第九章 软件测试

在软件开发过程中,特别是在开发大型软件系统的过程中，面对的问题是极其复杂。因此，在软件生命周期的每个阶段就不可避免地会产生差错。应该在每个阶段结束之前通过严格的技术审查，尽可能早地发现并纠正差错。但是，经验表明审查并不能发现所有差错，此外在编码过程中还不可避免地会引入新的错误。如果在软件正式运行之前，没有发现并纠正软件中的大部分差错，则这些差错迟早会在生产过程中暴露出来，那时不仅改正这些错误的代价更高，而且往往会造成很恶劣的后果。测试的目的就是要在软件投入生产性运行之前，尽可能多地发现软件中的错误。大量统计资料表明，软件测试的工作量往往占软件开发总工作量的 40%以上，在极端情况，测试软件所花费的成本，可能相当于软件工程其他开发步骤总成本的三到五倍。因此，必须高度重视软件测试工作。软件测试是软件质量保证的关键步骤，代表了规约、设计和编码的最终检查。

对软件进行测试的结果也是分析软件可靠性的重要依据。本章着重讨论软件测试概念、传统软件和面向对象软件测试的问题。

9.1 软件测试概述

既然软件测试是必不可少的阶段，那么我们必须首先对软件测试要有一个比较全面的认识。它的目标是什么？遵循的原则是什么？使用的方法又如何？等等。

9.1.1 软件测试目标

从表面看来，软件测试的目的与软件工程所有其他阶段的目的都相反。软件工程的其他阶段都是“建设性”的，软件工程师力图从抽象的概念出发，逐步设计出具体的软件系统，直到用一种适当的程序设计语言写出可以执行的程序代码。但是，在测试阶段测试人员努力设计出一系列测试方案，目的却是为了“破坏”已经建造好的软件系统——竭力证明程序中有错误不能按照预定要求正确工作。当然，这种反常仅仅是表面的，或者说是心理上的。暴露问题并不是软件测试的最终目的，发现问题是为了解决问题，测试阶段的根本目标是尽可能多地发现并排除软件中潜藏的错误，最终把一个高质量的软件系统交给用户使用。以下三点可以看作是测试的目标或定义：

(1)测试是为了发现程序中的错误而执行程序的过程；

(2)好的测试方案是尽可能发现迄今为止尚未发现的错误的测试方案；

(3)成功的测试是发现了至今为止尚未发现的错误的测试。

由此可以看出，测试的正确定义是“为了发现程序中的错误而执行程序的过程”。这和某些人通常想象的“测试是为了表明程序是正确的”,“成功的测试是没有发现错误的测试”等等是完全相反的。正确认识测试的目标是十分重要的，测试目标决定了测试方案的设计。如果为了表明程序是正确的，就会设计一些不易暴露错误的测试方案。相反，如果测试是为了发现程序中的错误，就会力求设计出最能暴露错误的测试方案。

测试只能查找出程序中的错误，不能证明程序中没有错误。如果成功构造了测试(根据上述目标)，则能够在软件中揭示错误。另外，测试能证实软件依据规约所具有的功能及其

性能需求，构造测试时的数据收集提供了软件可靠性以及软件整体质量的一些信息。

9.1.2 软件测试原则

在设计有效的测试用例之前，软件开发者必须理解软件测试的基本原则。软件测试的原则是：

（1）将“尽早地和不断地进行软件测试”作为软件开发者的座右铭。

不应把软件测试仅仅看做是软件开发的一个独立阶段，而应当把它贯穿到软件开发的各个阶段中。坚持软件开发的各个阶段的技术评审，这样才能在开发过程中尽早发现和预防错误，把出现的错误克服在早期，杜绝某些发生错误的隐患。

（2）测试用例应由测试输入数据和与之对应的预期输出结果这两部分组成。

测试以前应当根据测试的要求选择测试用例(Testcase)，用来检验程序员编制的程序，因此不但需要测试的输入数据，而且需要针对这些输入数据预期输出结果。

（3）程序员应避免检查自己的程序。

创建系统的软件开发人员并不是构造软件测试的最佳人选。这不能与程序的调试相混淆。调试由程序员自己来做可能更有效。程序员应尽可能避免测试自己编写的程序，程序开发小组也应尽可能避免测试本小组开发的程序。为了达到最佳效果，最好建立独立的软件测试小组或测试机构。

（4）在设计测试用例时，应当包括合理的输入条件和不合理的输入条件。合理的输入条件是指能验证程序正确的输入条件，不合理的输入条件是指异常的、临界的、可能引起问题的输入条件。软件系统处理非法命令的能力必须在测试时受到检验。用不合理的输入条件测试程序时，往往比用合理的输入条件进行测试能发现更多的错误。

（5）充分注意测试中的群集现象。

在被测程序段中，若发现错误数目多，则残存错误数目也比较多。这种错误群集性现象，已被许多程序的测试实践所证实。

根据这个规律，应当对错误群集的程序段进行重点测试，以提高测试投资的效益。

（6）严格执行测试计划，排除测试的随意性。

测试之前应仔细考虑测试的项目，对每项测试做出周密的计划，包括被测程序的功能、输入和输出、测试内容、进度安排、资源要求、测试用例的选择、测试的控制方式和过程等，还要包括系统的组装方式、跟踪规程、调试规程、回归测试的规定，以及评价标准等。对于测试计划，要明确规定，不要随意解释。

（7）应当对每个测试结果做全面检查。

有些错误的征兆在输出实测结果时已经明显地出现了，但是如果不仔细地、全面地检查测试结果，就会使这些错误被遗漏掉。所以必须对预期的输出结果明确定义，对实测的结果仔细分析检查，暴露错误。

（8）妥善保存测试计划、测试用例、出错统计和最终分析报告，为维护提供方便。

9.1.3 软件测试方法

测试任何产品都有两种方法，如果已经知道了产品应该具有的功能，可以通过测试来检验是否每个功能都能正常使用。如果知道产品内部工作过程，可以通过测试来检验产品内部动作是否按照规格说明书的规定正常进行。前一个方法称为黑盒测试，后一个方法称为白盒测试。

对于软件测试而言，黑盒测试法把程序看成一个黑盒子，完全不考虑程序的内部结构和处理过程。也就是说，黑盒测试是在程序接口进行的测试，它只检查程序功能是否能按照规格说明书的规定正常使用，程序是否能适当地接收输入数据产生正确的输出信息，并且保持外部信息(如，数据库或文件)的完整性。黑盒测试又称为功能测试。与黑盒测试法相反，白盒测试法是完全了解程序的结构和处理过程。这种方法按照程序内部的逻辑测试程序，检验程序中的每条通路是否都能按预定要求正确工作。白盒测试又称为结构测试。

粗看起来，不论采用上述哪种测试方法，只要对每一种可能的情况都进行测试，就可以得到完全正确的程序。包含所有可能情况的测试称为穷尽测试，对于实际程序而言，穷尽测试通常是不可能做到的。所以软件测试不可能发现程序中的所有错误。我们的目的是要通过测试保证软件的可靠性，因此，必须仔细设计测试方案，力争用尽可能少的测试发现尽可能多的错误。

9.1.4 软件测试与软件开发各阶段的关系

软件开发过程是一个自顶向下、逐步细化的过程，而测试过程则是自底向上、逐步集成的过程。低一级测试为上一级测试准备条件，如图 9-1 所示。首先对每个程序模块进行单元测试，消除程序模块内部在逻辑上和功能上的错误和缺陷。再对照软件设计进行集成测试，检测和排除子系统(或系统)结构上的错误。随后再对照需求分析，进行确认测试。最后，从全局出发运行系统，检验是否满足预期要求。(测试的过程将在本章第二节中详细讨论。)

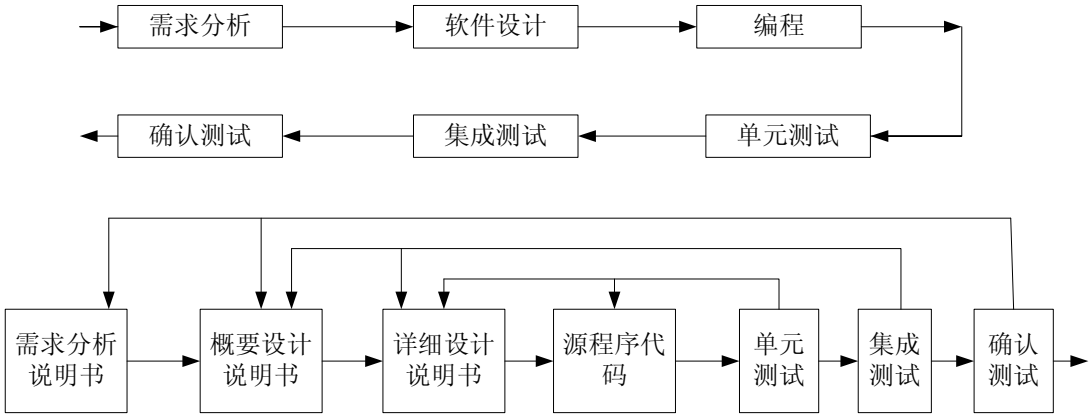

图 9-1　软件测试与软件开发的关系

与开发过程类似，每个步骤在逻辑上是前一个步骤的继续。大型软件系统通常由若干个子系统组成，每个子系统又由许多模块组成。软件系统测试的基本步骤可分为模块测试又称为单元测试、子系统测试、系统测试（不论是子系统测试还是系统测试，都兼有检测和组装两重含义，通常称为集成测试）、确认测试。关系重大的软件产品在验收之后并不立即投入生产性运行，而是再经过一段平行运行时间的考验。测试作为软件工程的一个阶段，它的根本任务是保证软件的质量，因此除了测试之外，还要做一些与测试密切相关的工作。这就是下面要讨论的内容。

9.1.5 测试信息流

测试信息流如图 9-2 所示。测试过程需要 3 类输入：

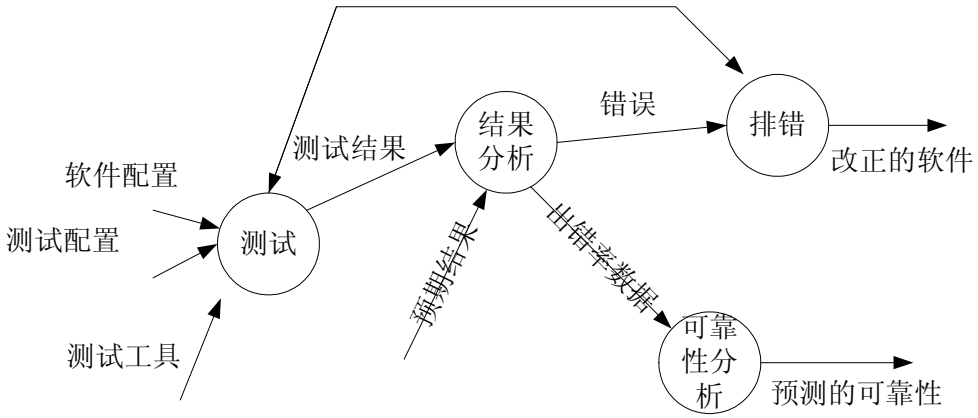

图 9-2 测试信息流

（1）软件配置　包括软件需求规格说明、软件设计规格说明、源代码等。

（2）测试配置　包括测试计划、测试用例、测试驱动程序等。

（3）测试工具　测试工具为测试的实施提供某种服务。例如，测试数据自动生成程序、静态分析程序、动态分析程序、测试结果分析程序以及驱动测试的工作等。

测试之后，用实测结果与预期结果进行比较。如发现出错的数据，就要进行调试。对已经发现的错误进行错误定位和确定出错性质，并纠正这些错误，同时修改相关的文档。修正后的文档一般都要经过再次测试，直到通过测试为止。

通过收集和分析测试结果数据，对软件建立可靠性模型。如果测试发现不了错误，那么可以肯定，测试配置考虑得不够细致充分，错误仍然潜伏在软件中。这些错误最终不得不由用户在使用中发现，并在维护时由开发者去改正。但那时改正错误的费用将比在开发阶段改正错误的费用要高若干倍。

9.1.6 错误分类

由于人们对错误有不同的理解和认识，所以目前还没有一个统一的错误分类方法。错误难于分类的原因，一方面是由于一个错误有许多征兆，因而它可以被归入不同的类。另一方面是因为把一个给定的错误归于哪一类，还与错误的来源和程序员的心理状态有关。

(1)按错误的影响和后果分类

①较小错误：只对系统输出有一些非实质性影响。例如，输出的数据格式不符要求等。

②中等错误：对系统的运行有局部影响。如输出的某些数据有错误或出现冗余。

③较严重错误：系统的行为因错误的干扰而出现明显不合情理的现象。例如，开出了 0．00 元的支票，系统的输出完全不可信赖。

④严重错误：系统运行不可跟踪，一时不能掌握其规律，时好时坏。

⑤非常严重的错误：系统运行中突然停机，其原因不明，无法软启动。

⑥最严重的错误：系统运行导致环境破坏，或是造成事故，引起生命、财产的损失。

(2)按错误的性质和范围分类

①功能错误

(a)规格说明错误：规格说明可能不完全，有二义性或自身矛盾。(b)功能错误：程序实现的功能与用户要求的不一致。这常常是由于规格说明中包含错误的功能、多余的功能或遗漏的功能所致。(c)测试错误：软件测试的设计与实施发生错误。软件测试自身也可能发生错误。(d)测试标准引起的错误：对软件测试的标准要选择适当，若测试标准太复杂，则导致测试过程出错的可能就大。

②系统错误

(a)外部接口错误：外部接口指如终端、打印机、通信线路等系统与外部环境通信的手段在使用中出错。 (b)内部接口错误：内部接口指程序之间的联系。它所发生的错误与程序内实现的细节有关。例如，设计协议错、输入／输出格式错、数据保护不可靠、子程序访问错等。(c)硬件结构错误：这类错误在于不能正确地理解硬件如何工作。例如，忽视或错误地理解分页机构、地址生成、通道容量、I／O 指令、中断处理、设备初始化和启动等而导致的出错。(d)操作系统错误：这类错误主要是由于不了解操作系统的工作机制而导致出错。当然，操作系统本身也有错误，但是一般用户很难发现这种错误。(e)软件结构错误：由于软件结构不合理或不清晰而引起的错误。这种错误通常与系统的负载有关，而且往往在系统满载时才出现。这是最难发现的一类错误。 (f)控制与顺序错误：这类错误包括忽视了时间因素而破坏了事件的顺序，猜测事件出现在指定的序列中，等待一个不可能发生的条件，漏掉先决条件，规定错误的优先级或程序状态，漏掉处理步骤，存在不正确的处理步骤或多余的处理步骤，等等。(g)资源管理错误：这类错误是由于不正确地使用资源而产生的。例如，使用未经获准的资源，使用后未释放资源，资源死锁，把资源链接在错误的队列中，等等。

③加工错误

(a)算术与操作错误：指在算术运算、函数求值和一般操作过程中发生的错误。(b)初始化错误：典型的错误有忘记初始化工作区，忘记初始化寄存器和数据区，错误地对循环控制变量赋初值，用不正确的格式、数据或类型进行初始化，等等。(c)控制和次序错误：这类错误与系统级同名错误类似，但它是局部错误。包括遗漏路径，不可达到的代码，不符合语法的循环嵌套，循环返回和终止的条件不正确，漏掉处理步骤或处理步骤有错，等等。(d)静态逻辑错误：这类错误主要包括不正确地使用 CASE 语句，在表达式中使用不正确的否定(例如用“>”代替“<”的否定)，对情况不适当地分解与组合，混淆“或”与“异或”等。

④数据错误

(a)动态数据错误：动态数据是在程序执行过程中暂时存在的数据。各种不同类型的动态数据在程序执行期间将共享一个共同的存储区域，若程序启动时对这个区域未初始化，就会导致数据出错。由于动态数据被破坏的位置可能与出错的位置在距离上相差很远，因此要发现这类错误比较困难。(b)静态数据错误：静态数据在内容和格式上都是固定的。它们直接或间接地出现在程序或数据库中。由编译程序或其他专门程序对它们做预处理。这是在程序执行前防止静态错误的好办法，但预处理也会出错。(c)数据内容错误：数据内容是指存储于存储单元或数据结构中的位串、字符串或数字。数据内容本身没有特定的含义，除非通过硬件或软件给予解释。数据内容错误就是由于内容被破坏或被错误地解释而造成的错误。(d)数据结构错误：数据结构是指数据元素的大小和组织形式。在同一存储区域中可以定义不同的数据结构。数据结构错误主要包括结构说明错误及把一个数据结构误当做另一类数据结构使用的错误。这是更危险的错误。(e)数据属性错误：数据属性是指数据内容的含义或语义。例如，整数、字符串、子程序等。数据属性错误主要包括对数据属性不正确地解释，例如错把整数当实数、允许不同类型数据混合运算而导致的错误等。

⑤代码错误

主要包括语法错误、打字错误、对语句或指令不正确理解所产生的错误。

(3)按软件生存期阶段分类

一般可把软件的逻辑错误按生存期不同阶段分为 4 类。

①问题定义(需求分析)错误

它们是在软件定义阶段，分析员研究用户的要求后所编写的文档中出现的错误。换句话说，这类错误是由于问题定义不满足用户的要求而导致的错误。

②规格说明错误

这类错误是指规格说明与问题定义不一致所产生的错误。可以将它们分成：(a)不一致性错误：规格说明中功能说明与问题定义发生矛盾。(b)冗余性错误：规格说明中某些功能说明与问题定义相比是多余的。(c)不完整性错误：规格说明中缺少某些必要的功能说明。(d)不可行错误：规格说明中有些功能要求是不可行的。(e)不可测试错误：有些功能的测试要求是不现实的。

③设计错误

这是在设计阶段产生的错误，它使系统的设计与需求规格说明中的功能说明不相符。可以细分为：(a)设计不完全错误：某些功能没有被设计，或设计得不完全。(b)算法错误：算法选择不合适。主要表现为算法的基本功能不满足功能要求、算法不可行或者算法的效率不符合要求。(c)模块接口错误：模块结构不合理，模块与外部数据库的界面不一致，模块之间的界面不一致。(d)控制逻辑错误：控制流程与规格说明不一致，控制结构不合理。(e)数据结构错误：数据设计不合理，与算法不匹配，数据结构不满足规格说明要求。

④编码错误

编码过程中的错误是多种多样的，大致有数据说明错、数据使用错、计算错、比较错、控制流错、界面错、输入 / 输出错及其他错误等几种。

在不同的开发阶段，错误的类型和表现形式是不同的，故应当采用不同的方法和策略来进行检测。

9.2 软件测试过程与策略

在上一节中对软件测试与软件开发关系进行了分析，本节主要讨论软件测试过程中的单元测试、集成测试、确认测试。

单元测试集中对用源代码实现的每个程序单元进行测试，检查各个程序模块是否正确地实现了规定的功能。然后，进行集成测试，根据软件设计规定的软件体系结构，把已测试过的模块组装起来，在组装时，检查程序结构组装的正确性。确认测试则是要检查已实现的软件是否满足了需求规格说明中所确定的各种需求，以及软件配置是否完全、正确。最后是系统测试，把已确认的软件纳入实际运行环境中，与其他系统成分组合在一起进行测试。

9.2.1 单元测试

单元测试针对程序模块，进行正确性检验的测试。其目的在于发现各模块内部可能存在的各种差错。单元测试需要从程序的内部结构出发设计测试用例。多个模块可以平行地独立进行单元测试。

（1）单元测试的内容

(a)模块接口测试：对通过被测模块的数据流进行测试。为此，对模块接口，包括参数表、调用子模块的参数、全程数据、文件输入 / 输出操作都必须检查。

(b)局部数据结构测试：设计测试用例检查数据类型说明、初始化、缺省值等方面的问

题，还要查清全程数据对模块的影响。

(c)路径测试：选择适当的测试用例，对模块中重要的执行路径进行测试。对基本执行路径和循环进行测试可以发现大量的路径错误。

(d)错误处理测试：检查模块的错误处理功能是否包含有错误或缺陷。例如，是否拒绝不合理的输入，出错的描述是否难以理解、是否对错误定位有误、是否出错原因报告有误、是否对错误条件的处理不正确，在对错误处理之前错误条件是否已经引起系统的干预，等等。

(e)边界测试：要特别注意数据流、控制流中刚好等于、大于或小于确定的比较值时出错的可能性。对这些地方要仔细地选择测试用例，认真加以测试。

此外，如果对模块运行时间有要求的话，还要专门进行关键路径测试，以确定最坏情况下和平均意义下影响模块运行时间的因素。这类信息对进行性能评价是十分有用的。

（2）单元测试的步骤

通常单元测试在编码阶段进行。在源程序代码编制完成，经过评审和验证，确认没有语法错误之后，就开始进行单元测试的测试用例设计。利用设计文档，设计可以验证程序功能、找出程序错误的多个测试用例。对于每一组输入，应有预期的正确结果。

模块并不是一个独立的程序，在考虑测试模块时，同时要考虑它和外界的联系，用一些辅助模块去模拟与被测模块相联系的其他模块。这些辅助模块就是驱动模块和桩模块。

(a)驱动模块：相当于被测模块的主程序。用它接收测试用例的测试数据，把这些数据传送给被测模块，最后输出实测结果。

(b)桩模块(又称存根模块、连接模块)：用以代替被测模块所调用的子模块。桩模块可以做少量的数据操作，不需要把子模块的所有功能都带进来，但不允许什么事情也不做。

被测模块、与被测模块相关的驱动模块和桩模块共同构成了一个“测试环境”，如图 9-3 所示。

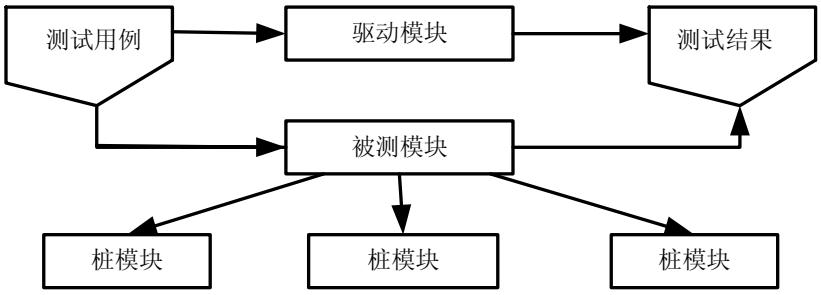

图 9-3 单元测试的测试环境

如果一个模块要完成多种功能，且以程序包或对象类的形式出现，例如 Ada 中的包、MODULA 中的模块、C++中的类。这时可以将这个模块看成由几个小程序组成。对其中的每个小程序先进行单元测试要做的工作，对关键模块还要做性能测试。对支持某些标准规程的程序，要着手进行互联测试。这种情况可称为模块测试，以区别单元测试。

为了进行单元测试必须编写测试软件，但是通常并不把它们作为软件产品的一部分交给用户。许多模块不能用简单的测试软件充分测试，为了减少开销可以使用下面将要介绍的渐增式测试方法，在集成测试的过程中同时完成对模块的详尽测试。

模块的内聚程度高可以简化单元测试过程。如果每个模块只完成一种功能，则需要的测试方案数目将明显减少，模块中的错误也更容易预测和发现。

9.2.2 集成测试

集成测试是组装软件的系统技术，即在单元测试的基础上，需要将所有模块按照设计要求组装成为系统。所以，这时需要考虑：①在把各个模块连接起来的时候，模块接口的数据是否会丢失。②一个模块的功能是否会对另一个模块的功能产生不利的影响。③各个子功能组合起来，能否达到预期要求的父功能。④全局数据结构是否有问题。⑤单个模块的误差累积起来，是否会放大，从而达到不能接受的程度。⑥单个模块的错误是否会导致数据库错误。

选择什么方式把模块组装起来形成一个可运行的系统，直接影响到模块测试用例的形式、所用测试工具的类型、模块编号的次序和测试的次序，以及生成测试用例的费用和调试的费用。通常，把模块组装成为系统的方式有两种方式：

（1）一次性集成方式

它是一种非增殖式集成方式，也叫做整体拼装。使用这种方式，首先对每个模块分别进行模块测试，然后再把所有模块组装在一起进行测试，最终得到要求的软件系统。由于程序中不可避免地存在涉及模块间接口、全局数据结构等方面的问题，所以一次试运行成功的可能性并不很大。

（2）增殖式集成方式

又称渐增式集成方式。首先对各个模块进行模块测试，然后将这些模块逐步组装成较大的系统，在组装的过程中边连接边测试，以发现连接过程中产生的问题。最后，通过增殖逐步组装成为要求的软件系统。这种方法实际上同时完成单元测试和集成测试。

（A）自顶向下的增殖方式：将模块按系统程序结构，沿控制层次自顶向下进行集成。这种增殖方式在测试过程中较早地验证了主要的控制和判断点。在一个功能划分合理的程序结构中，判断常出现在较高的层次，较早就能遇到。如果主要控制有问题，尽早发现它能够减少以后的返工。把模块结合进软件结构的具体过程由下述四个步骤完成：

第一步，对主控制模块进行测试，测试时用存根程序代替所有直接附属于主控制模块的模块；

第二步，根据选定的结合策略(深度优先或宽度优先)，每次用一个实际模块代换一个存根程序(新结合进来的模块往往又需要新的存根程序)；

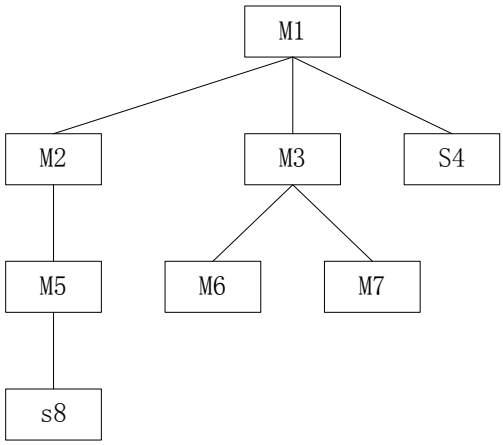

图 9-4 自顶向下结合

第三步，在结合进一个模块的同时进行测试；

第四步，为了保证加入模块没有引进新的错误，可能需要进行回归测试(即，全部或部分地重复以前做过的测试)。

从第二步开始不断地重复进行上述过程，直到构造起完整的软件结构为止。图 9-4 描绘了这个过程。假设选取深度优先的结合策略，软件结构已经部分地构造起来了，下一步桩模块 S8，将被模块 M8 取代。M8 可能本身又需要桩模块，以后这些桩模块也将被相应的模块所取代。

自顶向下的方法讲起来比较简单，但是实际使用时可能遇到逻辑上的问题。这类问题中最常见的是，为了充分地测试软件系统的较高层次，需要在较低层次上的处理。然而在自顶向下测试的初期，桩模块代替了低层次的模块，因此，在软件结构中没有重要的数据自下往上流。为了解决这个问题，测试人员有两种选择：

第一把许多测试推迟到用真实模块代替了存根程序以后再进行；

第二从层次系统的底部向上组装软件。

第一种方法失去了在特定的测试和组装特定的模块之间的精确对应关系，这可能导致在确定错误的位置和原因时发生困难。后一种方法称为自底向上的增殖方式，下面讨论这种方法。

(B)自底向上的增殖方式：从程序结构的最底层模块开始组装和测试。因为模块是自底向上进行组装，对于一个给定层次的模块，它的子模块(包括子模块的所有下属模块)已经组装并测试完成，所以不再需要桩模块。在模块的测试过程中需要从子模块得到的信息可以直接运行子模块得到。用下述步骤可以实现自底向上的结合策略：

第一 把低层模块组合成实现某个特定的软件子功能的族；

第二 编一个驱动模块(用于测试的控制模块)，协调测试数据的输入和输出；

第三 对由模块组成的子功能族进行测试；

第四 去掉驱动模块，沿软件结构自下向上移动，把子功能族组合起来形成更大的子功能族。

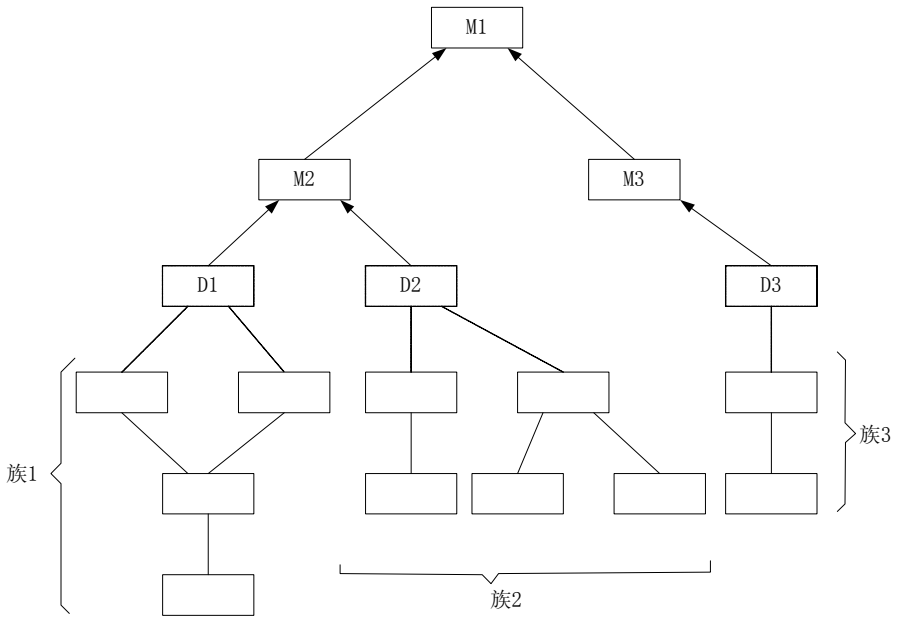

图 9-5 自底向上结合

上述第二步到第四步实质上构成了一个循环。图 9-5 描绘了自底向上的结合过程。首先把模块组合成族 1、族 2 和族 3，使用驱动程序对每个子功能族进行测试。族 1 和族 2 中的模块附属于模块 M2，去掉驱动模块 D1 和 D2，把这两个族直接同 M2 连接起来。类似地，在和模块 M3 结合之前去掉族 3 的驱动程序 D3。最终 M2 和 M3 这两个模块都与模块 M1 结合起来。

随着结合向上移动，对测试驱动模块的需要也减少了。事实上，如果软件结构的顶部两层用自顶向下的方法组装，可以明显减少驱动模块的数目，而且族的结合也将大大简化。可以看出，自底向上测试方法的优缺点与上述自顶向下测试方法的优缺点刚好相反。一般说来，纯粹自顶向下或纯粹自底向上的策略可能都不实用，人们在实践中创造出许多混合策略。

（C）混合增殖式测试

自顶向下增殖的方式和自底向上增殖的方式各有优缺点。自顶向下增殖方式的缺点是需要建立桩模块。要使桩模块能够模拟实际子模块的功能将是十分困难的。同时涉及复杂算法和真正输入／输出的模块一般在底层，它们是最容易出问题的模块，到组装和测试的后期才遇到这些模块，一旦发现问题，导致过多的回归测试。而自顶向下增殖方式的优点是能够较早地发现在主要控制方面的问题。自底向上增殖方式的缺点是“程序一直未能作为一个实体存在，直到最后一个模块加上去后才形成一个实体”。就是说，在自底向上组装和测试的过程中，对主要的控制直到最后才接触到。但这种方式的优点是不需要桩模块，而建立驱动模块一般比建立桩模块容易，同时由于涉及到复杂算法和真正输入／输出的模块最先得到组装和测试，可以把最容易出问题的部分在早期解决。此外自底向上增殖的方式可以实施多个模块的并行测试。

鉴于此，通常是把以上两种方式结合起来进行组装和测试。

①衍变的自顶向下的增殖测试：它的基本思想是强化对输入／输出模块和引入新算法模块的测试，并自底向上组装成为功能相当完整且相对独立的子系统，然后由主模块开始自顶向下进行增殖测试。

②自底向上、自顶向下的增殖测试：它首先对含读操作的子系统自底向上直至根结点模块进行组装和测试，然后对含写操作的子系统做自顶向下的组装与测试。

③回归测试：这种方式采取自顶向下的方式测试被修改的模块及其子模块，然后将这一部分视为子系统，再自底向上测试，以检查该子系统与其上级模块的接口是否适配。

9.2.3 确认测试

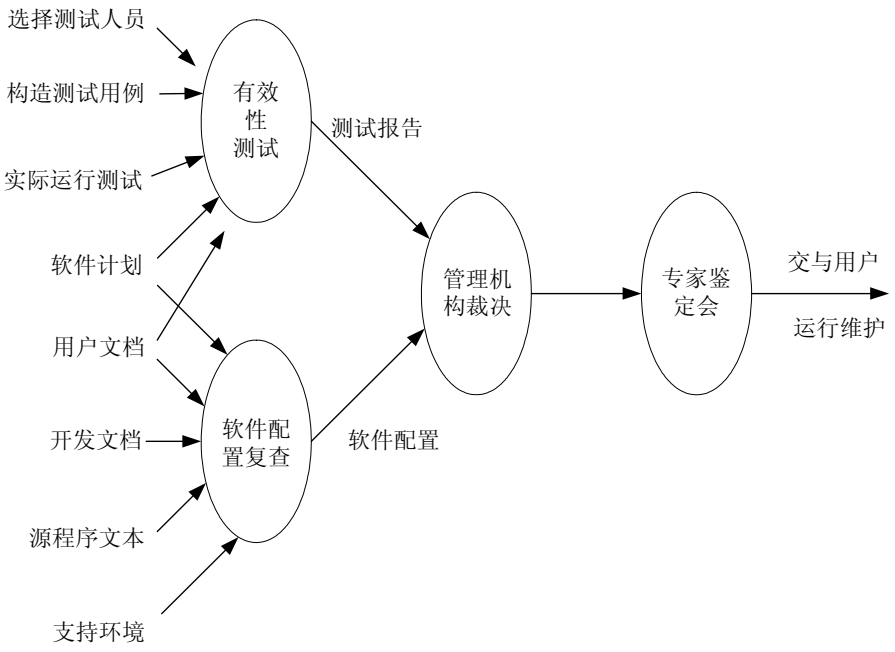

图 9-6 确定测试的步骤

经过集成测试，已经按照设计把所有模块组装成一个完整的软件系统，接口错误也已经基本排除了，接着就应该进一步验证软件的有效性，即验证软件的功能、性能及其他特性是否与用户的要求一致。这就是确认测试。在软件需求规格说明书描述了全部用户可见的软件属性，其中有一节叫做有效性准则，它包含的信息就是软件确认测试的基础。

确认测试阶段的工作如图 9-6 所示。首先，进行有效性测试以及软件配置复查；然后，进行验收测试和安装测试；在通过专家鉴定之后，才能成为可交付的软件。

① 有效性测试(功能测试)

有效性测试是在模拟的环境(可能就是开发的环境)下，运用黑盒测试的方法，验证被测软件是否满足需求规格说明书列出的需求。为此，需要首先制定测试计划，规定要做测试的种类。还需要制定一组测试步骤，描述具体的测试用例。通过实施预定的测试计划和测试步骤，确定软件的特性是否与需求相符，确保所有的软件功能需求都能得到满足，所有的软件性能需求都能达到，所有的文档都是正确且便于使用。同时，对其他软件需求，例如可移植性、兼容性、出错自动恢复、可维护性等，也都要进行测试，确认是否满足。

② 软件配置复查

软件配置复查的目的是保证软件配置的所有成分都齐全，各方面的质量都符合要求，具有维护阶段所必需的细节，而且已经编排好分类的目录。

除了按合同规定的内容和要求，由人工审查软件配置之外，在确认测试的过程中，应当严格遵守用户手册和操作手册中规定的使用步骤，以便检查这些文档资料的完整性和正确性。必须仔细记录发现的遗漏和错误，并且适当地补充和改正。

③ 验收测试

在通过了系统的有效性测试及软件配置审查之后，就应开始系统的验收测试。验收测试是以用户为主的测试。软件开发人员和 QA(Quality Assurance，质量保证)人员也应参加。由用户参加设计测试用例，使用用户界面输入测试数据，并分析测试的输出结果。一般使用生产中的实际数据进行测试。在测试过程中，除了考虑软件的功能和性能外，还应对软件的可移植性、兼容性、可维护性、错误的恢复功能等进行确认。

④ α 测试和β测试

在软件交付使用之后，用户将如何实际使用程序，对于开发者来说是无法预测的。因为用户在使用过程中常常会发生对使用方法的误解、异常的数据组合，以及产生对某些用户来说似乎是清晰的但对另一些用户来说却难以理解的输出等。如果软件是为多个用户开发的产品的时候，让每个用户逐个执行正式的验收测试是不切实际的。很多软件产品生产者采用一种称之为α测试和β测试的测试方法，以发现可能只有最终用户才能发现的错误。

α 测试是由一个用户在开发环境下进行的测试，也可以是公司内部的用户在模拟实际操作环境下进行的测试。这是在受控制的环境下进行的测试。α 测试的目的是评价软件产品的 FURPS (Function Usability Reliability Performance Support,即功能、可使用性、可靠性、性能和支持)。尤其注重产品的界面和特色。α 测试人员是除产品开发人员之外首先见到产品的人，他们提出的功能和修改意见是特别有价值的。α 测试可以从软件产品编码结束之时开始，或在模块(子系统)测试完成之后开始，也可以在确认测试过程中产品达到一定的稳定和可靠程度之后再开始。有关的手册(草稿)等应事先准备好。

β 测试是由软件的多个用户在一个或多个用户的实际使用环境下进行的测试。与α 测试不同的是，开发者通常不在测试现场。因而，β 测试是在开发者无法控制的环境下进行的软件现场应用。在β 测试中，由用户记下遇到的所有问题包括真实的以及主观认定的，定期向开发者报告，开发者在分析用户的报告之后，做出修改，最后将软件产品交付给全体用户使用。β 测试主要衡量产品的 FURPS。着重于产品的支持性，包括文档、客户培训和支持产品生产能力。只有当 α 测试达到一定的可靠程度时，才能开始β 测试。由于它处在整个测

试的最后阶段，不能指望这时发现主要问题。同时，产品的所有手册文本也应该在此阶段完全定稿。由于β测试的主要目标是测试可支持性，所以β测试应尽可能由主持产品发行的人员来管理。确认测试把软件系统作为单一的实体进行测试，测试内容与系统测试基本类似，但是它是在用户积极参与下进行的，而且可能主要使用实际数据(系统将来要处理的信息)进行测试。确认测试的目的是验证系统确实能够满足用户的需要，在这个测试步骤中发现的往往是系统需求说明书中的错误。

9.2.4 平行运行

关系重大的软件产品在验收之后往往并不立即投入生产性运行，而是要再经过一段平行运行时间的考验。所谓平行运行就是同时运行新开发出来的系统和将被它取代的旧系统，以便比较新旧两个系统的处理结果。这样做的具体目的有如下几点：

(1)可以在准生产环境中运行新系统而又不冒风险；

(2)用户能有一段熟悉新系统的时间；

(3)可以验证用户指南和使用手册之类的文档；

(4)能够以准生产模式对新系统进行全负荷测试，可以用测试结果验证性能指标。

9.3 设计测试方案

设计测试方案是测试阶段的关键技术问题。所谓测试方案包括预定要测试的功能，应该输入的测试数据和预期的结果。其中最困难的问题是设计测试用的输入数据(即，测试用例)。

不同的测试数据发现程序错误的能力差别很大，为了提高测试效率降低测试成本，应该选用高效的测试数据。因为不可能进行穷尽的测试，选用少量“最有效的”测试数据，做到尽可能完备的测试就更重要了。

设计测试方案的基本目标是，确定一组最可能发现某个错误或某类错误的测试数据。

已经研究出许多设计测试数据的技术，这些技术各有优缺点，没有哪一种是最好的，更没有哪一种可以代替其余所有技术；同一种技术在不同的应用场合效果可能相差很大，因此，通常需要联合使用多种设计测试数据的技术。

本节介绍的设计技术主要有：适用于黑盒测试的等价划分、边界值分析以及错误推测法等；适用于白盒测试的逻辑覆盖法。

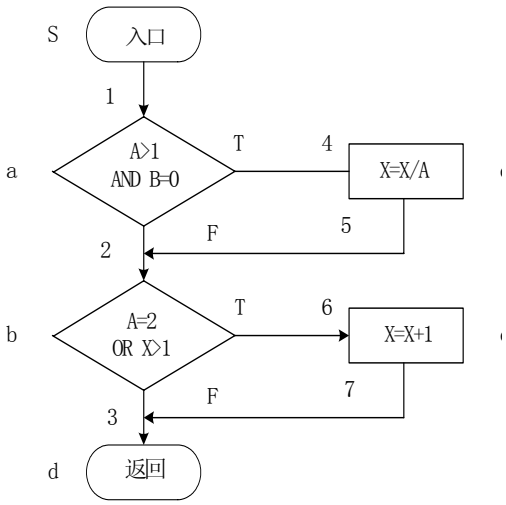

图 9-7 被测试模块的流程图

9.3.1 逻辑覆盖

有选择地执行程序中某些最有代表性的通路是对穷尽测试的唯一可行的替代办法。所谓逻辑覆盖是对一系列测试过程的总称，这组测试过程逐渐进行越来越完整的通路测试。测试数据执行(或叫覆盖)程序逻辑的程度可以划分

成哪些不同的等级呢?从覆盖源程序语句的详尽程度分析，大致有以下一些不同的覆盖标准：

1. 语句覆盖

为了暴露程序中的错误，至少每个语句应该执行一次。语句覆盖的含义是选择足够多的测试数据，使被测程序中每个语句至少执行一 次。

例如，图 9-7 是一个被测模块的流程图，它的源程序(用 PASCAL 语言书写)如下：

```
PROCEDURE  EXAMPLE(A，B：REAL；VAR  X:REAL)；
 BEGIN
  IF(A>1)AND(B=0)
  THEN  X：=X／A；
  IF(A=2)OR(X>1)
  THEN  X：＝X+1
 END；
```

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，在上面例子中两个判定条件都只测试了条件为真的情况，如果条件为假时处理有错误，显然不能发现。此外，语句覆盖只关心判定表达式的值，而没有分别测试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，以测试每个语句，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，如果程序中把第一个判定表达式中的逻辑运算符"AND"错写成“OR”，或把第二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并不能查出这些错误。

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，可以采用下述的逻辑覆盖标准。

2. 判定覆盖

判定覆盖又叫分支覆盖，它的含义是不仅每个语句必须至少执行一次，而且每个判定的每种可能的结果都应该至少执行一次，也就是每个判定的每个分支都至少执行一次。

对于上述例子来说，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判定覆盖：

I．A＝3，B=0，X＝3 (覆盖 sacbd)

II．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，例如，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

条件覆盖的含义是不仅每个语句至少执行一次，而且使判定表达式中的每个条件都取到各种可能的结果。

图 9-7 的例子中共有两个判定表达式，每个表达式中有两个条件，为了做到条件覆盖，应该选取测试数据使得在 a 点有下述各种结果出现：

A>1，A≤1，B＝0，B≠0

在 b 点有下述各种结果出现：

A＝2，A≠2，X>1，X≤1

只需要使用下面两组测试数据就可以达到上述覆盖标准：

Ⅰ．A＝2，B=0，X=4

(满足 A>1，B＝0，A＝2 和 X>1 的条件，执行路径 sacbed)

Ⅱ．A＝1，B＝1，X＝1

(满足 A≤1，B<>0，A<>2 和 X≤1 的条件，执行路径 sabd)

条件覆盖通常比判定覆盖强，因为它使判定表达式中每个条件都取到了两个不同的结果，判定覆盖却只关心整个判定表达式的值。例如，上面两组测试数据也同时满足判定覆盖标准。但是，也可能有相反的情况：虽然每个条件都取到了两个不同的结果，判定表达式却始终只取一个值。例如，如果使用下面两组测试数据，则只满足条件覆盖标准并不满足判定覆盖标准(第二个判定表达式的值总为真)：

Ⅰ．A=2，B＝0，X=1

(满足 A>1，B＝0，A＝2 和 X≤1 的条件，执行路径 sacbed)

Ⅱ．A＝1，B=1，X＝2

(满足 A≤1，B≠0，A≠2 和 X>1 的条件，执行路径 sabed)

4．判定／条件覆盖

既然判定覆盖不一定包含条件覆盖，条件覆盖也不一定包含判定覆盖，自然会提出一种能同时满足这两种覆盖标准的逻辑覆盖，这就是判定／条件覆盖。它的含义是选取足够多的测试数据，使得判定表达式中的每个条件都取到各种可能的值，而且每个判定表达式也都取到各种可能的结果。

对于图 9-7 的例子而言，下述两组测试数据满足判定／条件覆盖标准：

Ⅰ．A=2，B＝0，X＝4

Ⅱ．A=1，B＝1，X＝1

但是，这两组测试数据也就是为了满足条件覆盖标准最初选取的两组数据，因此，有时判定／条件覆盖也并不比条件覆盖更强。

5．条件组合覆盖

条件组合覆盖是更强的逻辑覆盖标准，它要求选取足够多的测试数据，使得每个判定表达式中条件的各种可能组合都至少出现一次。

对于图 9-7 的例子，共有八种可能的条件组合，它们是：

(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

和其他逻辑覆盖标准中的测试数据一样，条件组合(5)～(8)中的 X 值是指在程序流程图第二个判定框(b 点)的 X 值。

下面的四组测试数据可以使上面列出的八种条件组合每种至少出现一次：

Ⅰ．A＝2，B＝0，X＝4

(针对 1，5 两种组合，执行路径 sacbed)

Ⅱ．A=2，B＝1，X＝1

(针对 2，6 两种组合，执行路径 sabed)

Ⅲ．A＝1，B＝0，X＝2

(针对 3，7 两种组合，执行路径 sabed)

Ⅳ．A＝1，B＝1，X＝1

(针对 4，8 两种组合，执行路径 sabd)

显然，满足条件组合覆盖标准的测试数据，也一定满足判定覆盖、条件覆盖和判定 / 条件覆盖标准。因此，条件组合覆盖是前述几种覆盖标准中最强的。但是，满足条件组合覆盖标准的测试数据并不一定能使程序中的每条路径都执行到，例如，上述四组测试数据都没有测试到路径 sacbd。

以上根据测试数据对源程序语句检测的详尽程序，简单讨论了几种逻辑覆盖标准。在上面的分析过程中常常谈到测试数据执行的程序路径，显然，测试数据可以检测的程序路径的多少，也反映了对程序测试的详尽程度。从对程序路径的覆盖程度分析，能够提出下述一些主要的逻辑覆盖标准：

6．点覆盖

图论中点覆盖的概念定义如下：如果连通图 G 的子图 G′ 是连通的，而且包含 G 的所有节点，则称 G′ 是 G 的点覆盖。

按照从程序流程图导出程序图的方法，在正常情况下程序图是连通的有向图，图中每个节点相当于程序流程图的一个框(一个或多个语句)。

满足点覆盖标准要求选取足够多的测试数据，使得程序执行路径至少经过程序图中每个节点一次。显然，点覆盖标准和语句覆盖标准是相同的。

7．边覆盖

图论中边覆盖的定义是：如果连通图 G 的子图 G″ 是连通的，而且包含 G 的所有边，则称 G″ 是 G 的边覆盖。为了满足边覆盖的测试标准，要求选取足够多测试数据，使得程序执行路径至少经过程序图中每条边一次。

例如，图 9-8 是由图 9-7 得出的程序图。为了使程序执行路径经过程序图的边覆盖(1，2，3，4，5，6，7)，至少需要两组测试数据(分别执行路径 1—2—3 和 1—4—5—6—7；或分别执行路径 1—4—5—3 和 1—2—6—7)。

通常边覆盖和判定覆盖是一致的。例如，以前为满足判定覆盖标准而选取的下述两组测试数据，即可同时做到边覆盖：

Ⅰ．A＝3，B＝0，X＝3

(执行路径 1—4—5—3)

Ⅱ．A＝2，B=1，X＝1

(执行路径 1—2—6—7)

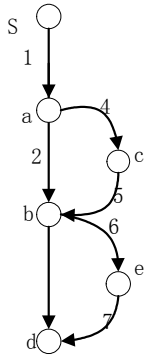

图 9-8　和图 9-7 对应的程序图

8．路径覆盖

路径覆盖的含义是，选取足够多测试数据，使程序的每条可能路径都至少执行一次(如果程序图中有环，则要求每个环至少经过一次)。

在图 9-8 的例子中共有四条可能的执行路径，它们是：1—2—3；1—2—6—7；1—4—5—3 和 1—4—5—6 一 7。因此，对于这个例子而言，为了做到路径覆盖必须设计四组测试数据。例如，下面四组测试数据可以满足路径覆盖的要求：

Ⅰ．A＝1，B＝1，X＝1(执行路径 1—2—3)

Ⅱ．A=1，b＝1，X=2(执行路径 1—2—6—7)

Ⅲ．A＝3，B＝0，X=1(执行路径 1—4—5—3)

Ⅳ．A＝2，B＝0，X＝4(执行路径 1—4—5—6—7)

路径覆盖是相当强的逻辑覆盖标准，它保证程序中每条可能的路径都至少执行一次，因此这样的测试数据更有代表性，暴露错误的能力也比较强。但是，为了做到路径覆盖只需考虑每个判定表达式的取值，并没有检验表达式中条件的各种可能组合情况。如果把路径覆盖和条件组合覆盖结合起来，可以设计出检错能力更强的测试数据。对于图 9-8 的例子，只要把路径覆盖的第三组测试数据和前面给出的条件组合覆盖的四组测试数据联合起来，共有五组测试数据，就可以做到既满足路径覆盖标准又满足条件组合覆盖标准。

9.3.2 等价划分

等价划分是用黑盒法设计测试方案的一种技术。前面讲过，穷尽的黑盒测试需要使用所有有效的和无效的输入数据来测试程序，通常这是不现实的。因此，只能选取少量最有代表性的输入数据，目的是要用较小的代价暴露出较多的程序错误。

如果把所有可能的输入数据（有效的和无效的）划分成若干个等价类，则可以做出下述假定：每类中的一个典型值在测试中的作用与这一类中所有其他值的作用相同。因此，可以从每个等价类中只取一组数据作为测试数据。这样选取的测试数据最有代表性，最可能发现程序中的错误。

使用等价划分法设计测试方案首先需要划分输入数据的等价类，为此需要研究程序的功能说明，从而确定输入数据的有效等价类和无效等价类。在确定输入数据的等价类时常常还需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。

划分等价类需要经验，下述几条启发式规则可能有助于等价类的划分：

•如果规定了输入值的范围，则可划分出一个有效的等价类(输入值在此范围内)，两个无效的等价类(输入值小于最小值或大于最大值)；

•如果规定了输入数据的个数，则类似地也可以划分出一个有效的等价类和两个无效的等价类；

•如果规定了输入数据的一组值，而且程序对不同输入值做不同处理，则每个允许的输入值是一个有效的等价类，此外还有一个无效的等价类(任一个不允许的输入值)；

• 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(从各种不同角度违反规则)；

• 如果规定了输入数据为整型，则可以划分出正整数、零和负整数等三个有效类；

• 如果程序的处理对象是表格，则应该使用空表，以及含一项或多项的表。

以上列出的启发式规则只是测试时可能遇到的情况中的很小一部分，实际情况千变万化，根本无法一一列出。为了正确划分等价类，一是要注意积累经验，二是要正确分析被测程序的功能。此外，在划分无效的等价类时还必须考虑编译程序的检错功能，一般说来，不需要设计测试数据用来暴露编译程序肯定能发现的错误。最后说明一点，上面列出的启发式规则虽然都是针对输入数据说的，但是其中绝大部分也同样适用于输出数据。

划分出等价类以后，根据等价类设计测试方案时主要使用下面两个步骤：

(1)设计一个新的测试方案以尽可能多地覆盖尚未被覆盖的有效等价类，复重这一步骤直到所有有效等价类都被覆盖为止；

(2)设计一个新的测试方案，使它覆盖一个而且只覆盖一个尚未被覆盖的无效等价类，重复这一步骤直到所有无效等价类都被覆盖为止。

注意，通常程序发现一类错误后就不再检查是否还有其他错误，因此，应该使每个测试

方案只覆盖一个无效的等价类。

下面用等价划分法设计一个简单程序的测试方案。

假设有一个把数字串转变成整数的函数。运行程序的计算机字长 16 位，用二进制补码表示整数。这个函数是用 PASCAL 语言编写的，它的说明如下：

function strtoint(dstr：shortstr)：integer；

函数的参数类型是 shortstr，它的说明是；

type shortstr＝array[1.. 6] Of char；

被处理的数字串是右对齐的，也就是说，如果数字串比六个字符短，则在它的左边补空格。如果数字串是负的，则负号和最高位数字紧相邻(负号在最高位数字左边一位)。

考虑到 PASCAL 编译程序固有的检错功能，测试时不需要使用长度不等于 6 的数组做实在参数，更不需要使用任何非字符数组类型的实在参数。

分析这个程序的规格说明，可以划分出如下等价类：

有效输人的等价类有

(1)1～6 个数字字符组成的数字串(最高位数字不是零)；

(2)最高位数字是零的数字串；

(3)最高位数字左邻是负号的数字串；

无效输入的等价类有

(4)空字符串(全是空格)；

(5)左部填充的字符既不是零也不是空格；

(6)最高位数字右面由数字和空格混合组成；

(7)最高位数字右面由数字和其他字符混合组成；

(8)负号与最高位数字之间有空格；

合法输出的等价类有

(9)在计算机能表示的最小负整数和零之间的负整数；

(10)零；

(11)在零和计算机能表示的最大正整数之间的正整数；

非法输出的等价类有

(12)比计算机能表示的最小负整数还小的负整数；

(13)比计算机能表示的最大正整数还大的正整数。

因为所用的计算机字长 16 位，用二进制补码表示整数，所以能表示的最小负整数是-32768，能表示的最大正整数是 32767。

根据上面划分出的等价类，可以设计出下述测试方案(注意，每个测试方案由三部分内容组成)：

(1)1～6 个数字组成的数字串，输出是合法的正整数。

输入：‘　　1’

预期的输出：1

(2)最高位数字是零的数字串，输出是合法的正整数。

输入：‘000001’

预期的输出：1

(3)负号与最高位数字紧相邻，输出合法的负整数。

输入：‘-00001’

预期的输出：-1

(4)最高位数字是零，输出也是零。

输入；‘000000’

预期的输出：0
(5)太小的负整数。
输入：‘-47561’
预期的输出：“错误——无效输入”
(6)太大的正整数。
输入：‘132767’
预期的输出：“错误——无效输入”
(7)空字符串。
输入：‘　　’
预期的输出：“错误——没有数字”
(8)字符串左部字符既不是零也不是空格。
输入：‘XXXXX1’
预期的输出：“错误——填充错”
(9)最高位数字后面有空格。
输入：‘　1　　2’
预期的输出:“错误——无效输入”
(10)最高位数字后面有其他字符。
输入：‘　1XX2’
预期的输出：“错误——无效输入”
(11)负号和最高位数字之间有空格。
输入：‘　-　12’
预期的输出：“错误——负号位置错”

9.3.3 边界值分析

经验表明，处理边界情况时程序最容易发生错误。例如，许多程序错误出现在下标、纯量、数据结构和循环等等的边界附近。因此，设计使程序运行在边界情况附近的测试方案，暴露出程序错误的可能性更大一些。

使用边界值分析方法设计测试方案首先应该确定边界情况，这需要经验和创造性，通常输入等价类和输出等价类的边界,就是应该着重测试的程序边界情况。按照边界值分析法，应该选取刚好等于、稍小于和稍大于等价类边界值的数据作为测试数据，而不是选取每个等价类内的典型值或任意值作为测试数据。一般,设计测试方案时总是联合使用等价划分和边界值分析两种技术。例如，为了测试前述的把数字串转变成整数的程序，除了上一小节已经用等价划分法设计出的测试方案外，还应该用边界值分析法再补充下述测试方案。

(12)使输出刚好等于最小的负整数。
输入：‘-32768’
预期的输出为：-32768
(13)使输出刚好等于最大的正整数。
输入；’32767’
预期的输出：32767

原来用等价划分法设计出来的测试方案(5)最好改为：
(5)使输出刚刚小于最小的负整数。
输入：‘-32769’
预期的输出：“错误——无效输入”

原来的测试方案(6)最好改为：

(6)使输出刚刚大于最大的正整数。

输入：‘32768’

预期的输出：“错误——无效输入”

此外，根据边界值分析方法的要求，应该分别使用长度为 0，1 和 6 的数字串作为测试数据。上一小节中设计的测试方案 1，2，3，4 和 7 已经包含了这些边界情况。

9.3.4 错误推测

使用边界值分析和等价划分技术，可以帮助我们设计出具有代表性的，因而也就容易暴露程序错误的测试方案。但是，不同类型不同特点的程序通常又有一些特殊的容易出错的情况。此外，有时分别使用每组测试数据时程序都能正常工作，这些输入数据的组合却可能检测出程序的错误。一般说来，即使是一个比较小的程序，可能的输入组合数也十分巨大，因此必须依靠测试人员的经验和直觉，从各种可能的测试方案中选出一些最可能引起程序出错的方案。对于推测程序中可能存在哪类错误，是挑选测试方案时的一个重要因素。错误推测法在很大程度上靠直觉和经验进行。它的基本想法是列举出程序中可能有的错误和容易发生错误的特殊情况，并且根据它们选择测试方案。本章第 9．2 节列出了模块中一些常见错误的清单，这些是模块测试经验的总结。对于程序中容易出错的情况也有一些经验总结出来，例如，输入数据为零或输出数据为零往往容易发生错误；如果输入或输出的数目允许变化(例如，被检索的或生成的表的项数)，则输入或输出的数目为 0 和 1 的情况(例如，表为空或只有一项)是容易出错的情况。还应该仔细分析程序规格说明书，注意找出其中遗漏或省略的部分，以便设计相应的测试方案，检测程序员对这些部分的处理是否正确。

经验还告诉我们，在一段程序中已经发现的错误数目往往和尚未发现的错误数成正比。例如，在 IBM OS／370 操作系统中，用户发现的全部错误的 47％只与该系统 4％的模块有关。因此，在进一步测试时要着重测试那些已发现了较多错误的程序段。

等价划分法和边界值分析法都只孤立地考虑各个输入数据的测试功效，而没有考虑多个输入数据的组合效应，可能会遗漏了输入数据易于出错的组合情况。利用判定表或判定树为工具是选择输入组合的一个有效途径，列出输入数据各种组合与程序应作的动作(及相应的输出结果)之间的对应关系，然后为判定表的每一列至少设计一个测试用例。

选择输入组合的另一个有效途径是把计算机测试和人工检查代码结合起来。例如，通过代码检查发现程序中两个模块使用并修改某些共享的变量，如果一个模块对这些变量的修改不正确，则会引起另一个模块出错，因此这是程序发生错误的一个可能的原因。应该设计测试方案，在程序的一次运行中同时检测这两个模块，特别要着重检测一个模块修改了共享变量后，另一个模块能否像预期的那样正常使用这些变量。反之，如果两个模块相互独立，则没有必要测试它们的输入组合情况。通过代码检查也能发现模块相互依赖的关系，例如，某个算术函数的输入是数字字符串，调用 9．3．2 节例子中的“strtoint”函数，把输入的数字串转变成内部形式的整数。在这种情况下，不仅必须测试这个转换函数，还应该测试调用它的算术函数在转换函数接收到无效输入时的响应。

9.3.5　实用测试策略

以上简单介绍了设计测试方案的几种基本方法，使用每种方法都能设计出一组有用的测试方案，但是没有一种方法能设计出全部测试方案。此外，不同方法各有所长，用一种方法

设计出的测试方案可能最容易发现某些类型的错误，对另外一些类型的错误可能不易发现。

因此，对软件系统进行实际测试时，应该联合使用各种设计测试方案的方法，形成一种综合策略。通常的做法是，用黑盒法设计基本的测试方案，再用白盒法补充一些必要的测试方案。具体地说，可以使用下述策略结合各种方法：

1．在任何情况下都应该使用边界值分析的方法。经验表明，用这种方法设计出的测试方案暴露程序错误的能力最强。注意，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．必要时用等价划分法补充测试方案。

3．必要时再用错误推测法补充测试方案。

4．对照程序逻辑，检查已经设计出的测试方案。可以根据对程序可靠性的要求采用不同的逻辑覆盖标准，如果现有测试方案的逻辑覆盖程度没达到要求的覆盖标准，则应再补充一些测试方案。

应该强调指出，即使使用上述综合策略设计测试方案，仍然不能保证测试能发现一切程序错误；但是，这个策略确实是在测试成本和测试效果之间的一个合理的折衷。通过前面的叙述可以看出，软件测试确实是一件十分艰巨繁重的工作。

最后，用一个简单例子结束这一节。

[例] 程序 TRIANGLE 读入三个整数值，这三个整数代表同一个三角形三条边的长度，程序根据这三个值判断三角形属于不等边、等腰或等边三角形中的哪一种。

综合使用边界值分析、等价划分和错误推测等技术，可以设计出下述 11 种应该测试的情况。

表 9-1 程序 TRIANGLE 的测试数据

测试功能	测试数据		
	a	b	C
1．等边	10，10，10		
2．等腰	10，10，17	10，17，10	17，10，10
3．不等边	8，10，12	8，12，10	10，12，8
4．非三角形	10，10，21	10，21，10	21，10，10
5．退化情况	10，5，5	5，10，5	5，5，10
6．零数据	0，0，0		
	0，0， 17	0，17，0	17，0，0
	0，10，12	12，0，10	12，10，0
7．负数据	-10，-10，-10		
	-10，-10，17	-10，17，-10	17，-10，-10
	-8，10，17	17，-8，10	10，17，-8
8．遗漏数据	—，—，—		
	10，—，—	—，10，—	—，—，10
	8，10，—	8，—，10	—，8，10
9．无效输入	A，B，C		
	=，+，*		
	8，10，A	8，A,10	A，10，8
	7E3，10．5，A	10.5.7E3,A	A，10，5，7E3

表 9-2　测试数据覆盖程度表

编　　号	测试数据	覆盖的边
l	10，10，10	l，2，3，4，5，6，7，8
2a	10，10，17	1，2，3，4，5，15，19，20，8
2b	10，17，10	1，2，3，4，14，18，19，20，8
2c	17，10，10	1，2，3，4，14，16，17，19，20，8
3a	8，10，12	1，2，3，4，14，16，21，22，8
3b	8，12，10	1，2，3，4，14，16，21，22，8
3c	10，12，8	1，2，3，4，14，16，21，22，8
4a	10，10，21	1，2，3，11，12，13，8
4b	10，21，10	1，2，10，12，13，8
4c	21，10，10	1，9，12，13，8

(1)正常的不等边三角形；
(2)正常的等边三角形；
(3)正常的等腰三角形，包括两条相等边的三种不同排列方法；

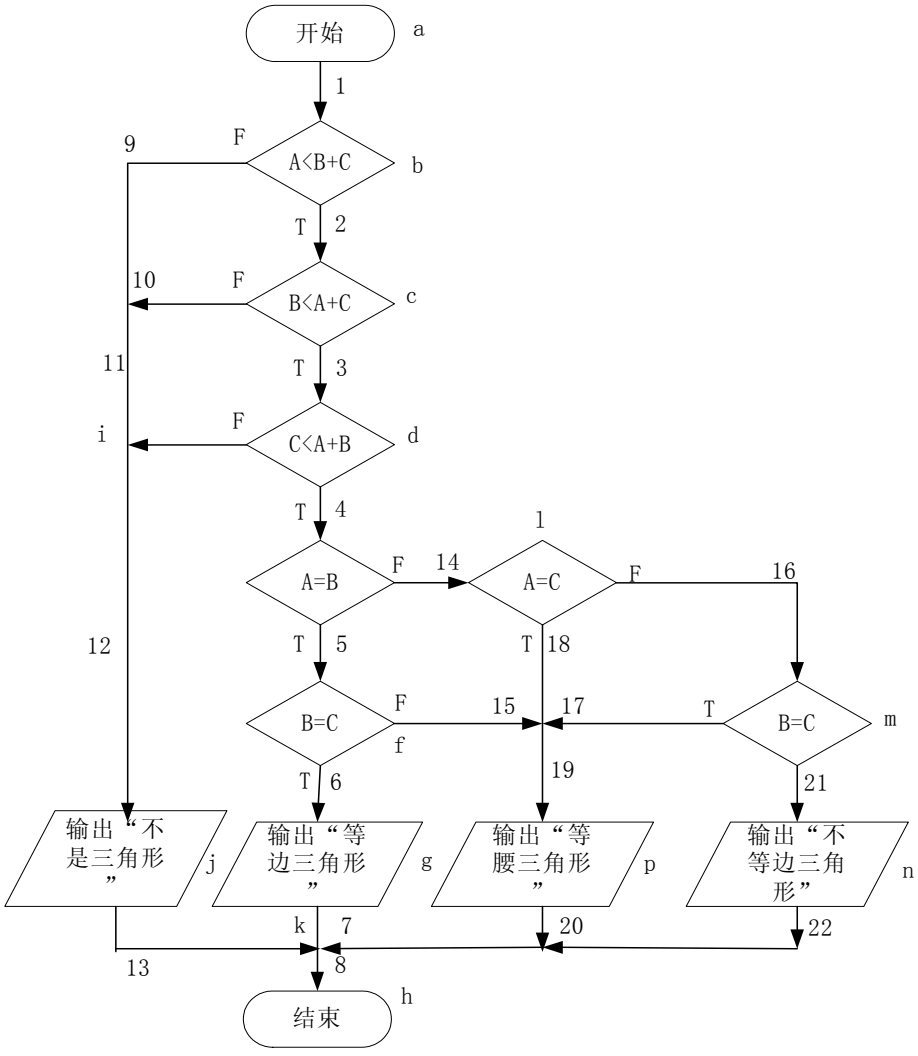

图 9-9 程序 TRIANGLE 的流程图

(4)退化的三角形(即，两边之和等于第三边)，包括三种不同排列方式；

(5)三条边不能构成三角形(即，两边之和小于第三边)，包括三种不同排列方法；

(6)一条边的长度为零，包括三种不同排列方法；

(7)两条边的长度为零，包括三种不同排列方法；

(8)三条边的长度全为零；

(9)输入数据中包含负整数；

(10)输入数据不全(不足三个正整数)；

(11)输入数据中包含非整数型的数据。

为了测试上述 11 种情况，设计出的测试数据列在表 9-1 中列出的第 1 种至第 5 种测试方案应该用不同整数值重复执行若干次，注意，这些整数值中应该包括程序可以接受的最大整数和最小整数，以及刚好大于和刚好小于这些极限值的整数。

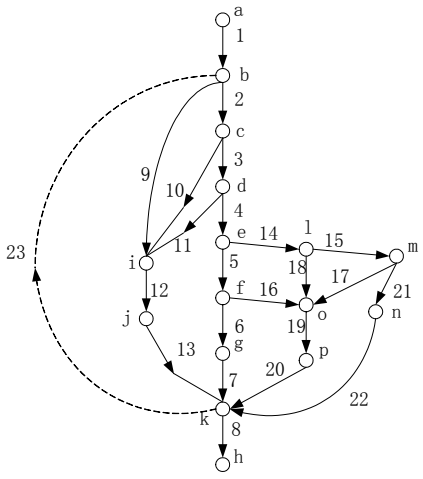

图 9-10　和图 9-9 对应的程序图

最后，检查上述测试数据的覆盖程度，通常应该做到边覆盖。为此需要知道程序的处理过程，假设程序 TRIANGlE 的流程图如图 9-9 所示，对应的程序图画在图 9-10 中。表 9-2　列出了第 1 种至第 4 种测试数据所覆盖的边。不难看出，仅仅这四种测试数据已经不仅做到了边覆盖，而且也做到了路径覆盖，也就是说，对于这个例子而言，用黑盒法设计出的测试方案已经足够多了，不需要再用白盒法补充设计测试方案了。

9.4 对 OOA 和 OOD 模型的测试

在 9.1.1 中，我们学习了测试的目标，简单地说，在实际可行的时间内应用可管理的工作去发现尽可能多的错误。虽然对面向对象软件而言，这个基本目标仍保持不变，但是 OO 程序的性质改变了测试的策略和测试战术。

也许有人会说，随着 OOA 和 OOD 的成熟，更多的设计模式复用将减轻 OO 系统的繁重测试量。准确地说，其反面才是真的。

每次复用是一个新的使用语境，要谨慎地重新测试。尤其是面向对象技术所独有的多态，继承，封装等新特点，为了获得面向对象系统的高可靠性，似乎可能将需要更多、而不是更少的测试。

为了充分地测试 OO 系统，必须做好三件事：(1)测试的定义必须扩大包括用于 OOA 和 OOD 模型的错误发现技术；(2)单元和集成测试策略必须有很大的改变；(3)测试用例的设计必须考虑 OO 软件的独特特征。

9.4.1 扩大测试的视角

面向对象软件的构造从分析和设计模型(见第七、八章)的创建开始。因为 OO 软件工程范型的演化性质，模型从对系统需求相对非正式的表示开始，逐步演化为详细的类模型、类连接和关系、系统设计和分配、以及对象设计(通过消息序列的对象连接模型)。在每个阶段，测试模型，以试图在错误传播到下一次递进前发现错误。

可以说 OO 分析和设计模型的复审是特别有用的，因为相同的语义结构(如，类、属性、

操作、消息)出现在分析、设计和代码阶段，因此，在分析阶段发现的类属性定义中的问题将影响设计或编码阶段，或甚至到下一次分析迭代才被发现并带来副作用。例如，考虑一个类，在第一次 OOA 迭代时定义了其中一系列属性，一个外来的无关属性被附于该类(由于对问题域的错误理解)，然后定义两个操纵该属性的操作。在复审时，一个领域专家指出该问题。通过在本阶段删除该无关属性，可在分析过程中避免下面的问题：

1．特殊的子类可能会产生无关的属性。应避免涉及不必要的子类创建工作。

2．对类定义的错误解释可能导致不正确的或无关的类关系。

3．为适应某些无关属性，可能会引起系统或它的类行为的不适当描述。

如果问题未在分析过程中被发现并进一步向前传播，在设计中可能产生下面的问题：

A．在系统设计阶段可能将类不合适的分配到子系统或任务。

B．可能花费不必要的工作去创建针对无关属性的操作的过程设计。

C．消息模型将是不正确的(因为必须为无关的操作设计消息)。

如果问题在设计阶段仍未被检测到，并传到编码活动中，则将花费大量的工作努力和精力去生成那些实现一个不必要的属性、一个不必要的操作、驱动对象间通信的消息以及很多其他相关问题的代码。此外，类的测试将花费更多的时间。一旦问题最终被发现，必须对系统进行修改，从而引致由于修改而产生到作用的很大的潜在可能性。

在它们的开发的后面阶段，OOA 和 OOD 模型提供了关于系统的结构和行为的实质性信息，为此，这些模型应该在生成代码前经受严格的复审。

所有面向对象模型应该被测试(在这个语境内，术语“测试”代表正规的技术复审)，以保证在模型的语法、语义和语用的语境内的正确性、完整性和一致性。

9.4.2 测试 OOA 和 OOD 模型

分析和设计模型不能进行传统意义上的测试，因为它们不能被执行。然而，可用正式的技术复审检查分析和设计模型的正确性和一致性。

1．OOA 和 OOD 模型的正确性

用于表示分析和设计模型的符号体系和语法是为了和项目选定的特定分析和设计方法相联系的，因此，语法正确性基于符号是否合适使用，而且对每个模型复审以保证保持合适的建模约定。

在分析和设计阶段，语义正确性必须基于模型对现实世界问题域的符合度来判断，如果模型精确地反应了现实世界(到这样一个细节程度，它对模型复审时所处的开发阶段是合适的)，则它语义是正确的。为了确定模型是否确实在事实上反应了现实世界，应该将它送给问题域专家，专家将检查类定义和类层次以发现遗漏和含混。评估类关系(实例连接)以确定它们是否精确地反应了现实世界的对象连接。

2．OOA 和 OOD 模型的一致性

对 OOA 和 OOD 模型的一致性判断可以通过考虑模型中实体间的关系。 为了评估一致性，应该检查每个类及其和其他类的连接。可运用类—责任—协作者(CRC)模型和对象—关系图。如我们在面向对象分析中提到，CRC 模型由 CRC 索引卡片构成，每个 CRC 卡片列出类名、类的责任(操作)、以及其协作者(其他类，类向它们发送消息并依赖于它们完成自己的责任)。协作蕴含了在 OO 系统的类之间的一系列关系(即，连接)，对象关系模型提供了类之间连接的图形表示。所有这些信息可以从 OOA 模型(见第七章面向对象分析)得到。

为了评估类模型，推荐采用以下面步骤：

1．再次考察 CRC 模型和对象—关系模型，进行交叉检查以保证由 OOA 模型所蕴含的协作适当地反应在二者中。

2．检查每个 CRC 索引卡片的描述以确定是否某被受权的责任是协作者的定义的一部分，例如，考虑为某 POS 结账系统定义的类，称为 credit sale，该类的 CRC 卡片如表 9.3 所示。

对于这组类和协作，我们问如果某责任(如，read credit card)被委托给指定的协作者(credit card)，是否完成该责任。即，类 credit card 是否可读。在本例的情形下，我们的回答 “是”。遍历对象关系以保证所有这样的连接是有效的。

3．反转该连接以保证每个被请求服务的协作者正在接收来自合理源的请求，例如，如果 credit card 类接收来自 credit sale 类的 purchase amount 请求，则将会有问题，因为 credit card 并不知道 purchase amount。

4．使用在第 3 步检查的反转连接，确定是否可能需要其他的类或责任是否被合适地在类间分组。

5．确定是否被广泛请求的责任可被组合为单个的责任，例如，read credit card 和 get authorization 在每种情况均发生，它们可以被组合为 validate credit request 责任，它结合了获取信用卡号及获得授权。

6．步骤 1 到 5 被迭代地应用到每个类，并贯穿 OOA 模型的每次演化。

表 9-3 一个用于复审的 CRC 卡片

类名: credit sale	
类的类型:交易事件	
类的特征:不确定的,原子的,顺序的,永久的,受保护的	
责任:	协作者:
Read creadit card	Credit card
Get authorizition	Credit authority
Post purchase amount	Product ticket
	Sales ledger
	Autdit file
Generate bill	bill

一旦已经创建了设计模型(第八章面向对象设计)，也应该进行对系统设计和对象设计的复审。系统设计描述了构成产品的子系统、子系统被分配到处理器的方式、以及类到子系统的分配。对象模型表示了每个类的细节和实现类间的协作所必需的消息序列活动。

通过检查在 OOA 阶段开发的对象—行为模型，和映射需要的系统行为到被设计用于完成该行为的子系统来进行系统设计的复审。也在系统行为的语境内复审并发性和任务分配，评估系统的行为状态以确定哪些行为并发地存在。

对象模型应该针对对象—关系网络来测试，以保证所有设计对象包含实现为每张 CRC 索引卡片定义的协作所必须的属性和操作。此外，使用传统的检查技术来实现操作细节的详细规约(即，实现操作的算法)。

9.5 面向对象的测试策略

传统的测试计算机软件的策略是从“小型测试”开始，逐步走向“大型测试”。用软件

测试的行话来陈述，我们从单元测试开始，然后逐步进入集成测试，最后是有效性和系统测试。在传统应用中，单元测试集中在最小的可编译程序单位——子程序(如，模块、子例程、进程)，一旦这些单元均被独立测试后，它被集成进程序结构中，这时要进行一系列的回归测试以发现由于模块的接口所带来的错误和新单元加入所导致的副作用，最后，系统被作为一个整体测试以保证发现在需求中的错误。

9.5.1 在 00 语境中的单元测试

当考虑面向对象软件时，单元的概念发生了变化。封装驱动了类和对象的定义，这意味着每个类和类的实例(对象)包装了属性(数据)和操纵这些数据的操作(也称为方法或服务)。而不是个体的模块。最小的可测试单位是封装的类或对象，类包含一组不同的操作，并且某特殊操作可能作为一组不同类的一部分存在，因此，单元测试的意义发生了较大变化。

我们不再孤立地测试单个操作(传统的单元测试观点)，而是将操作作为类的一部分。作为一个例子，考虑—个类层次，其中操作 A 针对超类定义并被一组子类继承，每个子类使用操作 A，但是它被应用于为每个子类定义的私有属性和操作的环境内。因为操作 A 被使用的语境有微妙的不同，有必要在每个子类的语境内测试操作 A。这意味着在面向对象的语境内在真空中测试操作 A(即传统的单元测试方法)是无效的。

对 OO 软件的类测试等价于传统软件的单元测试。和传统软件的单元测试不一样，它往往关注模块的算法细节和模块接口间流动的数据，OO 软件的类测试是由封装在类中的操作和类的状态行为所驱动的。

9.5.2 在 00 语境中的集成测试

因为面向对象软件没有层次的控制结构，传统的白顶向下和自底向上集成策略就没有意义，此外，—次集成一个操作到类中(传统的增量集成方法)一般是不可能的，这是由于构成类的成分的直接和间接的交互。

对 OO 软件的集成测试有两种不同策略，第一种称为基于线程的测试，集成对回应系统的一个输入或事件所需的一组类，每个线程被集成并分别测试，应用回归测试以保证没有产生副作用。第二种称为基于使用的测试，通过测试那些几乎不使用服务器类的类(称为独立类)而开始构造系统，在独立类测试完成后，下一层的使用独立类的类，称为依赖类，被测试。这个依赖类层次的测试序列一直持续到构造完整个系统。序列和传统集成不同，要尽可能避免使用驱动程序和桩模块（9．2．2)作为替代操作。

集群测试是 OO 软件集成测试的一步，这里一群协作类(通过检查 CRC 和对象一关系模型而确定的)通过设计试图发现协作中的错误的测试用例而被测试。

9.5.3 在 00 语境中的有效性测试

在有效性或系统层次，类连接的细节消失了。和传统有效性一样，OO 软件的有效性集中在用户可见的动作和用户可识别的系统输出。为了协助有效性测试的导出，测试员应该利用作为分析模型一部分的使用实例(面向对象分析一章)，使用实例提供了在用户交互需求中很可能发现错误的一个场景。传统的黑盒测试方法(9．1．3)可被用于驱动有效性测试，此外，测试用例可以从对象—行为模型和作为 OOA 的一部分的事件流图中导出。

9.6 OO 软件的测试用例设计

OO 软件的测试用例设计方法还正处于成型期，然而，对 OO 测试用例设计的整体方法可用：

1. 每个测试用例应该被唯一标识，并且和将被测试的类显式地相关联。
2. 应该陈述测试的目的。
3. 对每个测试应该开发一组测试步骤，应该包含：
   - 将被测试的对象的一组特定状态。
   - 将作为测试的结果使用的一组消息和操作。
   - 当测试对象时可能产生的一组例外。
   - 一组外部条件(即，为了适当地进行测试而必须存在的软件的外部环境的变化)。
   - 辅助理解或实现测试的补充信息。

和传统测试用例设计不同，传统测试是由软件的输入—加工—输出视图或个体模块的算法细节驱动的，面向对象测试关注于设计合适的操作序列以测试类的状态。

9.6.1 OO 概念的测试用例设计的含义

我们已经看到 OO 类是测试用例设计的目标。因为属性和操作是被封装的，对类之外操作的测试通常是徒劳的。虽然封装是 OO 的本质设计概念，但是它可能会成为测试的障碍，测试需要对对象的具体和抽象状态的报告。然而，封装却使得这些信息在某种程度上难于获得。除非提供了内置操作来报告类属性的值，否则，难于获得对对象的状态快照。

继承也造成了对测试用例设计者的挑战。我们已知道，即使是彻底复用的，对每个新的使用语境也需要重测试。此外，多重继承增加了需要测试的语境数量，从而使测试进一步复杂化。如果从超类导出的子类被用于相同的问题域，有可能对超类导出的测试用例集可以用于子类的测试，然而，如果子类被用于完全不同的语境，则超类的测试用例将没有多大用处，必须设计新的测试用例集。

9.6.2 传统测试用例设计方法的可用性

在 9.1.3 中描述的白盒测试方法可用于对为类定义的操作的测试，基本路径、循环测试或数据流技术可以帮助保证已经测试了操作中的每一条语句，然而，很多类操作的简洁结构导致某些人认为：将用于白盒测试的工作量用于类级别的测试可能会更好。

黑盒测试方法就象对传统软件工程方法开发的系统和对 OO 系统同样适用的，如我们在前面看到的 use cases，可以为黑盒及基于状态的测试的设计提供有用的输入。

9.6.3 基于故障的测试

在 OO 系统中基于故障的测试的目标是设计最有可能发现似乎可能的故障的测试。因为产品或系统必须符合客户需求，因此，完成基于故障的测试所需的初步计划是从分析模型开始。

测试员查找似乎可能的故障(即，系统实现中有可能产生错误的方面)，为了确定是否存在这些故障，设计测试用例以测试设计或代码。

考虑一个简单的例子。软件工程师经常在问题的边界处犯错误，例如，当测试 SQRT 操作(该操作对负数返回错误)时，我们尝试边界：一个靠近零的负数和零本身，“零本身”用于检查是否程序员犯了如下错误：

IF(X>0) calculate the _square_root()；

而不是正确的：

IF(X>=0) calculate the _square_root()；

作为另一个例子，考虑布尔表达式：

IF(a&&！b‖c)

多条件测试和相关的用于探查在该表达式中可能存在的故障的技术，如：

“&&”应该是”‖”

“!”在需要处被省去应该有括号包围“!b‖”

对每个可能的故障，我们设计迫使不正确的表达式失败的测试用例。在上面的表达式中，(a=0，b=0，c=0)将使得表达式得到预估的“假”值，如果“&&”已改为”‖”，则该代码做了错误的事情，有可能分叉到错误的路径。

当然，这些技术的有效性依赖于测试员如何感觉“似乎可能的故障”，如果 OO 系统中的真实故障被认为“难以置信的”，则本方法实质上不比任何随机测试技术好，然而，如果分析和设计模型可以提供对什么可能出错的深入洞察，则，基于故障的测试可以以相当低的工作量花费来发现大量的错误。

集成测试在消息连接中查找似乎可能的故障，在此语境下，会遇到三种类型的故障：未期望的结果、错误的操作／消息使用、不正确的调用。为了在函数(操作)调用时确定似乎可能的故障，必须检查操作的行为。

集成测试，对象的“行为”通过其属性被赋予的值而定义，测试应该检查属性以确定是否对对象行为的不同类型产生合适的值。

应该注意，集成测试试图在客户对象，而不是服务器对象中发现错误，用传统的术语来说，集成测试的关注点是确定是否调用代码中存在错误，而不是被调用代码中。用调用操作作为线索，这是发现实施调用代码的测试需求的一种方式。

9.6.4 OO 编程对测试的影响

根据 OOP 的方法，面向对象编程可能对测试有几种方式的影响：

- 某些类型的故障变得几乎不可能(不值得去测试)
- 某些类型的故障变得更加可能(值得进行测试)
- 出现某些新的故障类型

当调用一个操作时，可能很难确切知道执行什么代码，即，操作可能属于很多类之一。同样，也很难确定准确的参数类型／类，当代码访问参数时，可能得到一个未期望的值。

可以通过考虑如下的传统的函数调用来理解这种差异：

x=func(y)；

对传统软件，测试员需要考虑所有属于 func 的行为，其他则不需考虑。在 OO 语境中，测试员必须考虑 base：：func()、of derived：：func()等行为。每次 func 被调用，测试员必须考虑所有不同行为的集合，如果有了好的 OO 设计习惯并且限制了在超类和子类(用 C++的术语，称为基类和派生类)间的差异，则这是较为容易的。

测试 OO 的类操作类似于测试一段代码，它设置函数参数，然后调用该函数。继承是一

种方便的生成多态操作的方式，在调用点，关心的不是继承，而是多态。继承确实使得对测试需求的搜索更为直接。

由于 OO 系统的体系结构和构造，是否某些类型的故障更加可能，而其他类型的故障则几乎不可能吗?对 OO 系统而言，回答是“是”。例如，因为 OO 操作通常是较小的，往往存在更多的集成工作和更多的集成故障的机会，集成故障变得更加可能。

9.6.5 测试用例和类层次

如上节所述，继承并没有排除对所有派生类进行全面测试的需要，事实上，它确实使测试过程变得复杂。

考虑下面情形，类 base 包含了操作 inherited 和 redefined，类 derived 时 redefined 重定义以用于局部语境中，毫无疑问，必须测试 derived::redefined()操作，因为它表示了新的设计和新的代码。但是，必须重测试 derived::inherited()操作吗?

如果 derived::inherited()调用 redefined，而 redefined 的行为已经改变，derived::inherited()可能错误地处理这新行为，因此，即使其设计和代码没有改变，它需要被重新测试。然而，重要的是要注意，仅仅必须执行 derived::inherited()的所有测试的一个子集。如果 inherited 的设计和代码部分不依赖于 redefined(即，不调用它或任意间接调用它的代码)，则不需要在 derived 类中重测试该代码。

base::redefined()和 derived::redefined()是具有不同规约和实现的两个不同的操作，它们各自具有一组从规约和实现导出的测试需求，这些测试需求探查似乎可能的故障：集成故障、条件故障、边界故障等等。但是，操作可能是相似的，它们的测试需求的集合将交迭，OO 设计得越好，交迭就越大，仅仅需要对那些不能被 base::redefined()测试满足的 derived::redefined()的需求来导出新测试。

小结一下，base::redefined()测试被应用于类 derived 的对象，测试输入可能同时适合于 base 和 derived 类，但是，期望的结果可能在 derived 类中有所不同。

9.6.6 基于场景的测试设计

基于故障的测试忽略了两种主要的错误类型：(1)不正确的规约, (2)子系统间的交互。当和不正确的规约关联的错误发生时，产品不做用户希望的事情，它可能做错误的事情，或它可能省略了重要的功能。在任一情形下，质量(对要求的符合度)均受到影响。当一个子系统建立环境(如事件、数据流)的行为使得另一个子系统失败时，发生和子系统交互相关联的错误。

基于场景的测试关心用户做什么而不是产品做什么。它意味着捕获用户必须完成的任务(通过使用实例)，然后应用它们或它们的变体作为测试。

场景揭示交互错误，为了达到此目标，测试用例必须比基于故障的测试更复杂和更现实。基于场景的测试往往在单个测试中处理多个子系统(用户并不限制他们自己一次只用一个子系统)。

例如，考虑对文本编辑器的基于场景的测试的设计，下面是使用实例：

使用实例：确定最终草稿

背景：打印“最终”草稿、阅读它并常发现某些从屏幕上看是不明显的错误。

该使用实例描述当此事发生时产生事件的序列。

1．打印完整的文档。

2． 在文档中移动，修改某些页面。

3．当每页被修改后，打印它。

4．有时打印一系列页面。

该场景描述了两件事：测试和特定的用户需要。用户需要是明显的：(1)打印单页的方法 (2)打印一组页面的方法。当测试进行时，有必要在打印后测试编辑(以及相反)。测试员希望发现打印功能导致了编辑功能的错误，即，此两个软件功能不是合适的相互独立的。

使用实例：打印新拷贝

背景：某人向用户要求文档的一份新拷贝，它必须被打印。

1．打开文档。

2．打印文档。

3．关闭文档。

测试方法也是相当明显的，除非该文档未在任何地方出现过，它是在早期的任务中创建的，该任务对现在的任务有影响吗?在很多现代的编辑器中，文档记住它们上一次被如何打印，缺省情况下，它们下一次用相同的方式打印。在“确定最终草稿”场景之后，仅仅在菜单中选择“Print”并对话框里点击“Print”按钮，将使得上次修正的页面再打印一次，这样，按照编辑器，正确的场景应该是：使用实例,打印新拷贝。

1．打开文档。

2．选择菜单中的“Print”。

3．检查你是否将打印一系列页面，如果是，点击打印完整的文档。

4．点击“Print”按钮。

5．关闭文档。

但是，这个场景指明了一个潜在的规约错误，编辑器没有做用户希望它做的事。客户经常忽略在第 3 步中的检查，当他们走到打印机前发现只有一页，而他们需要 100 页时，问题就出来了，客户指出这一规约错误。

测试用例的设计者可能在测试设计中忽略这种依赖,但是,有可能在测试中会出现问题，测试员必须克服可能的情况。

9.6.7 测试表层结构和深层结构

表层结构指 OO 程序的外部可观察的结构，即，对终端用户立即可见的结构。不是处理函数，而是很多 OO 系统的用户可能被给定一些以某种方式操纵的对象。但是不管接口是什么，测试仍然基于用户任务进行。捕获这些任务涉及到理解、观察以及和代表性用户(以及很多值得考虑的非代表性用户)的交谈。

在细节上一定存在某些差异。例如，在传统的具有面向命令的界面的系统中，用户可能使用所有命令的列表作为检查表。如果不存在执行某命令的测试场景，测试可能忽略某些用户任务(或具有无用命令的界面)。在基于对象的界面中，测试员可能使用所有的对象列表作为检查表。

当设计者以一种新的或非传统的方式来看待系统时，则可以得到最好的测试。例如，如果系统或产品具有基于命令的界面，则当测试用例设计者假设操作是独立于对象的，将可以得到更彻底的测试。提出这样的问题：当使用打印机工作时，用户有可能希望使用该操作(它仅应用于扫描仪对象)吗?不管界面风格是什么，针对表层结构的测试用例设计应该同时使用对象和操作作为导向被忽视任务的线索。

深层结构指 OO 程序的内部技术细节，即通过检查设计和代码而理解的结构。深层结构测试被设计用以测试作为 OO 系统的子系统和对象设计(面向对象设计一章)的一部分而建立

的依赖、行为和通信机制。

分析和设计模型被用作深层结构测试的基础。例如，对象—关系图或子系统协作图描述了在对象和子系统间的可能对外不可见的协作。那么测试用例设计者会问：我们已经捕获了某些测试任务，它测试在对象—关系图或子系统协作图中记录的协作?如果没有，为什么?

类层次的设计表示提供了对继承结构的误入洞察，继承结构被用在基于故障的测试中。考虑如下一种情形：一个命名为 caller 的操作只有一个参数，并且该参数是到某基类的引用。当 caller 被传递给派生类时将发生什么事情?可能影响 caller 的行为有什么差异?对这些问题的回答可能导向特殊测试的设计。

9.7 其他专门环境要求的测试

随着计算机软件变得更为复杂，对特殊测试方法的需求也增加了。白盒和黑盒测试方法可以用于一些环境、体系结构和应用程序，但是有时还是需要专门的指南和方法。本节讨论用于软件工程师常见的特定环境、体系结构和应用程序的测试指南。

9.7.1 GUI 测试

图形用户界面(GUl)对软件工程师提出了有趣的挑战，因为 GUI 开发环境有可复用的构件，开发用户界面更加省时而且更加精确，同时，GUI 的复杂性也增加了，从而增加了设计和执行测试用例的难度。因为现代 GUI 有相同的观感，已经有一序列标准的测试。下列问题可以作为常见 GUI 测试的指南：

对于窗口：

- 窗口能否基于相关的输入或菜单命令适当地打开?
- 窗口能否改变大小、移动和滚动?
- 窗口中的数据内容能否用鼠标、功能键、方向箭头和键盘访问?
- 当被覆盖并重调用后，窗口能否正确地再生?
- 需要时能否使用所有窗口相关的功能?
- 所有窗口相关的功能是可操作的吗?
- 是否有相关的下拉式菜单、工具条、滚动条、对话框、按钮、图标和其他控制可为窗口所用，并适当地显示?
- 显示多个窗口时，窗口的名称是否被适当地表示?
- 活动窗口是否被适当地加亮?
- 如果使用多任务，是否所有的窗口被实时更新?
- 多次或不正确按鼠标是否会导致无法预料的副作用?
- 窗口的声音和颇色提示和窗门的操作顺序是否符合需求?
- 窗口是否正确地关闭?

对子下拉式菜单和鼠标操作：

- 菜单条是否显示在合适的语境中?
- 应用程序的菜单条是否显示系统相关的特性(如时钟显示)?
- 下拉式操作能正确工作吗?
- 菜单、调色板和工具条是否工作正确?
- 是否适当地列出了所有的菜单功能和下拉式子功能?
- 是否可以通过鼠标访问所有的菜单功能?

- 文本字体、大小和格式是否正确?
- 是否能够用其他的文本命令激活每个菜单功能?
- 菜单功能是否随当前的窗口操作加亮或变灰?
- 菜单功能是否正确执行?
- 菜单功能的名字是否具有自解释性?
- 菜单项是否有帮助，是否语境相关?
- 在整个交互式语境中，是否可以识别鼠标操作?
- 如果要求多次点击鼠标，是否能够在语境中正确识别?
- 如果鼠标有多个按钮，是否能够在语境中正确识别?
- 光标、处理指示器和识别指针是否随操作恰当地改变?

对于数据项：

- 字母数字数据项是否能够正确回显，并输入到系统中?
- 图形模式的数据项(如滑动条)是否正常工作?
- 是否能够识别非法数据?
- 数据输入消息是否可理解?

因为 GUI 操作相关的排列数很大，所以应当用自动化工具进行测试。近几年来市场上已有不少的 GUI 测试工具。

9.7.2 测试文档和帮助设施

术语“软件测试”造成一种假象，即测试用例是为程序及其操纵的数据准备的。回忆一下软件定义，要注意到测试必须扩展到软件的第三个元素——文档。

文档错误会同数据和代码错误一样给程序带来灾难性后果。为此，文档测试也是每个软件测试中有意义的一部分。文档测试可以分为两个步骤。第一步为正式的技术复审(见软件质量保证一章)，检查文档的编辑错误；第二步是活性测试，结合实际程序的使用而使用文档。

活性测试可使用类似于黑盒测试方法，基于图的测试可用于描述程序的使用。等价划分和边界值分析可以定义输入类型及其相关的交互，然后就可以按照文档跟踪程序的使用：

- 文档是否精确描述了如何使用各种使用模式?
- 交互顺序的描述是否精确?
- 例子是否精确?
- 术语、菜单描述和系统响应是否与实际程序一致?
- 是否能够很方便地在文档中定位指南?
- 是否能够很方便地使用文档排除错误?
- 文档的内容和索引是否精确完整?
- 文档的设计(布局、缩进和图形)是否便于信息的理解?
- 显示给用户的错误信息是否有更详细的文档解释?
- 如果使用超级链接，超级链接是否精确完整?

回答这些问题最可行的方法是按照程序的使用测试文档。标出所有错误和模糊的地方，以便重写。

9.7.3 实时系统测试

很多实时系统的时间依赖性和异步性给测试带来新的困难——时间。测试用例的设计者考虑的不仅是白盒和黑盒测试用例，而且包括事件处理(如中断处理)，数据的时间安排以及处理数据的任务(进程)的并发性。很多情况下，提供的测试数据有时使得实时系统在某状态下可以正常运行，而同样的数据在系统处于不同状态时有时又会导致错误。

例如，控制复印机的实时软件在机器复印时接收操作员的中断(如操作员按某些键如“reset”)不会产生错误，但是如果在夹纸时，按同样的键就会产生一个诊断代码，指明夹纸的位置信息将被丢失。

另外，实时系统的软件和硬件之间的密切关系也会导致测试问题，软件测试必须考虑硬件故障对软件处理的影响，这种故障很难实时仿真。

实时系统的综合性测试用例设计方法还有待进一步发展，但是，仍然已有了大致的四步策略：1 任务测试:测试实时系统的第一步是独立地测试各个任务。即对每个任务设计白盒和黑盒测试用例，并在测试时执行每个任务。任务测试能够发现逻辑和功能错误，但是不能发现时间和行为错误。2 行为测试:利用 CASE 工具创建的软件模型，就可能仿真实时系统，并按照外部事件的序列检查其行为,这些分析活动可作为创建实时系统时设计测试用例的基础。使用类似于等价划分的技术，可以对事件(如中断、控制信号和数据)分类测试，例如，复印机的事件可能是用户中断(如重置计数)、机器中断(如卡纸)、系统中断(如缺粉)和故障模式(如过热)。每种事件都可以独立测试，并且检查可执行系统的行为以检测是否有与事件处理相关的继发性错误。测试每种事件以后，以随机顺序和随机频率将事件传给系统，检查系统行为看是否有行为错误。3 任务间测试:在隔离了任务内部和系统行为错误以后，测试就要转向时间相关的错误。用不同的数据率和处理负载来测试与其他任务通讯的异步任务，看任务间的同步是否会产生错误。另外，测试通过消息队列和数据存储进行通讯的任务，以发现这些数据存储区域大小方面的错误。4 系统测试: 集成软件和硬件，并进行大范围的系统测试，以发现软件 / 硬件接口间的错误。

很多实时系统处理中断，所以，测试布尔事件的处理尤其重要。利用状态变迁图和控制规约，测试者可开发一系列可能的中断及其将发生的处理，设计测试用例以验证如下的系统特性：

- 是否能够正确赋予和处理中断的优先权?
- 每个中断的处理是否正确?
- 中断处理的性能(如处理时间)是否符合需求?
- 关键时刻有大量中断时，是否会导致功能和性能上的问题?

另外，也测试应当作为中断处理一部分的传输信息的全局数据区域，以评估潜在的副作用。

小　结

软件测试的目的是发现错误。为了完成这个目标，需要计划和进行一系列的测试步骤——单元、集成、确认和系统测试。单元测试和集成测试侧重于验证一个模块的功能和把模块集成到程序结构中去。确认测试用来验证软件需求，系统测试在软件集成为一个大的系统时才进行。

每一个测试步骤都是通过一系列有助于测试用例设计的系统化测试技术来完成的。更高质量的软件则需要更系统化的测试方法。我们所需要的是贯穿整个测试过程的一个整体

策略，而且在方法学上应当象基于分析、设计和编码的系统化软件开发一样地进行周密的计划。有两种不同的测试用例设计技术：白盒测试和黑盒测试。白盒测试注重于程序控制结构。测试用例要保证测试时程序的所有语句至少执行一次，而且检查了所有的逻辑条件。黑盒测试扩大了测试焦点，黑盒测试发现功能需求错误，而不考虑程序的内部工作。基于图的测试方法检查程序对象的行为及其之间的联系。特定的测试方法包括广泛的软件功能和应用区域，图形用户界面，客户 / 服务器结构，文档和帮助功能以及实时系统都需要专门的测试指南和技术。

在本章中，我们详细的探讨了测试策略的问题，考虑了达到主要测试目标最可能需要的步骤，以一种有序的和有效的方法来发现并纠正错误。

OO 测试的策略和战术与传统软件有很大不同。OO 测试的视角扩大到包括复审分析和设计模型，此外，测试的焦点从过程构件(模块)移向了类。

因为 OO 分析和设计模型以及产生源代码是语义耦合的，测试（以正式的技术复审的方式）在这些活动进行中开始，为此，CRC、对象—关系、和对象—行为模型的复审可视为第一阶段的测试。

一旦已经完成 OOP，可对每个类进行单元测试。类测试使用一系列不同的方法：基于故障的测试、随机测试和划分测试。每种方法均测试类中封装的操作。设计测试序列以保证相关的操作被处理。类的状态，由其属性的值表示，检查以确定是否存在错误。

集成测试可使用基于线程或基于使用的策略来完成。基于线程的测试集成一组相互协作以对某输入或事件作出回应的类。基于使用的测试按层次构造系统，从那些不使用服务器类的类开始。集成测试用例设计方法也可以使用随机和划分测试。此外，基于场景的测试和从行为模型导出的测试可用于测试类及其协作者。测试序列跟踪跨越类协作的操作流。

OO 系统有效性测试是面向黑盒的并可以通过应用对传统软件讨论的相同的黑盒方法来完成。然而，通过使用使用实例作为有效性测试的主要驱动基于场景的测试主宰了 OO 系统的有效性。

习　题

1 分析软件测试与软件调试的目的。

2 从下列关于软件测试的叙述中，选出 5 条正确的叙述。

(1)用黑盒法测试时，测试用例是根据程序内部逻辑设计的。

(2)尽量用公共过程或子程序去代替重复的代码段。

(3)测试是为了验证该软件已正确地实现了用户的要求。

(4)对于连锁型分支结构，若有n个判定语句，则有 2^n条路径。

(5)尽量采用复合的条件测试，以避免嵌套的分支结构。

(6)GOTO 语句概念简单，使用方便，在某些情况下，保留 GOTO 语句反能使写出的程序列加简洁。

(7)发现错误多的程序模块，残留在模块中的错误也多。

(8)黑盒测试方法中最有效的是因果图法。

(9)在做程序的单元测试时，桩(存根)模块比驱动模块容易编写。

(10)程序效率的提高主要应通过选择高效的算法来实现。

3 下面是选择排序的程序，其中 datalist 是数据表，它有两个数据成员。一是元素类型为 Element 的数组 V，另一个是数组大小 n。算法中用到两个操作，一是取某数组元素 V[i]的关键码操作 getKey()，一是交换两数组元素内容的操作 Swap()：

```
void  selectsort(datalist & list){
//对表 list.v[0]到 list.v[n-1]进行排序，n 是表当前长度
   for (int I=0;I<list.n-1;I++){
      int k=I;
//在 list,v[I].key 到 list.v[n-1].key 中找具有最小关键码的对象
     for (int j=I+1;j<list.n;j++)
        if (list.v[j].getkey()<list.v[k].getkey())  k=j;
        //当前具最小关键码的对象
        if (k!=i) swap(list.v[I],list.v[k]);     //交换
   }
}
```

(1)试计算此程序段的 McCabe 环路复杂性。

(2)用基本路径覆盖法给出测试路径。

(3)为各测试路径设计测试用例。

4 根据下面给出的规格说明，利用等价类划分的方法，给出足够的测试用例。

“一个程序读入 3 个整数”，把这 3 个数值看做一个三角形的 3 条边的长度值。这个程序要打印出信息，说明这个三角形是不条边的、是等腰的、还是等边的。”

5 设计对一个自动饮料售货机软件进行的黑盒测试。该软件的规格说明如下：

“有一个处理单价为 1 元 5 角钱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，在送出饮料的同时退还 5 角硬币。”

(1)试利用因果图法，建立该软件的因果图。

(2)设计测试该软件的全部测试用例。

6 应该由谁来进行确认测试？是软件开发者还是软件用户？为什么？

7 举例说明黑盒法测试能够给出“一切正常”，而白盒法测试可能发现错误。再举例说明白盒法测试能够给出“一切正常”，而黑盒法测试可能发现错误。

8 请描述为什么类是 OO 系统中测试的最小合理单位。

9 为什么“测试”应该从 OOA 和 OOD 活动开始?

10 用于集成测试的基于线程的和基于使用的策略间有什么不同?集群测试如何适合它?