第12章 软件测试

本章内容：

12.1 软件测试概述

12.2 测试方法

12.3 测试用例的设计

12.4 测试过程

12.5 调试

第12章 软件测试

软件测试是软件质量保证的关键阶段，是对软件设计和编码的最终检查。在软件开发的每一个阶段，人们使用了许多保证软件质量的方法分析、设计、实现软件，包括每阶段的复查。但是由于软件的特殊性，在工作中还是会存在错误。由于软件产品本身无形态，它是复杂的、知识高度密集的逻辑产品，没有一种软件方法可以保证在软件的设计和实现 过程中没有错误。在当前的软件开发中会将30%~40%的项目精力花在测试上。

12.1 软件测试概述

软件测试就是在软件投入运行前，对软件的需求分析、设计、实现编码进行最终审查。表面上看，在软件工程的其他阶段，都是建设性的，而软件测试是摧毁性的。但是，软件测试的最终目的是建立一个可靠性高的软件系统的一部分。它的定义为：是为了发现错误而执行程序的过程。

12.1.1 软件测试的目的

统计资料表明，测试的工作量约占整个项目开发工作量的40%左右，对于关系到人的生命安全的软件（如飞机飞行控制系统和核反应堆控制等），测试的工作量往往是其他阶段的3～5倍。那么，为什么要花这么多代价进行测试？其目的何在？软件测试要求认定刚开发的软件是错误的，它的目的是找出错误所在，而不是“说明程序能正确地执行它应有的功能”，也不是“表明程序没有错误”。如果是这样，那就会无意识地选择一些不易暴露错误的例子。

G.J.Myers在他的软件测试著作中对软件测试的目的提出了以下观点：

（1）软件测试是为了发现错误而执行程序的过程。

（2）一个好的测试用例能够发现至今尚未发现的错误。

（3）一个成功的测试是发现了至今尚未发现的错误的测试。

因此，测试阶段的基本任务是根据软件开发各阶段的文档资料和程序的内部结构，精心设计一组测试用例，它们能够系统地揭示不同类型的错误，并且耗费的时间和工作量最小。但是，已经找出错误的测试只能够说明已经发现的错误，但不能证明程序已经无错误。

12.1.2 软件测试的原则

在设计有效的测试用例之前，首先应该注意一些指导原则：

（1）测试用例由输入数据和预期的输出数据两部分组成。需要将程序运行后的结果和预期的输出相比较来测试程序。

（2）在输入数据的选择上，不仅要选择合理的输入数据，还要选择不合理的输入数据。这样可提高程序运行的可靠性。程序应该对不合理的输入数据给出相应提示。

（3）用穷举测试是不可能的。可以通过设计测试用例，充分覆盖所有的条件。

（4）应该在真正的测试工作开始之前很长时间内，就根据软件的需求和设计来制定测试计划，在测试工作开始后，要严格执行，排除随意性。

（5）长期保留测试用例。设计测试用例是一件耗费很大的工作，必须作为文档保存。因为测试不是一次完成的，在测试出错误并修改后，需要继续测试。同时，在以后的维护阶段仍然需要测试。

（6）对发现错误较多的程序段，应进行更深入的测试。Pareto原则表明，测试发现的错误中的80%是集中在20%的模块中。因为发现错误多的程序段，表明其质量较差，可能隐藏了更多的错误。同时在修改错误过程中又容易引入新的错误。

（7）为了达到最佳测试效果，应该有第三方来构造测试用例。避免程序员测试自己的程序。因为好的测试要求承认程序是有错误的，测试目的是发现错误，因此，程序员的心理状态是测试自己程序的障碍。另外，因为对需求说明的理解而引入错误自身则更难发现。

12.2 测试方法

软件测试方法一般分为动态测试方法与静态测试方法。动态测试方法中又根据测试用例的设计方法不同，分为黑盒测试与白盒测试两类。

12.2.1 静态测试

静态测试是采用人工检测和计算机辅助静态分析的手段对程序进行检测，方法如下：

（1）人工测试：是指不依靠计算机运行程序，而靠人工审查程序或评审软件。人工审查程序的重点是对编码质量进行检查，而软件审查除了审查编码还要对各阶段的软件产品（各种文档）进行复查。人工检测可以发现计算机不易发现的错误，特别是软件总体设计和详细设计阶段的错误。据统计，能有效地发现30%~70%的逻辑设计和编码错误，可以减少系统测试的总工作量。

（2）计算机辅助静态分析：指利用静态分析软件工具对被测试程序进行特性分析，从程序中提取一些信息，主要检查用错的局部变量和全程变量、不匹配参数、错误的循环嵌套、潜在的死循环及不会执行到的代码等。还可以分析各种类型的语句出现的次数、变量和常量的引用表、标识符的使用方式、过程的调用层次及违背编码规则等。静态分析中还可以用符号代替数值求得程序结果，以便对程序进行运算规律的检验。

12.2.2 动态测试

动态测试与静态测试相反，主要是设计一组输入数据，然后通过运行程序来发现错误。在软件的设计中，出现了大量的测试用例设计方法。测试任何工程化产品，一般有两种方法：

（1）了解了产品的功能，然后构造测试，来证实所有的功能是完全可执行的。

（2）知道测试产品的内部结构及处理过程，可以构造测试用例，对所有的结构都进行测试。

前一种方法称为黑盒测试法，后一种方法称为白盒测试法。

1. 黑盒测试法

该方法把被测试对象看成一个黑盒子，测试人员完全不考虑程序的内部结构和处理过程，只在软件的界面上进行测试，用来证实软件功能的可操作性，检查程序是否满足功能要求，是否能很好地接收数据，并产生正确的输出。因此，黑盒测试又称为功能测试或数据驱动测试。一般用来检验系统的基本特征。

黑盒测试的任务是发现以下错误：

（1）是否有不正确或遗漏了的功能。

（2）在界面上，能否正确地处理合理和不合理的输入数据，并产生正确的输出信息。

（3）访问外部信息是否有错。

（4）性能上是否满足要求等。

（5）初始化和终止错误。

用黑盒法测试时，必须在所有可能的输入条件和输出条件中确定测试数据。能否对每个数据都进行穷举测试呢？例如测试一个程序，需输入3个整数值。3个整数值的排列组合数为216×216×216=248≈3×1014。假设此程序执行，计算机一毫秒执行一次测试并得出评估，24小时不停地运行，则需要用时1万年！但这还不能算穷举测试，在黑盒测试时还包括输入一切不合法的数据。可见，穷举地输入测试数据进行黑盒测试是不可能的。

2. 白盒测试法

该方法把测试对象看作一个透明的盒子，测试人员能了解程序的内容结构和处理过程，以检查处理过程为目的，对程序中尽可能多的逻辑路径进行测试，在所有的点检验内部控制结构和数据结构是否和预期相同。

简单地来看，会认为通过全面的白盒测试法的程序将是“完全正确”的程序。但是同样，“全面”的白盒测试法也是不可能的。如测试一个循环20次的嵌套的IF语句，循环体中有5条路径。测试这个程序的执行路径为520，约为1014，如果每毫秒完成一个路径的测试，完成此程序的测试需3170年！

因为白盒测试不检查功能，因此即使每条路径都测试并正确了，程序仍可能有错。例如要求编写一个升序的程序，错编成降序程序（功能错误），就是穷举路径测试也无法发现。再如由于疏忽漏写了路径，白盒测试也发现不了。

所以，黑盒法和白盒法都不能使测试达到彻底。为了让有限的测试发现更多的错误，需精心设计测试用例。

黑盒法、白盒法是设计测试用例的基本策略，每一种方法对应着多种设计测试用例的技术，每种技术可达到一定的软件质量标准要求。下面分别介绍这两类方法对应的各种测试用例设计技术。

12.3 测试用例的设计

12.3.1 白盒技术

白盒测试是结构测试，所以一般都是以程序的内部逻辑结构为基础来设计测试用例。

1. 逻辑覆盖

追求程序内部的逻辑结构覆盖程序，当程序中有循环时，覆盖每条路径是不可能的，要设计使覆盖程序较高的或覆盖最有代表性的路径的测试用例。下面根据如图12-1所示的程序，分别讨论几种常用的覆盖技术。

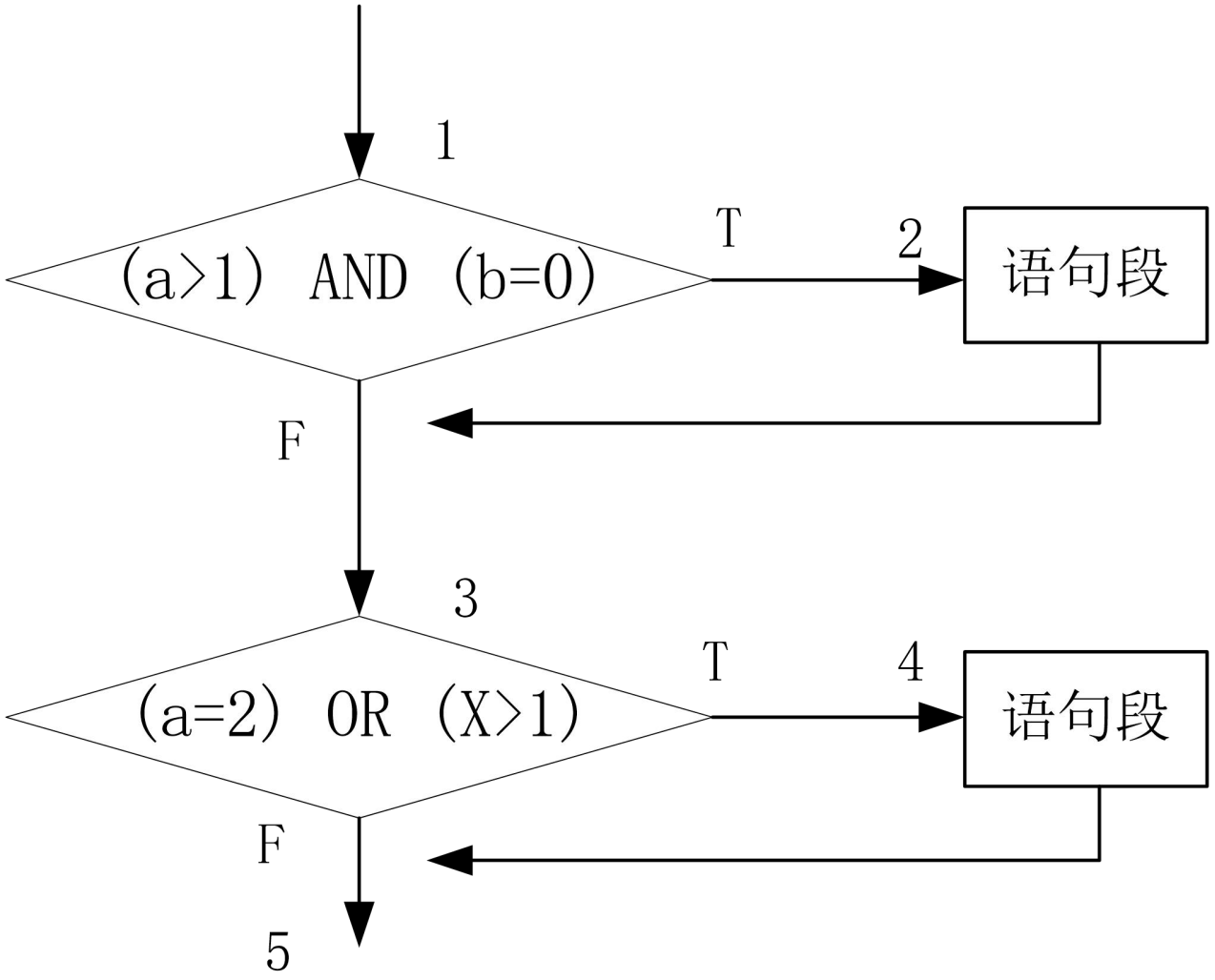


图12-1 一个被测试程序的流程图

1）语句覆盖

为了发现程序中的错误，程序中的每个语句都应该执行一次。语句覆盖是指使用足够多的测试数据，使被测试程序中每个语句至少执行一次。如图12-1所示是一个被测程序的程序流程图。

如果选择的测试数据可以执行路径124，就保证程序流程图中的四个语句至少执行一次，根据条件，选择a=2，b=0，x=3作为测试数据，就能达到语句覆盖的测试标准。

语句覆盖虽然全面地检验了每个语句，但它只测试了逻辑表达式为“真”的情况，如果将第一个逻辑表达式中的“AND”错写成“OR”、第二个逻辑表达式中将“x＞1”错写成“x＜1”，仍用上述数据进行测试，同样可以测试每一个语句，但是不能发现错误。因此，语句覆盖是比较弱的覆盖标准。

2）判定覆盖

相对语句覆盖技术中无法对检查出判定条件中的错误，判定覆盖则设计出足够多的测试用例，使得被测程序中每个判定表达式都执行一次“真”和一次“假”的运行，从而使程序的每一个分支至少都通过一次，因此判定覆盖也称分支覆盖。

对图12-1观察，只要测试用例能通过路径124、135或者125、134，就可以让每个分支都被执行一次。为了达到这个测试目的，可以选择两组数据：

a=3， b=0， x=1（通过路径125）

a=2， b=1， x=2（通过路径134）

对于多分支（嵌套IF、CASE）的判定，判定覆盖要使得每一个判定表达式获得每一种可能的值来测试。同样是每个分支都被执行一次。

只要执行了判定覆盖测试，则语句覆盖肯定也测试了。因为如果通过了各个分支，则各个语句也执行了。但该测试仍不能检查出所有的判定条件的错误，上述数据只覆盖了全部路径的一半，如果将第二个判定表达式中的“x＞1”错写成“x＜1”，仍查不出错误。

3）条件覆盖

条件覆盖测试将使得判定表达式中每个条件的各种可能的值都至少出现一次。在上述程序中有4个条件：

a＞1， b=0， a=2， x＞1

条件覆盖测试要求选择足够的数据，使得第一个判定表达式有下述各种结果出现：

a＞1， b=0， a≤1， b≠0

并使第二个判定表达式出现如下的结果：

a=2， x＞1， a≠2， x≤1

才能达到条件覆盖的标准。

选择以下两组测试数据，就可满足上述要求：

a=2， b=0， x=3（满足a＞1， b=0， a=2，x＞1，通过路径124）

a=1， b=1， x=1（满足a≤1， b≠0， a≠2，x≤1，通过路径135）

以上两组测试覆盖了判断表达式的所有可能，还覆盖了所有判断的取“真”分支和“假”分支。在此测试数据下，条件覆盖比判断覆盖好。

但是如果选择另外一组测试数据：

a=1， b=0， x=3（满足a≤1， b=0， a≠2， x＞1）

a=2， b=1， x=1（满足a＞1， b≠0， a＝2， x≤1）

虽然覆盖了所有条件的可能值，满足条件覆盖，但在第一个判定表达式中只能取“假”和在第二个判定表达式中取“真”，即只测试了路径134，连语句覆盖都不满足。所以满足条件覆盖不一定满足判定覆盖，若结合条件覆盖和判定覆盖就得到了判定/条件覆盖测试技术。

4）判定/条件覆盖

该覆盖标准指设计足够的测试用例，使得判定表达式中的每个条件的所有可能取值至少出现一次，并使每个判定表达式所有可能的结果也至少出现一次。对于上述程序，选择以下两组测试用例：

a=2， b=0， x=3

a=1， b=1， x=1

该组测试数据不仅能满足判定条件覆盖要求，也满足条件覆盖的测试要求。

从表面上看，判定/条件覆盖似乎测试了所有条件的取值，但因为在条件组合中的某些条件的真假会屏蔽其他条件的结果，判定/条件覆盖还有不完善的方面。例如在含有“与”运算的判定表达式中，第一个条件为“假”，则这个表达式中的后面几个条件的值均不起作用；而在含有“或”运算的表达式中，第一个条件为“真”，后边其他条件也不起作用，此时如果后边其他条件写错就不能测试出来。

5）条件组合覆盖

条件组合覆盖是比较强的覆盖标准，顾名思义，按此标准设计的测试用例，使得每个判定表达式中条件的各种可能的值的组合都至少出现一次。

上述程序中，两个判定表达式共有4个条件，因此有8种组合：

① a＞1，b=0　　② a＞1，b≠0

③ a≤1，b=0　　④ a≤1，b≠0

⑤ a=2，x＞1　　⑥ a=2，x≤1

⑦ a≠2，x＞1　　⑧ a≠2，x≤1

下面4组测试用例就可以满足条件组合覆盖标准：

a=0，b=0，x=2　覆盖条件组合①和⑤，通过路径124

a=2，b=1，x=1　覆盖条件组合②和⑥，通过路径134

a=1，b=0，x=2　覆盖条件组合③和⑦，通过路径134

a=1，b=1，x=1　覆盖条件组合④和⑧，通过路径135

显然，满足条件组合覆盖的测试一定满足“判定覆盖”、“条件覆盖”和“判定/条件覆盖”，因为每个判定表达式、每个条件都不止一次地取到过“真”、“假”值。但是，该组测试数据没有能通过125这条路径，不能测试出这条路径中存在的错误。

6）路径覆盖

因为存在选择语句，因此从输入到输出有多条路径。路径覆盖就是要求设计足够多的测试数据，可以覆盖被测程序中所有可能的路径。

该程序中，共有4条路径，选择以下测试用例，就可以覆盖程序中的4条路径：

a=2，b=0，x=2　覆盖路径124，覆盖条件组合①和⑤

a=2，b=1，x=1　覆盖路径134，覆盖条件组合②和⑥

a=1，b=1，x=1　覆盖路径135，覆盖条件组合④和⑧

a=3，b=0，x=1　覆盖路径125，覆盖条件组合①和⑧

可以看出满足路径覆盖却未满足条件组合覆盖。

表12-1列出了这六种覆盖标准的要求。

发现错误能力	覆盖标准	要求
弱 ↓ 强	语句覆盖	每条语句至少执行一次
	判定覆盖	每个判定的每个分支至少执行一次
	条件覆盖	每个判定的每个条件应取到各种可能的值
	判定/条件覆盖	同时满足判定覆盖和条件覆盖
	条件组合覆盖	每个判定中各条件的每一种组合至少出现一次
	路径覆盖	使程序中每一条可能的路径至少执行一次

表12-1 6种覆盖标准的对比

在前五种测试技术中，都是针对单个判定或判定的各个条件值上，其中条件组合覆盖发现错误能力最强，凡满足其标准的测试用例，也必然满足前四种覆盖标准。

路径覆盖测试则根据各判定表达式取值的组合，使程序沿着不同的路径执行，查错能力强。但由于它是从各判定的整体组合发出设计测试用例的，可能使测试用例达不到条件组合的要求。在实际的逻辑覆盖测试中，一般以条件组合覆盖为主设计测试用例，然后再补充部分用例，以达到路径覆盖测试标准。

2. 循环覆盖

除了选择结构外，循环也是程序的主要逻辑结构，要覆盖含有循环结构的所有路径是不可能的，但可通过限制循环次数来测试，一般用以下的方法来设计对循环结构的测试。

1）单循环

设n为可允许执行循环的最大次数。可设计测试数据实现下列的情况：

（1）跳过循环。

（2）只执行循环一次。

（3）执行循环m次，其中m＜n。

（4）执行循环n次和n+1次。

2）嵌套循环

对嵌套循环结构的测试步骤为：

（1）将外循环固定，对内层进行单循环测试。

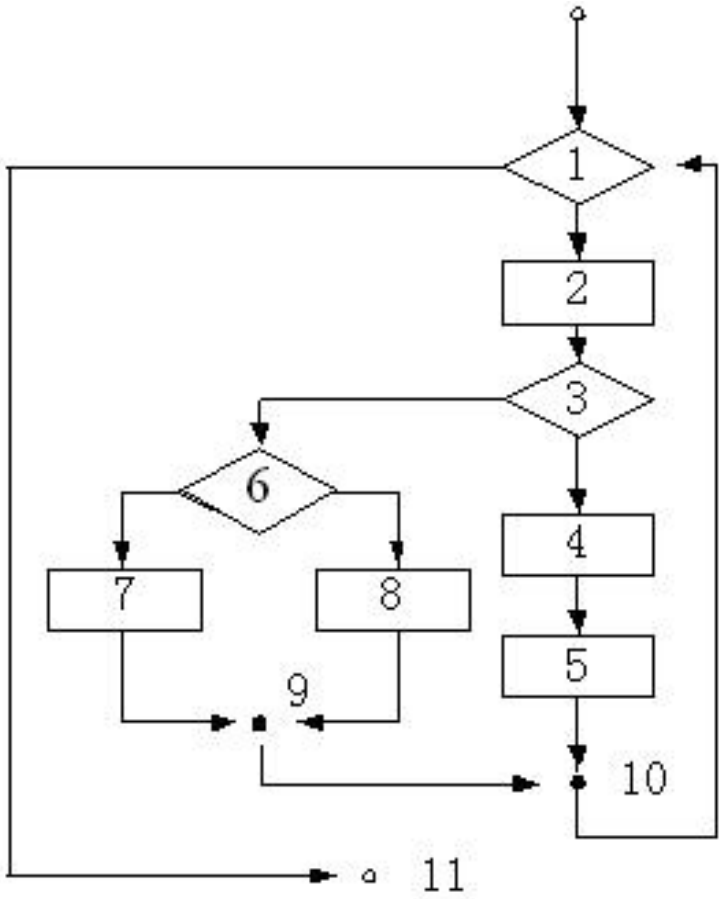
（2）由里向外，进行下一层的循环测试。

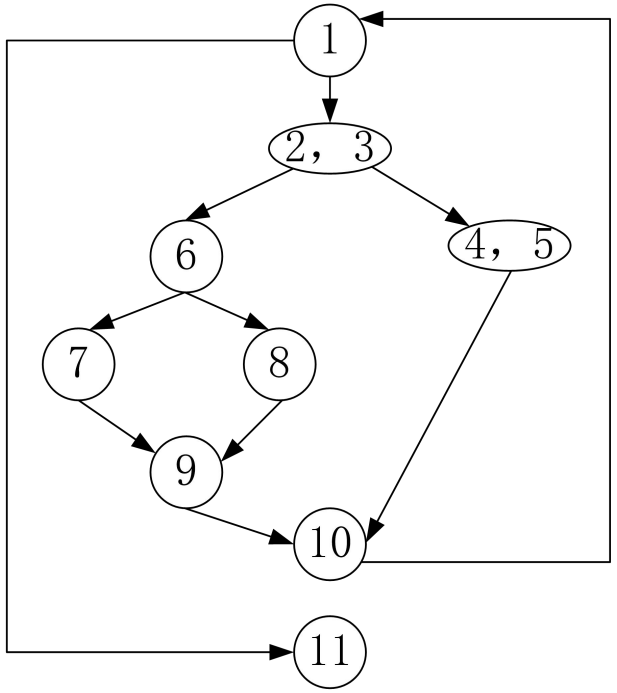
3. 基本路径测试

图12-1的例子只有4条路径。但在实际问题中，一个不太复杂的程序路径都是一个庞大的数字。将覆盖的路径数压缩到一定的限度内可简化测试，例如，循环体只执行一次。基本路径测试就是在程序流程图的基础上，通过分析导出基本路径集合，从而设计测试用例，保证这些路径至少通过一次。

设计基本路径测试的步骤为：

（1）以详细设计或源程序为基础，导出程序流程图的拓扑结构——程序图。程序图是简化了的程序流程图，它是反映程序流程的有向图，其中小圆圈称为结点，代表了流程图中每个处理符号（矩形、菱形框），用箭头的连线表示控制流向，称为程序图中的边或路径。图12-2（a）是一个程序流程图，可以将它转换成图12-2（b）的程序图（假设菱形框表示的判断内设有复合的条件）。



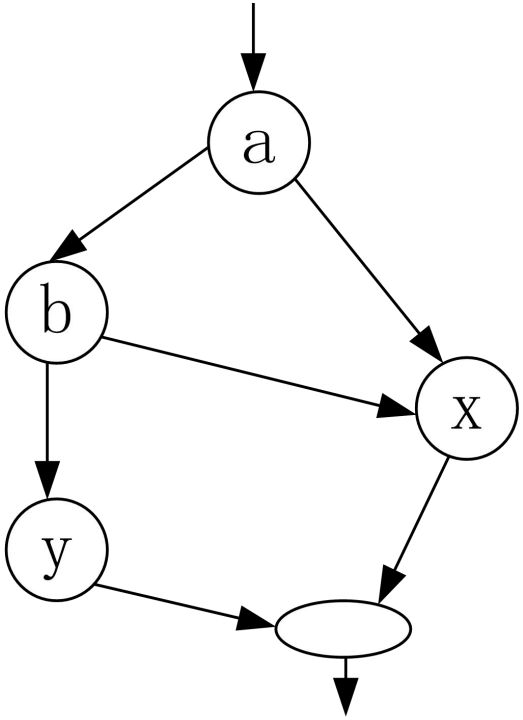


（a）程序流程图　　（b）程序图

图12-2 程序流程图和程序图

在转换时注意，一条边必须终止于一结点，在选择结构中的分支汇聚处即使无语句也应有汇聚结点；若判断中的逻辑表达式是复合条件，应分解为一系列只有单个条件的嵌套判断，如对于图12-3（a）的复合条件的判定应画成图12-3（b）所示的程序图。

```
If a or b
then x else
     y
```



（a）程序　　（b）程序图

图12-3 复合条件的程序图

（2）程序图G的环路复杂性V(G)的计算。McCabe定义程序图的环路复杂性为此平面图中区域的个数。区域个数为边和结点圈定的封闭区域数加上图形外的区域数1。

例如，图12-2（b）的V(G)=4，还可以按如下两种方法计算：

V(G)= 判定结点数+1=3+1=4。

V(G)= 边的数量-节点数量+2=11-9+2=4。

（3）确定只包含独立路径的基本路径集。环路复杂性可导出程序基本路径集合中的独立路径条数，这是确保程序中每个执行语句至少执行一次所必需的测试用例数目的上界。独立路径是指包括一组以前没有处理的语句或条件的一条路径。

从程序图来看，一条独立路径是至少包含有一条在其他独立路径中未有过的边的路径，例如，在图12-2（b）所示的图中，一组独立的路径是：

path1：1—11

path2：1—2—3—4—5—10—1—11

path3：1—2—3—6—8—9—10—1—11

path4：1—2—3—6—7—9—10—1—11

从例中可知，一条新的路径必须包含有一条新边。这四条路径组成了图12-2（b）所示的程序图的一个基本路径集，4是构成这个基本路径集的独立路径数的上界，这也是设计测试用例的数目。只要测试用例确保这些基本路径的执行，就可以使程序中每个可执行语句至少执行一次，每个条件的取“真”和取“假”分支也能得到测试。基本路径集不是惟一的，对于给定的程序图，可以得到不同的基本路径集。

（4）设计测试用例，确保基本路径集合中每条路径的执行。

12.3.2 黑盒技术

黑盒测试是注重于测试软件的功能需求，因此需要研究需求说明和概要设计说明中有关程序功能或输入、输出之间的关系等信息，并且要根据功能得到预期的输出结果，从而与测试后的结果进行分析比较。

黑盒测试不是白盒测试的替代，而是检查出和白盒测试不同类型的错误。黑盒测试一般是在测试的后期使用。用黑盒技术设计测试用例的方法一般有以下4种，但没有一种方法能提供一组完整的测试用例，以检查程序的全部功能。在实际测试中应该把各种方法结合起来使用。

1. 等价类划分

从前面的叙述中知道，不可能用所有可能的输入数据来测试程序，而只能从输入数据中选择一个子集进行测试。等价类划分是选择测试子集的办法。它将输入数据域按有效的或无效的（也称合理的或不合理的）划分成若干个等价类，认为测试等价类的代表值的结果就等于对该类其他值的测试。

也就是说，如果从某个等价类中任选一个测试用例未发现程序错误，则认为该类中其他测试用例也不会发现程序的错误。这样就把漫无边际的随机测试改变为有少数的、有针对性的等价类测试，能有效地提高测试效率。利用等价类测试的步骤如下：

1）划分等价类

从程序的功能说明（如需求说明书）中找出每个输入条件（通常是一句话或一个短语），然后将每一个输入条件划分成为两个或多个等价类，将其列表，其格式为表12-2所示。

输入条件	合理等价类	不合理等价类
…	…	…

表12-2 等价类表

表12-2中合理等价类是指各种正确的输入数据，不合理的等价类是其他错误的输入数据。划分等价类没有一个完整的法则，在具体划分等价类时，可以遵照如下几条经验：

（1）如果某个输入条件规定了取值范围或值的个数，则可确定一个合理的等价类（输入值或数在此范围内）和两个不合理等价类（输入值或个数小于这个范围的最小值或大于这个范围的最大值）。

（2）如果规定了输入数据的一组值，而且程序对不同的输入值做不同的处理，则每个允许的输入值是一个合理等价类，此外还有一个不合理等价类（任何一个不允许的输入值）。

（3）如果规定了输入数据必须遵循的规则，可确定一个合理等价类（条例规则）和若干个不合理等价类（从各种不同角度违反规则）。

（4）如果已划分的等价类中各元素在程序中的处理方式不同，则应将此等价类进一步划分为更小的等价类。

正确分析被测程序的功能和输入条件是正确划分等价类的基础。

2）确定测试用例

完成等价类的划分后，就可以按以下步骤设计测试用例：

（1）为每一个等价类编号。

（2）设计一个合理等价类的测试用例，对于各个输入条件，使其尽可能多地覆盖尚未被覆盖过的合理等价类。重复这步，直到覆盖了所有的合理等价类。

（3）设计一个不合理等价类的测试用例。因为在输入中有一个错误存在时，往往会屏蔽掉其他错误显示，因此设计不合理的等价类的测试数据时，只覆盖一个不合理等价类。重复这一步，直到所有不合理等价类被覆盖。

例如，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1990年1月至1999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此范围内，则显示输入错误信息。该系统规定日期由年、月的6位数字字符组成，前4位代表年，后两位代表月。现用等价类划分法设计测试用例，来测试程序的“日期检查功能”。

（1）划分等价类并编号：划分成3个有效等价类，7个无效等价类，如表12-3所示。

输入等价类	合理等价类	不合理等价类
报表日期的类型及长度	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1990~1999之间	6.小于1990 7.大于1999
月份范围	8.在1~12之间	9.等于0 10.大于12

表12-3 “报表日期”的输入条件的等价类表

（2）为合理等价类设计测试用例，对于表中编号为1，5，8对应的3个合理等价类，用一个测试用例覆盖。

测试数据	期望结果	覆盖范围
199905	输入有效	1，5，8

（3）为每一个不合理等价类至少设计一个测试用例：

测试数据	期望结果	覆盖范围
99MAY	输入无效	2
19995	输入无效	3
1999005	输入无效	4
198912	输入无效	6
200001	输入无效	7
199900	输入无效	9
199913	输入无效	10

在不合理的测试用例中，不能出现相同的测试用例，否则相当于一个测试用例覆盖了一个以上不合理等价类，比如不能用99MAY来同时覆盖2和3。否则即使出错，也不知道原因是2还是3。

等价类划分法比随机选择测试用例要好得多，但这个方法的缺点是没有注意选择某些高效的、能够发现更多错误的测试用例。

2. 边界值分析

边界值是指输入等价类或输出等价类边界上的值。实践经验表明，程序往往在处理边界情况时发生错误。因此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。

例如，在做三角形设计时，要输入三角形的3个边长A，B，C。这三个数值应当满足A＞0，B＞0，C＞0，A＋B＞C，A＋C＞B，B＋C＞A，才能构成三角形。但如果把6个不等式中的任何一个“＞”错写成“≥”，那个不能构成三角形的问题恰出现在容易被疏忽的边界附近。在选择测试用例时，选择边界附近的值就能发现被疏忽的问题。

在划分等价类的基础上采用边界值分析方法设计测试用例，可以直接取那些等价类的边界值，选取正好等于、刚刚大于或刚刚小于边界值的测试数据。有以下的一些设计原则：

（1）如果输入条件规定了值的范围，可以选择正好等于边界值的数据作为合理的测试用例，同时还要选择刚好越过边界值的数据作为不合理的测试用例。如输入值的范围是[1，100]，可取0，1，100，101等值作为测试数据。

（2）如果输入条件指出了输入数据的个数，则按最大个数、最小个数、比最小个数少1及比最大个数多1等情况分别设计测试用例。如一个输入文件可包括1~255个记录，则分别设计有1个记录、255个记录，以及0个记录和256个记录的输入文件的测试用例。

（3）对每个输出条件分别按照以上两个原则确定输出值的边界情况。如一个学生成绩管理系统规定，只能查询95~98级大学生的各科成绩，可以设计测试用例，除了查询范围内的学生的学生成绩，还需设计查询94级、99级学生成绩的测试用例（不合理输出等价类）。

由于输出值的边界与输入值的边界没有必然的对应关系，所以要检查输出值的边界不一定可能，要产生超出输出值之外的结果也不一定能做到，但必要时还需试一试。

（4）如果程序的需求说明给出的输入或输出域是个有序集合（如顺序文件、线性表和链表等），则应选取集合的第一个元素和最后一个元素作为测试用例。

对上述报表处理系统中的报表日期输入条件，以下用边界值分析设计测试用例。

程序中判断输入日期（年月）是否有效，假设使用如下语句：

IF（ReportDate＜= MaxDate＝AND（ReportDate＞=MinDate）

THEN产生指定日期报表

ELSE显示错误信息

ENDIF

如果将程序中的“＜＝”误写为“＜”，则上例的等价类划分中所有测试用例都不能发现这一错误，采用边界值分析法的测试用例，如表12-4所示。

显然采用测试用例发现程序中的错误要更彻底一些

输入等价类	测试用例说明	测试数据	期望结果	选取理由
报表日期的类型及长度	1个数字字符	5	显示出错	仅有一个合法字符
	5个数字字符	19995	显示出错	比有效长度少1
	7个数字字符	199905	显示出错	比有效长度多1
	有1个非数字字符	1999.5	显示出错	只有一个非法字符
	全部是非数字字符	May---	显示出错	6个非法字符
	6个数字字符	199905	显示有效	类型及长度均有效
日期范围	在有效范围边界上选取数据	199001	输入有效	最小日期
		199912	输入有效	最大日期
		199000	显示出错	刚好小于最小日期
		199913	显示出错	刚好大于最大日期
月份范围	月份为1月	199801	输入有效	最小月份
	月份为12月	199812	输入有效	最大月份
	月份＜1	199800	显示出错	刚好小于最小月份
	月份＞12	199813	显示出错	刚好大于最大月份

表12-4 “报表日期”边界值分析法测试用例

3. 错误推测

在测试程序时，人们根据经验或直觉推测程序中可能存在的各种错误，从而有针对性地编写检查这些错误的测试用例，这就是错误推测法。

错误推测法没有确定的步骤，凭经验进行。它的基本思想是列出程序中可能发生错误的情况，根据这些情况选择测试用例。如输入、输出数据为零是容易发生错误的情况；又如，输入表格为空或输入表格只有一行是容易出错的情况等。

例如，对于一个排序程序，列出以下几项需特别测试的情况：

（1）输入表为空。

（2）输入表只含一个元素。

（3）输入表中所有元素均相同。

（4）输入表中已排好序。

又如，测试一个采用二分法的检索程序，考虑以下情况：

（1）表中只有一个元素。

（2）表长是2的幂。

（3）表长是2的幂减1或2的幂加1。

因此，要根据具体情况具体分析。

4. 因果图

等价类划分和边界值分析方法都只是孤立地考虑各个输入数据的测试功能，而没有考虑多个输入数据的组合引起的错误。因果图能有效地检测输入条件的各种组合可能会引起的错误。因果图的基本原理是将自然语言描述的功能说明转换为判定表，最后为判定表的每一列设计一测试用例。

5. 综合策略

没有哪一种测试方法是最好的，发现错误能力最强的。每种方法都适合于发现某种特定类型的错误。因此在实际测试中，常常联合使用各种测试方法，形成综合策略，通常先用黑盒测试法设计基本的测试用例，再用白盒法补充一些必要的测试用例，方法如下：

（1）在任何情况下都应使用边界值分析法，用这种方法设计的用例暴露程序错误能力强。设计用例时，应该既包括输入数据的边界情况又包括输出数据的边界情况。

（2）必要时用等价类划分方法补充一些测试用例。

（3）再用错误推测法补充测试用例。

（4）检查上述测试用例的逻辑覆盖程度，如未满足所要求的覆盖标准，再增加例子。

（5）如果需求说明中含有输入条件的组合情况，则一开始就可使用因果图法。

12.4 测试过程

12.4.1 软件测试过程中的信息

软件测试时需要以下三类信息：

（1）软件配置：指需求说明书、设计说明书和源程序等。

（2）测试配置：指测试方案、测试用例和测试驱动程序等。

（3）测试工具：指计算机辅助测试的有关工具。

软件经过测试以后，要根据预期的结果对测试的结果进行评估，对于出现的错误要报告，并修改相应文档。修改后的程序往往要经过再次测试，直到满意为止。

在分析结果的同时，要对软件可靠性进行评价，如果总是出现需要修改设计的严重错误，软件的质量和可靠性就值得怀疑，同时也需进一步测试；如果软件功能能正确完成，出现的错误易修改，可以断定软件的质量和可靠性可以接受或者所做的测试还不足以发现严重错误；如果测试发现不了错误，那么应该修改测试方案、用例，要考虑错误仍潜伏在软件中，应考虑重新制定测试方案，设计测试用例。

12.4.2 软件测试的步骤与各开发阶段的关系

测试并不只是针对编码进行的测试，一般在软件产品交付使用之前要经过单元测试、集成测试、确认测试和系统测试。如图12-4所示为软件测试经历的步骤。

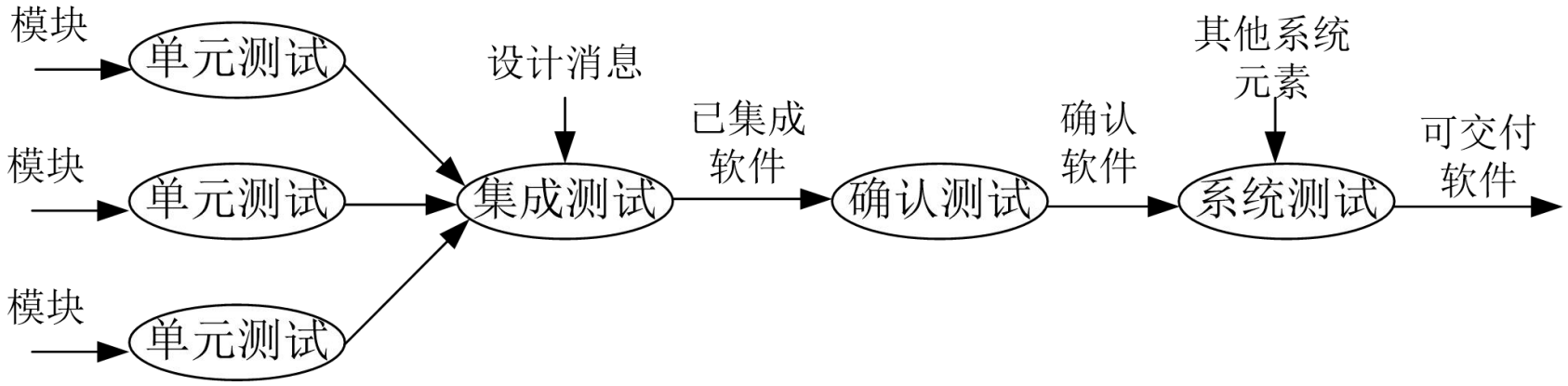


图12-4 软件测试步骤

单元测试检查软件设计中的最小单元——模块。检查各个模块是否正确实现规定的功能，接口是否正确，从而发现模块在编码中或算法中的错误。对模块进行测试之前，必须先通过编译程序检查语法错误，然后根据该阶段涉及编码和详细设计的文档进行测试。当所有的单元测试完成后，将各模块组装起来进行集成测试，主要是检查各单元之间的接口问题，单元之间的相互影响问题以及与设计相关的软件体系结构的有关问题。完成集成测试之后，接口之间的问题已经被修改了。

确认测试则主要检查已实现的软件是否满足需求说明书中确定了的各种功能。系统测试指把已确定的软件与其他系统元素（如硬件、其他支持软件、数据和人工等）结合在一起进行测试，主要包括恢复测试、安全测试、压力测试和性能测试。如图12-5所示列出了软件工程各阶段与各种测试的关系。

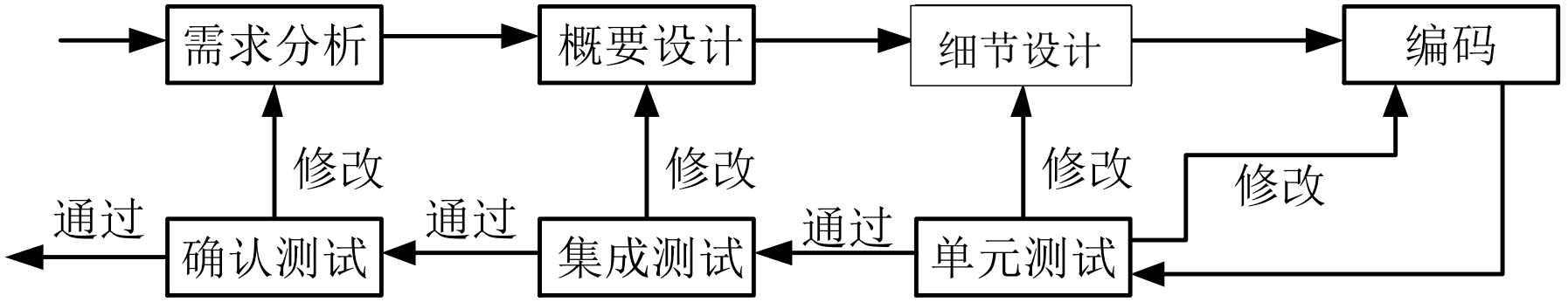


图12-5 软件测试与软件开发过程的关系

12.4.3 单元测试

1. 测试的内容

单元测试主要针对模块的5个基本特征进行测试。

1）模块接口

模块接口测试主要是保证数据能否正确地通过单元。检查的主要内容是实参和形参的参数个数、数据类型及对应关系是否一致。当模块通过文件进行输入/输出时，要检查文件的具体描述（包括文件的定义、记录的描述、缓冲区的大小和记录是否匹配及文件的处理方式等）是否正确。

2）局部数据结构

局部数据结构主要检查以下几方面的错误：类型说明不正确或不一致；初始化或缺省值错误；不正确的变量名字；数据类型不相容；上溢、下溢或地址错等。

除了检查局部数据外，还应注意全局数据对模块的影响。

3）重要的执行路径

重要模块要进行基本路径测试，仔细地选择测试路径是单元测试的一项基本任务。注意选择测试用例能发现不正确的计算、错误的比较或不适当的控制流而造成的错误。

4）错误处理

错误处理主要测试程序处理错误的能力，检查是否存在以下问题：不能正确处理外部输入错误或内部处理引起的错误；对发生的错误不能正确描述或描述内容难以理解；所显示的错误与真正的错误不一致，例如条件处理不正确；在错误处理之前，系统已进行干预等。

5）边界条件

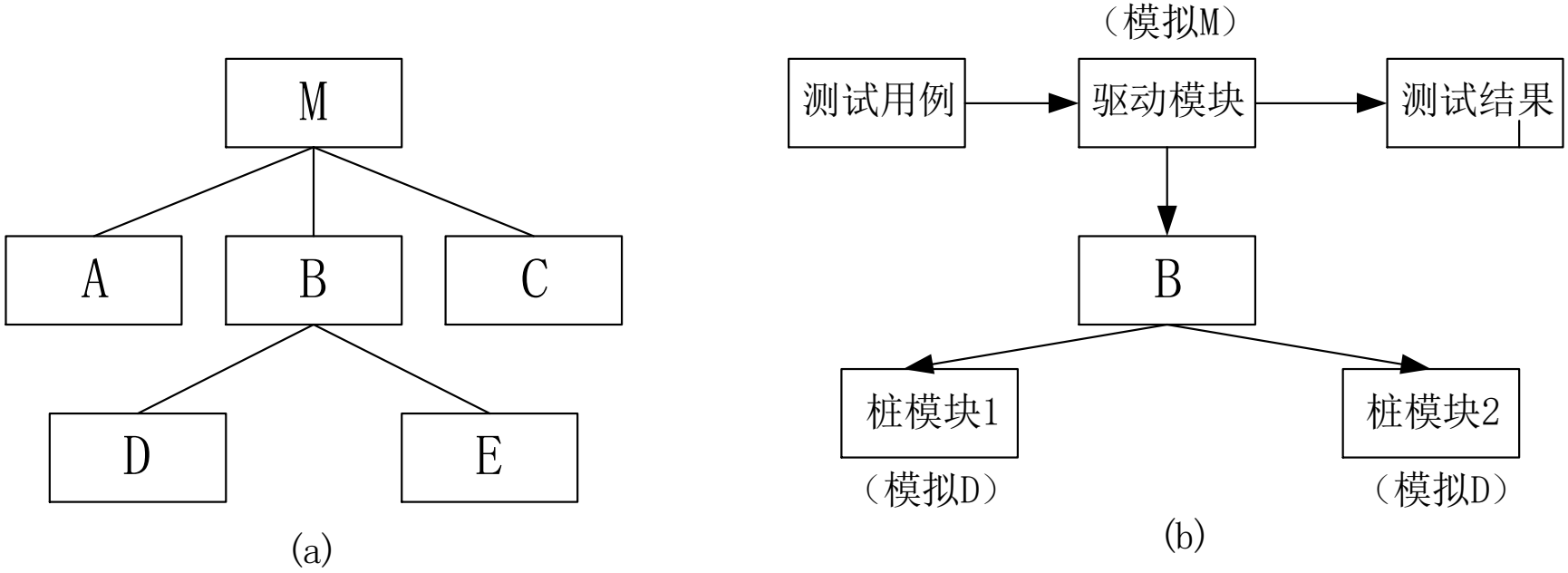
程序最容易在边界上出错，如输入/输出数据的等价类边界，选择条件和循环条件的边界、复杂数据结构（如表）的边界等都应进行测试。

2. 测试的方法

由于被测试的模块处于整个软件结构的某一层位置上，一般是被其他模块调用或调用其他模块，其本身不能进行单独运行，因此在单元测试时，需要为被测模块设计驱动模块（driver）和桩（stub）模块。

驱动模块的作用是用来模拟被测模块的上级调用模块，功能要比真正的上级模块简单得多，仅仅是接收被测模块的测试结果并输出。桩模块则用来代替被测模块所调用的模块。它的作用是提供被测模块所需的信息。图12-6（a）表示被测软件的结构，而图12-6（b）表示用驱动模块和桩模块建立测试模块B的环境。

驱动模块和桩模块的编写给软件开发带来额外开销，但是设计这些模块对测试是必要的。



（a）软件结构　　（b）模块B的测试环境

图12-6 单元测试的测试环境

12.4.4 集成测试

1. 集成测试的目的

集成测试是指在单元测试的基础上，将所有模块按照设计要求组装成一个完整的系统而进行的测试，故也称组装测试或联合测试。实践证明，单个模块能正常工作，组装后不见得仍能正常工作，这是因为：

（1）单元测试中使用的驱动模块和模块都是模拟相应模块的功能，与它们所代替的模块并不完全等效，因此单元测试有不彻底、不严格的情况。

（2）各个模块组装起来，穿越模块接口的数据可能会丢失。

（3）组合后，一个模块的功能可能会对另一个模块的功能产生不利的影响。

（4）各个模块的功能组合起来可能达不到预期要求的功能。

（5）单个模块可以接受的误差，组装起来可能累积和放大到不能接受的程度。

（6）模块组合之后，全局数据可能会对模块产生影响。

因此必须要进行集成测试，用于发现模块组装中可能出现的问题，最终构成一个符合要求的程序结构。

2. 集成测试的方法

集成测试的方法主要有自底向上测试和自顶向下测试。

1）自底向上测试

该测试是首先对每个模块分别进行单元测试，然后再把所有的模块按设计要求组装在一起进行的测试，测试时不再需要桩模块。

2）自顶向下测试

该测试首先集成主控模块，然后自顶向下逐个把未经过测试的模块组装到已经测试过的模块上去，进行集成测试。每加入一个新模块就进行一次集成的测试，重复此过程直至程序组装完毕。

3）自顶向下与自底向上测试的区别

自顶向下与自底向上测试的区别有如下几点：

（1）自底向上方法把单元测试和集成测试分成两个不同的阶段，前一阶段完成模块的单元测试，后一阶段完成集成测试。而自顶向下测试把单元测试与集成测试合在一起，同时完成。

（2）自顶向下可以较早地发现接口之间的错误，自底向上最后组装时才发现。

（3）自顶向下有利于排错，发生错误往往和最近加进来的模块有关，而自底向上发现接口错误推迟到最后，很难判断是哪一部分接口出错。

（4）自顶向下比较彻底，已测试的模块和新的模块组装在一起再测试。

（5）自顶向下占用的时间较多，但自底向上需更多的驱动模块，也占用一些时间。

（6）自底向上开始可并行测试所有模块，能充分利用人力，对测试大型软件很有意义。

在软件开发中，根据软件错误发现越早代价越低等特点，采用自顶向下方法测试较好，但在实际开发中，常将两种方法结合起来，在进行自顶向下测试中，同时组织人力对一些模块分别测试，然后将这些测试过的模块再用自顶向下逐步结合进软件系统中去。

3. 组装模块的方法

1）自顶向下结合

该方法从主控模块开始进行集成，测试中不需要编写驱动模块，只需要编写桩模块。然后沿被测程序的控制路径逐步向下测试，从而把各个模块都结合进来，在集成各个模块中，有两种组合策略：

（1）深度优先策略：先从软件结构中选择一条主控路径，把该路径上的模块一个个结合进来进行测试，则某个完整的功能会被实现和测试，接着再结合其他需要优先考虑的路径。主控路径一般选择系统的关键路径或输入、输出路径。图12-7是一个软件结构图。

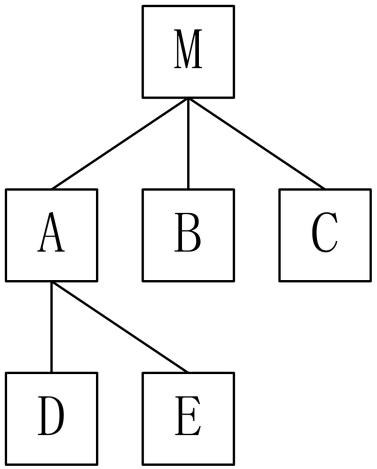


图12-7 一个软件结构图

图12-8是自顶向下以深度优先策略组装模块的例子，其中Si模块代表桩模块。

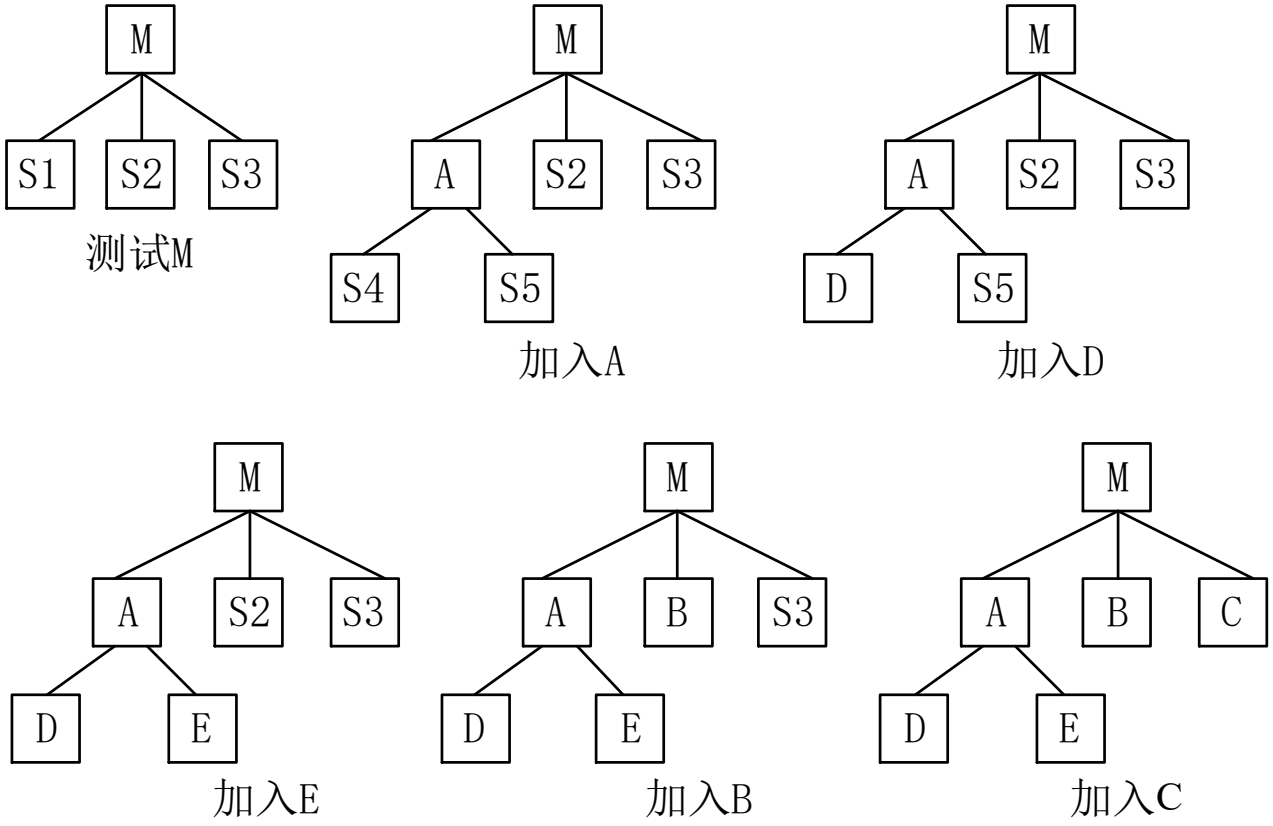


图12-8 采用深度优先策略自顶向下结合模块的过程

（2）宽度优先策略：逐层结合直接下属的所有模块。如对于图12-7的例子，结合顺序为M，A，B，C，D，E。

自顶向下测试的优点是能较早地发现高层模块接口、控制等方面的问题；程序在初期对功能性的验证，对增强开发人员和客户的信心都有好处。缺点是桩模块不可能完全等效于真正的底层模块，因此许多测试只有推迟到用实际模块代替桩模块之后才能完成；测试中要设计较多的桩模块，测试开销大；早期不能并行工作，不能充分利用资源。

2）自底向上结合

与自顶向下不同，只需编写驱动模块，不需编写桩模块。测试步骤为：

（1）把低层模块组合成实现一个个特定功能的族。如图12-9所示。

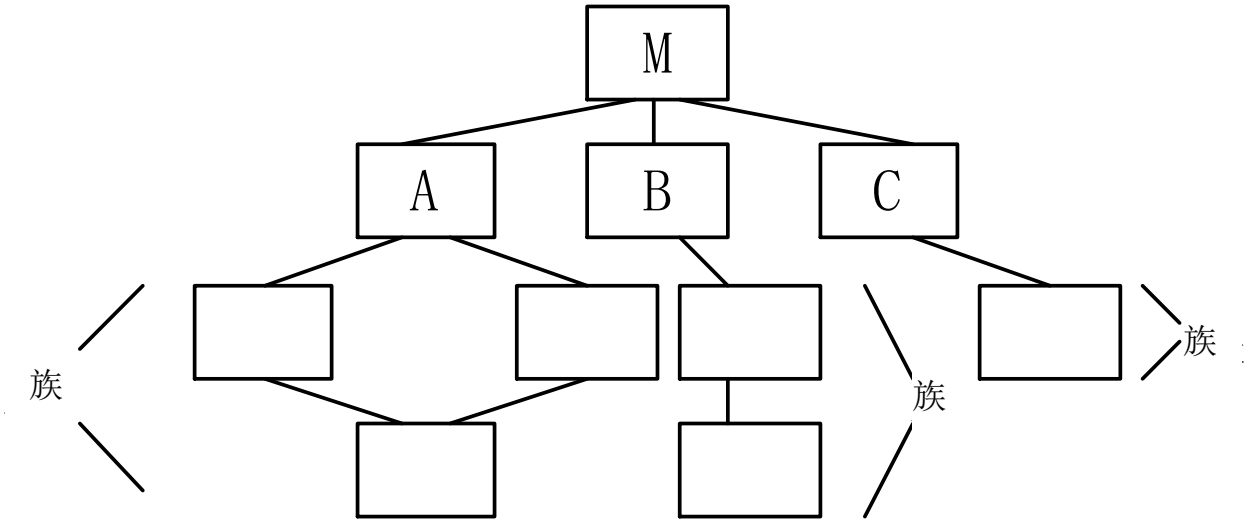


图12-9 一个软件结构图

（2）为每一个族编写一个驱动模块，以协调测试用例的输入和测试结果的输出。如图12-10所示，其中di模块为驱动模块。

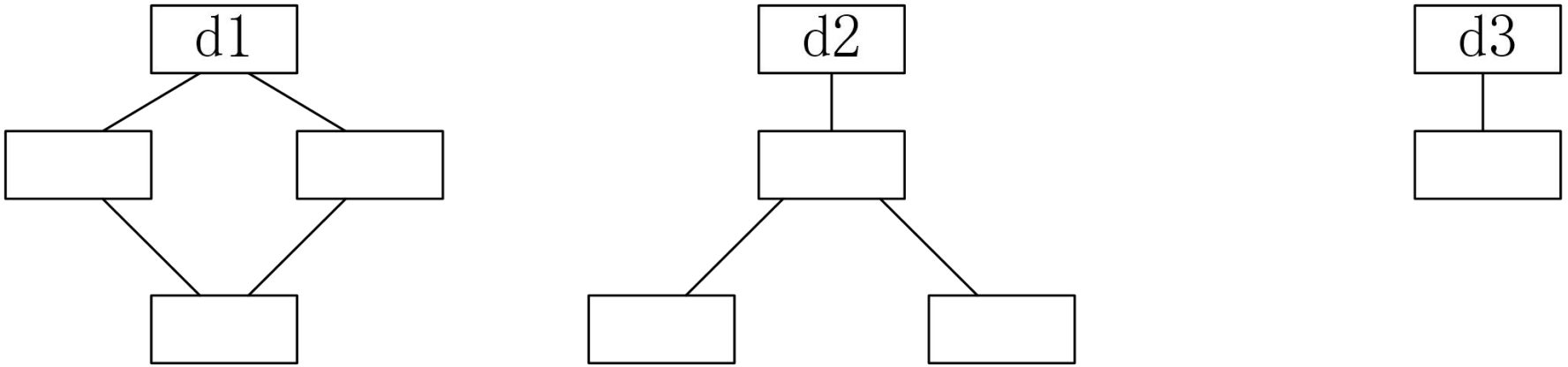


图12-10 为每个族分别进行测试

（3）对模块族进行测试。

（4）按软件控制结构图依次向上扩展，用实际模块替换驱动模块，形成一个个更大的族。如图12-11所示。

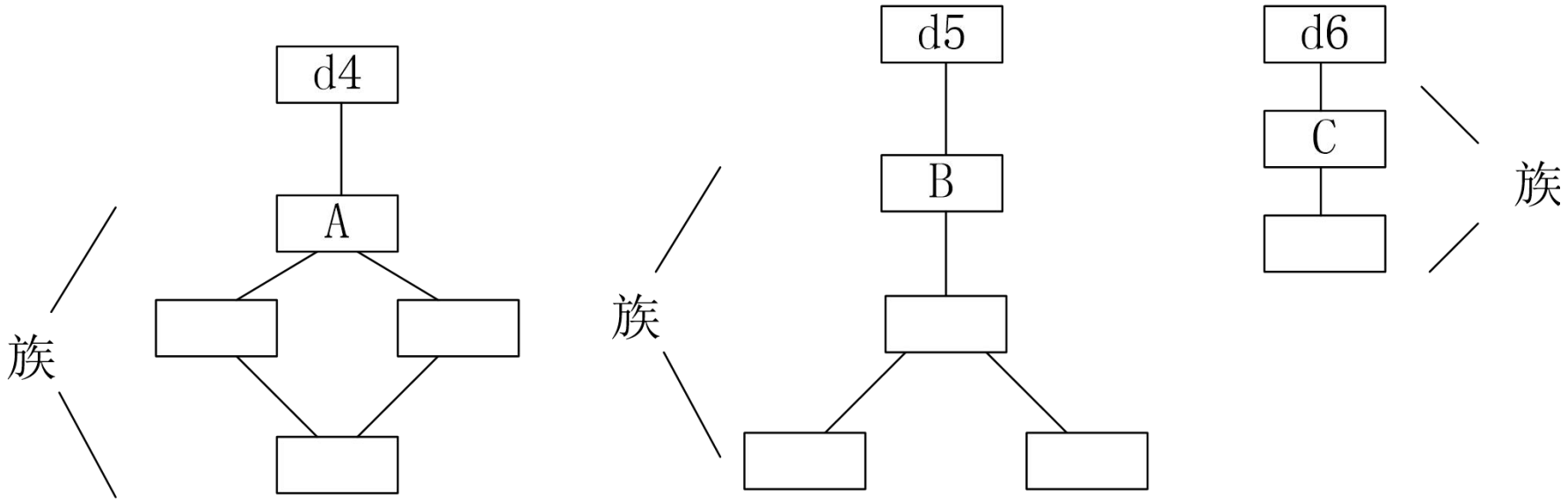


图12-11 形成3个更大的族进一步测试

（5）重复（2）~（4）步，直至软件系统全部测试完毕。

自底向上测试的优点是：随着测试的进行，驱动模块数逐步减少；比较容易设计测试用例；在早期可以并行工作，充分利用软硬件资源；低层模块的错误能较早发现。其缺点是：系统整体功能到最后才能测试；软件决策性的错误发现得晚，而上层模块的问题是全局性的问题，影响范围大。

由于自顶向下测试和自底向上测试的方法各有利弊，实际应用时，应根据软件的特点、任务的进度安排选择合适的方法。一般是将这两种测试方法结合起来，低层模块使用自底向上结合的方法组装成子系统，然后由主模块开始自顶向下对各子系统进行集成测试。

无论使用自顶向下和自底向上测试方法，都需要使用回归测试。因为随着新模块的添加，软件就发生改变，模块之间的影响也存在，所以要对已经测试过的模块重新测试。

12.4.5 确认测试

当集成测试完成后，软件的组装就完成了。软件测试的最后一部分确认测试就开始了。确认测试又称有效性测试。它的任务是检查软件的功能与性能是否与需求说明书中确定的指标相符合。因而需求说明是确认测试的基础。

确认测试阶段有进行确认测试与软件配置审查两项工作。

1. 进行确认测试

确认测试一般是在模拟环境下运用黑盒测试方法，由专门测试人员和用户参加的测试。确认测试需要需求说明书、用户手册等文档，要制定测试计划，确定测试的项目，说明测试的内容，描述具体的测试用例。测试用例应选用实际运用的数据。测试结束后，应写出测试分析报告。

经过确认测试后，可能有两种情况：

（1）功能、性能与需求说明一致，该软件系统是可以接受的。

（2）功能、性能与需求说明有差距，要提交一份问题报告。对这样的错误进行修改，工作量非常大，必须同用户协商。

2. 软件配置审查

软件配置审查的任务是检查软件的所有文档资料的完整性、正确性。如发现遗漏和错误，应补充和改正。同时要编排好目录，为以后的软件维护工作奠定基础。

软件系统只是计算机系统中的一个组成部分，软件经过确认后，最终还要与系统中的其他部分（如计算机硬件、外部设备、某些支持软件、数据及人员）结合在一起，在实际使用环境下运行，测试其能否协调工作，这就是所谓的系统测试，系统测试有关的内容不在软件工程范围内。

12.5 调试

1. 调试的目的

软件测试的目的是尽可能多地发现程序中的错误，而调试则是进行了成功的测试之后才开始的工作。调试的目的是确定错误的原因和位置，并改正错误，因此调试也称为纠错。

调试是程序员自己进行的技巧性很强的工作，确定发生错误的内在原因和位置几乎占整个调试工作量的90%左右。调试工作的困难与人的心理因素和技术因素都有关系，需要繁重的脑力劳动和丰富的经验。调试技术缺乏系统的理论研究，因此调试方法多是实践中的经验积累。

2. 调试技术

1）简单的调试方法

（1）在程序中插入打印语句。

该方法的优点是动态显示程序的运行状况，比较容易检查源程序的有关信息。缺点是低效率，可能输出大量无关的数据，发现错误带有偶然性。同时还要修改程序，这种修改可能会掩盖错误、改变关键的时间关系或把新的错误引入程序，因此一般是在可能出错的地方插入打印语句。调试完毕要记着将打印语句删除或注释掉。

（2）运行部分程序。

有时为了测试某些被怀疑为有错的程序段，整个程序反复执行多次，使很多时间浪费在执行已经是正确的程序段上。在此情况下，应设法使被测试程序只执行需要检查的程序段，以提高效率。可通过注释程序或开发语言所带的调试工具来查找错误。

2）归纳法调试

归纳法调试从测试结果发现的错误入手，分析它们之间的联系，导出错误原因的假设，然后再证明或否定这个假设。归纳法调试的具体步骤如下：

（1）收集有关数据：列出程序做对了什么、做错了什么的全部信息。

（2）组织数据：整理数据以便发现规律，使用分类法构造一线索表。

（3）提出假设：分析线索之间的关系，导出一个或多个错误原因的假设。如果不能推测一个假设，再选用测试用例去测试，以便得到更多的数据。如果有多个假设，首先选择可能性最大的一个。

（4）证明假设：假设不是事实，需要证明假设是否合理。不经证明就根据假设改错，只能纠正错误的一种表现（即消除错误的征兆）或只纠正一部分错误。如果不能证明这个假设成立，需要提出下一个假设。

3）演绎法调试

演绎法相当于是一种排错法。演绎法调试先列出所有可能的错误原因的假设，然后利用测试数据排除不适当的假设，最后再用测试数据验证余下的假设确实是出错的原因。演绎法调试的具体步骤如下：

（1）列出所有可能的错误原因的假设：把可能的错误原因列成表，不需要完全解释，仅是一些可能因素的假设。

（2）排除不适当的假设：应仔细分析已有的数据，寻找矛盾，力求排除前一步列出的所有原因。如果都排除了，则需补充一些测试用例，以建立新的假设；如果保留下来的假设多于一个，则选择可能性最大的原因做基本的假设。

（3）精化余下的假设：利用已知的线索，进一步求精余下的假设，使之更具体化，以便可以精确地确定出错位置。

（4）证明余下的假设：做法同归纳法。

4）回溯法调试

对小型程序寻找错误位置，回溯法是有效方法。该方法从程序产生错误的地方出发，人工沿程序的逻辑路径反向搜索，直到找到错误的原因为止。例如，从打印语句出错开始，通过看到的变量值，从相反的执行路径查询该变量值从何而来。

小结

本章主要介绍了软件测试。测试的目的是为了发现错误，而不是去证明程序正确。完善的测试用例是能够用最少的数据发现最多的错误的测试集。

测试方法分为静态测试和动态测试，静态测试不需要运行程序，检查程序的逻辑结构和算法，动态测试是通过运行程序来发现程序的错误。有两种不同的测试用例设计技术：黑盒测试和白盒测试。

小结

白盒测试是以了解程序的内部结构为基础的。测试用例要求保证每条语句至少执行一次，每个判断条件至少执行一次，每条路径也要执行一次。

黑盒测试是一种对功能的测试。不需要知道程序的内部结构。它的目的是发现功能错误。它侧重于划分程序的输入和输出域。主要的测试用例设计方法是划分等价类。

小结

要保证软件的质量，在软件交付使用之前应该对软件进行单元测试、集成测试和确认测试。每种测试都有它自己的发现错误的重点。在对单元进行测试时，要为测试编写“桩”模块和“驱动”模块；在集成测试时可以是自顶向下和自底向上的测试方法。实际测试时要综合应用各种测试方法。

谢谢