

现在有许多这种类型的环境，包括ADW、Foundation和Bachman Product Set。根据现在市场中面向商业的CASE环境的比例，可以想象出将来会有更高档的这种类型的环境出现。

13.13　公用工具的基础结构

欧洲信息技术研究战略计划(ESPRIT)开发了一个支持CASE工具的基础结构。可移植通用工具环境(PCTE, Portable Common Tool Environment)[Thomas,1989]并不是一个环境，尽管它的名字叫环境。相反，它是一个为CASE工具提供服务的基础结构，就像UNIX为用户产品提供操作系统服务一样(PCTE 中的“通用”的意思是指“公用的”或“没有版权保护的”)。

PCTE开始得到广泛的认可。例如，在1990年ECMA(European Computer Manufacturers Associatioin)接受PCTE作为ECMA标准149。后来，PCTE的C和Ada接口被包含进ECMA标准中。另外，PCTE也服从国际标准。Emeraude和IBM等组织实现了PCTE。

希望在将来，将有更多CASE工具遵从PCTE标准，PCTE将在各种计算机上被实现。一个支持PCTE标准的工具可以在任何支持PCTE的机器上运行。这样将可以使各种CASE工具得到更广泛的使用。这样也会导致更好的软件开发过程和更加高效的软件产品的开发。

在面向对象的技术方面，致力于为面向对象系统开发公用环境的对象管理组(OMG, object Management Group)成立了。特别是，公用对象请求代理程序体系结构(CORBA, Common Object Request Broker Architecture)在分布式环境中支持不同机器上的软件应用程序的移植。这样，如果一个分布式应用软件根据一个特定的网络协议在一个给定的操作系统上运行，它也可以在其他支持CORBA的任何机器上运行。特别是，CORBA应该为面向对象的CASE工具提供一个共同的环境。HP公司ORB Plus是目前市场上已有供应的CORBA实现环境中的之一个。

13.14　各类环境的比较

就像没有一种编程语言是最好的那样，没有一种环境是对任何产品和开发商都适用的。在本章中讨论的各类环境都有自己的优点和缺点，选择一个差的环境甚至比不使用任何环境更糟糕。例如，如13.11.1节所述，基于技术的环境本质上是实现人工过程自动化。如果一个开发商选择了一个基于技术的环境，而该环境并不适合于该软件的开发，那么，使用这个CASE环境只会得不偿失。

一个更糟糕的情况是，开发商在选择环境时忽视了3.7节的建议，即只有在开发组织达到成熟度的CMM 3级水平时，才能使用CASE环境。当然，任何一个开发组织都应该使用CASE工具，通常，使用一个平台也是无害的。然而，每一个环境都会给使用它的软件组织强加一个自动的软件过程。如果该组织的软件过程处于成熟度3或更高，则使用一个环境将会使软件过程实现自动化，对软件生产的各个方面都带来极大的帮助。但是，如果开发组织仅仅在水平1或2，则没有这样一个过程存在。对于一个不存在的过程进行自动化，也就是引入一个CASE环境，只会带来混乱。

13.15　实现和集成阶段的度量

在12.16.2节中讨论了许多种实现和集成阶段的复杂性度量，包括代码行、McCabe的环路复杂度以及软件科学的度量。

从测试的观点看，相关的度量包括测试用例的总数和测试失误的测试用例的数目。在代码检测时还必须维护常见故障的统计数字。错误的总数是很重要的，因为，如果一个模块或对象

中的错误大于一个极值，那么这个模块或对象就要重新设计和编码(12.21.1节)。同时，细节统计还必须考虑检测到的错误类型。典型的错误包括对设计的错误理解、缺少初始化、变量不一致等。将这些错误数据加入到检测列表中，以便在以后的产品测试中使用。一套强大的实现和集成阶段的度量与基于统计的测试相联系。为了弄明白这些度量是如何使用的，我们考虑其中的一个零错误技术。这个技术用来决定一件产品的测试要经历多长时间。它的基本思想是，如果一个产品被检测到没有错误发生的时间越长，这个产品存在错误的可能性就越小。这个技术指明该产品在无一个错误的情况下还必须测试多长时间。可以合理地推断，随着测试过程的进行，发生错误的机率就会呈指数下降。在没有一个错误发生的情况下的测试小时数用以下公式计算[Brettschneider,1989]：

$$\frac{\ln\left(\frac{f_{target}}{0.5 + f_{target}}\right)}{\ln\left(\frac{0.5 + f_{target}}{f_{total} + f_{target}}\right)} \times t_h$$

其中f_{target}指的是期望的错误数目；f_{total}指的是到目前为止检测到的错误数目；th指的是，到上次错误为止测试所用的总共时间；ln是为e为底的对数。

例如，假设一个软件有50 000行代码，合同规定，在所交付的产品中每1 000行代码不超过0.02个错误，这意味着交付产品中允许有一个错误，所以$f_{target} = 1$。假设软件迄今为止已测试了400小时。在这段时间里检测到20个错误，而且从上次错误到现在产品已运行了50小时，这就意味着$f_{total} = 20$，$t_h = 400\text{-}50 = 350$小时，根据公式：

$$\frac{\ln\left(\frac{1}{0.5 + 1}\right)}{\ln\left(\frac{0.5 + 1}{20 + 1}\right)} \times 350 = 54 \text{ 小时}$$

这个软件已经无故障地运行了50个小时，所以还需要继续测试4个小时即可。当然，如果在这最后4小时有错误发生，需要再根据公式重新进行计算。

13.16 MSG实例研究：实现和集成阶段

MSG实例研究完整的Java实现在附录H中给出。为了帮助读者阅读，首先是按字母顺序列出所有的类和方法。

可以认为实现就是直接将详细设计(附录F)转化为代码，即设计人员把精心制作的设计交给编程组，而后者只需用Java加以实现。程序员在代码中加入一些注释，给维护人员带来方便。下一章将讨论维护阶段的困难，以及如何简化维护阶段的任务。

本章回顾

实现和集成两个阶段必须并行处理(13.1节)。自顶向下、自底而上和三明治式实现和集成方法在13.11节到13.13节介绍并做了比较。面向对象产品的实现和集成在13.14节做了讨论。13.2节讨论了实现和集成阶段不同类型的测试，包括13.4节的产品测试和13.5节的验收测试。13.3节用户图形界面的集成测试是一个重要问题。在13.6节讨论实现集成阶段的CASE工具，针对整个软件过程的CASE工具在13.7节讨论了。基于编程语言、面向结构的和工具集环境分别在13.8、13.9、13.10节讨论。集成CASE工具的问题在13.11节讨论。面向商业应用的环境在13.12节讨论，13.13节讨论了公用工具基础结构。然后在13.14节讨论了各类环境的比较，