第 24 章

Software Engineering: A Practitioner's Approach, Eighth Edition

测试面向对象的应用

要点浏览

概念：面向对象（OO）软件的体系结构是包含协作类的一系列分层的子系统。这些系统的每个元素（子系统和类）所执行的功能都有助于满足系统需求。有必要在各种不同的层次上测试面向对象系统，尽力发现当类之间存在协作以及子系统穿越体系结构层通信时可能发生的错误。

人员：面向对象测试由软件工程师和测试专家执行。

重要性：在将程序交付给客户之前，必须运行程序，试图去除所有的错误，使得客户免受糟糕软件产品的折磨。为了发现尽可能多的错误，必须进行系统的测试，并且必须使用严格的技术来设计测试用例。

步骤：面向对象测试在策略上类似传统系统的测试，但在战术上是不同的。面向对象分析和设计模型在结构和内容上类似于最终的面向对象程序，因此“测试”开始于对这些模型的评审。一旦代码已经生成，面向对象测试就开始进行“小规模”的类测试。设计一系列测试以检查类操作及一个类与其他类协作时是否存在错误。当将类集成起来构成子系统时，应用基于线程的测试、基于使用的测试、簇测试以及基于故障的测试方法彻底检查协作类。最后，运用用例（作为分析模型的一部分开发）发现软件确认级的错误。

工作产品：设计并文档化一组测试用例来检查类、协作和行为。定义期望的结果，并记录实际结果。

质量保证措施：测试时应改变观点，努力去“破坏”软件！规范化地设计测试用例，并对测试用例进行周密的评审。

在第 23 章已经提到，简单地说，测试的目标就是在可行的时间期限内，以可行的工作量发现最大可能数量的错误。虽然这个基本目标对于面向对象软件没有改变，但面向对象程序的本质改变了测试策略和测试战术。

可以断定，由于可复用类库规模的增大，更多的复用会缓解对面向对象系统进行繁重测试的需求。然而确切地说，相反的情况的确也是存在的。Binder[Bin94b] 在讨论这种情况时说道：

> 每次复用都是一种新的使用环境，重新测试需要谨慎。为了在面向对象系统中获得高可靠性，似乎需要更多的测试，而不是更少的测试。

为了充分测试面向对象的系统，必须做三件事情：（1）对测试的定义进行扩展，使其包括应用于面向对象分析和设计模型的错误发现技术；（2）单元测试和集成测试策略必须彻底改变；（3）测试用例设计必须考虑面向对象软件的独特性质。

关键概念

类测试
簇测试
一致性
基于故障的测试
多类测试
面向对象模型
划分测试
随机测试
基于场景的测试
测试方法
测试策略
基于线程的测试
基于使用的测试

24.1　扩展测试的视野

面向对象软件的构造开始于分析和设计模型的创建[⊖]。由于面向对象软件工程模式的进化特性，这些模型开始于系统需求的不太正式的表示，并进化到更详细的类模型、类关系、系统设计和分配以及对象设计（通过消息传递来合并对象连接模型）。在每一个阶段，都要对模型进行“测试”，尽量在错误传播到下一轮迭代之前发现错误。

建议　尽管面向对象分析和设计模型的评审是测试面向对象应用不可分割的一部分，但要认识到这是不够的，还要实施可运行的测试。

可以肯定，面向对象分析和设计模型的评审非常有用，因为相同的语义结构（例如，类、属性、操作、消息）出现在分析、设计和代码层次。因此，在分析期间所发现的类属性的定义问题会防止副作用的发生。如果问题直到设计或编码阶段（或者是分析的下一轮迭代）还没有发现，副作用就会发生。

例如，在分析的第一轮迭代中，考虑定义了很多属性的一个类。有一个无关的属性被扩展到类中（由于对问题域的错误理解），然后指定了两个操作来处理此属性。对分析模型进行了评审，领域专家指出了这个问题。在这个阶段去除无关的属性，可以在分析阶段避免下面的问题和不必要的工作量。

1. 可能会生成特殊的子类，以适应不必要的属性或例外。去除无关的属性后，与创建不必要的子类相关的工作就可以避免。
2. 类定义的错误解释可能导致不正确或多余的类关系。
3. 为了适应无关的属性，系统的行为或类可能被赋予不适当的特性。

524

如果问题没有在分析期间被发现以致于进一步传播，则在设计期间会发生以下问题（早期的评审可以避免这些问题的发生）。

1. 在系统设计期间，可能会发生将类错误地分配给子系统和任务的情况。
2. 可能会扩展不必要的设计工作，比如为涉及无关属性的操作创建过程设计。
3. 消息模型可能不正确（因为会为无关的操作设计消息）。

引述　我们使用的工具对我们的思考习惯具有深远的影响，因此，也对我们的思考能力具有深远的影响。

Edsger Dijkstra

如果问题没有在设计期间检测出来，以致于传递到编码活动中，那么将大幅增加生成代码的工作量，用于实现不必要的属性、两个不必要的操作、驱动对象间通信的消息以及很多其他相关的问题。另外，类的测试会消耗更多不必要的时间。一旦最终发现了这个问题，一定要对系统执行修改，以处理由变更所引起的潜在副作用。

在开发的后期，面向对象分析（OOA）和面向对象设计（OOD）模型提供了有关系统结构和行为的实质性信息。因此，在代码生成之前，需要对这些模型进行严格的评审。

应该在模型的语法、语义和语用方面对所有的面向对象模型进行正确性、完整性和一致性测试（在这里，术语测试包括技术评审）。

24.2　测试 OOA 和 OOD 模型

不能在传统意义上对分析和设计模型进行测试，因为这些模型是不能运行的。然而，可以使用技术评审（第 20 章）检查模型的正确性和一致性。

⊖ 分析建模和设计建模技术在本书第二部分介绍。基本的面向对象概念在附录 2 中介绍。

24.2.1　OOA 和 OOD 模型的正确性

用于表示分析和设计模型的符号和语法是与为项目所选择的特定分析和设计方法连接在一起的。由于语法的正确性是基于符号表示的正确使用来判断的，因此必须对每个模型进行评审以确保维持了正确的建模习惯。

在分析和设计期间，可以根据模型是否符合真实世界的问题域来评估模型的语义正确性。如果模型准确地反映了现实世界（详细程度与模型被评审的开发阶段相适应），则在语
525 义上是正确的。实际上，为了确定模型是否反映了现实世界的需求，应该将其介绍给问题领域的专家，由专家检查类定义以及层次中遗漏和不清楚的地方。要对类关系（实例连接）进行评估，确定这些关系是否准确地反映了现实世界的对象连接[⊖]。

24.2.2　面向对象模型的一致性

面向对象模型的一致性可以通过这样的方法来判断："考虑模型中实体之间的关系。不一致的分析模型或设计模型在某一部分中的表示没有正确地反映到模型的其他部分"[McG94]。

为了评估一致性，应该检查每个类及其与其他类的连接。可以使用类－职责－协作者（Class-Responsibility-Collaborator，CRC）模型和对象－关系图来辅助此活动。如在第 10 章所学到的，CRC 模型由 CRC 索引卡片组成。每张 CRC 卡片都列出了类的名称、职责（操作）和协作者（接收其消息的其他类及完成其职责所依赖的其他类）。协作意味着面向对象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。这些信息都可以从分析模型（第 10 章）中获得。

推荐使用下面的步骤对类模型进行评估 [McG94]。

1. **检查 CRC 模型和对象－关系模型**。对这两个模型做交叉检查，确保需求模型所蕴含的所有协作都已正确地反映在这两个模型中。
2. **检查每一张 CRC 索引卡片的描述以确定委托职责是协作者定义的一部分**。例如，考虑为销售积分结账系统定义的类（称为 CreditSale），这个类的 CRC 索引卡片如图 24-1 所示。

类的名称：credit sale	
类的类型：transaction event	
类的特性：nontangible, atornio, esquential, pemanert, guarded	
职责：	协作者
读信用卡	信用卡
取得授权	信用权利
显示购物金额	产品票
	销售总账
	审计文件
生成账单	账单

图 24-1　用于评审的 CRC 索引卡片实例

⊖ 对于面向对象系统，在对照现实世界的使用场景追踪分析和设计模型方面，用例是非常有价值的。

对于这组类和协作，例如，将职责（例如读信用卡）委托给已命名的协作者（CreditCard），看看此协作者是否完成了这项职责。也就是说，类 CreditCard 是否具有读卡操作？在此实例中，回答是肯定的。遍历对象－关系模型，确保所有此类连接都是有效的。

3. **反转连接，确保每个提供服务的协作者都从合理的地方收到请求。**例如，如果 CreditCard 类收到了来自 CreditSale 类的请求 purchase amount，那么就有问题了。CreditCard 不知道购物金额是多少。 526
4. **使用步骤 3 中反转后的连接，确定是否真正需要其他类，或者职责在类之间的组织是否合适。**
5. **确定是否可以将广泛请求的多个职责组合为一个职责。**例如，读信用卡和取得授权在每一种情形下都会发生，可以将这两个职责组合为验证信用请求（validate credit request）职责，此职责包括取得信用卡号和取得授权。

可以将步骤 1 ～步骤 5 反复应用到每个类及需求模型的每一次评估中。

一旦创建了设计模型（第 12 ～ 18 章），就可以进行系统设计和对象设计的评审了。系统设计描述总体的产品体系结构、组成产品的子系统、将子系统分配给处理器的方式、将类分配给子系统的方式以及用户界面的设计。对象模型描述每个类的细节以及实现类之间的协作所必需的消息传送活动。

系统设计评审是这样进行的：检查面向对象分析期间所开发的对象－行为模型，并将所需要的系统行为映射到为完成此行为而设计的子系统上。在系统行为的范畴内也要对并发和任务分配进行评审。对系统的行为状态进行评估以确定并发行为。使用用例进行用户界面设计。

对照对象－关系网检查对象模型，确保所有的设计对象都包括必要的属性和操作，以实现为每个 CRC 索引卡片所定义的协作。另外，要对操作细节的详细规格说明（即实现操作的算法）进行评审。 527

24.3　面向对象测试策略

如在第 22 章讲到的，经典的软件测试策略从“小范围”开始，并逐步过渡到“软件整体”。用软件测试的行话来说（第 23 章），就是先从单元测试开始，然后过渡到集成测试，并以确认测试和系统测试结束。在传统的应用中，单元测试关注最小的可编译程序单元——子程序（例如，构件、模块、子程序、程序）。一旦完成了一个单元的单独测试，就将其集成到程序结构中，并进行一系列的回归测试，以发现模块的接口错误及由于加入新模块所引发的副作用。最后，将系统作为一个整体进行测试，确保发现需求方面的错误。

24.3.1　面向对象环境中的单元测试

考虑面向对象软件时，单元的概念发生了变化。封装是类和对象定义的驱动力，也就是说，每个类和类的每个实例（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）。最小的可测试单元是封装了的类，而不是单独的模块。由于一个类可以包括很多不同的操作，并且一个特定的操作又可以是很多不同类的一部分，因此，单元测试的含义发生了巨大的变化。

关键点　在面向对象软件中，最小的可测试“单元”是类，类测试是由封装在类中的操作和类的状态行为驱动的。

我们已经不可能再独立地测试单一的操作了（独立地测试单一的操作是单元测试的传统观点），而是要作为类的一部分进行操作。例如，考虑在一个类层次中，为超类定义了操作 *X*()，并且很多子类继承了此操作。每个子类都使用操作 *X*()，但是此操作是在为每个子类所定义的私有属性和操作的环境中应用的。由于使用操作 *X*() 的环境具有微妙的差异，因此，有必要在每个子类的环境中测试操作 *X*()。这就意味着在真空中测试操作 *X*()（传统的单元测试方法）在面向对象的环境中是无效的。

面向对象软件的*类测试*等同于传统软件的单元测试[⊖]。面向对象软件的类测试与传统软
528 件的单元测试是不同的，传统软件的单元测试倾向于关注模块的算法细节和流经模块接口的数据，而面向对象软件的类测试由封装在类中的操作和类的状态行为驱动。

24.3.2 面向对象环境中的集成测试

由于面向对象软件不具有层次控制结构，因此传统的自顶向下和自底向上的集成策略是没有意义的。另外，由于“组成类的构件之间的直接和非直接的交互”[Ber93]，因此每次将一个操作集成到类中通常是不可能的。

关键点 面向对象软件的集成测试是对响应一个给定事件所需要的一组类进行测试。

面向对象系统的集成测试有两种不同的策略 [Bin94a]。第一种集成策略是*基于线程的测试*，将响应系统的一个输入或一个事件所需要的一组类集成到一起。每个线程单独集成和测试，并应用回归测试确保不产生副作用。第二种集成策略是*基于使用的测试*，通过测试那些很少使用服务器类的类（称为*独立类*）开始系统的构建。测试完独立类之后，测试使用独立类的下一层类（称为*依赖类*）。按照这样的顺序逐层测试依赖类，直到整个系统构建完成。与传统集成不同，在可能的情况下，这种策略避免了作为替换操作的驱动模块和桩模块的使用（第 23 章）。

簇测试 [McG94] 是面向对象软件集成测试中的一个步骤。通过设计试图发现协作错误的测试用例，对一簇协作类（通过检查 CRC 和对象 – 关系模型来确定）进行测试。

24.3.3 面向对象环境中的确认测试

在确认级或系统级，类连接的细节消失了。如传统的确认方法一样，面向对象软件的确认关注用户可见的动作和用户可以辨别的来自系统的输出。为了辅助确认测试的导出，测试人员应该拟定出用例（第 9 章和第 10 章），用例是需求模型的一部分，提供了最有可能发现用户交互需求方面错误的场景。

传统的黑盒测试方法（第 23 章）可用于驱动确认测试。另外，测试人员可以选择从对象 – 行为模型导出测试用例，也可以从创建的事件流图（OOA 的一部分）导出测试用例。

引述 我将测试人员看成是项目的护卫。开发人员专注于创造成功，而测试人员则守护在他们左右，使其免遭失败。

James Bach

24.4 面向对象测试方法

面向对象体系结构导致封装了协作类的一系列分层子系统的产生。每
529 个系统成分（子系统和类）完成的功能都有助于满足系统需求。有必要在不同的层次上测试面向对象系统，以发现错误。在类相互协作以及子系统

⊖ 面向对象类的测试用例设计方法在 24.4 ～ 24.6 节讨论。

穿越体系结构层通信时可能出现这些错误。

面向对象软件的测试用例设计方法还在不断改进，然而，对于面向对象测试用例的设计，Berard 已经提出了总体方法 [Ber93]：

1. 每个测试用例都应该被唯一地标识，并明确地与被测试的类相关联。
2. 应该叙述测试的目的。
3. 应该为每一个测试开发测试步骤，并包括以下内容：将要测试的类的指定状态列表；作为测试结果要进行检查的消息和操作列表；对类进行测试时可能发生的异常列表；外部条件列表（即软件外部环境的变更，为了正确地进行测试，这种环境必须存在）；有助于理解或实现测试的补充信息。

面向对象测试与传统的测试用例设计是不同的，传统的测试用例是通过软件的输入 – 处理 – 输出视图或单个模块的算法细节来设计的，而面向对象测试侧重于设计适当的操作序列以检查类的状态。

24.4.1 面向对象概念的测试用例设计含义

经过分析模型和设计模型的演变，类成为了测试用例设计的目标。由于操作和属性是封装的，因此从类的外面测试操作通常是徒劳的。尽管封装是面向对象的重要设计概念，但它可能成为测试的一个小障碍。如 Binder [Bin94a] 所述："测试需要报告对象的具体状态和抽象状态。"然而，封装使获取这些信息有些困难，除非提供内置操作来报告类的属性值，否则，可能很难获得一个对象的状态快照。

网络资源 一些极好的有关面向对象测试的论文集和资源可在 https://www.thecsiac.com 找到。

继承也为测试用例设计提出了额外的挑战。我们已经注意到，即使已取得复用，每个新的使用环境也需要重新测试。另外，由于增加了所需测试环境的数量，因此多重继承[⊖]使测试进一步复杂化 [Bin94a]。若将从超类派生的子类实例用于相同的问题域，则测试子类时，使用超类中生成的测试用例集是可能的。然而，若子类用在一个完全不同的环境中，则超类的测试用例将具有很小的可应用性，因而必须设计新的测试用例集。

530

24.4.2 传统测试用例设计方法的可应用性

第 23 章描述的白盒测试方法可以应用于类中定义的操作。基本路径、循环测试或数据流技术有助于确保一个操作中的每条语句都测试到。然而，许多类操作的简洁结构使某些人认为：用于白盒测试的工作投入最好直接用于类层次的测试。

与利用传统的软件工程方法所开发的系统一样，黑盒测试方法也适用于面向对象系统。如我们在第 23 章提到的，用例可为黑盒测试和基于状态的测试设计提供有用的输入。

关键点 基于故障的测试策略是假设一组似乎可能出现的故障，然后导出测试去证明每个假设。

24.4.3 基于故障的测试[⊖]

在面向对象系统中，基于故障的测试目标是设计测试以使其最有可能发现似乎可能出现的故障（以下称为似然故障）。由于产品或系统必须符合

⊖ 应非常小心使用的一个面向对象概念。

⊖ 24.4.3 节和 24.4.4 节是从 Brain Marick 发布在因特网新闻组 comp.testing 上的文章中摘录的，已得到作者的许可。有关该主题的详细信息见 [Mar94]。应该注意到，24.4.3 节和 24.4.4 节讨论的技术也适用于传统软件。

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的。测试人员查找似然故障（即系统的实现中有可能产生错误的方面）。为了确定这些故障是否存在，需要设计测试用例以检查设计或代码。

当然，这些技术的有效性依赖于测试人员如何理解似然故障。若在面向对象系统中真正的故障被理解为“没有道理”的，则这种方法实际上并不比任何随机测试技术好。然而，若分析模型和设计模型可以洞察有可能出错的事物，则基于故障的测试可以花费相当少的工作量而发现大量的错误。

集成测试寻找的是操作调用或信息连接中的似然错误。在这种环境下，可以发现三种错误：非预期的结果，使用了错误的操作/消息，以及不正确的调用。为确定函数（操作）调用时的似然故障，必须检查操作的行为。

提问 在操作调用和消息连接中会遇到哪些类型的故障？

集成测试适用于属性，同样也适用于操作。对象的“行为”通过赋予属性值来定义。测试应该检查属性以确定不同类型的对象行为是否存在合适的值。

集成测试试图发现用户对象而不是服务对象中的错误，注意到这一点很重要。用传统的术语来说，集成测试的重点是确定调用代码而不是被调用代码中是否存在错误。以操作调用为线索，这是找出调用代码的测试需求的一种方式。

531

24.4.4　基于场景的测试设计

基于故障的测试忽略了两种主要类型的错误：（1）不正确的规格说明；（2）子系统间的交互。当出现了与不正确的规格说明相关的错误时，产品并不做客户希望的事情，而是有可能做错误的事情或漏掉重要的功能。但是，在这两种情况下，质量（对需求的符合性）均会受到损害。当一个子系统的行为创建的环境（例如事件、数据流）使另一个子系统失效时，则出现了与子系统交互相关的错误。

基于场景的测试关心用户做什么，而不是产品做什么。这意味着捕获用户必须完成的任务（通过用例），然后在测试时使用它们及其变体。

关键点 基于场景的测试将发现任何角色与软件进行交互时出现的错误。

场景可以发现交互错误。为了达到这个目标，测试用例必须比基于故障的测试更复杂且更切合实际。基于场景的测试倾向于用单一测试检查多个子系统（用户并不限制自己一次只用一个子系统）。

24.5　类级可应用的测试方法

“小范围”测试侧重于单个类及该类封装的方法。面向对象测试期间，随机测试和分割是用于检查类的测试方法。

24.5.1　面向对象类的随机测试

为简要说明这些方法，考虑一个银行应用，其中 Account 类有下列操作：open()、setup()、deposit()、withdraw()、balance()、summarize()、creditLimit() 及 close()[Kir94]。其中，每个操作均可应用于 Account 类，但问题的本质隐含了一些限制（例如，账号必须在其他操作可应用之前打开，在所有操作完成之后关闭）。即使有了这些限制，仍存在很多种操作排列。一个 Account 对象的最小行为的生命历史包含以下操作：

建议 随机测试可能的排列数可以变得相当大。与正交数组测试相类似的策略可以用于提高测试的有效性。

open•setup•deposit•withdraw•close

这表示 Account 的最小测试序列。然而，可以在这个序列中发生大量其他行为：

open•setup•deposit•[deposit|withdraw|balance|summarize|creditLimit]^n•withdraw•close

532

可以随机产生一些不同的操作序列，例如：

测试用例 r_1: open•setup•deposit•deposit•balance•summarize•withdraw•close

测试用例 r_2: open•setup•deposit•withdraw•deposit•balance•creditLimit•withdraw•close

执行这些序列和其他随机顺序测试，以检查不同类实例的生命历史。

SafeHome 类测试

[场景] Shakira 的工作间。

[人物] Jamie 与 Shakira，SafeHome 软件工程团队成员，负责安全功能的测试用例设计。

[对话]

Shakira：我已经为 Detector 类（图 14-4）开发了一些测试，你知道，这个类允许访问安全功能的所有 Sensor 对象。你熟悉它吗？

Jamie（笑）：当然，它允许你加入“小狗焦虑症”传感器。

Shakira：而且是唯一的一个。不管怎么样，它包含 4 个操作的接口：read()、enable()、disable() 和 test()。在传感器可读之前，它必须被激活。一旦激活，就可以进行读和测试，且随时可以中止它，除非正在处理警报条件。因此，我定义了检查其行为生命历史的简单测试序列（向 Jamie 展示下述序列）。

#1: enable•test•read•disable

Jamie：不错。但你还得做更多的测试！

Shakira：我知道。这里有我提出的其他测试序列。

（向 Shakira 展示下述序列）

#2: enable•test*[read]^n•test•disable

#3: [read]^n

#4: enable*disable•[test | read]

Jamie：看我能否理解这些测试序列的意图。#1 通过一个正常的生命历史，属于常规使用。#2 重复 read() 操作 n 次，那是一个可能出现的场景。#3 在传感器激活之前尽力读取它……那应该产生某种错误信息，对吗？#4 激活和中止传感器，然后尽力读取它，这与 #2 不是一样的吗？

Shakira：实际上不一样。在 #4 中，传感器已经被激活，#4 测试的实际上是 disable() 操作是否像其预期的一样有效工作。disable() 之后，read() 或 test() 应该产生错误信息。若没有，则说明 disable() 操作中有错误。

Jamie：太棒了，记住这 4 个测试必须应用于每一类传感器，因为所有这些操作根据其传感器类型可能略有不同。

Shakira：不用担心，那是计划之中的事。

24.5.2　类级的划分测试

与传统软件的等价划分（第 23 章）基本相似，划分测试（partition testing）可减小测试特定类所需的测试用例数量。对输入和输出进行分类，设计测试用例以检查每个分类。但划分类别是如何得到的呢？

提问　在类级上什么测试选项可供使用？

533

基于状态的划分就是根据它们改变类状态的能力对类操作进行分类。例如，考虑 Account 类，状态操作包括 deposit() 和 withdraw()，而非状态操作包括

balance()、summarize() 和 creditLimit()。将改变状态的操作和不改变状态的操作分开，分别进行测试，因此：

测试用例 p_1：open•setup•deposit•deposit•withdraw•withdraw•close

测试用例 p_2：open•setup•deposit•summarize•creditLimit•withdraw•close

测试用例 p_1 检查改变状态的操作，而测试用例 p_2 检查不改变状态的操作（除了那些最小测试序列中的操作）。

也可以应用其他类型的划分测试。基于属性的划分就是根据它们所使用的属性对类操作进行分类。基于类别的划分就是根据每个操作所完成的一般功能对类操作进行分类。

引述 对于面向对象的开发，定义单元测试和集成测试范围的边界是不同的。在这个过程中，可以在很多点上设计和检查测试。因此，“设计一点儿，编码一点儿”变成了“设计一点儿，编码一点儿，测试一点儿”。

Robert Binder

24.6　类间测试用例设计

当开始集成面向对象系统时，测试用例的设计变得更为复杂。在这个阶段必须开始类间协作的测试。为说明“类间测试用例生成”[Kir94]，我们扩展 24.5 节中讨论的银行例子，让它包括图 24-2 中的类与协作。图中箭头的方向指明消息传递的方向，标注则指明作为消息隐含的协作的结果而调用的操作。

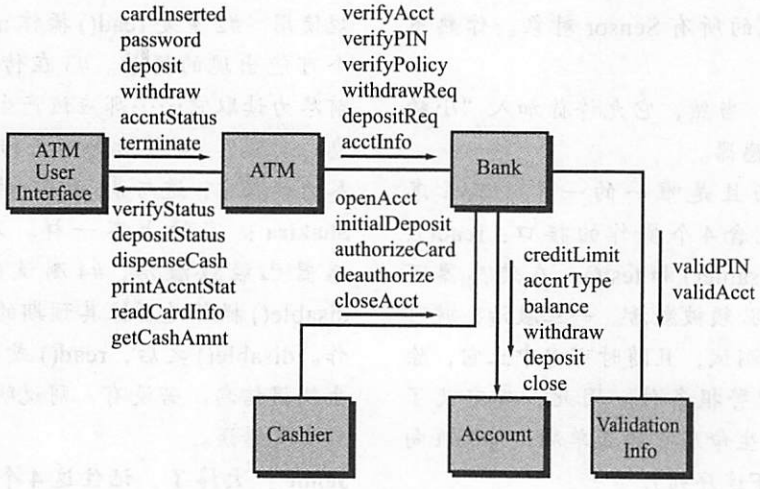

图 24-2　银行应用的类协作图 [Kir94]

与单个类的测试相类似，类协作测试可以通过运用随机和划分方法、基于场景测试及行为测试来完成。

24.6.1　多类测试

Kirani 和 Tsai[Kir94] 提出了利用下列步骤生成多类随机测试用例的方法：

1. 对每个客户类，使用类操作列表来生成一系列随机测试序列。这些操作将向其他服务类发送消息。

534

2. 对生成的每个消息，确定协作类和服务对象中的相应操作。
3. 对服务对象中的每个操作（已被来自客户对象的消息调用），确定它传送的消息。
4. 对每个消息，确定下一层被调用的操作，并将其引入到测试序列中。

为便于说明 [Kir94]，考虑 Bank 类相对于 ATM 类的操作序列（图 24-2）：

```
verifyAcct•verifyPIN•[[verifyPolicy•withdrawReq]|depositReq|acctInfoREQ]ⁿ
```

Bank 类的一个随机测试用例可以是：

测试用例 r_3：`verifyAcct•verifyPIN•depositReq`

为考虑涉及该测试的协作者，考虑与测试用例 r_3 中提到的操作相关的消息。为了执行 verifyAcct() 与 verifyPIN()，Bank 类必须与 ValidationInfo 类协作。为了执行 depositReq()，Bank 类必须与 Account 类协作。因此，检查这些协作的新测试用例为：

测试用例 r_4：

```
Test case r4 = verifyAcct [Bank:validAcctValidationInfo]•verifyPIN
[Bank: validPinValidationInfo]•depositReq [Bank: depositaccount]
```

多个类的划分测试方法与单个类的划分测试方法类似，单个类的划分测试方法如在 24.5.2 节讨论的那样。然而，可以对测试序列进行扩展，以包括那些通过发送给协作类的消息而激活的操作。另一种划分测试方法基于特殊类的接口。参看图 24-2，Bank 类从 ATM 类和 Cashier 类接收消息，因此，可以通过将 Bank 类中的操作划分为服务于 ATM 类的操作和 535
服务于 Cashier 类的操作对其进行测试。基于状态的划分（24.5.2 节）可用于进一步细化上述划分。

24.6.2　从行为模型导出的测试

在第 11 章中，我们已讨论过用状态图表示类的动态行为模型。类的状态图可用于辅助生成检查类（以及与该类的协作类）的动态行为的测试序列。图 24-3[Kir94] 给出了前面讨论的 Account 类的状态图。根据该图，初始变换经过了 Empty acct 状态和 Setup acct 状态，该类实例的绝大多数行为发生在 Working acct 状态。最终的 Withdrawal 和结束账户操作使得 Account 类分别向 Nonworking acct 状态和 Dead acct 状态发生转换。

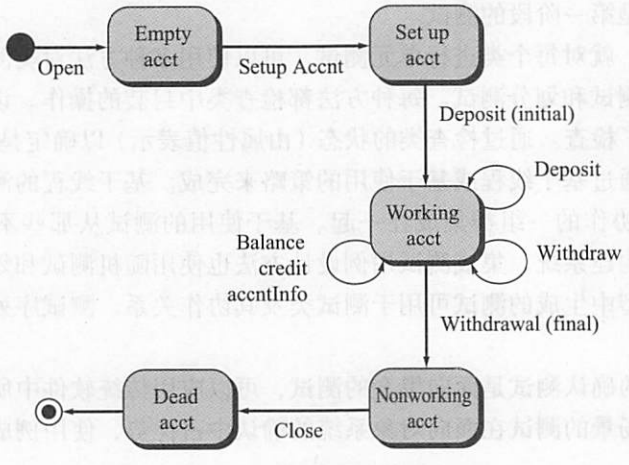

图 24-3　Account 类的状态转换图 [Kir94]

将要设计的测试应该覆盖所有的状态，也就是说，操作序列应该使 Account 类能够向所有可允许的状态转换：

测试用例 s_1：`open•setupAccnt•deposit (initial)•withdraw (final)•close`

应该注意到，这个序列与 24.5.2 节所讨论的最小测试序列相同。下面将其他测试序列加入最小测试序列中：

测试用例 s_2: open•setupAccnt•deposit(initial)•deposit•balance•credit•withdraw(final)•close

536 测试用例 s_3: open•setupAccnt•deposit(initial)•deposit•withdraw•accntInfo•withdraw (final)•close

可以设计更多的测试用例以保证该类的所有行为已被充分检查。在该类的行为与一个或多个类产生协作的情况下，可以用多个状态图来追踪系统的行为流。

可以通过“广度优先”[McG94] 的方式来遍历状态模型。在这里，广度优先意味着一个测试用例检查单个转换，之后在测试新的转换时，仅使用前面已经测试过的转换。

考虑银行系统中的一个 CreditCard 对象。CreditCard 对象的初始状态为 undefined（即未提供信用卡号）。在销售过程中一旦读取信用卡，对象就进入了 defined 状态，即属性 card number、expiration date 以及银行专用的标识符被定义。当信用卡被发送以请求授权时，它处于 submitted 状态，当接收到授权时，它处于 approved 状态。可以通过设计使转换发生的测试用例来测试 CreditCard 对象从一个状态到另一个状态的转换。对这种测试类型的广度优先方法在检查 undefined 和 defined 之前不会检查 submitted 状态。若这样做了，它就使用了尚未经过测试的转换，从而违反了广度优先准则。

24.7　小结

面向对象测试的总体目标是以最少的工作量发现最多的错误，这与传统软件的测试目标是一样的。但是，面向对象测试的策略和战术与传统软件有很大差异。测试的视角扩展到了需求模型和设计模型，另外，测试的重点从过程构件（模块）移向了类。

由于面向对象需求模型、设计模型和最终的源代码在语义上是有联系的，在建模活动期间，测试（以技术评审的形式）就开始了。因此，可以将 CRC、对象 – 关系模型、对象 – 行为模型的评审看成是第一阶段的测试。

一旦有了代码，就对每个类进行单元测试。可以使用多种方法对类测试进行设计：基于故障的测试、随机测试和划分测试。每种方法都检查类中封装的操作。设计测试序列以保证对相关的操作进行了检查。通过检查类的状态（由属性值表示）以确定是否存在错误。

集成测试可以通过基于线程或基于使用的策略来完成。基于线程的测试将为了响应一个输入或事件而相互协作的一组类集成在一起。基于使用的测试从那些不使用服务类的类开始，以分层的方式构建系统。集成测试用例设计方法也使用随机测试和划分测试。另外，基
537 于场景和从行为模型中生成的测试可用于测试类及其协作关系。测试序列追踪类间协作的操作流。

面向对象系统的确认测试是面向黑盒的测试，可以应用传统软件中所讨论的黑盒方法来完成。然而，基于场景的测试在面向对象系统的确认中占优势，使用例成为确认测试的主要驱动者。

习题与思考题

24.1　用自己的话描述为什么在面向对象系统中类是最小的合理测试单元。

24.2　若现有类已进行了彻底的测试，为什么我们还是必须对从现有类实例化的子类进行重新测试？

我们可以使用为现有类设计的测试用例吗?

24.3 为什么“测试”应该从面向对象分析和设计开始?

24.4 为 SafeHome 导出一组 CRC 索引卡片，按照 24.2.2 节讲述的步骤确定是否存在不一致性。

24.5 基于线程和基于使用的集成测试策略有什么不同? 簇测试如何适应?

24.6 将随机测试和划分方法运用到设计 SafeHome 系统时定义的三个类。产生用于展示操作调用序列的测试用例。

24.7 运用多类测试及从 SafeHome 设计的行为模型中生成的测试。

24.8 运用随机测试、划分方法、多类测试及 24.5 节和 24.6 节所描述的银行应用的行为模型导出的测试，再另外生成 4 个测试。

扩展阅读与信息资源

第 22 章和第 23 章的推荐读物与阅读信息部分所列出的很多测试方面的书都在一定程度上讨论了面向对象系统的测试。Bashir 和 Goel（《Testing Object-Oriented Software》, Springer, 2012）、Schach（《Object-Oriented and Classical Software Engineering》, McGraw-Hill, 8th ed., 2010）以及 Bruege 和 Dutoit（《Object-Oriented Software Engineering Using UML, Patterns, and Java》, Prentice Hall, 3rd ed., 2009）在更广的软件工程实践环境中考虑面向对象测试。Jorgensen（《Software Testing: A Craftsman's Approach》, Auerbach, 3rd ed., 2008）讨论了形式化技术和面向对象技术。Yurga（《Testing and Testability of Object-Oriented Software Systems via Metrics: A Metrics-Based Approach to the Testing Process and Testability of Object-Oriented Software Systems》, LAP Lambert, 2011）、Sykes 和 McGregor（《Practical Guide to Testing Object-Oriented Software》, Addison-Wesley, 2001）、Binder（《Testing Object-Oriented Systems》, Addison-Wesley, 1999）以及 Kung 和他的同事（《Testing Object-Oriented Software》, Wiley-IEEE Computer Society Press, 1998）都非常详细地描述了面向对象测试。Freeman 和 Pryce（《Growing Object-Oriented Software, Guided by Tests》, Addison-Wesley, 2009）讨论了面向对象软件的测试驱动设计。Denney（《Use Case Levels of Test: A Four-Step Strategy for Building Time and Innovation in Software Test Design》, CreateSpace Independent Publishing, 2012）讨论了可以应用于测试面向对象系统的技术。

从网上可以获得大量有关面向对象测试方法的信息。最新的有关测试技术的参考文献可在 SEPA 网站 www.mhhe.com/pressman 找到。

538
~
539