第 3 章　单元测试

本章要点

单元测试的定义；单元测试同集成测试和系统测试的区别；单元测试环境的组成；单元测试的分析方法；单元测试的用例设计方法；单元测试的过程；单元测试举例。

本章目标

- 掌握单元测试的概念
- 了解单元测试的误区
- 了解单元测试与集成测试和系统测试的区别
- 掌握单元测试的策略
- 掌握单元测试分析的方法
- 掌握单元测试用例设计方法

通俗一点来说，工厂在组装一台电视机之前，对每个元件所进行的测试就类似于软件开发过程中的单元测试。对于程序员来说，单元测试是每天都必做的工作，如：除了极简单的函数外，每写完一个函数，总是要执行一下，看看它所实现的功能是否正常；甚至有时还要想办法输出一些数据，如：弹出窗口应该显示的信息。当然这种单元测试属于非正规的临时单元测试，针对代码的测试很不完整，代码覆盖率要超过 70%都很困难。很多人尤其是初学者可能会问：为什么要进行烦人的单元测试？那些刚刚接触完全测试概念的开发人员也常常会考虑到这个问题。原因就在于未经测试覆盖的代码可能会遗留大量细小的错误，这些错误还会互相影响，当 BUG 暴露出来的时候难于调试，会大幅度提高后期测试和维护成本，也降低了开发商的市场竞争力。

在传统的软件测试过程中，单元测试是最早开始进行的测试，是在代码编写完成之后才进行的测试，使用最多的技术是白盒测试。它也被认为是集成测试的基础，因为只有通过了单元测试的模块，才可以把它们集成到一起进行集成测试。否则，即使集成测试通过了，投入使用的软件也会像地基不牢的摩天大楼一样暗藏着很多不安全因素。随着软件开发技术的不断进步，以及人们对软件测试工作重要性认识的增强，这个阶段的测试通常在项目详细设计阶段就已经开始了。由于在软件开发周期后期可能会因为需求变更或功能完善等原因对某个单元的代码做一些改动，因此不妨把单元测试看成是一种活动，从详细设计开始一直贯穿于整个项目开发的生命周期中。

总之，单元测试是十分重要的，通常被认为是提高软件质量，降低开发成本的必由之路。

3.1　单元测试概述

传统软件对“单元”一词有各种定义，如：

- 单元是可以编译和执行的最小软件组件。
- 单元是决不会指派给多个设计人员开发的软件组件。

实际上，“单元”的概念和被测应用（AUT）的设计方法，以及在其开发过程中采用的实现技术有关。基本单元必须具备一定的基本属性，有明确的规格定义，以及包含与其他部分接口的明确定义等，并且能够清晰地与同一程序中的其他单元划分开来。

对于结构化的编程语言而言，程序单元通常指程序中定义的函数或子程序，单元测试就是指对函数或子程序所进行的测试。但有时候也可以把紧密相关的一组函数或过程看做一个单元，举例来说：如果函数 A 只调用另一个函数 B，并且函数 B 和函数 A 的代码总处于一定的范围之内，那么在执行单元测试时，就可以将 A 和 B 合并为一个单元进行测试。对于面向对象的编程语言而言，程序单元通常指特定的一个具体的类或相关的多个类，单元测试主要是指对类的测试；但有时候，在一个类特别复杂时，就会把方法作为一个单元进行测试。对于同面向对象软件关联密切的 GUI 应用程序而言，单元测试一般是在“按钮级”进行。对 Ada 语言来说，开发人员可以选择在独立的过程和函数，或是在 Ada 包的级别上进行单元测试。

那么，什么是单元测试？通常而言，单元测试是在软件开发过程中要进行的最低级别的测试活动，或者说是针对软件设计的最小单位——程序模块，进行正确性检验的测试工作，其目的在于发现每个程序模块内部可能存在的差错。在单元测试活动中，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试，主要工作分为两个步骤：人工静态检查和动态执行跟踪。前者主要是保证代码算法的逻辑正确性（尽量通过人工检查发现代码的逻辑错误）、清晰性、规范性、一致性、算法高效性，并尽可能地发现程序中没有发现的错误。后者就是通过设计测试用例，执行待测程序来跟踪比较实际结果与预期结果来发现错误。

经验表明，使用人工静态检查法能够有效地发现 30%～70%的逻辑设计和编码错误。但是代码中仍会有大量的隐性错误无法通过视觉检查发现，必须通过跟踪调试法细心分析才能够捕捉到。所以，动态跟踪调试方法也成了单元测试的重点与难点。

单元测试应该在什么时候进行最好？笼统地说就是越早越好，那么应该早到什么程度？XP 开发理论讲究 TDD，即测试驱动开发，先编写测试代码，再进行开发。但在实际的工作中，可以不必过分强调先测试什么后测试什么。一些有经验的开发人员建议，先编写产品函数的框架，然后编写测试函数，再针对产品函数的功能编写测试用例，最后编写产品函数的代码，每写一个功能点都运行测试，并随时补充测试用例。所谓先编写产品函数的框架，是指先编写函数空的实现，有返回值的随便返回一个值，编译通过后再编写测试代码，这时，函数名、参数表、返回类型都应该确定下来了，所编写的测试代码以后需修改的可能性比较小。

单元测试是其他级别测试工作展开的基础，其重要性不言而喻。那么，单元测试应该由谁来完成比较合适呢？也许读者会想当然地认为单元测试应该由测试人员来进行，其实这种想法是错误的。单元测试与其他测试不同，单元测试可看做是编码工作的一部分，在编码的过程中考虑测试问题，得到的将是更优质的代码，因为在这时程序员对代码应该做些什么了解得最清楚。如果不这样做，等到某个模块崩溃时程序员可能已经忘记了代码是怎样工作的。即使是在强大的工作压力下，也必须重新回想才能把它弄清楚，这又要花费许多时间。进一步说，这样做出的更正往往不会那么彻底，可能更脆弱。因此一般应该由程序员完成单元测试工作，并且在提交产品代码的同时也提交测试代码。当然，为了确保软件质量，测试部门可以对其测试工作做一定程度的审核。

单元测试的分工大致如下：一般由开发组在开发组组长监督下进行，保证使用合适的测试技术，根据单元测试计划和测试说明文档中制定的要求，执行充分的测试；由编写该单元的开发组中的成员设计所需要的测试用例，测试该单元并修改缺陷。其中，测试包括设计和执行测试脚本和测试用例，并且要记录测试结果和单元测试日志。另外在进行单元测试时，最好要有一个专人负责监控测试过程，见证各个测试用例的运行结果。当然，可以从开发组中选取一人担任，也可以由质量保证代表担任。

经常有人问，既然单元测试对开发人员来说如此重要，那么对于客户或最终使用者而言也是这么重要吗？它与验收测试有关吗？这个问题很难回答。事实上，在做单元测试时常常并不关心整个产品的确认、验证和正确性等，甚至也不关心性能方面的问题，而主要是证明代码的行为和我们的期望一致。因此单元测试的对象常常是一些规模很小、非常独立的片断。只有所有单独部分的行为都通过了验证，确保它和我们的期望一致，才能够建立起我们对产品的信心，从而开始做进一步的集成测试。如果在程序员不能够确信正在书写的这些代码和预期一致的情况下，做其他任何形式的测试只不过是在浪费时间而已。

足够的单元测试不但能够使开发工作变得更轻松，而且会对设计工作的改进提供帮助，甚至大大减少花费在调试上面的时间。

总之，单元测试的目标就是验证开发人员所书写的编码是否可以按照其所设想的方式执行而产出符合预期值的结果，确保产生符合其需求的可靠程序单元。符合需求的代码通常应该具备以下性质：正确性、清晰性、规范性、一致性、高效性等（根据优先级别排序）。

（1）正确性是指代码逻辑必须正确，能够实现预期的功能。

（2）清晰性是指代码必须简明、易懂，注释准确没有歧义。

（3）规范性是指代码必须符合企业或部门所定义的共同规范，包括命名规则，代码风格等。

（4）一致性是指代码必须在命名上（如：相同功能的变量尽量采用相同的标识符）、风格上都保持统一。

（5）高效性是指代码不但要满足以上性质，而且需要尽可能减少代码的执行时间。

3.1.1　单元测试误区

如果在单元测试阶段就应该发现的 BUG 遗留到软件开发的后期阶段，那么那时修改它将会浪费大量的项目资源，因为与缺陷所在单元相关联的模块测试以及包括该单元在内的集成测试都要进行回归测试。好的单元测试就在于能否尽早发现更多的 BUG，从而降低软件开发的成本。

在软件开发过程中，最可怕的就是需求被频繁地修改和变动，因为这些变化最终都要反映在代码中。也就是说，编码本身出现的 BUG 并不多，更多 BUG 的产生是由于变化后的代码破坏了源代码功能造成的，所以只要发生了变化，必须保证进行完整的回归测试。高质量的单元测试会大大简化系统的集成过程，因为所有被集成的单元都是可以信赖的。

能否把单元测试工作做好的关键就在于我们是否拥有正确的测试思想，明确测试所针对的目标，并且能够很好地监控和管理测试过程，适当地使用某些自动化工具来支持测试过程。科学合理地安排和进行这些活动，可以使我们在最低开发成本下得到可靠的软件。

为了使读者更清晰地认识到单元测试的重要性，下面列举一些在实际工作中对单元测试的

误解并加以澄清。

（1）单元测试是一种浪费时间的工作。编码工作完成之后，开发人员常常希望能够尽快进行软件的集成工作，这样就可以看到实际的系统开始运行工作了，而把单元测试活动看做是通往这个阶段点的障碍，推迟了对整个系统进行联调的时间。

事实上，没有经过单元测试而集成的系统能够正常工作的可能性是很小的，而且存在各种类型的 BUG，软件甚至无法运行。接下来不得不将大量的时间花费在跟踪那些隐藏在各个单元里的 BUG 上面，当然不排除个别情况下，这些 BUG 可能是微不足道的，但是总的来说，这样会延长软件的开发周期，而且当系统投入使用时也无法确保它能够可靠运行。

在实际工作中，进行计划完整的单元测试和编写实际的代码所花费的精力大致上是相同的。单元测试工作一旦完成了，很多 BUG 将被纠正，在手头拥有稳定可靠的单元模块的情况下，开发人员能够进行更高效的系统集成工作，这才是真实意义上的进展。所以说单元测试不是在浪费时间。

使用一些单元测试支持工具可以使单元测试更加简单和有效，但这不是必需的。单元测试即使是在没有工具支持的情况下也是一项非常有意义的活动。

（2）单元测试只能证明代码做了什么。那些没有首先为每个单元编写一个详细的规格说明而直接进入编码阶段的开发人员经常会有这样的抱怨。当编码完成以后，要执行该单元的测试任务时，他们就阅读这些代码，查看这些代码实际上做了什么，也就是将测试工作完全建立在代码的基础上。当然，所有这些测试工作能够表明的事情就是编译器工作正常，此外无法证明任何事情。他们也许能够发现（希望能够）罕见的编译器 BUG，但是他们能够做的仅仅是这些。

如果他们首先写好一个详细的规格说明，使测试能够以规格说明为基础，那么代码就能够针对它的规格说明，而不是针对自身进行测试。这样的测试仍然能够抓住编译器的 BUG，同时也能找到更多的编码错误，甚至是一些规格说明中的错误。好的规格说明可以使测试的质量更高。

在实践中可能会出现这样的情况：一个开发人员接到一个只有代码没有规格说明的单元的测试任务。此时，应该怎样做才能更好地进行单元测试呢？第一步就是理解这个单元原本要做什么，而不是它实际上做了什么。比较有效的方法就是通过阅读程序代码和注释，以及调用它和被它调用的相关代码，然后倒推出一个概要的规格说明，也可以用手工或使用某种工具画出流程图，这些都是非常有帮助的。然后对这个概要规格说明进行走读，以确保这个规格说明没有基本的错误，然后根据它来设计单元测试。

（3）我是个很棒的程序员，我是不是可以不进行单元测试？在每个开发团队中都可能有非常擅长于编程的开发人员，他们开发的软件总是可以最先正常运行。因此有人认为他们所开发的代码就可以不用进行单元测试了。

在现实生活中，每个人都会犯错误。更何况真正的软件系统是非常复杂的，即使是编程高手也无法保证不犯任何错误。因此，我们不要希望那些没有进行充分测试和 BUG 修改过程的软件系统可以正常工作。虽然编码工作不是一个可以一次性通过的过程，但是开发人员可以通过开发一些可重复的单元测试来节省测试时间。

（4）集成测试能捕捉到所有的 BUG。在前面的讨论中，我们已经从侧面对这个问题进行了阐述。这个论点不正确的原因就在于，如果没有首先做单元测试就进行软件集成工作，并且

软件规模越大集成就越复杂，不难想象，开发人员花费大量的时间很可能仅仅是为了使软件能够运行，导致实际的测试方案无法执行，因为没有经过单元测试的模块可能存在着很多软件缺陷，甚至会导致软件无法集成。

即使软件可以运行，开发人员又要面对这样的问题：在考虑软件全局复杂性的前提下对每个单元进行全面的测试。这是一件非常困难的事情，甚至在建立一种单元调用的测试条件时，要全面地考虑单元被调用时的各种入口参数。因此在软件集成阶段，对单元功能全面测试的复杂程度远远超过独立进行的单元测试过程。最后将导致无法进行全面的测试，忽略甚至遗漏了很多缺陷。

让我们类比一下，假设我们要清洗一台已经完全装配好的食物加工机器。无论你喷了多少水和清洁剂，一些食物的小碎片还是会粘在机器的死角位置，只有任其腐烂并等待以后再想办法。但我们换个角度想想，如果这台机器是拆开的，这些死角也许就不存在或者更容易接触到了，并且每一部分都可以毫不费力地进行清洗。

（5）单元测试的成本效率不高。开发组织的测试水平高低是与他们对那些未发现的 BUG 潜在后果的重视程度成正比的。如果被软件开发人员所忽视的一个小小的 BUG（但是用户可不会这样认为）经常出现，甚至有时会发生死机的情况，那么这将会影响软件开发组织的信誉，使用户失去对开发者的信任。

很多研究成果表明，无论什么时候做出修改都要进行完整的回归测试，在生命周期中尽早地对软件产品进行测试将使效率和质量得到最好的保证。BUG 发现得越晚，修改它所需的成本就越高。因此从经济角度来看，应该尽可能早地查找和修改 BUG。而单元测试就是一个在修改费用变得过高之前，能够在早期抓住 BUG 的最佳时机。

与集成测试和系统测试等其他级别的测试相比，单元测试的创建更简单，维护更容易。与那些复杂且旷日持久的集成测试，或是不稳定的软件系统测试相比，单元测试所需的费用是很低的。

图 3-1 摘自《实用软件度量》（Capers Jones，McGraw-Hill 1991），它列出了准备测试、执行测试和修改缺陷所花费的时间（以一个功能点为基准），这些数据显示出单元测试的成本效率大约是集成测试的两倍、系统测试的三倍（参见条形图）。术语域测试（Field Test）的含义是在软件投入使用以后，针对某个领域所做的所有测试活动。

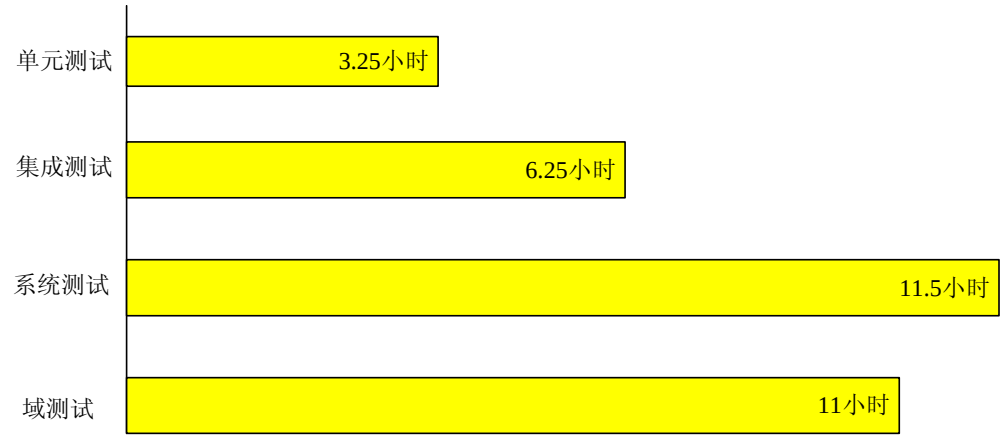

图 3-1　测试成本效率对比

这个图表并不表示开发人员不应该进行后续阶段的测试，而真正意图是向我们表明尽可能早地排除尽可能多的 BUG，可以减少以后各个阶段测试的费用。

总之，在经过了单元测试之后，系统集成过程将会大大地简化。开发人员可以将精力集中在单元之间的交互作用和全局的功能实现上，而不会陷入充满很多 BUG 的单元之中不能自拔。

3.1.2 单元测试与集成测试的区别

由测试 V 模型可知，单元测试与集成测试的主要区别在于测试的对象不同。单元测试的对象是实现具体功能的单元，一般对应详细设计中所描述的设计单元。往往在详细设计阶段把这些模块分给不同的开发小组。集成测试是针对概要设计所包含的模块以及模块组合进行的测试。

单元测试所使用的主要测试方法是基于代码的白盒测试。而集成测试所使用的主要测试方法是基于功能的黑盒测试。

因为集成测试要在所有要集成的模块都通过了单元测试之后才能进行，也就是说在测试时间上，集成测试要晚于单元测试，所以单元测试的好坏直接影响着集成测试。

单元测试的工作内容包括模块内程序的逻辑、功能、参数传递、变量引用、出错处理、以及需求和设计中具体的要求等方面的测试。集成测试的工作内容主要是验证各个接口、接口之间的数据传递关系，以及模块组合后能否达到预期效果。

虽然单元测试和集成测试有一些区别，但是二者之间也有着千丝万缕的联系。目前集成测试和单元测试的界限趋向模糊。有时也会在单元测试中引入集成测试的方法，如：为了减少编写桩模块代码的工作量，有时也采用的自底向上的测试方法。总之，无论是单元测试还是集成测试，其最终目的都是为了发现软件开发过程中所引入的错误，所以也无需一味地追究二者的区别。

3.1.3 单元测试与系统测试的区别

单元测试与系统测试的区别不仅仅在于测试的对象和测试的层次的不同，最重要的区别是测试性质不同。单元测试的执行早于系统测试，测试的是软件单元的具体实现、内部逻辑结构以及数据流向等。系统测试属于后期测试，主要是根据需求规格说明书进行的，是从用户角度来进行的功能测试和性能测试等，以证明系统是否满足用户的需求。

单元测试中发现的错误容易定位，并且多个单元测试可以并行进行；而系统测试发现的错误比较难定位。

这里以单元测试与系统测试阶段所做的验收测试为例来进一步说明二者之间的区别。单元测试注重系统的内部，比如体系构造、系统的框架结构等，是要保证系统各个部分得以安全地正常执行。验收测试则是在系统可以正常运行这一概念之上才进行的，更注意系统的细节部分，比如系统界面是否美观，系统的操作是否人性化等。单元测试是从开发者的角度考虑的，而验收测试则是从用户的角度出发。

3.2 单元测试环境

单元测试环境的建立是单元测试工作进行的前提和基础，在测试过程中所起到的作用不言

而喻。显然，单元测试的环境并不一定是系统投入使用后所需的真实环境。那么，我们应该建立一个什么样的环境才能够满足单元测试的要求呢？本节将向读者介绍如何建立单元测试的环境。

由于一个模块或一个方法（Method）并不是一个独立的程序，在考虑测试它时要同时考虑它和外界的联系，因此要用到一些辅助模块，来模拟与所测模块相联系的其他模块。一般把这些辅助模块分为两种：

（1）驱动模块（driver）：相当于所测模块的主程序。它接收测试数据，把这些数据传送给所测模块，最后再输出实际测试结果。

（2）桩模块（stub）：用于代替所测模块调用的子模块。桩模块可以进行少量的数据操作，不需要实现子模块的所有功能，但要根据需要来实现或代替子模块的一部分功能。

这样，所测模块和与它相关的驱动模块及桩模块共同构成了一个“测试环境”，如图 3-2 所示。为了能够正确地测试软件，驱动模块和桩模块的编写，特别是桩模块可能需要模拟实际子模块的功能，因此桩模块的开发并不是很轻松。我们常常希望驱动模块和桩模块的开发工作比较简单，实际开销相对低些。比如说，有时候因为编写桩模块是困难费时的，我们就会尽量避免编写桩模块，即在项目进度管理时将实际桩模块的代码编写工作安排在被测模块前编写，以提高实际桩模块的测试频率，从而更有效地保证产品的质量，提高测试工作的效率。但是，为了保证能够向上一级模块提供稳定可靠的实际桩模块，为后续模块测试打下良好的基础，驱动模块还是必不可少的。但遗憾的是，仅用简单的驱动模块和桩模块有时不能完成某些模块的测试任务。

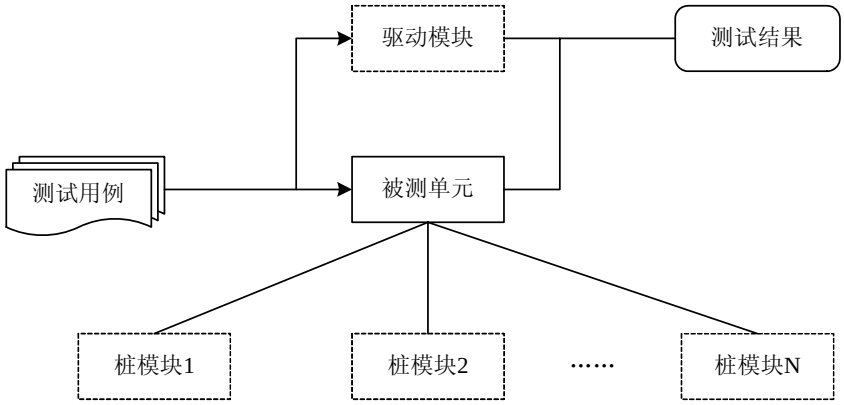

图 3-2　单元测试环境

为了确保可以高质量地完成单元测试，在设计桩模块和驱动模块时最好多考虑一些环境因素（所有的潜在输入和实际环境的代表物都需要考虑），如：系统时钟、文件状态、单元加载地点，以及与实际环境相同的编译器、操作系统、计算机等，这些都要在测试设计过程中给予关注。

对于每一个包或子系统我们可以根据所编写的测试用例编写一个测试模块类来做驱动模块，用于测试包中所有的待测试模块。最好不要在每个类中用一个测试函数的方法来测试跟踪类中所有的方法。这样的好处在于：

（1）能够同时测试包中所有的方法或模块，也可以方便地测试跟踪指定的模块或方法。

（2）能够联合使用所有测试用例对同一段代码执行测试，发现问题。

（3）便于回归测试，当某个模块做了修改之后，只要执行测试类就可以执行所有被测的模块或方法。这样不但能够方便地检查、跟踪所修改的代码，而且能够检查出修改对包内相关模块或方法所造成的影响，使修改引进的错误得以及时发现。

（4）复用测试方法，使测试单元保持持久性，并可以用既有的测试来编写相关测试。

（5）将测试代码与产品代码分开，使代码更清晰、简洁；提高测试代码与被测代码的可维护性。

在建立单元测试环境时，除了会需要一些桩模块和驱动模块以便使被测对象能够运行起来之外，还要模拟生成测试数据或状态，为单元运行准备动态环境。为了便于测试工作的顺利开展，最好还要考虑对测试过程的支持，比如：测试结果的统计、分析和保留，测试覆盖率的记录等。另外，测试人员在构建单元测试环境时可借助很多测试工具。如：在对使用 Java 语言开发的程序进行单元测试时可以借助 Junit，在本书后面的相关章节中将会对 Junit 工具进行专门的介绍，并列举出使用 Junit 进行测试的例子。

在这里读者要注意的是，驱动模块和桩模块是测试时使用的软件，不是软件产品的组成部分，不能和最终的软件一起提交。

最后要强调的一点是，测试代码也是用一般的方式编写和编译的，它和项目中的普通源码是一样的。测试代码可能偶尔会用到某些额外的程序库，但是除此之外，测试代码再也没有任何特别之处——它们也只是普通代码而已。

3.3 单元测试策略

单元测试涉及到的测试技术通常有：针对被测单元需求的功能测试、用于代码评审和代码走读的静态测试、白盒测试、状态转换测试和非功能测试。大部分情况下会选择使用白盒测试，参与测试的主要人员均为开发组成员，因为他们对单元的结构十分了解。这里的非功能测试是指对单元的性能、压力或者可靠性的测试，并不是单元测试的重点，但在适当的时候也要进行，因为单元模块性能的好坏会间接地影响整个系统的性能。

为了提高单元测试的质量，只了解这些单元测试技术还远远不够，还要选择合适的测试策略。在选择测试策略时，主要考虑如下 3 种方式：自顶向下（Top Down Unit Testing）的单元测试策略、自底向上的单元测试策略（Bottom up Unit Testing）和孤立的单元测试策略（Isolation Unit Testing）。

3.3.1 自顶向下的单元测试策略

- 步骤：

（1）从最顶层开始，把顶层调用的单元做成桩模块。

（2）对第二层测试，使用上面已测试的单元做驱动模块。

（3）依次类推，直到全部单元测试结束。

- 优点：可以在集成测试之前为系统提供早期的集成途径。由于详细设计一般都是自顶向下进行设计的，这样自顶向下的单元测试测试策略在顺序上同详细设计一致，因此测试可以与详细设计和编码工作重叠进行。

- 缺点：单元测试被桩模块控制，随着单元测试的不断进行，测试过程也会变得越来越复杂，测试难度以及开发和维护的成本都不断增加；要求的低层次的结构覆盖率也难以得到保证；由于需求变更或其他原因而必须更改任何一个单元时，就必须重新测试该单元下层调用的所有单元；低层单元测试依赖顶层测试，无法进行并行测试，使测试进度受到不同程度的影响，延长测试周期。
- 总结：从上述分析中，不难看出该测试策略的成本要高于孤立的单元测试成本，因此从测试成本方面来考虑，并不是最佳的单元测试策略。在实际工作中，当单元已经通过独立测试后，我们可以选择此方法。

3.3.2　自底向上的单元测试

- 步骤：

（1）先对模块调用图上的最底层模块进行测试，模拟调用该模块的模块为驱动模块。
（2）其次，对上一层模块进行单元测试，用已经被测试过的模块做桩模块。
（3）依次类推，直到全部单元测试结束。

- 优点：不需要单独设计桩模块；无需依赖结构设计，可以直接从功能设计中获取测试用例；可以为系统提供早期的集成途径；在详细设计文档中缺少结构细节时可以使用该测试策略。
- 缺点：随着单元测试的不断进行，测试过程会变得越来越复杂，测试周期延长，测试和维护的成本增加；随着各个基本单元逐步加入，系统会变得异常庞大，因此测试人员不容易控制；越接近顶层的模块的测试其结构覆盖率就越难以保证；另外，顶层测试易受底层模块变更的影响，任何一个模块修改之后，直接或间接调用该模块的所有单元都要重新测试。由于只有在底层单元测试完毕之后才能够进行顶层单元的测试，所以并行性不好。另外，自底向上的单元测试也不能和详细设计、编码同步进行。
- 总结：相对其他测试策略而言，该测试策略比较合理，尤其是需要考虑对象或复用时。它属于面向功能的测试，而非面向结构的测试。对那些以高覆盖率为目标或者软件开发时间紧张的软件项目来说，这种测试方法不适用。

3.3.3　孤立测试

- 步骤：无需考虑每个模块与其他模块之间的关系，分别为每个模块单独设计桩模块和驱动模块，逐一完成所有单元模块的测试。
- 优点：该方法简单、容易操作，因此所需测试时间短，能够达到高覆盖率。因为一次测试只需要测试一个单元，其驱动模块比自底向上的驱动模块设计简单，而其桩模块的设计也比自顶向下策略中使用的桩模块简单。另外，各模块之间不存在依赖性，所以单元测试可以并行进行。如果在测试中增添人员，可以缩短项目开发时间。
- 缺点：不能为集成测试提供早期的集成途径。依赖结构设计信息，需要设计多个桩模块和驱动模块，增加了额外的测试成本。
- 总结：该方法是比较理想的单元测试方法。如辅助适当的集成测试策略，有利于缩短项目的开发时间。

3.3.4 综合测试

在单元测试中，为了有效地减少开发桩模块的工作量，可以考虑综合自底向上测试策略和孤立测试策略。

下面，结合两票系统中数据初始化模块的代码进行说明：

```
    public void initialData(){//数据初始化
    try{
    m.setColor(Color.gray);
    m.fillRect(InitialLeft,InitialTop,InitialLeft+InitialWidth,InitialTop+Init
ialHeight);
                //进度条背景框结束
    paint_tishi(16711680,65535,"系统数据初始化请稍后",InitialLeft,InitialTop,
InitialLeft +InitialWidth*1/40,InitialTop+InitialHeight);
    init_sys();
    paint_tishi(16711680,65535,"系统数据初始化请稍后",InitialLeft,InitialTop,
InitialLeft+ InitialWidth*1/20,InitialTop+InitialHeight);
    init_ini2();//系统设置
    paint_tishi(16711680,65535,"母线数据初始化请稍后",InitialLeft,InitialTop,
InitialLeft +InitialWidth*2/20,InitialTop+InitialHeight);
    init_mLine();//母线
    paint_tishi(16711680,65535,"刀闸数据初始化请稍后",InitialLeft,InitialTop,
InitialLeft +InitialWidth*3/20,InitialTop+InitialHeight);
    init_dz();//刀闸
    paint_tishi(16711680,65535,"开关数据初始化请稍后",InitialLeft,InitialTop,
InitialLeft +InitialWidth*4/20,InitialTop+InitialHeight);
    …
    init_bhkMenu();//菜单
    init_BhData();//保护
    init_CzpShowData();//操作票内容显示
    paint_tishi(16711680,65535,"数据初始化完成",InitialLeft,InitialTop,InitialLeft
+InitialWidth*17/20,InitialTop+InitialHeight);
    init_CzpShowData();//操作票内容显示
    //init_czpdb();//操作票内容显示
    paint_tishi(16711680,65535,"字典数据初始化请稍后",InitialLeft,InitialTop,
InitialLeft+InitialWidth*18/20,InitialTop+InitialHeight);
    init_zd();//字典初始化
    init_CzpDDYdangan();//操作票档案
    paint_tishi(16711680,65535,"控件初始化请稍后",InitialLeft,InitialTop,
InitialLeft+InitialWidth*19/20,InitialTop+InitialHeight);
    …
    }

    public void paint_dz(int i,boolean flag){//画刀闸
    int color,color1,color2,x,Y,L,W,H,x1,y1,x2,y2,x3,y3,x4,y4,x5,y5;
    x=mydata_dz[i].getx()*stretch;
```

```
Y=mydata_dz[i].getY()*stretch;
L=mydata_dz[i].getlength()*stretch;
W=mydata_dz[i].getWidth()*stretch;
H=mydata_dz[i].getHeight()*stretch;
//颜色设定
if (flag)
{
  color1=Voltage_color(mydata_dz[i].getlevel(),8);
    color2=65280;
    }
  else
    {
    color1=backcolor;
    color2=backcolor;
    }
   x1=x-H/2-1*stretch;
   y1=Y-H/2-1*stretch;
   x2=x+H/2+1*stretch;
   y2=Y+H/2+1*stretch;
   m.setColor(new Color(backcolor));
   m.fillRect(x1,y1,x2,y2);//不显示出来
   color=mydata_dz[i].getcolor();
   //绘制刀闸
   if(mydata_dz[i].getType()==2)
   {//跌落式熔断器
    …
   if(mydata_dz[i].getState()==1)
   {//状态：合
    …
   }
  else if(mydata_dz[i].getState()==0)
   {//状态：断
   …}
   }
   else
   {//普通刀闸 0,1
     if(mydata_dz[i].getMode()==0)//根据不同模式来画
         …
    …
     }
public void paint_kg(int i,boolean flag){//画开关
…}
…
```

在对上面的代码进行测试时，首先要对各个元件的 paint 函数进行单元测试，然后直接使用 paint 函数作为桩来测试 initialData 函数。如果单纯依据策略二，在测试 initialData 函数时，需要使测试用例同时覆盖 initialData 函数和各个 paint 函数和 init 函数；而如果使用孤立测试

策略，直接使用各个 paint 函数和 init 函数，并仍然把它们理解为桩函数，这样在设计用例的时候就不用关注各个 paint 函数和 init 函数是怎样执行的了，只需验证 initialData 函数是否实现了设计的功能即可。而考虑覆盖率时，只需考虑 initialData 的覆盖率就可以了。

3.4 单元测试分析

有一些开发者在做单元测试的时候，只编写一个测试，让所有代码从头到尾跑一次，只测试一条能够正确执行的路径。如果测试通过了，就认为测试工作已经完成了。但是，现实工作中的单元测试，远远没有这么简单，常常会遇到各种糟糕的情况，如：代码运行后常常会抛出异常；硬盘的剩余空间不足；网络出现故障；缓冲区溢出；代码潜藏着一些 BUG 等。这就是软件开发“工程”的特点。土木设计师在设计一座桥梁的时候，必须要考虑桥梁的负载、强风的影响，地震、洪水等灾害发生的情况。显然，我们在测试这座桥梁的时候，不能够简单地选择在风和日丽的一天，只要一辆车顺利通过就结束测试。因此，我们在做单元测试的时候，要确认这段代码是否在任何情况下都和期望的一直一致，比如：突然断电、网络故障、硬盘空间不足等情况。因此，在做单元测试时要做完善和全面的单元测试分析，以确定测试内容。初学者往往会觉得很茫然，不知道如何下手，本节将从不同侧面对这个问题进行详细的讨论。

对于方法或类来说，我们很难一下子发现所有可能出问题的地方，找到所有潜藏的 BUG。经验丰富的人常常能够轻松地分析出系统最可能出现的问题，但是对于没有经验或经验很少的工作人员来讲，是很难做到这一点的。可是，最终用户在使用过程中迟早会发现程序中遗留的 BUG，因此下面总结了一些测试分析的指导原则，希望使读者在测试工作中能够有的放矢、有章可循。

一般可以从如下几个方面进行分析和测试。

（1）判断得到的结果是否正确。因为对于测试而言，首要的任务就是察看一下所期望的结果是否正确，即对结果进行验证。例如：在两票系统中，需要对生成一次和二次系统图所需要的初始化的数据进行验证；需要对系统图页面上弹出的菜单项进行验证等。

这些通常都是简单的测试，可能在需求说明中就已经规定了。假如没有明确的文档，可以询问相关人员，自己确定一些需求，或者安排用户参与以便及时获得反馈，对自己确定的需求假设进行调整。因为在整个软件开发的生命周期中，需求的更新都有可能使得判断代码“正确”的标准改变。对于那些涉及大量测试数据的测试，可考虑使用一个独立的数据文件来存储这些数据，做单元测试时直接读取这些数据，但前提是在使用数据文件之前要进行仔细检查，以免引入不必要的错误。总之，无论有什么困难都必须要想尽各种办法确认结果的正确性。

（2）判断是否满足所有的边界条件。边界条件是指软件计划的操作界限所在的边缘条件。边界条件测试是单元测试中最后也是最重要的一项任务。众所周知，软件经常在边界上失效，采用边界值分析技术，针对边界值及其左、右设计测试用例，很有可能发现新的错误。边界上的错误常常是防不胜防的。在使用边界值测试的方法时，不妨结合实际项目参考以下测试技巧：

1）输入了完全伪造或者和要求不一致的数据，如：给程序提供一个根本就不存在的文件

或路径。

2）输入一个格式错误的数据，如：一个类似 softwaretesting@sohu 这样的没有顶层域名的电子邮件地址。

3）提供一个空值或者不完整的值（如：null 和 0.0）。

4）与意料之中的值相差很远的值，例如：把人员年龄输入为 10000 岁。

5）假如一个列表中不允许有重复的数值存在，就可以给它传入一组存在重复数值的列表；如果某个字段的值要求惟一，那么可以输入两个或多个相同的数值来进行测试。

6）如果要求按照一定的顺序来存储一些数据，那么可以输入一些顺序打乱的数据来做测试。

7）对于一些做了安全限制的部分，要做一些尽量通过各种途径尝试能否绕过安全限制的测试。如：很多系统都分成好几个子系统，或者对于系统的一些功能进行了权限设置，因此没有权限的工作人员是不能使用相关功能的，那么我们可以以没有权限的用户身份登录系统，测试是否能够登录成功或能否使用并没有被赋予权限的功能。

8）如果功能的启用有一定的顺序限制，就用和期望不一致的顺序来进行测试。如：一般系统都要求用户正式进入工作页面之前先登录，那么我们就可以测试一下是否不登录也能进入工作页面。

除了这些之外，可能还会存在更多的边界值情况，有的书中把边界值测试归纳为 7 种情形，即：一致性（实际数值是否和预期的一致）、顺序性（实际数值是否像预期的那样有序或者无序）、区间性（实际值是否位于合理的最小值和最大值之内）、依赖性（代码是否引用了一些不在代码本身控制范围之内的外部资源）、存在性（实际数值是否非 null、非 0、在一个集合中等等）、基数性（是否恰好有足够的值）、相对或者绝对的时间性（所有事情的发生是否有序？是否是在正确的时刻？是否恰好及时？）。读者在实际工作中也应该注意积累这方面的经验，提高软件测试的技巧。

（3）分析能否使用反向关联检查。在实际程序中，有一些方法可以使用反向的逻辑关系来验证它们。例如：为了检查某一条记录是否成功地插入了数据库，可以使用查询语句来验证。但要注意的是：原方法和它的反向测试中的错误可能会同时掩盖一些 BUG。所以，可能的话尽量使用不同的方法来做反向测试。

（4）分析是否能使用其他手段来交叉检查结果。一般而言，对某个值进行计算会有一种以上的算法，但我们会因考虑到运行效率或其他方面的原因而选择其中的一种。那么，可以使用剩下算法中的一个进行交叉检查的方法来进行测试。其实，也可以利用升级前的版本中实现同样功能的代码来检查新版本。

另外，对于面向对象的应用软件，还可以使用与类本身有关联关系的不同数据来做测试。如：在两票系统中，在进行某个人的开具工作票数目统计时，可以利用总的工作票数目去掉其他人开具的工作票数目的方法来检查。

（5）分析是否可以强制一些错误发生。在实际使用过程当中，总会有意想不到的各种各样的情况和错误发生，如：网络出现了故障、磁盘空间不足、内存不足等。如果某个方法依赖于网络、数据库或 Servlet 引擎等，那么就要模拟这些外部条件产生错误的情况来做单元测试。但这并不一定要求测试人员手工来模拟，可以借助一些相关的工具，如：EasyMock（可以把它集成到 Junit 测试框架中）。

（6）分析模块接口。数据在接口处出错就好像丢掉了进入大门的钥匙，无法进行下一步的工作，只有在数据能正确流入、流出模块的前提下，其他测试才有意义。测试接口正确与否应该考虑下列因素：

1）输入的实际参数与形式参数的个数是否相同。

2）输入的实际参数与形式参数的属性是否匹配。

3）输入的实际参数与形式参数的量纲是否一致。

4）调用其他模块时所给实际参数的个数是否与被调模块的形参个数相同。

5）调用其他模块时所给实际参数的属性是否与被调模块的形参属性匹配。

6）调用其他模块时所给实际参数的量纲是否与被调模块的形参量纲一致。

7）调用预定义函数时所用参数的个数、属性和次序是否正确。

8）是否存在与当前入口点无关的参数引用。

9）是否修改了只读型参数。

10）对全程变量的定义各模块是否一致。

11）是否把某些约束作为参数传递。

如果模块内包括外部输入输出，还应该考虑下列因素：

1）文件属性是否正确。

2）OPEN/CLOSE 语句是否正确。

3）格式说明与输入/输出语句是否匹配。

4）缓冲区大小与记录长度是否匹配。

5）文件使用前是否已经打开。

6）是否处理了文件尾。

7）是否处理了输入/输出错误。

8）输出信息中是否有文字性错误。

（7）分析局部数据结构。局部数据结构往往是错误的根源，对其检查主要是为了保证临时存储在模块内的数据在程序执行过程中完整、正确，因此应仔细设计测试用例，力求发现下面几类错误：

1）被测模块中是否存在不合适或不一致的数据类型说明。

2）被测模块中是否残留未赋值或未初始化的变量。

3）被测模块中是否存在错误的初始值或错误的默认值。

4）被测模块中是否有不正确的变量名（拼错或不正确的截断）。

5）被测模块中是否存在数据结构的不一致。

6）被测模块中是否会出现上溢、下溢和地址异常。

除了局部数据结构外，如果可能，单元测试时还应该查清全局数据（例如 FORTRAN 的公用区）对模块的影响。

（8）分析独立路径。在模块中应对每一条独立执行路径进行测试，单元测试的基本任务是保证模块中每条语句至少执行一次。此时设计测试用例是为了发现因计算错误、比较不正确和控制流不适当而造成的错误，发现这些错误的最常用且最有效的测试技术就是基本路径测试和循环测试。常见的错误通常包括：

1）运算符优先级理解或使用错误。

2）混合类型运算错误。

3）变量初始化错误。

4）精度不够。

5）表达式符号错误。

比较判断与控制流常常紧密相关，此类测试用例还应致力于发现下列错误：

1）不同数据类型的对象之间进行比较。

2）错误地使用逻辑运算符或优先级。

3）因计算机表示的局限性，期望理论上相等而实际上不相等的两个量相等。

4）比较运算或变量出错。

5）循环终止条件不可能出现。

6）迭代发散时不能退出。

7）错误地修改了循环变量。

（9）分析出错处理是否正确。一个好的设计应能预见各种出错条件，并进行适当的出错处理，即预设各种出错处理通路。出错处理是模块功能的一部分，这种带有预见性的机制保证了在程序出错时，对出错部分及时修补，因此出错处理通路同样需要认真测试，此类测试应着重检查下列问题：

1）输出的出错信息难以理解。

2）错误陈述中未能提供足够的出错定位信息。

3）显示错误与实际遇到的错误不符。

4）异常处理不得当。

5）在程序进行出错处理前，错误条件已经引发系统的干预。

3.5　单元测试用例设计

通过前面的学习读者已经了解到，测试用例就是测试数据及与之相关的测试规程的一个特定集合，它是为验证被测试程序（为测试路径或验证是否符合特定需求）而产生的。在单元测试过程中，测试用例的设计应与复审工作相结合，根据设计信息选取测试数据，将增加发现上述各类错误的可能性；另外，在确定测试用例的同时，应给出期望结果，以便进行测试分析和判断。单元测试用例的设计既可以使用白盒测试也可以使用黑盒测试，但以白盒测试为主。

白盒测试进入的前提条件是测试人员已经对被测试对象有了一定的了解，基本上明确了被测试软件的逻辑结构。具体过程就是针对程序逻辑结构设计和加载测试用例，驱动程序执行，检查在不同点程序的状态，以确定实际的状态是否与预期的状态一致。

一般来说，为了度量测试的完整性，测试工作中通常要求达到一定的覆盖率要求。因为通过覆盖率的统计可以知道测试是否充分，对软件的哪个部分所做的测试不够，指导我们如何设计增加覆盖率的测试用例。这样就能够提高测试质量，尽量避免设计无效的用例。

在白盒测试的范畴内通常使用下面几种测试覆盖率来度量测试，如：语句覆盖、判定覆盖、条件覆盖、判定条件覆盖、路径覆盖等。白盒测试最低应该达到的覆盖率目标是：语句覆盖率达到 100%，分支覆盖率达到 100%，覆盖程序中主要的路径，主要路径是指完成需求和

设计功能的代码所在的路径和程序异常处理执行到的路径。

测试人员在实际工作中要根据不同的覆盖要求来设计面向代码的单元测试用例，运行测试用例后至少应该实现如下几个覆盖需求：

（1）对程序模块的所有独立的执行路径至少覆盖一次。

（2）对所有的逻辑判定，真假两种情况都至少覆盖一次。

（3）在循环的边界和运行界限内执行循环体。

（4）测试内部数据结构的有效性等。

黑盒测试是要首先了解软件产品具备的功能和性能等需求，再根据需求设计一批测试用例以验证程序内部活动是否符合设计要求的活动。在黑盒测试范畴内通常使用功能覆盖率来度量测试的完整性。而功能覆盖率中最常见的就是需求覆盖，目的就是通过设计一定的测试用例，使得每个需求点都被测试到。其次，还包括接口覆盖（又叫入口点覆盖），其目的就是通过设计一定的测试用例使系统的每个接口都被测试到。黑盒测试应达到的覆盖率目标是：程序单元正确地实现了需求和设计上要求的所有功能，满足性能要求，同时程序单元要有可靠性和安全性。

测试人员在实际工作中至少应该设计能够覆盖如下需求的基于功能的单元测试用例：

（1）测试程序单元的功能是否实现。

（2）测试程序单元性能是否满足要求（可选）。

（3）是否有可选的其他测试特性，如边界、余量、安全性、可靠性、强度测试、人机交互界面测试等。

无论是白盒测试还是黑盒测试，每个测试用例都应该包含下面 4 个关键元素：

（1）被测单元模块初始状态声明，即测试用例的开始状态（仅适用于被测单元维持了调用中间状态的情况）。

（2）被测单元的输入，包含由被测单元读入的任何外部数据值。

（3）该测试用例实际测试的代码，用被测单元的功能和测试用例设计中使用的分析来说明，如：单元中哪一个决策条件被测试。

（4）测试用例的期望输出结果（在测试进行之前的测试说明中定义）。

1. 测试用例设计步骤

下面介绍进行测试用例设计，书写测试说明的 7 步通用过程

步骤1：使被测单元运行

在单元测试说明中，应该把使用一种简单方法就能够执行被测单元的测试用例作为第一个测试用例，因为当这个测试用例运行成功时可以增强人的自信心。如果运行失败，最好选择一个更简单的输入对被测单元进行测试/调试。

这个阶段适合的技术有：

1）模块设计说明导出的测试。

2）对等区间划分。

步骤2：正面测试（Positive Testing）

正面测试的测试用例用于验证被测单元能够执行应该完成的工作。测试设计者应该查阅相关的设计说明，每个测试用例应该测试模块设计说明中一项或多项陈述。如果涉及多个设计说明，最好使测试用例的序列对应一个模块单元的主设计说明。

这个阶段适合的技术有：

1）设计说明导出的测试。

2）对等区间划分。

3）状态转换测试

步骤 3：负面测试（Negative Testing）

负面测试用于验证软件不执行其不应该完成的工作。这一步骤主要依赖于错误猜测，需要依靠测试设计者的经验判断可能出现问题的位置。

适合的技术有：

1）错误猜测。

2）边界值分析。

3）内部边界值测试。

4）状态转换测试。

步骤 4：模块设计需求中其他测试特性用例设计

如果需要，应该针对性能、余量、安全需要、保密需求等设计测试用例。在有安全保密需求的情况下，重视安全保密分析和验证是必要的。针对安全保密问题的测试用例应该在测试说明中进行标注。同时应该加入更多的测试用例测试所有的保密和安全问题。

适合的技术：设计说明导出的测试。

步骤 5：覆盖率测试用例设计

应该增加更多的测试用例到单元测试说明中以达到特定的测试覆盖率目标。一旦覆盖测试设计好，就可以构造测试过程和执行测试。覆盖率测试一般要求语句覆盖率和判断覆盖率。

适合的技术：

1）分支测试。

2）条件测试。

3）数据定义－使用测试。

4）状态转换测试。

步骤 6：测试执行

使用上述 5 个步骤设计的测试说明在大多数情况下可以实现一个比较完整的单元测试。到这一步，就可以使用测试说明构造实际的测试过程和用于执行测试的测试过程。该测试过程可能是特定测试工具的一个测试脚本。测试过程的执行可以查出模块单元的错误，然后进行修复和重新测试。在测试过程中的动态分析可以产生代码覆盖率测量值，以指示是否已经达到了覆盖目标。因此需要在测试设计说明中增加一个完善代码覆盖率的步骤。

步骤 7：完善代码覆盖

由于模块单元的设计文档规范不一，测试设计中可能引入人为的错误，测试执行后，复杂的决策条件、循环和分支的覆盖率目标可能并没有达到，这时需要进行分析找出原因。导致一些重要执行路径没有被覆盖的可能原因有：

1）不可行路径或条件。应该标注测试说明解释该路径或条件没有被测试的原因。

2）不可到达或冗余代码。对这种情况的正确处理方法是删除这些代码。但这种分析容易出错，特别是使用防卫式程序设计技术（Defensive Programming Techniques）时，如有疑义，

这些防卫性程序代码就不要删除。

3）测试用例不足。对这种情况应该重新提炼测试用例，设计更多的测试用例并添加到测试说明中以覆盖没有执行过的路径。

理想情况下，覆盖完善阶段应该在不阅读实际代码的情况下进行。然而，实际上，为达到覆盖率目标，看一下实际代码也是需要的。

适合的技术：

1）分支测试。

2）条件测试。

3）设计定义——试验测试。

4）状态转换测试。

2. 面向对象应用程序的单元测试用例设计

前面所讲述的内容属于针对传统软件进行测试的技术，其中的大部分测试方法已在第 2 章中进行了详细的叙述，这里不再赘述。读者可能会问，在对面向对象软件进行单元测试时也使用这些方法吗？回答是肯定的。但是，还会用到一些不同的测试用例设计技术，接下来将会对几种常见的方法进行介绍。

自 20 世纪 80 年代中后期以来，面向对象软件开发技术发展迅速，获得了越来越广泛的应用，在面向对象的分析、设计技术以及面向对象的程序设计语言方面，均获得了很丰富的研究成果。与之相比，面向对象软件测试技术的研究还相对薄弱。例如，对面向对象的程序测试应当分为多少级尚未达成共识。基于结构的传统集成策略并不完全适于面向对象的程序。这是因为面向对象的程序的执行实际上是执行一个由消息连接起来的方法序列，而这个方法序列往往是由外部事件驱动的。在面向对象语言中，虽然信息隐藏和封装使得类具有较好的独立性，有利于提高软件的易测试性和保证软件的质量，但是，这些机制与继承机制和动态绑定给软件测试带来了新的课题。尤其是面向对象软件中类与类之间的集成测试和类中各个方法之间的集成测试具有特别重要的意义，与传统语言书写的软件相比，集成测试的方法和策略也应该有所不同。

从目前的研究现状来看，研究较多地集中在类和对象状态的测试方面。面向对象程序设计的继承和动态联编所带来的多态性对软件测试的影响，虽然有所论及，但是不仅缺乏针对这一特点的测试方法，而且还有许多问题有待进一步的研究。

软件测试中的另一个重要问题是测试的充分性问题，充分性准则对软件测试的揭错能力具有重要影响。对传统语言的软件测试已经存在多种充分性准则，但对面向对象的软件测试，目前尚无普遍接受的充分性准则。对这些方面的深入研究将会产生真正对软件测试的理论与实践有指导意义、有影响的成果。

对 OO 软件的类测试相当于传统软件的单元测试，但与传统软件的单元测试不同。单元测试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。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。封装使得我们难于获得对象的状态，继承也给测试带来了难度，即使是彻底复用的，对每个新的使用语境也需要重新测试。多重继承更增加了需要测试的语境的数量，使测试进一步复杂化。如果从超类导出的测试用例被用于相同的问题域，有这些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途，必须设计新的测试

用例集。

在类的生命周期中，类测试只是一个初始的测试阶段。类作为独立的成分可以多次在不同的应用系统中重复使用，这些成分的用户要求每个类是可靠的，并无需了解其实现细节。这样的类要尽可能多地进行测试，因为我们关心的是类单元本身，而不是它所处的上下文，如类库中的 List、Stack 等基本类。

类的测试用例可以先根据其中的方法设计，然后扩展到方法之间的调用关系。如果类中的方法都已定义了前置/后置条件，则可以此作为开发对各方法进行测试所用的测试用例的参考。一般情况下，根据方法的前置、后置条件以及关于类的约束条件，利用一些传统的测试方法，也能设计出较完善的测试用例。

类测试一般也采用两种传统的测试方式：功能性测试和结构性测试，即黑盒测试和白盒测试。功能性测试以类的规格说明为基础，它主要检查类是否符合其规格说明的要求。

（1）功能性测试。功能性测试包括两个层次：类的规格说明和方法的规格说明。

1）类的规格说明：类的规格说明是各方法规格说明的组合及对类所表示概念的广义描述。对数据类型的形式化描述也可以用来对类进行定义，但类比类型的含义更广泛，具有更确切的语义，尤其是类之间的继承关系也被表示出来了。

一个 Java 类的规格说明具有多层性，但对它的用户来说，它只包括了在类定义公共区中方法的说明，子类所能见到的父类是其 public 和 protected 区域中的内容，一个类中所定义的方法可分为 3 个存取层次：public，protected 和 private。这些方法可以各自分开独立考虑，一个类是所有这些的综合。

2）方法的规格说明：每个独立方法的规格说明可以用其前置/后置条件描述。根据前置条件选择相应的测试用例，就可以检查其产生的输出是否满足后置条件而完成对独立方法的测试，对独立方法的测试与对独立过程的测试方法类似。

（2）结构性测试。结构性测试对类中的方法进行测试，它把类作为一个单元来进行测试。测试分为两层：第一层考虑类中各独立方法的代码；第二层考虑方法之间的相互作用，每个方法的测试要求能针对其所有的输入情况。对于一个类的测试要保证类在其状态的代表集上能够正确工作，构造函数的参数选择以及消息序列的选择都要满足这一准则。因此，在这两个不同的测试层次上应分别做到：

1）方法的单独测试：结构性测试的第一层是考虑各独立的方法，这可以与过程的测试采用同样的方法，两者之间最大的差别在于方法改变了它所在实例的状态，这就要取得隐藏的状态信息来估算测试的结果。传给其他对象的消息被忽略，而以桩来代替，并根据所传的消息返回相应的值。测试数据要求能完全覆盖类中代码，可以用传统的测试技术来获取。

2）方法的综合测试：第二层要考虑一个方法调用本对象类中的其他方法和从一个类向其他类发送信息的情况。单独测试一个方法时，只考虑其本身执行的情况，而没有考虑动作的顺序问题。综合测试用例中加入了激发这些调用的信息，以检查它们是否正确运行。对于同一类中方法之间的调用，一般只需要极少甚至不用附加数据，因为方法都是对类进行存取，故这一类测试的准则是要求遍历类的所有主要状态。

（3）基于对象—状态转移图的面向对象软件测试。面向对象设计方法通常采用状态转移图建立对象的动态行为模型。状态转移图用于刻画对象响应各种事件时状态发生转移的情况，

结点表示对象的某个可能状态，结点之间的有向边通常用“事件/动作”标出。

在如图 3-3 的所示示例中，当对象处于状态 A 时，若接收到事件 e，则执行相应的操作 a 并转移到状态 B。因此，对象的状态随各种外来事件发生怎样的变化，是考察对象行为的一个重要方面。

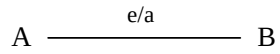

图 3-3 对象—状态转移图

基于状态的测试是通过检查对象的状态在执行某个方法后是否会转移到预期状态的一种测试技术。使用该技术能够检验类中的方法能否正确地交互，即类中的方法是否能通过对象的状态正确地通信。对象的状态是通过对象的数据成员的值反映出来的，所以检查对象的状态实际上就是跟踪监视对象数据成员的值的变化。如果某个方法执行后对象的状态未能按预期的方式改变，则说明该方法含有错误。

状态转移图中的结点代表对象的逻辑状态，而非所有可能的实际状态。理论上讲，对象的状态空间是对象所有数据成员定义域的笛卡尔乘积。当对象含有多个数据成员时，要对对象所有的可能状态进行测试是不现实的，这就需要对对象的状态空间进行简化，同时又不失对数据成员取值的“覆盖面”。简化对象状态空间的基本思想类似于黑盒测试中常用的对等区间的方法。依据软件设计规范或分析程序源代码，可以从对象数据成员的取值域中找到一些特殊值和一般性的区间。特殊值是设计规范里说明有特殊意义，在程序源代码中逻辑上需特殊处理的取值。位于一般性区间中的值不需要区别各个值的差别，在逻辑上以同样方式处理。例如下面的类定义：

```
…
public class Account{
string  name;
int accNum;
int balance;
…
}
```

依据常识可知，特殊值情况：name=NULL，accNum=0，balance=0；一般区间内：name！＝NULL，accNum<0 或 accNum>0，balance<0 或 balance>0。

进行基于状态的测试时，首先要对被测试的类进行扩充定义，即增加一些用于设置和检查对象状态的方法。通常是对每一个数据成员设置一个改变其取值的方法。另一项重要工作是编写作为主控的测试驱动程序，如果被测试的对象在执行某个方法时还要调用其他对象的方法，则需编写桩程序代替其他对象的方法。测试过程为：首先生成对象，接着向对象发送消息把对象状态设置到测试实例指定的状态，再发送消息调用对象的方法，最后检查对象的状态是否按预期的方式发生变化。

下面给出基于状态测试的主要步骤：

1）依据设计文档，或者通过分析对象数据成员的取值情况空间，得到被测试类的状态转移图。

2）给被测试的类加入用于设置和检查对象状态的新方法，导出对象的逻辑状态。

3）对于状态转移图中的每个状态，确定该状态是哪些方法的合法起始状态，即在该状态时，对象允许执行哪些操作。

4）在每个状态，从类中方法的调用关系图最下层开始，逐一测试类中的方法。

5）测试每个方法时，根据对象当前状态确定出对方法的执行路径有特殊影响的参数值，将各种可能组合作为参数进行测试。

（4）类的数据流测试。数据流测试是一种白盒测试方法，它利用程序的数据流之间的关系来指导测试的选择。现有的数据流测试技术能够用于类的单个方法测试及类中通过消息相互作用的方法的测试，但这些技术没有考虑到当类的用户以随机的顺序激发一系列的方法时引起的数据流交互关系。为了解决这个问题，测试研究人员提出了一种新的数据流测试方法，这个方法支持各种类的数据流交互关系。对于类中的单个方法及类中相互作用的方法，可以采用类似于一般的数据流测试方法；对于可以从外部访问类的方法，以及可以以任何顺序调用类时，我们计算数据流信息，并利用它来测试这些方法之间可能的交互关系。

1）数据流分析。当数据流测试用于单个过程的单元测试时，定义—引用对可利用传统的迭代的数据流分析方法来计算，这种方法利用一个控制流图（Control Flow Graph）来表示程序，其中的节点表示程序语句，边表示不同语句的控制流，且每一个控制流图都加上了一个入口和一个出口。为了将数据流测试技术应用到交互式过程中，需要有更精确的计算。过程间数据流分析（Interprocedural Dataflow Analysis）可以计算定义在一个过程中，而引用又在另一个过程中的定义-引用对，这种技术可以计算全局变量的定义—引用对，另外它在计算定义—引用对时还考虑指针变量及别名的影响。

利用上面的算法为程序建立一个过程间控制流图，它把单个的过程和控制流图结合在一起，并把每一个调用点用一个调用结点和一个返回结点代替，通过加入从调用结点到输入结点的边及从输出节点到返回结点的边表示过程的调用，从而把整个控制流图联系在一起。

2）类及类测试。类是个独立的程序单位，它应该有一个类名并包括属性说明和服务说明两个主要部分，对象是类的一个实例。不失一般性，下面构造一个类的模型。下面的程序代码是一个计算学生成绩的例子，它包括公开的方法如构造函数 Student ()，addCourse(int grade)，deleteCourse(int courseNum)，display()等。

```
1 public class Student{
2 int N=10;
3 private String name;
4 private int idNumber;
5 private float average;
6 private int a[N];
7 public Student(String name, int idNumber1) {
8  String name= name;
9  int idNumber=idNumber1;
10 average=0;
11 for(int i=0;i<N;i++) a[i]=-1;
12 }
```

```
13 private float averageGrade(){
14 int i; int j=0;
15 float sum=0;
16 for(i=0;i<N;i++)
17 if(a[i]>=0)
18 { sum+=a[i]; j++; }
19 float average1=sum/j;
20 return average1;
21 }
22 public Boolean addCourse(int grade) {
23 int i;
24 for(i=O;i<N;i++)
25 if(a[i]<0)
26 break;
27 if(i<N){
28 a[i]=grade;
29 average=averageGrade();
30 return true;
31 }
32 else
33 return false;
34 }
35 public Boolean deleteCourse(int courseNum) {
36 int i;
37 if(courseNum>N)
38 return false;
39 else {
40 for(i=0;i<N;i++)
41 if(i==courseNum-1)
42 break;
43 a[i]=一1;
44 average=averageGrade();
45 return true;
46 }
47 }
48 public void display(){
49 system.out.println(name+"的平均成绩是"+average);
50 }
51 }
```

我们用类调用图（Class Call Graph）来表示类的调用结构，在图 3-4 中，结点表示方法，边表示方法间的过程调用，图 3-4 为类 Student 的类调用图，addCourse()和 deleteCourse ()调用 averageGrade()，则有一条由 addCourse()指向 averageGrade()的边以及一条由 deleteCourse()指向 averageGrade()的边；图中还有一些虚线，它表示从类外部发给这些公开方法的消息。

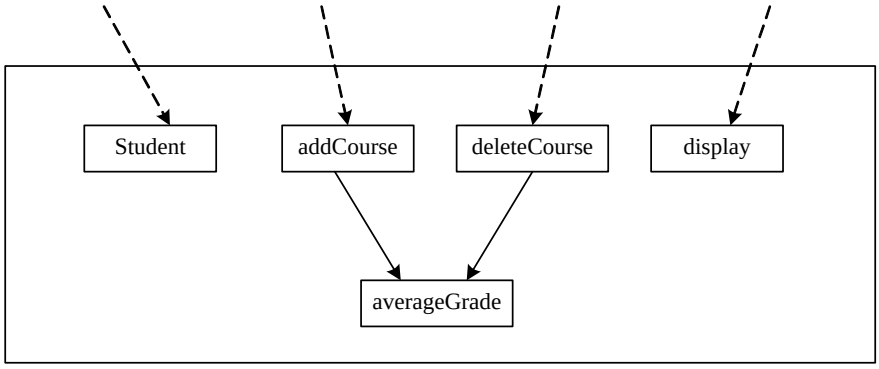

图 3-4　Student 的类调用图

我们对类进行三级测试，定义如下：

a）方法内部测试（Intra-method testing）：测试单个方法，这级测试相当于单元测试。

b）方法间测试（Inter-method testing）：在类中与其他方法一起测试一个直接或间接调用的公开方法，这级测试相当于集成测试。

c）类内部测试（Intra-class testing）：测试公开方法在各种调用顺序时的相互作用关系，由于类的调用能够激发一系列不同顺序的方法，可以用类内部测试来确定类的相互作用关系顺序，但由于公开方法的调用顺序是无限的，只能测试其中一个子集。

为了说明这些级别的测试，可以结合 Student 类来进行描述。在对类 Student 进行方法内部测试时，分别测试每一个方法（共有 5 个）；在对 addCourse()进行方法间测试时则要把 addCourse()和 averageGrade()等方法集成起来；类似地在对 deleteCourse()进行方法间测试时则要把 deleteCourse()和 averageGrade()等方法集成起来。

由于构造函数 Student 及方法 display 没有调用其他的方法，对它们进行方法内和方法间的测试是等价的。为了进行类内部测试，可以选择诸如<Student,addCourse, display>和<Student, addCourse, addCourse, deleteCourse, display>这样的测试序列。

3）数据流测试。为了支持现有的类内部测试（Intra-class testing）技术，我们需要一个基于代码的测试技术来识别需要测试的类的部件，这种技术就是数据流测试，它考虑所有的类变量及程序点说明的定义—引用对（Def-Use pairs）。在类中共有3种定义—引用对需要测试，这3种类型分别与前面所定义的相对应，设C为需要测试的类，d表示为一个包含定义（definition）的状态，u为包含引用（use）的状态，则3种定义—引用时的定义如下：

- 方法内部定义—引用对（Intra-method def-use pairs）：设 M 为类 C 中的一个方法，如果 d 和 u 都在 M 中，且存在一个程序 P 调用 M，则在 P 中当 M 激发时，(d,u)为一个引用对，那么(d,u)为一个方法内部定义—引用对。
- 方法间定义—引用对（Inter-method def-use pairs）：设 M_0 为 C 中的一个公开方法，{M_1，M_2,…,M_n}为 C 直接或间接调用的方法集。设 d 在 M_i 中，u 在 M_j 中，且 M_i、M_j 都在{M_1,M_2,…,M_n}中，如果存在一个程序 P 调用 M_0，则在 P 中当 M_0 激发且 $M_i \neq M_j$ 或 M_i、M_j 被同一个方法分别激发时，(d,u)为一个引用对，那么(d,u)为一个方法间定义—引用对。

- 类内部定义—引用对（Intra-class def-use pairs）：设 M_0 为 C 中的一个公开方法，$\{M_1,M_2,\cdots,M_n\}$为 C 直接或间接调用的方法集，设 N_0 为 C 中的一个公开方法（可能与方法 M_0 相同），$\{N_1,N_2,\cdots,N_n\}$为 C 直接或间接调用的方法集，设 d 在$\{M_1,M_2,\cdots,M_n\}$的某个方法中，u 在$\{N_1,N_2,\cdots,N_n\}$的某个方法中，如果存在一个程序 P 调用 M_0 和 N_0，且在 P 中(d,u)为一个引用对，并且在 d 执行之后，u 执行之前，M_0 的调用就中止了，那么(d,u)为一个类内部定义—引用对。

一般来说，方法内部定义—引用对出现在单个的方法中，且测试定义—引用对的相互作用时也限于这些方法中。例如，在 Student 类中，averageGrade 方法包含一个关于 sum 的方法内部定义—引用对(18,19)，即变量 sum 在 18 行中定义，sum 的引用则在 19 行中。方法间定义—引用对出现在单个公开方法被调用后方法之间相互作用之中，定义出现在一个方法中，引用则出现在通过公开方法直接或间接调用这个方法的另一个方法中。例如在类 Student 中，addCourse 方法调用 averageGrade()方法，接收 a[i]的值并使用在 addCourse 方法中，定义引用对(28,18)是一个方法间定义—引用对，即 a[i]的定义出现在方法 addCourse 中（28 行）而 a[i]的使用出现在方法 averageGrade 中（18 行）。

类内部定义—引用对出现在一系列公开方法被激发时。例如，在方法序列<addCourse, display>中，addCourse 通过调用 averageGrade 计算出平均分 average，而 display 则显示该学生的有关信息（包括 average）。在该调用序列中，在程序的 44 行对 average 进行了定义，而在程序的 49 行对该变量进行了引用，这样(44,49)就构成了类内部定义—引用对。

上面所提及的 3 种定义—引用对于类的测试是非常有用的。例如，当使用 all-uses 数据流覆盖准则时，则方法内的定义—引用对(18,19)就能检测 averageGrade 是否能正确执行，计算出学生的平均分 average。方法间的定义—引用对(28,18)可以检测增加的课程成绩是否正确地存放到数组中相应的位置上。类内部定义—引用对(44,49)可以检测是否能将增加课程后的平均分信息取出来。类内部定义一引用对还有一个好处，就是可以指导测试者选取应该运行的方法序列和不必运行的方法序列。例如，为了执行类内定义—引用对(44,49)，必须测试方法序列<addCourse, display>，然而没有类内定义—引用对开始于方法 display 而结束于方法 addCourse，因此测试者可以不必测试方法序列<display,addCourse>。

4）计算类的数据流信息。为了支持类的数据流测试，必须计算类的各种定义—引用对。前面描述的算法对于计算方法内部及方法间的定义—引用对是有用的，但由于它需要一个完整的程序来构造一个控制流图，因此不能直接用于计算类内部定义—引用对。为了计算类内部定义—引用对，我们必须考虑当一系列的公开方法被调用时的相互作用。可以考虑建立一个图来描述这些相互作用，然后用类似的算法来计算它。

为了计算类的 3 种定义—引用对，可以构造一个类控制流图（Class Control Flow Graph，CCFG），其算法如下：

a）为类构造类调用图，作为类控制流图的初值。

b）把框架（frame）加入到类调用图中。

c）根据相应的控制流图替换类调用图中的每一个调用结点，具体实现方法是：对于类 C 中的每一个方法 M，在类调用图中用方法 M 的控制流图替代方法 M 的调用结点，并更新相应的边。

d）用调用结点和返回结点替换调用点，具体实现方法是：对于类调用图中的每一个表示

类 C 中调用方法 M 的调用点 S，用一个调用结点和返回结点代替调用点 S。

e）把单个的控制流图连接起来，具体实现方法是：对于类控制流图中的每一个方法 M，加上一条从框架调用结点到输入结点的边和一条从输出结点到框架返回结点的边，其中输入结点和输出结点都在方法 M 的控制流图中。

f）返回完整的类控制流图。

具体实施时，我们为类 C 构造一个类调用图，并且把它作为类控制流图的原形。用一个框架把类调用图包围起来，这个框架有助于数据流分析，它是类的一个驱动模块，可以模拟调用公开方法的随机序列。一个框架包含 5 个结点：框架输入（Frame Entry）和框架输出（Frame Exit），它们分别表示从框架输入和输出，框架循环（Frame Loop），它促进方法的序列化，框架调用（Frame Call）和框架返回（Frame Return），它们分别表示从任何公开方法进行的调用和返回。框架还包括 4 条边：（框架输入，框架循环）、（框架循环，框架调用）、（框架循环，框架输出）和（框架返回，框架循环）。图 3-5 为类 Student 包含框架的类调用图，在此图中，框架结点用虚线框表示。此时构造的类控制流图中框架和类调用图并没有联系。

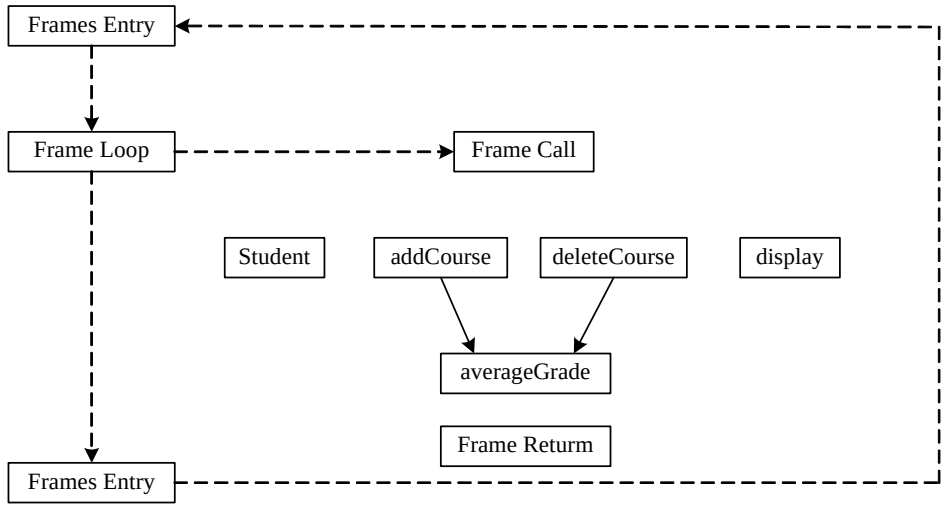

图 3-5　Student 包含框架的类调用图

有了类的控制流图，就可以基于各种逻辑覆盖标准来设计测试用例，从而通过运行这些测试用例对类进行测试，发现代码中隐藏的缺陷。

本节介绍了类的数据流测试技术，定义了 3 种数据流测试：①方法内部测试，用于测试单个类方法；②方法间测试，用于测试一个类中不同方法通过过程调用的相互作用；③类内部测试，用于测试一系列方法的调用。为了区分这 3 种测试的定义—引用对，我们把类作为一个单输入、单输出的程序，并为这个程序构造一个类控制流图。

3.6　单元测试过程

前面已经了解了有关单元测试的一些常识性知识，下面简单介绍一下单元测试的过程。工作性质的不同，决定了工作的侧重点也会不同，因此程序开发人员在单元测试的过程中关注

得更多的是程序代码本身和已经实现的功能。因此，站在他们的角度看，单元测试的过程就是在编写测试方法之前，首先考虑如何对需要测试的方法进行测试，然后编写测试代码；下一步就是运行某个测试，或者同时运行该单元的所有测试，确保所有测试都通过。目的就是在不直接引入 BUG 的同时，也不破坏程序的其他部分。最后就是检查和分析测试结果。读者可能担心这样会很麻烦，其实现在有很多单元测试工具可供开发人员使用，能够大大地简化测试过程。相比较而言，专业测试人员则应该关注更加完整的整个测试过程。但二者的出发点和实质是一样的，都是为了达到使软件质量能够得到保证的目标。图 3-6 从宏观的角度概括了单元测试的工作过程图。

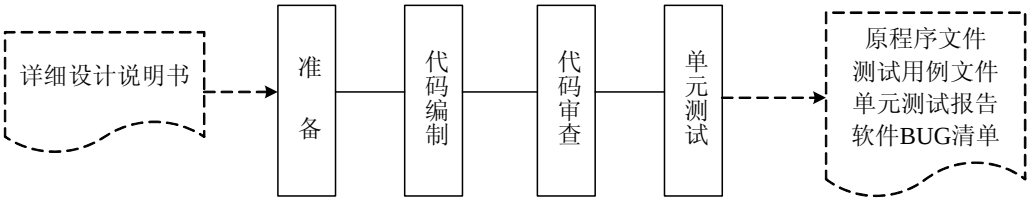

图 3-6　单元测试工作过程

1. 单元测试进入和退出准则

单元测试进入和退出准则如表 3-1 和表 3-2 所示。

表 3-1　进入准则

要素	判断准则
详细设计说明书 单元测试用例	● 经过审查 ● 获得批准 ● 进入配置库

表 3-2　退出准则

要素	判断准则
源代码文件 源代码文件清单	● 源代码文件获得批准 ● 源代码文件进入配置库的源代码区 ● 测试用例源代码通过同级评审
软件 BUG 清单 单元测试报告	● 提交测试负责人 ● 提交软件产品配置管理

注意：与代码相关的数据文件、修改日志、编译环境文件和源程序文件清单也包含在源代码文件中。

2. 单元测试过程

（1）准备阶段。

1）培训。

a）根据程序员的实际水平进行有关编程语言、编程规范、编程方法、编程工具、调试方法、配置管理等方面的培训。

b）根据测试人员的实际水平进行有关测试方法、测试工具、问题汇报方法等方面的培训；

有关被测产品的功能培训。

2）准备开发及测试工具和环境，如有必要在各编码组内对临时的编译环境和调试方法进行约定。

3）对详细设计说明书需要做进一步确认工作，保证接口、工作流程的一致性。如果是多人参与开发，还需根据实际情况对参与人员进行设计讲解工作。

4）根据单元划分情况编写单元测试用例，并审查是否达到测试需求。

（2）编制阶段。

1）程序员依据详细设计，进行程序单元的编制工作（包括建立相关的构造环境），并调试和检查。

2）在更正测试问题时，修改源码和测试用例，提交新的源码文件。

（3）代码审查阶段。将编制的源代码文件进行静态代码审查，填写代码审查表（作为单元测试报告附录形式提交）。

在代码审查阶段，必须执行的活动有以下几个项目：

1）检查算法的逻辑正确性。确定所编写代码的算法、数据结构定义（如：队列、堆栈等）是否实现了模块或方法所要求的功能。

2）模块接口的正确性检查。确定形式参数个数、数据类型、顺序是否正确；确定返回值类型及返回值的正确性。

3）输入参数有没有做正确性检查。如果没有做正确性检查，要确定该参数是否的确无需做参数正确性检查，否则要添加参数的正确性检查。经验表明，缺少参数正确性检查的代码是造成软件系统不稳定的主要原因之一。

4）调用其他方法接口的正确性。检查实参类型正确与否，传入的参数值正确与否，个数正确与否，特别是具有多态的方法。检查返回值正确与否，有没有误解返回值所表示的意思。最好对每个被调用的方法的返回值用显示代码做正确性检查，如果被调用方法出现异常或错误程序应该给予反馈，并添加适当的出错处理代码。

5）出错处理。模块代码要求能预见出错的条件，并设置适当的出错处理，以便一旦在程序出错时，能对出错程序重做安排，保证其逻辑的正确性，这种出错处理应当是模块功能的一部分。若出现下列情况之一，则表明模块的错误处理功能包含有错误或缺陷：出错的描述难以理解；出错的描述不足以对错误定位，不足以确定出错的原因；显示的错误信息与实际的错误原因不符；对错误条件的处理不正确；在对错误进行处理之前，错误条件已经引起系统的干预等。

6）保证表达式、SQL 语句的正确性。检查所编写的 SQL 语句的语法、逻辑的正确性。对表达式应该保证不含二义性，对于容易产生歧义的表达式或运算符优先级（如：《、=、》、&&、||、++、--等）可以采用扩号“（）”运算符避免二义性，这样一方面能够保证代码的正确可靠，同时也能够提高代码的可读性。

7）检查常量或全局变量使用的正确性。确定所使用的常量或全局变量的取值和数值、数据类型；保证常量每次引用同它的取值、数值和类型的一致性。

8）表示符定义的规范一致性。保证变量命名能够见名知意，并且简洁但不宜过长或过短、规范、容易记忆、最好能够拼读，并尽量保证用相同的表示符代表相同功能，不要将不同的功能用相同的表示符表示，更不要用相同的表示符代表不同的功能意义。

9）程序风格的一致性、规范性。代码必须能保证符合企业规范，保证所有成员的代码风格一致、规范、工整。例如对数组做循环，不要一会儿采用下标变量从下到上的方式（如：for(i=0;i++;i<10)），一会儿又采用从上到下的方式（如：for(i=10;i--;i>0)）；应该尽量采用统一的方式，或者统一从下到上，或者统一从上到下。建议采用 for 循环和 while 循环，不要采用 do{}while 循环等。

10）检查程序中使用到的神秘数字是否采用了表示符定义。神秘的数字包括各种常数、数组的大小、字符位置、变换因子以及程序中出现的其他以文字形式写出的数值。在程序源代码里，如果一个具有原本形式的数对其本身的重要性或作用没提供任何指示性信息，那么它们也会导致程序难以理解和修改。对于这类神秘数字必须采用相应的标量来表示。如果该数字在整个系统中都可能使用到，务必将它定义为全局常量；如果该神秘数字在一个类中使用，可将其定义为类的属性（Attribute）；如果该神秘数字只在一个方法中出现，务必将其定义为局部变量或常量。

11）检查代码是否可以优化，算法效率是否最高。如：SQL 语句是否可以优化，是否可以用一条 SQL 语句代替程序中的多条 SQL 语句的功能，循环是否必要，循环中的语句是否可以抽出到循环之外等。

12）检查程序是否清晰简洁容易理解。需要注意的是，冗长的程序并不一定不是清晰的。

13）检查方法内部注释是否完整，是否清晰简洁，是否正确地反映了代码的功能。错误的注释比没有注释更糟。还要检查是否做了多余的注释，对于简单的一看就懂的代码没有必要注释。

14）检查注释文档是否完整。对包、类、属性、方法功能、参数、返回值的注释是否正确且容易理解；是否会落了或多了某个参数的注释，参数类型是否正确，参数的限定值是否正确。特别是对于形式参数与返回值中关于神秘数值的注释，如：对于类型参数，应该指出①代表什么；②代表什么；③代表什么等。对于返回结果集（Result Set）的注释，应该注释结果集中包含哪些字段及字段类型、字段顺序等。

（4）单元测试阶段。

1）从配置库获取源码文件，设计测试用例，执行测试用例，并利用相关测试工具对单元代码进行测试，将测试结论填写到单元测试报告和软件 BUG 清单中。

2）把软件 BUG 清单和测试用例执行结果提交测试负责人，并纳入质量管理。对源码文件进行的测试，视程序存在缺陷的情况，可能要重复进行，直至问题解决。

3）单元测试的执行者，一般情况下可由程序的编码者担当，特殊情况可由独立于编码者的测试人员进行。

（5）评审、提交阶段。

1）对源代码文件进行同级评审，给出评审结论（由审查人员填写产品批准表），并将其提交到配置库中。

2）上述过程完成后，开发人员应提交源代码、代码清单、单元测试用例代码及单元测试报告。测试人员提交该版本的软件 BUG 清单，代码审查人员提交产品批准表。

上面所列出的测试环节可供读者参考，在具体的单元测试过程中可能会因实际工作要求的不同和具体单元测试目标的不同有所增加、补充或修改，当然也有一些公司内部会专门规定相关的单元测试流程和单元测试规范。

3.7　单元测试举例

从上一节所介绍的单元测试过程中，我们已经了解到单元测试的大体过程。在本节中，将根据两票系统项目的实际情况，以该项目的一个单元测试为例来说明如何进行单元测试。当然，这里忽略了单元测试过程中的很多细节，只介绍单元测试过程中的关键环节，如：根据被测试单元的实际情况制定的单元测试计划；测试分析和设计；测试的执行等方面。

1. 单元测试计划

（1）简介。这份文档的目标是详细描述对两票系统的可以实现在二次系统图上进行图形开票模块的功能验证的测试过程。本文档所关注的特征来自于需求文档，包括二次图模块和需求定义。需求文档的标识是 TwoBill-RD-02，它在文档控制系统中，读者可以从如下的网址中获得。http://www.tmtcointernal. com/usr/ www/docstores/design/requirements/TwoBill-RD-02.doc。

（2）本测试的总目标。测试该模块是否实现了需求文档中定义的所有功能。

（3）完整性需求。在测试该模块是否实现了需求文档中定义的所有功能之前，应该做如下两项工作①测试数据初始化是否无误；②测试二次图开票 GUI 界面是否与系统维护模块显示一致。

（4）单元测试终止标准。

1）硬件资源不足或故障造成软件无法运行。

2）软件运行后无法正确显示（如：因数据初始化有误造成 GUI 界面同系统维护模块显示不一致或不正确等）。

3）所有功能测试均已经完成。

（5）资源需求。

1）软件资源。

操作系统：Windows 2000

Web 服务器：Tomcat 5.0

数据库服务器：SQL Server 2000

浏览器：Microsoft IE 6.0

测试工具：Junit

2）硬件资源。同开发用 PC 机配置相同即可。

3）测试进度。如表 3-3 所示。

表 3-3　二次系统图图形开票模块

任务	开始日期	结束日期
配置并调试测试环境	10/01	10/03
GUI 界面测试	10/04	10/06
所有功能测试	10/07	10/10

（6）准备测试的特征。以下特征将被测试，以确保该模块能满足需求规格说明书中指定的需求：

需求 2.2.1 用户界面

需求 2.2.2 弹出菜单

需求 2.2.3 图形开票

（7）不准备测试的特征。本次测试将不考虑是否能够同一次系统图图形开票模块的集成。

（8）测试方法。该单元测试方法包括功能测试、GUI 测试和数据测试。

- 功能测试：对需求定义中所描述的所有功能进行测试。
- GUI 测试：我们将使用 Microsoft IE 对图形用户界面进行测试，对所有功能以及功能间的切换进行测试，对于弹出菜单的测试借助 IBM Rational Functional Tester 工具进行。
- 数据测试：对绘制二次系统图的初始化数据进行测试，主要借助 Junit 测试工具来实现。

（9）通过/失败标准。每个测试用例的通过/失败标准都由它预期的结果来描述。如果在执行一个测试用例时得到了预期的结果，那么测试就通过了。如果在执行一测试用例时没有得到预期的结果，那么测试就失败了。如果因为构建中存在一些阻碍性缺陷而未能执行某项测试，则该测试的结果将记为“受阻”。

要让该模块退出单元测试阶段，需要至少在软件的一个构件上运行本测试计划定义的所有测试用例。所有运行的测试都要通过，在测试结束时不能有未修复的灾难级错误。

（10）测试结束后须提交的产物。包括以下几个文档：

- 本测试计划。
- 测试规格说明文档。
- 测试结果报告。
- 向测试经理和开发经理提交的每日测试状态报告。
- 缺陷报告。

（11）测试执行人员。由开发人员来执行该模块的测试。

（12）风险和应急计划。如果在执行该模块的测试之前，二次系统图的图片可能还没有完全准备好，那么测试负责人将参加其他模块的开发设计会议。

2. 测试的设计和开发

前面已经提到过，测试设计和开发要以测试计划作为输入。在设计测试时，首先应该明确测试目标（细化测试的方法和范围）；确定每个测试的输入规格说明；确定每个测试使用的测试配置；复查测试设计的覆盖率和测试的准确度。

在对本单元进行测试时所采用的方法是：先完成黑盒测试，然后统计白盒覆盖率，再针对未覆盖的逻辑单位设计测试用例覆盖它，例如，先检查是否有语句未覆盖，有的话设计测试用例覆盖它，然后用同样方法完成条件覆盖、分支覆盖和路径覆盖。这样，既检验了黑盒测试的完整性，又避免了重复的工作，达到了非常高的测试完整性。不过，这些工作可不是手工能完成的，必须借助于工具。用于白盒测试的工具有很多种，读者可参考第 7 章的相关内容。

然后，就应该开发测试用例，应该注意的是在测试用例中应该尽可能详细地描述测试过

程，对于耗时的测试进行自动化。

最后，验证和调试测试。

下面详细介绍该阶段的具体过程，drawbh 类的代码如下。

```
import java.awt.*;
import java.awt.Graphics;
import java.io.*;

import java.text.*;
import java.awt.Image;
import java.awt.image.*;
import java.awt.event.*;//为了利用鼠标事件
import java.applet.Applet;//为了利用 Applet
import java.lang.*;
import java.net.*;
import java.sql.*;

import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JPopupMenu;

public class drawbh extends JApplet
implements MouseListener, MouseMotionListener,ItemListener,ActionListener{
       …
       …
        db_Czp datadb=new db_Czp();//数据连接
        ResultSet rs=null;
        String sqlString=null;
        String bds_tablename;//所选用的表
       bds_tablename="bds20";
       int idx_bhdyname;//mydata_bh 数组维数
        int idx_bhbhname;
        int idx_bhkname;//mydata_bhk 数组维数
        MyData_bhk mydata_bhk[][]=new MyData_bhk[arrSize1][arrSize2];
        MyData_bh  mydata_bh[][]=new MyData_bh[arrSize1][arrSize2];
        int bh;
        int bh1;
        int bhk;
       …
    public void init_bh(){//保护数据初始化
      try{
      sqlString="select * from " +bds_tablename + "_abh";
        rs=datadb.executeQuery(sqlString);
```

```
  idx_bhdyname=0;
  idx_bhbhname=0;
 mydata_bh[idx_bhdyname][idx_bhbhname]=new MyData_bh();

 if(rs.next()){
 mydata_bh[idx_bhdyname][idx_bhbhname].setidx(rs.getInt("idx"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setflagidx(rs.getInt("flagidx"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setx(rs.getInt("x"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setY(rs.getInt("y"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setbdsname(rs.getString("bdsname"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setdyname(rs.getString("dyname"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setBHname(rs.getString("bhname"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setBhkindidx(rs.getInt("bhkindidx"));
mydata_bh[idx_bhdyname][idx_bhbhname].setstation(rs.getInt("station"));
mydata_bh[idx_bhdyname][idx_bhbhname].setCzRul0(rs.getString("czrul0"));
mydata_bh[idx_bhdyname][idx_bhbhname].setCzRul1(rs.getString("czrul1"));
 bh=mydata_bh[0][0].getidx();
   }
 …
 …
 for(int idx_bhdyname=1;idx_bhdyname<=bh;idx_bhdyname++){
 if(rs.next()){
 idx_bhbhname=0;
 mydata_bh[idx_bhdyname][idx_bhbhname]=new MyData_bh();
 mydata_bh[idx_bhdyname][idx_bhbhname].setidx(rs.getInt("idx"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setflagidx(rs.getInt("flagidx"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setx(rs.getInt("x"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setY(rs.getInt("y"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setbdsname(rs.getString("bdsname"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setdyname(rs.getString("dyname"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setBHname(rs.getString("bhname"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setBhkindidx(rs.getInt("bhkindidx"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setstation(rs.getInt("station"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setCzRul0(rs.getString("czrul0"));
 mydata_bh[idx_bhdyname][idx_bhbhname].setCzRul1(rs.getString("czrul1"));
//while(rs.next())
 bh1=mydata_bh[idx_bhdyname][0].getidx();
           }
      for(idx_bhbhname=1;idx_bhbhname<=bh1;idx_bhbhname++)
      {
      if(rs.next()){
      mydata_bh[idx_bhdyname][idx_bhbhname]=new MyData_bh();
      mydata_bh[idx_bhdyname][idx_bhbhname].setidx(rs.getInt("idx"));
```

```
                mydata_bh[idx_bhdyname][idx_bhbhname].setflagidx(rs.getInt
("flagidx"));
                mydata_bh[idx_bhdyname][idx_bhbhname].setx(rs.getInt("x"));
                mydata_bh[idx_bhdyname][idx_bhbhname].setY(rs.getInt("y"));
                mydata_bh[idx_bhdyname][idx_bhbhname].setbdsname(rs.getString
("bdsname"));
                mydata_bh[idx_bhdyname][idx_bhbhname].setdyname(rs.getString
("dyname"));
                mydata_bh[idx_bhdyname][idx_bhbhname].setBHname(rs.getString
("bhname"));
                mydata_bh[idx_bhdyname][idx_bhbhname].setBhkindidx(rs.getInt
("bhkindidx"));
                mydata_bh[idx_bhdyname][idx_bhbhname].setstation(rs.getInt
("station"));
                mydata_bh[idx_bhdyname][idx_bhbhname].setCzRul0(rs.getString
("czrul0"));
                mydata_bh[idx_bhdyname][idx_bhbhname].setCzRul1(rs.getString
("czrul1"));

                        }
                 }
                 //  idx_bhbhname--;
                                                  }
                //idx_bhdyname--;

             datadb.closeResultSet(rs);
                }
                catch(SQLException e) {
                 e.printStackTrace();
                                 }
                              }


            public void init_bhk(){//保护种类数据初始化
             try{
               int i=0;
               int j=0;
               sqlString="select * from " + bds_tablename + "_abhk";
               rs=datadb.executeQuery(sqlString);
               idx_bhkname=0;
               idx_bhcz=0;

               if(rs.next()){

               mydata_bhk[idx_bhkname][idx_bhcz]=new MyData_bhk();
               mydata_bhk[idx_bhkname][idx_bhcz].setidx(rs.getInt("idx"));
               mydata_bhk[idx_bhkname][idx_bhcz].setflagidx(rs.getInt("flagidx"));
```

```
    mydata_bhk[idx_bhkname][idx_bhcz].setname(rs.getString("name"));
    mydata_bhk[idx_bhkname][idx_bhcz].setczts(rs.getString("czts"));
    mydata_bhk[idx_bhkname][idx_bhcz].setcznr(rs.getString("cznr"));
    mydata_bhk[idx_bhkname][idx_bhcz].setState(rs.getInt("state"));
    mydata_bhk[idx_bhkname][idx_bhcz].setBMPname (rs.getString("bmpname"));

    //while(rs.next())
    bhk=mydata_bhk[0][0].getidx();
             }

for(int idx_bhkname=1;idx_bhkname<=bhk;idx_bhkname++){

     for(int idx_bhcz=0;idx_bhcz<=10;idx_bhcz++){
          if(rs.next()){

    mydata_bhk[idx_bhkname][idx_bhcz]=new MyData_bhk();
    mydata_bhk[idx_bhkname][idx_bhcz].setidx(rs.getInt("idx"));
    mydata_bhk[idx_bhkname][idx_bhcz].setflagidx(rs.getInt("flagidx"));
    mydata_bhk[idx_bhkname][idx_bhcz].setname(rs.getString("name"));
    mydata_bhk[idx_bhkname][idx_bhcz].setczts(rs.getString("czts"));
    mydata_bhk[idx_bhkname][idx_bhcz].setcznr(rs.getString("cznr"));
    mydata_bhk[idx_bhkname][idx_bhcz].setState(rs.getInt("state"));
    mydata_bhk[idx_bhkname][idx_bhcz].setBMPname (rs.getString("bmpname"));


                    }
                                              }
     }
     datadb.closeResultSet(rs);
     }
     catch(SQLException e) {
      e.printStackTrace(); }
     }

  public void init_bhscreen(){//保护屏列表初始化


 for(int i=1;i<=bh;i++){
 bhscreen.add(mydata_bh[i][0].getdyname());
                }
                      }
public boolean SearcBhidx(int xe,int ye){//判断选中的是哪一个保护(bhname)
boolean isinrect =false;
int j;
double x1,y1;
double x2,y2;
```

```
          for(j=1;j<=mydata_bh[index+1][0].getidx();j++){

             x1=mydata_bh[index+1][j].getx();
             y1=mydata_bh[index+1][j].getY();
             String pictureName;
             pictureName=mydata_bhk[mydata_bh[index+1][j].getBhkindidx()]
[mydata_bh[index+1][j].getstation()].getBMPname();
             try{
             int num=pictureName.indexOf(".");
             if(num!=-1){
             String str3;
             str3=pictureName.substring(0,num);
             img=getImage(getCodeBase(),"image/"+str3+".gif");
                      }
             w=img.getWidth(this);
             h=img.getHeight(this);
             //double pictureBl;
             // pictureBl=0.0378;
             // x2 = x1 + w * pictureBl;
             //y2 = y1 + h * pictureBl;
             x2 = x1+250+w;
             y2 = y1+50+h;
           if(xe>(x1+250)&&xe<x2&&ye>(y1+50)&&ye<y2){
             ThisBHidx=j;
             isinrect =true;
                                          }
           }
         catch(NullPointerException e){}
                                              }
         return isinrect;
       }
          public void PopMenu(int ThisBHidx1){//弹出菜单初始化
          //调用 SearcBhidx()方法可得到 ThisBHidx 的值
          popmenu.removeAll();
         //int ThisBHKindidx;
         if(ThisBHidx1!=0){
         ThisBHKindidx = mydata_bh[index+1][ThisBHidx1].getBhkindidx();
         int station;
         station = mydata_bh[index+1][ThisBHidx1].getstation();
         //Thisstate = station;

         for(int j=1;j<=14;j++){
          item[j]=new JMenuItem();
          item[j].setEnabled(true);
          item[j].addActionListener(this);
                          }
```

```
          for(i=1;i<=mydata_bhk[ThisBHKindidx][0].getidx();i++){

              item[i].setVisible(true);
              item[i].setText(mydata_bhk[ThisBHKindidx][i].getczts());
              //'判断菜单是否有效
              if(mydata_bhk[ThisBHKindidx][i].getState()==0){//'不改变状态
              if(mydata_bhk[ThisBHKindidx][i].getBMPname().equals(mydata_bhk
[ThisBHKindidx][station].getBMPname()))
                   item[i].setEnabled(true);
                else
                   item[i].setEnabled(false);
                                                 }

              else
                 {
                    if(!mydata_bhk[ThisBHKindidx][i].getBMPname().equals
(mydata_bhk[ThisBHKindidx][station].getBMPname()))
                   item[i].setEnabled(true);
                 else
                   item[i].setEnabled(false);
                 }
                                                        }
               item[mydata_bhk[ThisBHKindidx][0].getidx()+1].setVisible(true);
               item[mydata_bhk[ThisBHKindidx][0].getidx()+1].setText("取消");
               for(int j=mydata_bhk[ThisBHKindidx][0].getidx()+2;j<=14;j++){
               item[j].setVisible(false);
                                                        }
                                    }
               // for(int j=1;j<=mydata_bhk[index+1][0].getidx();j++){
                for(int j=1;j<=mydata_bhk[ThisBHKindidx][0].getidx();j++){
               popmenu.add(item[j]);
                                                          }
               popmenu.addSeparator();
               // popmenu.add(item[mydata_bhk[index+1][0].getidx()+1]);
               popmenu.add(item[mydata_bhk[ThisBHKindidx][0].getidx()+1]);
               popmenu.validate();
                                          }
              …
              …
               public void mouseMoved(MouseEvent e){ }
               public void mouseClicked(MouseEvent e){}
               public void mouseEntered(MouseEvent e){}
               public void mouseExited(MouseEvent e){}
               public void mouseReleased(MouseEvent e){}
               public void mouseDragged(MouseEvent e){}
          }
```

从上述的代码中，可以看到这是一个 Applet，因此需要嵌入到 HTML 文件中依赖浏览器才能够运行，进而显示运行界面。为了对该类进行功能测试，我们编辑了一个简单的 HTML 文件，即：

```
<html>
<body>
<applet code=" applet.unedu.tlpe.drawbh.class" width=1000 height=800> </applet>
</body>
</html>
```

然后，双击并运行该文件，得到的界面如图 3-7 所示。

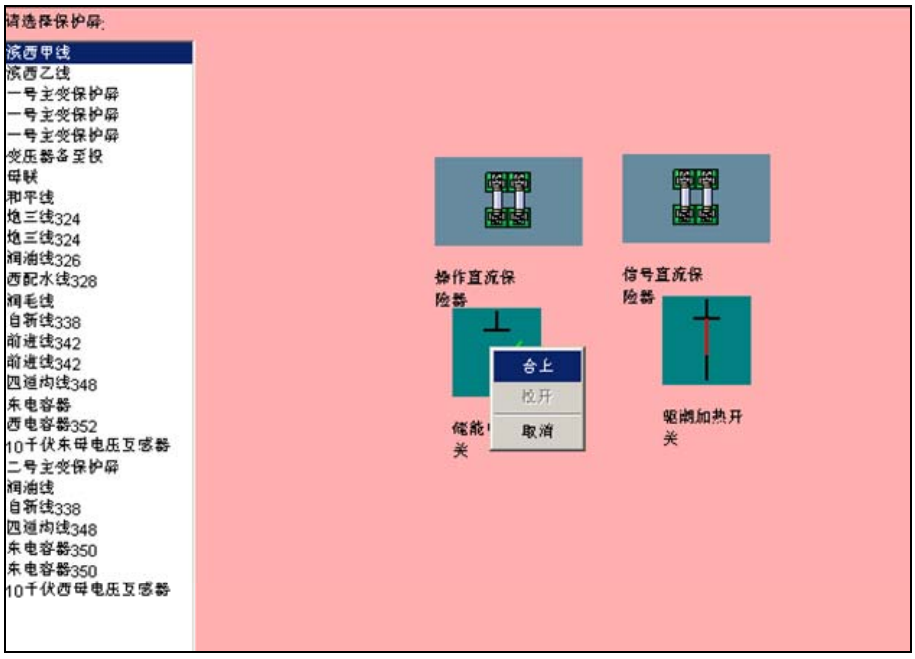

图 3-7　显示弹出菜单的二次系统图界面

该类所实现的功能如下：显示某个分厂（通过 bds_tablename 变量来确定）的二次系统图，可以实现图形开票功能，例如：在二次系统图界面选中了储能电源单击右键后，界面如图 3-7 所示，然后选择“合上”菜单项，储能开关由拉开状态（如图 3-8 所示）转换成合上状态（如图 3-9 所示）。同时，将操作内容（1.合上滨西甲线储能电源开关）写入操作票内容显示和文字编辑界面（如图 3-10 所示）。

首先，确定对该模块所进行的黑盒测试，所进行的测试项如下。

（1）界面测试。

1.1 测试在保护屏列表中所显示的名称是否正确。

1.2 测试二次系统图显示是否正确，如：当选择了滨西甲线后，在界面的右侧区域是否正常显示了与该保护屏相对应的元件，即：如图 3-7 所示的“操作直流保险器”、“信号直流保险器”、“储能电源开关”、“驱潮加热开关”。

1.3 测试选中每类元件单击右键后，能否显示弹出菜单，以及所显示的弹出菜单项是否正确。

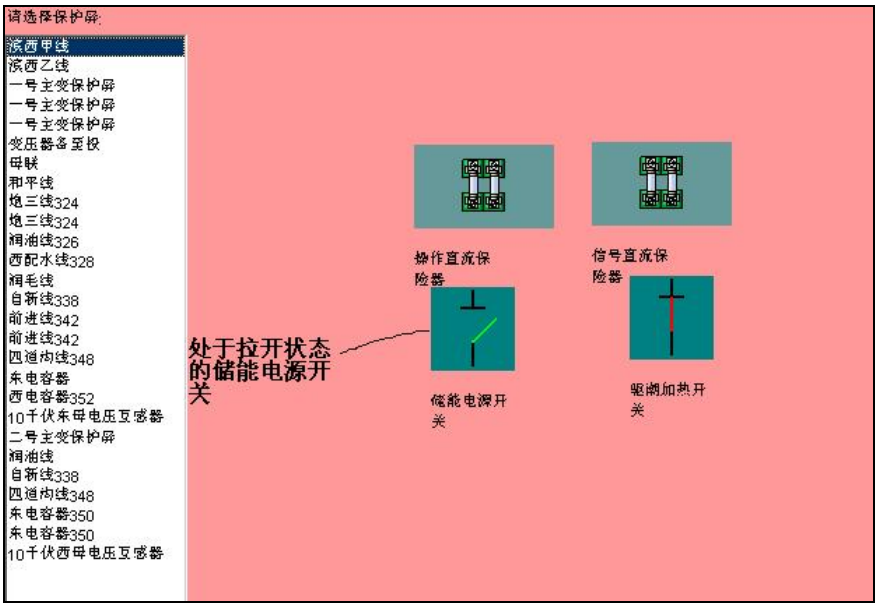

图 3-8 储能电源开关位于“拉开”状态的二次系统图界面

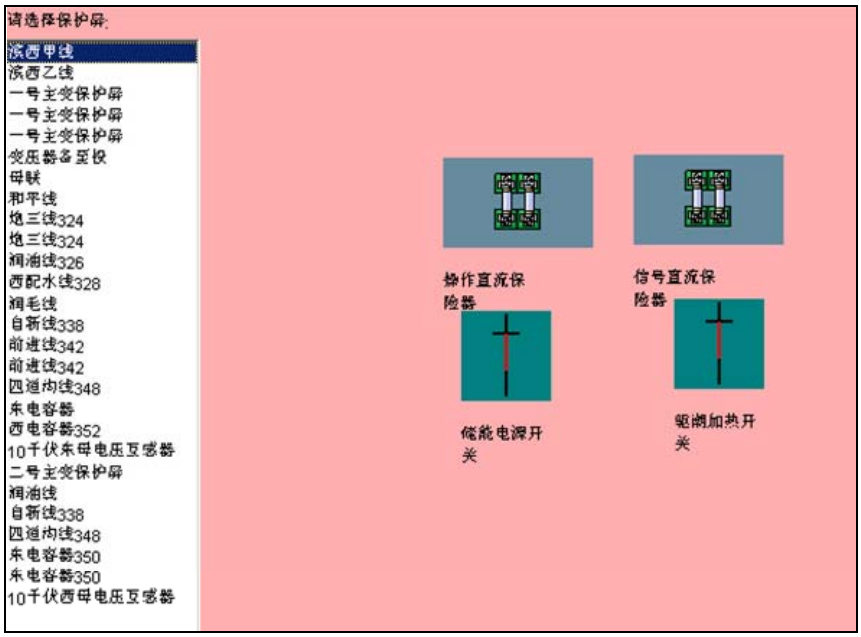

图 3-9 储能电源开关位于“合上”状态的二次系统图界面

（2）功能测试。

2.1 测试生成的操作内容是否和预期结果一致。如：在图 3-7 中选中了“储能电源开关”单击右键，然后选择“合上”菜单项后，写入操作票内容显示和文字编辑界面的内容是否为：“1.合上滨西甲线储能电源开关”。

2.2 操作结束后，元件的状态是否按照相应的操作进行转换。如：在处于“拉开”状态的“储能电源开关”上执行了“合上”操作后，“储能电源开关”是否处于“合上状态”。

为了进行上述所列出的测试内容，设计如下测试用例。

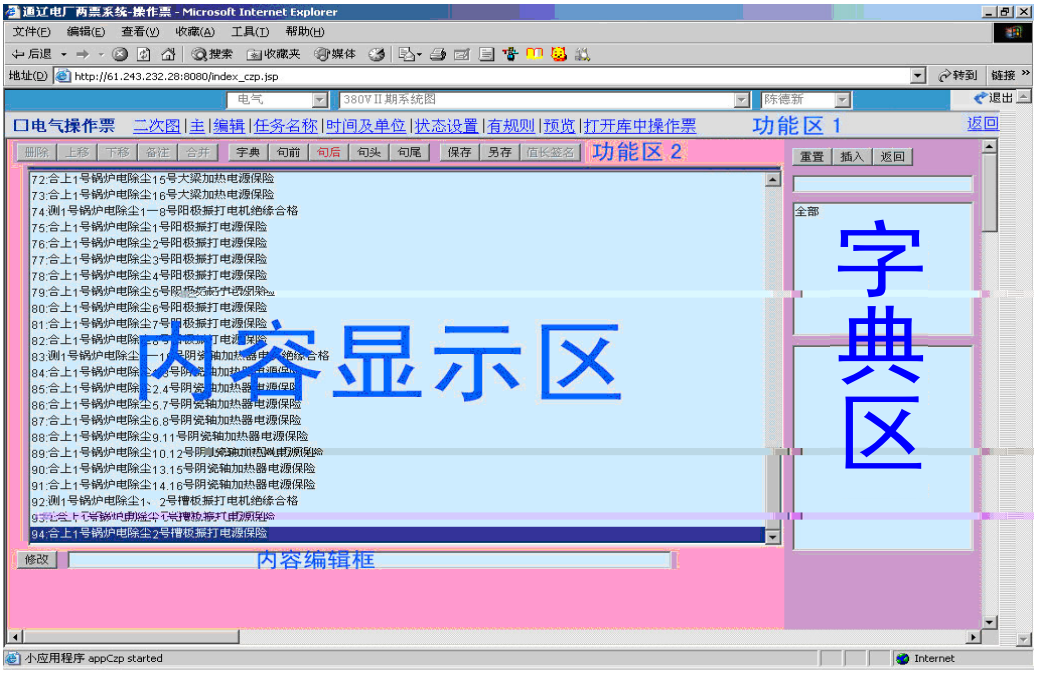

图 3-10　操作票子系统的操作票内容显示和文字编辑界面

◆　1.1 项测试用例

由于电厂的每个分厂都包括多个系统图，而每个系统图又包含多个保护屏，因此我们不可能对所有数据组合进行测试，也没有必要进行这样的测试。在进行该项测试时，采用边界值分析和等价类划分的方法首先设计一个测试用例，如表 3-4 所示。

表 3-4　界面测试用例 1

序号	输入	预期输出
Test case 1	bds_tablename=“bds10”	保护屏列表的第一项显示“滨西甲线”、第二项显示“滨西乙线”、第五项显示“一号主变保护屏”、倒数第二项显示“东电容器 350”、最后一项显示“10 千伏西母电压互感器”

之所以选取这些数据作为测试用例，基于的考虑如下：表示第一、二个保护屏和倒数第一、二个保护屏名称的数据位于数据库中的边界，从而可以判断是否从数据库中提取了所有表示保护屏名称的数据并正确显示在保护屏列表的相应位置。另外随机选取了一个位于数据库的中间部分的数据，作为补充测试数据。

◆　1.2 项测试用例

每个保护屏所包含的元件是不同的，同样使用等价类和边界值分析的方法进行二次图页面显示的测试，分别针对第一、二、五和倒数第一、二保护屏设计了测试用例，测试用例如表 3-5 所示。

表 3-5 界面测试用例 2

序号	前提	输入	预期输出
Test case 1	bds_tablename=“bds10”	选择保护屏列表的第一项，即“滨西甲线”	界面的右侧区域显示的元件有：“操作直流保险器”、“信号直流保险器”、“储能电源开关”、“驱潮加热开关”
Test case 2	bds_tablename=“bds10”	选择保护屏列表的第二项，即“滨西乙线”	“操作直流保险器”、“信号直流保险器”、“保护压板”、“重合闸”
Test case 3	bds_tablename=“bds10”	选择保护屏列表的第五项，即“一号主变保护屏”	“熔断器”、“保护压板”、“重合闸”
Test case 4	bds_tablename=“bds10”	选择保护屏列表的倒数第二项，即“东电容器 350”	“信号直流保险器”、“保护压板”、“重合闸”
Test case 5	bds_tablename=“bds10”	选择保护屏列表的倒数第一项，即“10 千伏西母电压互感器”	“信号直流保险器”、“储能电源开关”、“保护压板”

◆ 1.3 项测试用例

由于每个保护屏中都包括了多个元件，我们不可能对每个分厂的所有二次系统图中的所有保护屏中的元件上显示的弹出菜单进行测试。因此，在进行该项测试用例的设计时也采用了等价类划分的方法，即在每类元件中分别选择其中的一个来进行弹出菜单测试，测试用例如表 3-6 所示。

表 3-6 界面测试用例 3

序号	输入	预期输出
Test case 1	选中“滨西甲线”保护屏、“操作直流保险器”元件，单击右键	显示的弹出菜单项有：“合上”，“拉开”，“取消”
Test case 2	选中“滨西甲线”保护屏、“储能电源开关”元件，单击右键	显示的弹出菜单项有：“合上”，“拉开”，“取消”
Test case 3	选中“滨西乙线”保护屏、“保护压板”元件，单击右键	显示的弹出菜单项有：“投入”，“退出”，“检查在插入位置”，“检查在退出位置”
Test case 4	选中“滨西乙线”保护屏、“重合闸”元件，单击右键	显示的弹出菜单项有：“投入”，“退出”，“检查在插入位置”，“检查在退出位置”
Test case 5	选中“和平线”保护屏、“熔断器”元件，单击右键	显示的弹出菜单项有：“装上”“取下”“取消”
Test case 6	选中“和平线”保护屏、“刀闸”元件，单击右键	显示的弹出菜单项有：“合上”，“拉开”，“取消”

◆ 2.1 项和 2.2 项的测试用例

同 1.3 项测试相比，这两项测试所涉及到的数据组合还要多，因为每类元件的弹出菜单还要分别包括不同的菜单项。为了能够尽可能地对该模块进行比较完善的测试，我们针对每类元件分别进行了不同操作的测试。为了减少重复劳动，这里采用把两项测试合并起来的方法，针对各种元件所做的操作设计测试用例，但表 3-7 只给出了针对开关元件所做的操作进行测试的用例，针对其他元件进行操作的测试用例与此类似。

表 3-7　功能测试用例 1

序号	输入	预期输出
Test case 1	选中“滨西甲线”保护屏、“储能电源开关”元件，单击右键，然后选择“合上”	电源开关状态由“拉开”转为“合上”，同时在操作票内容显示与文字编辑界面显示的操作内容为：1.合上滨西甲线储能电源开关
Test case 2	选中“滨西甲线”保护屏、“储能电源开关”元件，单击右键，然后选择“拉开”	电源开关状态由“合上”转为“拉开”，同时在操作票内容显示与文字编辑界面显示的操作内容为：1.拉开滨西甲线储能电源开关
Test case 3	选中“滨西甲线”保护屏、“储能电源开关”元件，单击右键，然后选择“取消”	电源开关状态不变，同时在操作票内容显示与文字编辑界面显示的操作内容为空

接下来，对该模块进行白盒测试。因为 drawbh 类没有特别复杂的分支和判断条件，因此功能测试用例基本上覆盖了大部分代码。但是，还不能够测试程序是否实现了规格说明所没有要求的功能，不能够覆盖所有状态转换，不能够测试是否定义了多余的变量。因此，还必须对该类进行有针对性的白盒测试，如变量定义—引用测试或类的数据流测试等。读者可以参考本书中的相关章节。此外，为了使读者能够更好地理解如何在单元测试过程中运用白盒测试技术，下面以黑盒测试所没有覆盖到的 init_bh()方法为例介绍如何使用白盒测试技术对两票系统进行单元测试，对该方法进行测试的主要步骤如下：

1）选取测试用例。选定的测试用例如下：

Test case1

目的：检查生成某个分厂的二次系统图所用的保护屏初始化数据是否为预期值。

前提：确定变电所（bds_tablename）值；与数据库建立数据连接。

输入：从数据库中取得相关字段值

预期输出：mydata_bh[i][0].getdyname();为每个变电所的第 i 单元相对应的单元名。

因每个变电所有很多单元，不必对每个单元都进行测试，我们只对每个变电所的第一、第二和最后一个单元名称进行比较。如：bdstablename="bds20"，那么，mydata_bh[1][0].getdyname()的值应为“六热一号线 1601 保护屏”；

mydata_bh[2][0].getdyname()的值应为“六热二号线 1602 保护屏”；

mydata_bh[bh][0].getdyname()的值应为“一号所用变”。

Test case2

目的：检查与该变电所的每个单元相对应的保护屏名称（bhname）是否正确。

前提：确定变电所（bdstablename）值；与数据库建立数据连接。

输入：从数据库中取得相关字段值。

预期输出：这里只须对第一、第二和最后一个单元的保护名称（第一和第二个保护名称）进行比较就可以了。即判断：

mydata_bh[1][1].getbhname()的值应为“投过流 I 段压板”；

mydata_bh[1][2].getbhname()的值应为“投过流 II、III 段压板”；

mydata_bh[2][1].getbhname()的值应为“投过流 I 段压板”；

mydata_bh[2][2].getbhname()的值应为“投过流 II、III 段压板”；

mydata_bh[bh][1].getbhname()的值应为“一号所用变二次空气开关”;

mydata_bh[bh][2].getbhname()的值应为“一号所用变一次保险器”。

2）设计测试驱动程序。首先要声明的一点是，在这里我们所编写的测试驱动代码是基于 Junit 框架的。因为这样不但可以利用自动化测试工具已经为我们准备好的测试环境，而且有利于测试结果的统计和分析，以及为应用程序的回归测试打下坚实的基础。后面的相关章节将对 Junit 框架进行专门介绍，可供未接触过 Junit 的读者进行参考。测试驱动程序的代码如下：

```
import junit.framework.*;
public class DrawbhTest extends TestCase {

public Drawbh drawbh1;
//初始化

protected void setUp() {
drawbh1=new Drawbh();
drawbh1.init_bh();
}
public static Test suite() {
return new TestSuite(DrawbhTest.class);

}
public void testdyname1() {

assertEquals("这次测试第一个单元名","mydata_bh[1][0].getdyname()","六热一号线
1601 保护屏");
}
public void testdyname2() {

assertEquals("这次测试第二个单元名","mydata_bh[2][0].getdyname()","六热二号线
1602 保护屏");
}
public void testdyname3() {
assertEquals("这次测试第三个单元名","mydata_bh[bh][0].getdyname()","一号所用变");

}
public void testdyname1bhname1(){
assertEquals("这次测试第一个单元相对应的第一个保护名", "mydata_bh[1]
[1].getbhname()","投过流 I 段压板");
}
public void testdyname1bhname1(){
assertEquals("这次测试第一个单元相对应的第二个保护名", "mydata_bh[1]
[2].getbhname()","投过流 II、III 段压板");
}
```

```
public static void main (String[] args) {

//文本方式
junit.textui.TestRunner.run(suite());
//Swingui 方式
//junit.swingui.TestRunner.run(suite().getClass());
//awtui 方式
//junit.awtui.TestRunner.run(suite().getClass());
}

}
```

由于单元测试也是一个循环迭代的过程，也就是需要进行多次回归测试，以防改动的代码给其他代码造成不良影响。因此，要注意维护测试驱动程序。

为了避免不必要的重复劳动，首先统计黑盒测试没有覆盖到的代码，然后针对这些代码补充一些测试用例。通过统计我们发现了一些没有覆盖到的代码，如：以下代码中的第 9 行代码。

```
public void mousePressed(MouseEvent e)
{
if(SearcBhidx(e.getX(),e.getY()))
{
PopMenu(ThisBHidx);
popmenu.show(this,e.getX(),e.getY());
}
else
popmenu.setVisible(false);
}
```

针对该行代码我们补充了如表 3-8 所示的测试用例。

表 3-8　补充测试用例 1

序号	输入	预期输出
Test case 1	鼠标直接单击界面右侧区域空白处	页面不显示弹出菜单
Test case 2	鼠标单击某个元件，如：储能电源开关，显示弹出菜单后，再单击界面右侧区域空白处	储能电源开关上所显示的弹出菜单消失

最后，通过错误猜测技术进一步补充更多的测试用例，如：根据错误猜测我们设计了这样一个测试用例，如表 3-9 所示。

表 3-9　补充测试用例 2

序号	输入	预期输出
Test case 1	删除（备份到另外一个目录上）一部分图片文件，再运行程序	界面上应该显示未载入的图片信息

但实际运行该测试用例后，我们发现界面上并未显示任何提示，下面请读者首先看一下该类中载入图片的这一段代码：

```
String pictureName;//图片名
pictureName=mydata_bhk[mydata_bh[index+1][i].getBhkindidx()][mydata_bh[inde
```

```
x+1][i].getstation()].getBMPname();
    try{
    int num=pictureName.indexOf(".");

    if(num!=-1){
    String str3;
    str3=pictureName.substring(0,num);

    img=getImage(getCodeBase(),"image/"+str3+".gif");
    }
    w=img.getWidth(this);
    h=img.getHeight(this);
    g.drawImage(img,x1+250,y1+50,w,h,this);}
    catch(NullPointerException e){}
```

在这一段代码中，我们发现没有对图片不全的情况进行处理的代码，因此必须进行完善。

3. 测试的执行

这一步很简单，我们是使用 JCreator 开发环境来运行测试的。但初学者要注意在所使用的 Java 语言开发环境中是否已经正确地集成了测试工具。在本例中，我们在运行测试驱动程序之前就需要把 Junit 工具包，即 junit.jar 加入到 JCreator 环境中。

4. 测试结果

在第一轮测试中，我们发现数据初始化问题有误，通过排查发现是由于数组下标引用错误而引起的。改正了该缺陷后，进行了第二轮测试，得到了预期的测试结果。

5. 测试总结

（略）

在很多书籍以及相关的软件测试资料中，很多人都认为单元测试就是白盒测试，这很容易使读者混淆单元测试和白盒测试。那么，请读者思考一下：这样的说法对吗？如果你认为不对，那么单元测试和白盒测试有什么不同？

提示：①白盒测试是一种测试用例设计技术；②单元测试是软件测试过程中的一个测试级别，在测试的过程中经常用到的测试用例设计技术就是白盒测试。但读者要注意的一点是，一个单元的测试可能要用到不止一种白盒测试技术。另外，在实际测试过程中，可以把白盒测试技术和黑盒测试结合起来进行单元测试。在上面所举的例子中就是如此。

3.8 单元测试经验总结

测试人员在进行测试的过程中，应该注意积累测试工作经验，这样可以缩短单元测试的时间，提高测试效果和效率。如：

（1）在做单元测试的过程中，要灵活选用测试用例设计技术，如本章中的两票系统单元测试过程中，首先使用黑盒测试用例设计技术，然后根据相应的覆盖率统计再补充白盒测试用例。既减少了测试工作的重复，又保证了单元测试的完整性。

（2）设计驱动程序时，要保证测试逻辑的正确性。否则，即使代码正确也不能保证测试通过。测试通过后，不要随便删除测试代码，以便在后期的软件维护过程中进行回归测试。确

保变更的代码没有对软件的其他部分造成不良影响。当然，需求变更的情况除外。

（3）有时候可能会遇到这样的情况，代码是绝对正确合理的，可是却不能通过相应的测试。那么，此时应该考虑是否发生了需求变更。

（4）如果测试没有达到相应的测试覆盖要求，可以针对未覆盖的代码补充测试用例。

（5）应该尽量开发简单的测试驱动代码，增强其可读性。最重要的是，单元测试代码中不能包含分支和逻辑语句，因为这样意味着有多个测试在同时进行，将会使测试代码变得难以理解和维护。

（6）尽量开发易于执行的测试，增强我们对测试代码的信心。一般情况下，有两种类型的测试代码：一种是无需任何配置就能够运行的；一种是必须进行相关的配置才能够运行的。显然，我们所需要的是前者。

（7）避免各个测试之间存在任何的关联，以便在需要时单独运行每个测试。

本章小结

通过单元测试，我们验证开发人员所书写的编码是可以按照其所设想的方式执行的，产出了符合预期值的结果。这就实现了单元测试的目的。与后面阶段的测试相比，单元测试的创建更简单，维护更容易，并且可以更方便地进行重复。从全程的费用来考虑， 相比起那些复杂且旷日持久的集成测试，或是不稳定的软件系统测试来说，单元测试所需的费用是很低的。

在模块单元设计完毕之后的开发阶段就要开始单元测试。值得注意的是，如果在书写代码之前就进行单元测试，测试设计就会显得更加灵活，因为一旦代码完成，对软件的测试可能就受制于代码，倾向于测试该段代码完成什么功能，而不是真正的测试，我们需要做的应该是测试这段代码应该做什么。因此，应该把单元测试的设计放在详细设计阶段。

习题

1．什么是单元测试？

2．测试用例设计的步骤有哪些？

3．单元测试的策略有哪些？每种测试策略具有哪些优点和缺点？

4．单元测试与集成测试、系统测试各有哪些区别？

5．针对你所熟悉的一个简单的小程序，根据其特点考虑一下如何对其进行单元测试，并试着使用基本的黑盒或白盒测试技术设计一些测试用例和测试数据。