第 11 章　面向对象分析及设计

人们很容易变得只专注于C++的语法，而忽略如何及为何使用这些技术来编写程序。

本章介绍以下内容：

- 如何使用面向对象分析来理解要解决的问题。
- 如何使用面向对象设计来创建健壮、可扩展和可靠的解决方案。
- 如何使用统一建模语言（Unified Modeling Language，UML）来编写分析和设计文档。

11.1　建立模型

要控制复杂程度，必须创建域模型。模型旨在创建有意义的现实世界抽象。这种抽象应比现实世界简单但又精确地反映现实世界，以便可以使用这个模型来预测现实世界中事物的行为。

地球仪是一个经典模型。该模型不是地球本身；人们绝不会将地球仪与地球混为一谈，但地球仪很好地反映了地球，我们可通过研究地球仪来了解地球。

当然有相当程度的简化。笔者女儿的地球仪不会下雨、发洪水、地震等，但当 Sams 的高级管理人员询问为什么我的手稿延期时，而我需要当面做出解释（你看我正在努力干着，但在一个隐喻中纠缠不清而花了几个小时来理清）时，可以用她的地球仪预测从我家飞到印第安纳波利斯要花多长时间。

如果模型比它模拟的事物还复杂，将没有多大用处。喜剧演员 Steren Wright 讽刺说：我有一张地图，在这个地图上一英寸就等于一英寸，我住在E5。

面向对象软件设计旨在要设计良好的模型。它由两个重要部分组成：建模语言和过程。

11.2　软件设计：建模语言

建模语言是面向对象分析和设计中最不重要的方面；遗憾的是，它常常最受重视。建模语言只不过是一种有关如何在媒介（如纸张或计算机系统）上，使用某种形式（如图形、文本或符号）表示模型的约定。例如，可以容易地决定将类画成三角形，将继承关系画成虚线。这样，可以按图 11.1 那样绘制大竺葵模型。

从该图可知，天竺葵是一种特殊的花。如果你和我达成一致，按这样的方式绘制继承（泛化-具体化）图，将能够很好地彼此沟通。随着时间的推移，可能要建立很多复杂关系的模型，因此将开发一套复杂的图形绘制约定和规则。

当然，需要向每个同事解释这些约定，而每位新员工和合作方也需要学习这些约定。我们可能需要与其他公司打交道，而这些公司有自己的约定，在这种情况下，必须花些时间协商出共同的约定，以弥合不可避免的误解。

如果业界的每个人都使用同一种建模语言，将非常方便。从这种意义上说，如果世界上每个人使用同一种口头语言，将非常方便。

软件分析和设计的世界语是 UML：统一建模语言。UML 规范的任务是回答诸如“应该如何表示继承关

系”等问题。使用 UML 时，图 11.1 所示的关系应绘制成图 11.2 所示的那样。

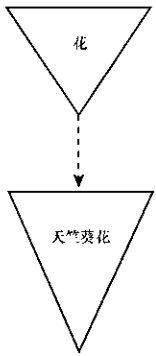

图 11.1　普通-具体关系

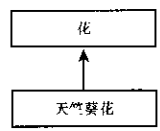

图 11.2　表示继承关系的 UML 图

在 UML 中，类用矩形表示，继承用带箭头线段表示。有趣的是，箭头从具体类指向普通类。箭头方向可能并不直观，但这无关紧要；只要大家达成一致，在了解表示方法后，使能进行沟通。

UML 的细节相当简单。这些图通常并不难使用和理解，我们将在它们出现时进行介绍。虽然全面介绍 UML 需要一整本书的篇幅，但在 90%的情况下，你只使用一个很小的 UML 表示法子集，而这个子集学习起来很容易。

11.3　软件设计：过程

面向对象分析和设计的过程比建模语言要复杂和重要得多，然而具有讽刺意味的是，读者却很少听说它。

这是因为关于建模语言的争论已经平息，整个行业决定将 UML 作为主要标准；而关于过程的争论还在激烈地进行。

方法是一种建模语言和一个过程。方法常常被误解为方法学，方法学是研究方法的。

方法学家是开发或研究方法的人。通常，方法学家开发并发表自己的方法。3 位著名的方法学家及其方法是：Grady Booch 及其开发的 Booch 方法、Ivar Jacobson 及其开发的面向对象软件工程和 James Rumbaugh 及其开发的对象建模技术（OMT）。这 3 位一起创建了 Rational Unified Process（以前叫 Objectory），它既是一种方法，也是 Rational Software 公司的一种商业产品。这 3 位都曾受雇于 IBM 的 Rational Software 分部，在那里他们被亲切地称为“岁寒三友（Three Amigos）”。

本章大概地遵循他们的方法，而不盲目地信奉学术理论——制造出产品比遵守方法更重要。其他方法也有一些可取之处，后面制定可行的框架时，将零星地介绍其中一些有用的内容。并非每位实践者都认可这种方法，但议读者广泛地阅读有关软件工程实践的文献，自己判断哪些应是最佳实践。如果你受雇于某个公司，而该公司将某种方法作为官方最佳实践，你需要遵守该方法，并达到公司要求的一致程度。

软件设计过程可能是迭代性的。在这种情况下，在开发软件时，需要反复地完成整个过程以努力加深对需求的认识。设计指导着实现，但在实现中发现的细节将反馈到设计中。采用这种方法是，不要试图以有序、线性的方式一次性开发出项目，而应迭代性地完成项目的各个部分，不断地改进设计和优化实现。

11.3.1　迭代式开发和瀑布式开发

迭代式开发不同于瀑布式开发。在瀑布式开发中，一个阶段的输出将成为下一阶段的输入。就像难以沿瀑布逆流而上一样，采用这种开发方法时，不存在回到前一个阶段的问题（如图 11.3 所示）。

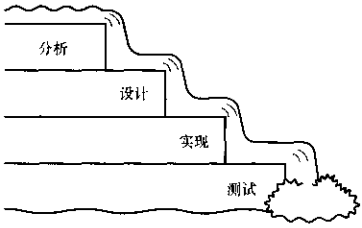

图 11.3 瀑布式开发方法

在瀑布式开发过程中，对需求进行细化，并由客户签字同意（是的，这是我想要的）；然后将确定的需求提供给设计人员。设计人员制定设计方案，并将其提供给程序员，由他们来实现设计方案。程序员再将代码提交给质量保证（QA）人员，后者对代码进行测试后，将其交付给客户。从理论上说，这很不错，但实践起来可能是场灾难。

11.3.2 迭代式开发过程

在迭代式开发中，从概念（对要创建的系统的想法）开始。随着对细节进行分析，想法将成熟和演变。

对需求有大致了解后，开始设计；对设计阶段出现的问题有深入认识后，可能回过头来修改需求。在设计阶段，也可能开始建立原型并实现产品。开发阶段出现的问题将反馈到设计中，甚至影响你对需求的认识。最重要的是，你只设计和实现整个产品的一部分，并不断地在设计和实现阶段之间反复。

虽然这种过程的步骤被迭代性反复，但几乎不能用循环方式来描述它们。下面按顺序描述它们。

迭代式开发过程的步骤如下：

第 1 步：概念化

概念化是“愿景”，它是描述伟大思想的一句话。

第 2 步：分析

分析是认识需求的过程。

第 3 步：设计

设计是创建类模型的过程，将根据设计来编写代码。

第 4 步：实现

实现就是编写代码（例如，使用 C++）。

第 5 步：测试

测试确保代码正确。

第 6 步：交付（rollout）

交付是将程序交给客户。

注意：这些步骤不同于 Rational Unified Process 中的如下阶段：

- 开始（inception）；
- 精制（elaboration）；
- 构建（construction）；
- 迁移（Transition）。

也不同于 Rational Unified Process 中的如下工作流程：

- 业务建模；
- 需求；
- 分析和设计；
- 实现；

- 测试;
- 部署;
- 配置和变更管理;
- 项目管理;
- 环境。

不要误解，实际上，在一个产品的开发过程中，你将经过这些步骤多次。如果循环执行每个步骤，迭代式开发过程将难以表述和理解。

这个过程看起来很容易，本章余下的内容将介绍细节。

争论

关于迭代式设计过程中每个阶段将发生什么存在无休止的争论，甚至对如何给这些阶段命名也如此。

窍诀在于，这些无关紧要。

几乎所有面向对象过程的基本步骤都相同：确定要创建什么样的程序、设计解决方案并实现该设计方案。

虽然有关对象技术的新闻组和邮件列表多如牛毛、但面向对象分析和设计的精髓很简单。本章将提供完成这种过程的实用方法，读者可将其作为建立应用程序体系结构的基石。

所有这些工作旨在编写出满足需求、可靠、可扩展和易于维护的代码。最重要的是，目标是在规定的时间和预算内提供高质量的代码。

11.4　第 1 步：概念化阶段——从愿景开始

所有不同寻常的软件一开始都只是愿景（vision）。某人对一种产品有深刻的洞察力，认为很值得开发。在企业中，某人预见到产品或服务的远大前景，建议企业开发或提供。委员会很少能够提供引人注目的看法。

面向对象分析和设计的第一个阶段是用一句话（至多一小段话）来陈述这种愿景。愿景开发的指导原则，随着开发的进行，为实现愿景走到一起的开发小组应参照愿景，并在必要时对其进行修订。

即使愿景是由营销部门的委员会提出的，也应将某个人指定为“愿景提出者”，其职责是充当这盏圣灯的看守人。随着开发的进行，需求将演变。时间安排和面市要求可能（也应该）改变程序的第一次迭代中要完成的工作，但愿景提出者必须坚守核心理念，确保产品准确地反映核心愿景。这种忠实的信念（坚守承诺）伴随着项目的完成。如果对愿景视而不见，产品将注定以失败告终。

阐明愿景的概念化阶段很短。这可能是灵感出现的一瞬间加上将灵感记录下来的时间。在其他项目中，阐明愿景需要经过复杂，有时是激烈争论的范围确定（scoping）阶段，在这个阶段，参与的人员或小组必须就愿景的内容达成一致。在这个阶段，将什么包含在愿景中，将什么剔除，是项目成功的决定性因素，因为在完成这种工作的过程中，将对费用进行初步估算。

作为面向对象专家，你通常是在愿景提出后才参与到项目中的。

11.5　第 2 步：分析阶段——收集需求

有些公司将需求和愿景混为一谈。有价值的愿景是必不可少的，但光有愿景还不够。为进入分析阶段，必须知道产品将被如何使用以及它必须如何完成任务。分析阶段旨在阐明和收集需求。分析阶段的最终结果是，制作一份需求文档。需求文档的第一节是用例分析。

11.5.1　用例

分析、设计和实现的驱动力是用例。用例是对产品将被如何使用的高级描述。用例不仅推动分析，还推

动设计，帮助你确定类，且在测试产品时特别重要。

创建一组健壮、全面的用例可能是分析阶段最重要的任务。这里对域专家的依赖性最大，域专家是对要收集的业务需求最有发言权的专家。

用例不关心用户界面的细节，也不关心要构建的系统的内部细节，而将重点放在需要发生的交互以及协同工作以得到所需结果的人和系统（被称为参与者，actor）上。

总之，下面是一些定义：

- **用例**：描述软件将被如何使用。
- **域专家**：在要开发的产品针对的业务领域具备专业知识的人员。
- **参与者**：与要开发的系统进行交互的人或系统。

用例是对参与者和系统之间交互的描述。就用例分析而言，系统被视为一个“黑盒子”。参与者“发送一条消息”给系统，然后发生某些事件：返回信息、系统状态改变、宇宙飞船改变方向等。

要收集所有需求，仅有用例还不够，但它们是重要组成部分，通常最受关注。其他部分包括业务规则、数据元素、性能方面的技术需求、安全性等。

1. 确定参与者

需要注意的是，并非所有的参与者都是人；与要创建的系统交互的系统也是参与者。因此，如果开发的是自动柜员机（ATM）系统，则顾客、银行职员、与这个新系统交互的其他系统（如抵押跟踪系统或学生贷款系统）都可能是参与者。参与者的基本特征如下：

- 位于系统外部；
- 与系统交互。

提示：起步阶段常常是用例分析最困难的部分。通常，最好先召开一次“脑力激荡”会议。只记下与新系统交互的人和系统的列表。

人指的是角色：银行职员、经理、顾客等。同一个人可以有多种角色。

对于刚提到的 ATM 范例，角色清单中包括：

- 顾客；
- 银行职员；
- 后台办公系统；
- 将钱放入 ATM 中的人。

首先不要超越显而易见的清单。只需确定 3～4 个参与者，就可以开始编写用例。每个参与者都以不同的方式与系统交互，需要在用例中收集这些交互。

2. 确定第一批用例

必须找到一个切入点。对 ATM 范例而言，从顾客角色开始。顾客角色将执行哪些操作呢？通过脑力激荡，可得到下述针对顾客的用例：

- 查询余额；
- 存款；
- 取款；
- 转账；
- 开立账户；
- 注销账户。

应区分“顾客向支票账户存款”和“顾客向储蓄账户存款”，还是像上述清单所示将这些行为合并为“顾客账户中存款”呢？答案取决于在域（被模拟的现实世界环境，这里为银行系统）中这种区分是否有意义。

要决定这些操作是一个用例还是两个，必须询问机制是否不同（这些存款操作是否有重大不同）以及结

果是否不同（系统是否以不同的方式进行响应）。就存款而言，这两个问题的答案都是否定的：顾客向这两种账户中存款的方式基本相同，结果也极其相似：ATM 的响应是，增加相应账户的余额。

鉴于无论是向支票账户还是储蓄账户存款中，参与者和系统的行为和响应或多或少地相同，这两个用例实际上是一个用例。后面提供用例场景时，读者可尝试这两种变体，看它们是否有不同。

考虑每个参与者时，可以通过提出以下问题发现其他用例：

- 为什么参与者使用该系统？

顾客使用该系统来取钱、存款或查询账户余额。

- 参与者希望通过每种请求获得什么样的结果？

增加账户余额或取钱去购物。

- 什么事情导致参与者现在使用该系统？

他可能刚发工资或他正在去购物的途中。

- 要使用该系统，参与者必须做什么？

插入 ATM 卡。

这表明需要一个针对顾客登录系统的用例。

- 参与者必须向系统提供什么信息？

输入个人 ID 号。

这表明需要一个用于获得和编辑个人 ID 号的用例。

- 参与者希望从系统获得什么信息？

存款余额等。

通过重点关注域中对象的属性，常常可以发现其他用例。顾客有姓名、PIN 和账号，有管理这些对象的用例吗？账户有账号、存款余额、交易记录，这些信息被收集到了用例中吗？

详细分析顾客用例后，确定用例清单的下一步是确定其他每个参与者的用例。下述清单列出了 ATM 范例的第一组用例：

- 顾客查询存款余额。
- 顾客存款。
- 顾客取款。
- 顾客转账。
- 顾客开立账户。
- 顾客注销账户。
- 顾客登录账户。
- 顾客最近的交易记录。
- 银行职员登录到特殊的管理账户。
- 银行职员调整顾客的账户。
- 后台办公系统根据外部活动更新用户的账户。
- 用户账户的变更反映到后台办公系统中。
- ATM 发出信号，指出没有现金了。
- 银行技术人员给 ATM 补充现金。

3. 创建域模型

确定初略的用例后，便使用细化的域模型来编制需求文档。域模型是一份文档，记录了你知道的有关域（要开发的程序针对的业务领域）的全部信息。作为域模型的一部分，你创建域对象，对用例中提到的所有对象进行描述。至此，ATM 范例包括如下对象：顾客、银行职员、后台办公系统、支票账户、储蓄账户等。

对于每个域对象，需要收集基本信息，如对象名称（顾客、账户等）、对象是否是参与者、对象的主要属性和行为等。很多建模工具支持用在类描述中收集这些信息。图 11.4 说明了如何使用建模工具 Rational Rose 来收集这些信息。

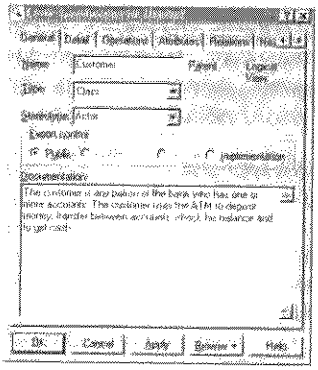

图11.4 Rational Rose

这里描述的并不是设计时将使用的类（虽然设计时可能使用类似的类），而是需求域中对象的类，认识到这一点非常重要。这是记录系统需求的文档，而不是系统将如何满足这些需求的文档。

可以使用UML通过图示来描述ATM范例域中对象之间的关系，这里使用的绘图约定与后面在设计中用来描述类之间关系的约定相同。这是UML的优点：可以在项目的每个阶段使用相同的表示法。

例如，通过使用有关类和泛化关系的UML约定，可以获知支票账户和储蓄账户都是更普遍的概念银行账户的具体化，如图11.5所示。

在图11.5中，方框表示各种域对象，带箭头的线段表示泛化关系。UML规定，箭头从具体类指向更通用的基类。因此，支票账户和储蓄账户都指向银行账户，表示它们都是银行账户的具体化。

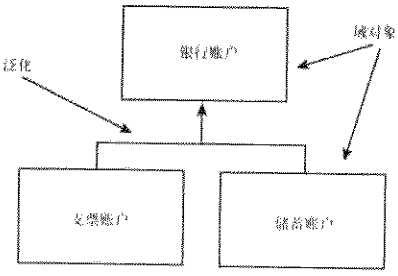

图11.5 具体化

注意：同样，需要指出的是，这里要绘制的是域需求中类之间的关系，以后可能决定创建CheckingAccount对象和BankAccount对象，并使用继承实现这种关系；但这些都是设计阶段的决策。在分析阶段，只需要在文档中记录对需求域的认识。

UML是一种丰富的建模语言，可以表示任意数目的关系。但在分析阶段，收集的主要关系如下：

- 泛化；
- 包含；
- 关联

（1）泛化

泛化常常相当于继承，但两者之间存在一个明显而有意义的区别。泛化描述的是关系，而继承是泛化的

（3）关联

在域分析期间，常用的第三种关系是简单关联。关联意味着两个对象以某种方式交互，但具体的交互方式不清楚。在设计阶段，将更准确地阐述这种交互关系；但就分析而言，建议指出对象 A 和对象 B 交互，至于包含的是泛化关系还是包含关系不必指出。在 UML 中，在两个对象之间绘制一条线段来表示关联，如图 11.8 所示。

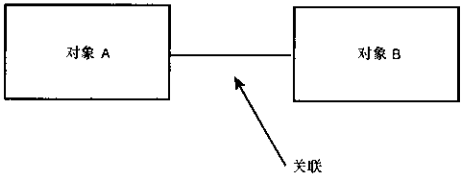

图 11.8 关联

图 11.8 表明，对象 A 以某种方式与对象 B 相关联。

4. 建立场景

有了一组初步的用例和图示域对象之间关系的工具后，可以将这些用例规范化，使之更为详细。

每个用例都可被划分成一系列场景。场景（scenario）是对一组具体情形的描述，将用例的各种元素区分开来。例如，用例“顾客取款”可能有如下场景：

- 顾客要求从支票账户提取 300 美元，从现金槽中拿走现金，然后系统打印凭条。
- 顾客要求从支票账户提取 300 美元，但存款余额只有 200 美元。顾客被告知支票账户中没有足够的余额。
- 顾客要求从支票账户提取 300 美元，但他当天已从该账户提取了 100 美元，而每天最多只能提取 300 美元。顾客被告知这一问题，然后他选择只提取 200 美元。
- 顾客要求从支票账户提取 300 美元，但打印凭条的纸用完了。顾客被告知这一问题，然后他选择不打印凭条。

每个场景都是原始用例的变体。通常，这些变体都是特殊情形（账户中金额不足、机器中金额不足等）。有时候，变体考虑了用例决策的细微变化，例如，顾客在提款之前是否要转账？

并非每种可能的场景都需要考虑，而应找出那些能确定系统需求或与参与者交互细节的场景。

5. 制订指导原则

作为方法的一部分，应该制订记录场景的指导原则。这些指导原则收集在需求文档中。通常，需要确保每个场景都包括如下内容：

- 前置条件：场景开始前，哪些条件必须满足。
- 触发器：什么导致场景开始。
- 参与者执行什么操作。
- 系统导致了什么样的结果和变化。
- 参与者接受什么样的反馈。
- 是否有重复性的活动发生，什么导致它们结束。
- 场景的逻辑流程描述。
- 什么导致场景结束。
- 后置条件：场景结束时，哪些条件必须满足。

另外，需要为每个用例和场景命名。因此，情形可能如下：

用例：顾客取款。

场景：从支票账户中成功取款。

前置条件：顾客已登录系统。

触发器：顾客请求“提款”。

描述：顾客选择从支票账户取款。账户中有足够的余额，ATM 中有足够的现金和凭条用纸，且网络正常运行。ATM 提示顾客输入要提取的数额，顾客输入 300 美元，这是本次取款的有效数额。机器送出 300 美元并打印凭条，然后顾客取走现金和凭条。

后置条件：顾客的账户减少 300 美元，顾客拥有 300 美元现金。

该用例可用如图 11.9 所示的非常简单的图形来表示。

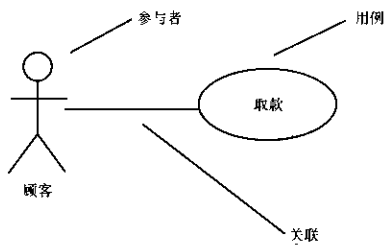

图 11.9　用例图

这里除参与者（顾客）和系统之间交互的高度抽象外，没有提供其他信息。说明用例之间的交互时，这种图将更有用些。这里说更有用些是因为只有两种可能的交互：<<uses>>和<<extends>>。构造型（stereotype）<<uses>>表示一个用例是另一个的超集。例如，不首先登录就不能取款，图 11.10 所示的图形说明了这种关系。

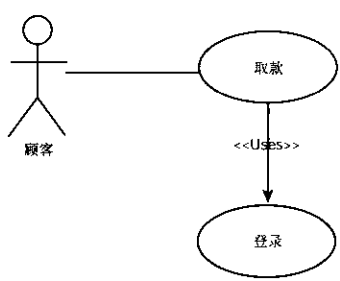

图 11.10　构造型<<uses>>

图 11.10 表明，“取款”用例使用了“登录”用例，因此“登录”是“取款”的一部分。

构造型<<extends>>用于表示条件关系以及类似于继承的东西。然而在对象建模领域，有关<<uses>>和<<extends>>之间的区别如此令人迷惑，以致于很多开发人员认为<<extends>>的含义并不能充分地被人理解，因此将其搁置不用。从个人而言，作者在需要复制并粘贴整个用例时使用<<uses>>，而在某些可定义的条件满足时才使用用例时，使用<<extends>>。

（1）交互图

虽然用例图本身的价值有限，但可以结合使用图和用例，从而极大地改善对交互的理解和记录。例如，场景“取款”表示了如下域对象之间的交互：顾客、支票账户和用户界面。可以使用交互图（也叫协同图）来说明这种交互，如图 11.11 所示。

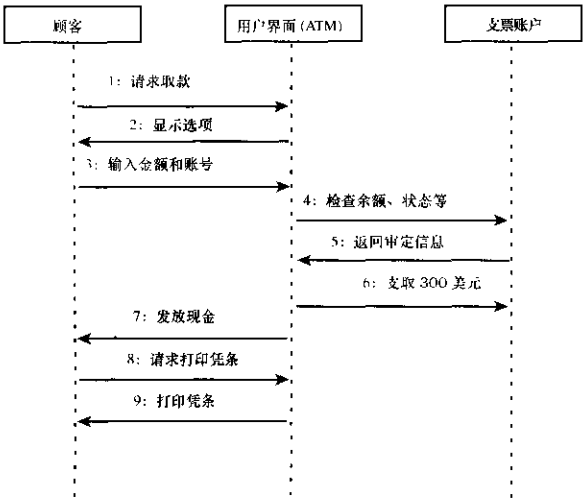

图 11.11 UML 交互图

图 11.11 所示的交互图提供了有关该场景的细节，而这些细节只靠阅读文本很难了解。交互的对象是域对象，整个 ATM/UI 被视为一个对象，在任何细节中，只涉及银行账户。

这个相当简单的 ATM 范例只说明了一组抽象的交互，但明确这些交互的细节对于理解新系统的问题域和需求大有裨益。

（2）创建包

对于任何非常复杂的问题，需要编写很多用例，UML 允许你将用例组合成包。

包就像目录或文件夹，它是一组建模对象（类、参与者等）。为控制用例的复杂度，可以按任何对问题来说有意义的特征将用例组合成包。可以按账户类型、存款还是取款、顾客类型或对你有意义的特征来组合用例。更重要的是，同一个用例可以出现在不同的包中，这提供了极大的灵活性。

11.5.2 应用分析

除创建用例外，需求文档还必须包含客户的假设、约束（有关硬件和操作系统的需求）、安全性和性能等方面的信息。这些需求是特定客户的先决条件，这些条件通常在设计和实现阶段确定，但客户已经为你决定了。

应用需求（有时被称为技术性需求）通常是由要与现有系统交互决定的。在这种情况下，理解现有系统的功能和工作原理是分析中不可或缺的一部分。

理想情况下，你分析问题、设计解决方案，然后决定哪种平台和操作系统最适合你的设计。这是理想情况，很少遇到。更常见的情况是，客户已经在特定操作系统或硬件平台上进行了投资。客户的业务计划要求你设计的软件在现有系统上运行，因此必须尽早收集这些需求并进行相应的设计。

11.5.3 系统分析

有些软件是独立运行的，只与最终用户交互。然而，你常常被要求与已有系统交互。系统分析是收集要与之交互的系统的所有细节的过程。新系统将充当服务器，为现有系统提供服务，还是充当客户？在系统之间的接口方面，可以进行协商还是必须采用现有的标准？另一个系统是稳定的，还是你必须不断对付一个移

动的目标呢？

这些问题及相关的问题必须在设计新系统前的分析阶段解决。另外，还需要尽量收集与其他系统交互隐含的约束和限制。它们会降低新系统的响应速度吗？它们会消耗资源和计算时间，从而对新系统提出很高的要求吗？

11.5.4　规划文档

知道系统必须做什么以及如何做后，就该着手创建有关时间和预算的文档了。通常，时间表是由客户规定：必须在 18 个月内完成。理想情况下，你根据需求估算设计和实现解决方案所需的时间。这是理想情况；现实情况是，大多数系统都有时间和成本方面的限制，真正的技巧在于，计算在指定的时间和预算下能完成多少所需的功能。

下面是确定项目预算和时间表时，必须记住的两条指导原则：

- 在知道区间的情况下，上限可能是乐观的。
- Liberty 定律指出，完成任何工作的时间都比预期的要长，即使你已经考虑了该定律。

鉴于这些现实情况，必须考虑工作的先后顺序，首先完成最重要的工作。不应有这样的期望：它如此简单，肯定有时间去完成。重要的是，用完时间时，现有的产品能够运行，可作为第一个发行版本。建造大桥时，如果用完了时间，没有机会铺设自行车道，这很糟糕；但仍可以通车并开始收费。如果时间用完时，大桥只建到了河流中间，将更糟糕。

关于规划文档，需要了解的一点是，它们通常是错误的。在过程的早期，几乎不可能给出对项目周期的可靠估计。有了需求后，可以准确地确定设计需要多长时间，准确地估计实现需要多长时间，合理猜测测试需要多长时间。然后必须至少加上 20%～25%的“活动空间”，随着项目的进展，对项目有更深入的了解后，可以缩短段时间。

注意：在规划文档中包含“活动空间”并非是不编写规划文档的借口。它只是警告不要过早地依赖于规划文档。随着项目的进展，你将对系统的工作原理有更深入的认识，估计的准确度也会提高。

11.5.5　可视化

需求文档的最后一部分是可视化。可视化是一个虚构的名称，指的是图表、图片、屏幕截图、原型及其他任何可视化表示，用于帮助思考和设计产品的图形用户界面。

对很多大型项目来说，可以开发一个完整的原型来帮助你（和客户）理解系统将如何运行。在有些小组中，原型是生动的需求文档；“真正”的系统被设计用来实现原型演示的功能。

11.5.6　可交付品

在分析和设计阶段的结尾，你将创建一系列文档（通常被称为可交付品，deliverable）。表 11.1 列出了分析阶段的一些可交付品。有多个小组需要使用这些文档。客户根据这些文档来确信你理解了他们需要什么，最终用户根据这些文档给项目提出反馈和指导；而项目小组根据它们来设计和实现代码。很多这类文档还给文档制作小组和质量保证小组提供了至关重要的材料，让他们知道系统应如何运行。

表 11.1　　在项目开发的分析阶段创建的可交付品

可交付品	描　述
用例报告	详细说明用例、场景、构造型、前置条件、后置条件和可视化的文档
域分析	描述域对象间关系的文档和图表
分析协同图	描述问题域中对象间交互的协同图
分析活动图	描述问题域中对象间交互的活动图
系统分析	描述将在其上创建项目的低级和硬件系统的报告和图
应用分析文档	描述客户对该项目的特定需求的报告和图

续表

可交付品	描　　述
运行期约束报告	描述性能特征和客户加入的约束的报告
成本和规划文档	包含指示项目进度、里程碑和成本的图表的报告

11.6　第 3 步：设计阶段

分析的重点是理解问题域，而过程的下一步——设计的重点是创建解决方案。设计是将对需求的认识转化为能够用软件实现的模型的过程。该过程的结果是设计文档。

设计文档分两部分：类设计和体系结构机制（Architectural Mechanism）。类设计部分又分为静态设计（详细描述各种类以及它们的关系和特征）和动态设计（详细描述类之间是如何交互的）。

设计文档中的体系结构机制部分提供有关如何实现对象持久性、并行性、分布式对象系统等的细节。本章余下的篇幅将重点放在设计文档的类设计方面；而本书其余的章节将解释如何实现各种体系结构机制。

11.6.1　什么是类

作为 C++程序员，读者已经习惯了创建类。正规的设计方法要求将概念 C++类和设计类分开，虽然它们密切相关。使用代码编写的 C++类是你设计的类的实现。存在一一对应的关系：设计方案中的每个类对应于代码中的一个类，但不要将这两种类混为一谈。显然，可以使用另一种语言来实现你设计的类，而类定义的语法可能不同。

大多数时候，讨论这两种类时并不加以区分，因为差别是高度抽象的。当你说，在模型中，Cat 类有一个 Meow()方法时，这意味着将在 C++类中也加入一个 Meow()方法。

使用 UML 图表示的是设计模型中的类，而在可编译的代码中表示的是实现中的 C++类。这种区分是有意义的，但很微妙。

对很多新手来说，最大的障碍是找出一组初始类以及理解什么是设计良好的类。一种简单的技巧是，写出所有用例场景，然后为每个名词创建一个类。请看下面的用例场景：

“顾客”选择从“支票账户”提取“现金”。“账户”中有足够的现金，“ATM”中有足够的现金和“凭条”用纸，且“网络”正常运行。ATM 请顾客输入要“提取”的“金额”，顾客要求提取 300 美元，这是有效的提取金额。“机器”发放 300 美元并打印出凭条，顾客取走“钱”和凭条。

你可能从这个场景中提取出下述类：

- 顾客；
- 现金；
- 支票账户；
- 账户；
- 凭条；
- ATM；
- 网络；
- 金额；
- 取款；
- 机器；
- 钱。

然后合并同义词得到如下清单，并为每个名词创建一个类：

- 顾客；
- 现金（钱、金额、取款）；

- 活期账户；
- 账户；
- 凭条；
- ATM（机器）；
- 网络。

当目前为止，这种方法很不错。然后，你可能使用图来描述某些类之间一些显而易见的关系，如图 11.12 所示。

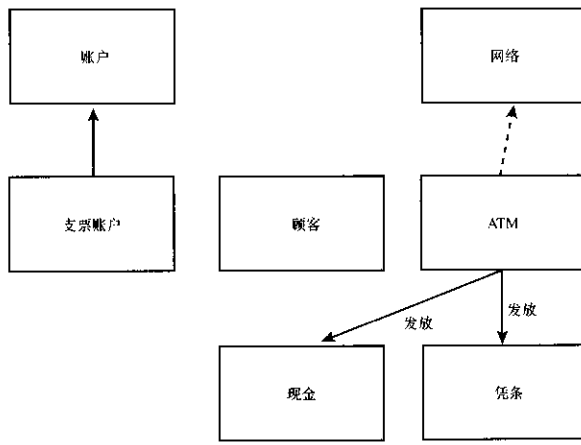

图 11.12　初步类

11.6.2　转换

前一节所做的与其说是从场景中提取名词，不如说是将域分析中的对象转换为设计中的对象。这是一个很好的开端。通常，域中的很多对象在设计中都有代理人（surrogtae）。将对象称为代理人旨在将 ATM 打印的实际凭条同设计中的对象区分开来，后者只是用代码实现的智能抽象。

你很可能发现，大多数域对象都在设计中有其代表，也就是说，域对象和设计对象之间存在一一对应的关系。然而，有时候，一个域对象在设计中是用一个系列对象表示；还有时候，一系列域对象可能被一个设计对象表示。

在图 11.12 中，支票账户是账户的具体化。你并没有着手查找泛化关系，但上述这个是不言自明的，因此收集到了。同样，根据域分析可知，ATM 发放现金和凭条，因此这种信息被立刻收集到设计中。

顾客和支票账户之间的关系不那么明显。我们知道它们之间存在某种关系，但具体细节并不明显，因此先将它放一放再说。

11.6.3　其他转换

转换域对象后，可以开始确定其他有用的设计阶段对象。通常，每个参与者都是一个类。一个不错的切入点是新系统和现有系统之间的接口，这种接口应封装到一个接口类中。然而，在考虑数据库和其他外部存储介质时一定要小心。通常，最好让每个类负责管理自己的永久性问题，即如何在用户会话之间存储和检索。当然，这些设计类可以使用通用类来访问文件或数据库，但最常见的情况是，操作系统或数据库厂商会提供这样的类。

这些接口类让你能够将你的系统与其他系统的交互封装起来，从而在其他系统发生变化时不影响你的代

码。接口类让你能够修改自己的设计或根据其他系统设计的变更做相应的修改，而不影响其他的代码。只要两个系统继续支持协商好的接口，它们就可以彼此独立地变更。

1. 数据操纵

同样，你可能需要创建用于操纵数据的类。如果必须将数据从一种格式转换为另一种格式（例如，从华氏转换为摄氏或从英制转换为公制），可能想将这些转换封装到一个特殊类中。在需要将数据转换为其他系统要求的格式或通过 Internet 进行传输所要求的格式时，可以使用这种技术；总之，每当需要将数据转换为指定的格式时，都可以将转换协议封装到数据操纵类中。

2. 视图和报告

系统生成的每一个“视图”和“报表”（如果生成了很多报告，则为每一组报告）都是一个候选类。报表采用的规则（如何收集和显示信息）可封装在视图类中。

3. 设备

如果你的系统要与设备交互或操纵设备（如打印机、相机、调制解调器、扫描仪等），应将设备协议的细节封装到一个类中。同样，通过为设备接口创建类，可以添加使用新协议的新设备，而不会影响其他代码：只须创建一个支持相同接口（或其派生接口）的新接口类即可。

11.6.4 建立静态模型

确定初步类集后，需要建立它们的关系和交互模型。为清晰起见，首先解释静态模型，然后解释动态模型。在实际设计过程中，可以自由地在静态和动态模型间切换，加入两种模型的细节，实际上是随着认识的深入，添加新类并进行补充。

静态模型主要关注三个方面：职责、属性和关系。其中最重要几首先需要关注的是每个类的职责。最重要的指导原则是，每个类应负责一件事情。

这并不是说每个类只能有一种方法，相反，很多类都有几十个方法。但这些方法必须是相关和内聚的：也就是说，它们必须互相关联，对类完成某方面职责的功能有贡献。

在设计良好的系统中，每个对象都是定义明确、易于理解并承担某方面职责的类的实例。类通常将额外的职责指派给其他相关的类。通过创建只关心某方面的类，有助于创建可维护性非常高的代码。

为控制类的职责，首先使用 CRC 卡来进行设计工作将大有裨益。

1. CRC 卡

CRC 表示类（Class）、职责（Responsibility）和协同（Collaboration）。CRC 卡是一张 4×6 的索引卡。这种技术含量低的简单工具让你能够和其他人员一起工作，以理解初始类集的主要职责。你们拿一叠 4×6 的空索引卡，围座在会议桌周围，召开一系列 CRC 卡会议。

（1）如何召开 CRC 会议

对于大型项目或组件，参加每次 CRC 会议的理想人数为 3～6 名，再多就不方便了。应有一名协调员，其职责是确保会议正常进行，帮助与会者记录获悉的信息。至少应有一名高级软件设计师与会，它最好是在面向对象分析和设计方面有丰富经验。另外，至少应有一两位域专家，他们了解系统需求，在有关系统应如何运行方面能够提供专家级建议。

对于 CRC 会议，最重要的是不能有重要的管理人员与会。这是一个创造性的、自由发挥的会议，不能受需要打动老板的影响。这里的目的是探索、冒险、确定类的职责以及了解它们如何交互。

开始 CRC 会议之前，应让与会人员围着会议桌就坐，并摆放一摞 4×6 的索引卡。在每张 CRC 卡的开头写下类名。在卡片中间画一条竖线，左边写上“职责”，右边写上“协同”。

首先为已确定的最重要的类填写卡片。在每张卡片的背面，写下一两句定义。还可以写下这个类是哪个

类的具体化，如果在填写 CRC 卡时这一点显而易见。只写出超类：在类名下面填写派生出当前类的类名。

（2）将重点放在职责上

CRC 会议的重点是确定每个类的职责。别太在意属性，只确定最基本和最明显的属性。重要的工作是确定职责。如果在完成职责时，类必须将工作指定给另一个类，在“协同”下放记录这些信息。

随着会议的进行，注意职责列表。如果 4×6 卡的空间不够，应考虑这个类所做的工作是不是太多。别忘了，每个类都应负责某个方面的工作，列出的各种职责应是内聚和相关的，即它们应协同工作来完成类的整体职责。

此时，不要关注关系，也不要考虑类的接口（哪个方法应是公有的，哪个方法应是私有的）。关注重点是每个类做些什么。

（3）拟人化和用例驱动

CRC 卡的重要功能是将类拟人化，即赋予每个类以人一样的品性。其工作原理如下：有了初步的类集后，回到 CRC 场景。将 CRC 随意分散在桌面上，将其与场景一起分析。例如，回到下面的场景：

顾客选择从支票账户取款。账户中有足够的余额，ATM 有足够的现金和凭条用纸，网络运行正常。ATM 请顾客输入取款金额，顾客请求提取 300 美元，这是有效的取款金额。机器发放 300 美元并打印凭条，顾客取走现金和凭条。

假设 CRC 会议有 5 位与会者：Amy 是协调人和面向对象设计人员；Barry 是首席程序员；Charlie 是客户；Dorris 是域专家；Ed 是一名程序员。

Amy 拿起表示 CheckingAccount 类的 CRC 卡说：“我告诉顾客可提取多少钱。顾客让我给他 300 美元。我发送一条消息给 ATM，告诉他提供 300 元现金。”Barry 拿起他的卡片说：“我是 ATM，我发放 300 美元并给 Amy 发送一条消息，告诉她将存款余额减去 300 美元。现在，ATM 中少了 300 美元，我告诉谁呢？要记录这一点吗？”Charlie 说：我认为需要一个对象来记录 ATM 中的现金。Ed 说：“不，ATM 应该知道自己有多少现金，这种信息应是 ATM 的一部分。”Amy 不同意：不，必须有人协调现金的发放。ATM 需要知道是否有现金以及顾客的账户中是否有足够的余额，且要数出钱并知道何时关上出钱口。它应将负责记录手上现金的职责交给他人：某种内部账户。知道手上有多少现金的人可以通知后台办公系统何时该补充现金。不然的话，ATM 做的工作太多了。

讨论继续进行。通过拿起卡片互相交流，确定了是否需要将职责委托给他人以及委托的时机，每个类都变得活灵活现，并明晰了其职责。当小组在设计问题上争论得不可开交时，协调人可以做出决定，让小组继续往下讨论。

（4）CRC 卡的局限性

虽然 CRC 卡是一个启动设计的强有力工具，但它们有固有的局限性。第一个问题是，它们的可扩展性不强。在非常复杂的项目中，需要的 CRC 卡数量可能非常大，仅记录这些卡片就很困难。

CRC 卡也不能用于记录类之间的关系。虽然它们能够用于记录协同，但协同的本质使其难以模拟。通过查看 CRC 卡，无法知道类之间的包含关系和泛化关系等。CRC 卡也没有记录属性，难以根据 CRC 卡来编写代码。最重要的是，CRC 卡是静态的，虽然可以将类之间的交互付诸行动，但 CRC 卡本身没有记录这种信息。

总之，CRC 卡让你能够有良好的开端，但要建立健壮、完整的设计模型，需要将 CRC 卡转换为 UML 图。虽然过渡到 UML 并不太难，但这是一条单行道。一旦将 CRC 卡转换为 UML 图，就无法回头，你将 CRC 卡搁在一边，并不回过头使用它们。使这两种模型保持同步太难了。

（5）将 CRC 卡转换为 UML 图

每张 CRC 卡都可直接转换为用 UML 模拟的类。职责转换为类方法，同时将已确定的所有属性加入到类中。卡片背面的类定义转换为类文档。图 11.13 是根据 CRC 卡 CheckingAccount 创建的 UML 类图。该 CRC 卡的内容如下：

类名：CheckingAccount。
超类：Account。
职责：
记录当前余额；
接受存款和转出；
开支票；
转入；
记录当天可通过 ATM 提取的余额。
协同：
其他账户；
后台办公系统；
ATM。

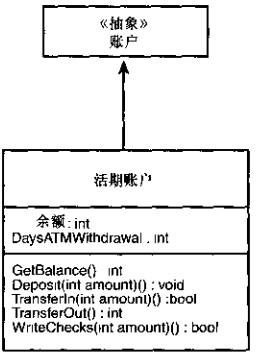

图 11.13 CheckingAccount 类的 UML 图

2. 类之间的关系

用 UML 表示类后，可以将注意力转向各个类之间的关系。需要对其建模的主要关系如下：

- 泛化；
- 关联；
- 聚集；
- 组合。

在 C++中，泛化关系是通过公有继承实现的。然而，从设计的角度看，你较少关注机制，而更关注语义：这种关系意味着什么。

你在分析阶段分析泛化关系，但现在将注意力转向设计中的对象而不是域中的对象。现在的工作是，将相关类共有的功能提取出来，放到能够封装共有职责的基类中。

提取共有功能时，将该功能从具体类移到更普通的类。也就是说，如果发现支票账户和储蓄账户都需要转入和转出款项的方法，则将 TransferFunds( )方法移到账户基类中。从派生类中提取的功能越多，设计的多态性越强。

多重继承是 C++支持而 Java 不支持的一项功能，虽然 Java 的多重接口提供了类似的功能。多重继承允许类继承多个基类，从而获得多个类的成员和方法。

经验表明，应慎用多重继承，因为它可能将设计和实现复杂化。很多最初用多重继承解决的问题现在都用聚集来解决。也就是说，多重继承是一个强大的工具，设计可能要求一个类具体化多个类的行为。

（1）多重继承和包含

对象是其各部分的和吗？将 Car 对象模拟为具体化的 SteeringWheel、Door 和 Tire（如图 11.14 所示）合理吗？

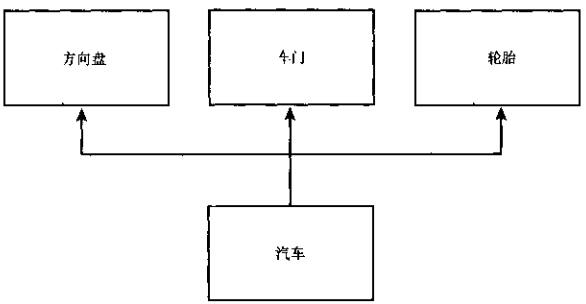

图 11.14 错误的继承

重要的是回到基本原理上来：公有继承应总是模拟泛化关系。对此一种通俗的表述是，继承应模拟 is-a（是一个）关系。要模拟 has-a（有一个）关系（如汽车有一个方向盘），应使用聚集，如图 11.15 所示。

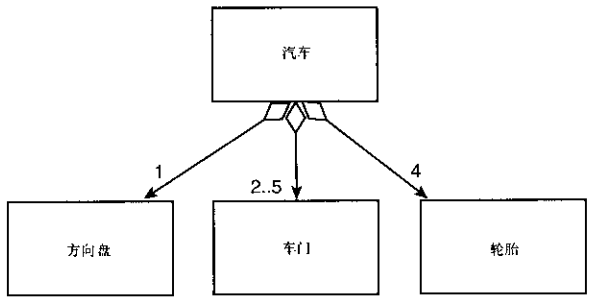

图 11.15　聚集

图 11.15 表明，一辆汽车有一个方向盘、4 个车轮和 2~5 扇车门。这是一个更准确的轿车与其部件间关系的模型。注意，图中的菱形不是实心的，因为这种关联为聚集而不是组合。组合意味着控制对象的生命周期。虽然汽车有轮胎和车门，但轮胎和车门在成为汽车的一部分前已经存在，在不再是汽车的一部分后还将继续存在。

图 11.16 模拟了组合。这个模型指出，身体不仅是一个头、两只胳膊和两条腿的聚集，且这些对象（头、胳膊、腿）创建身体时被创建，当身体消失时它们也将消失。也就是说，它们不能独立地存在，身体是由这些对象组成，它们的生命周期彼此相关。

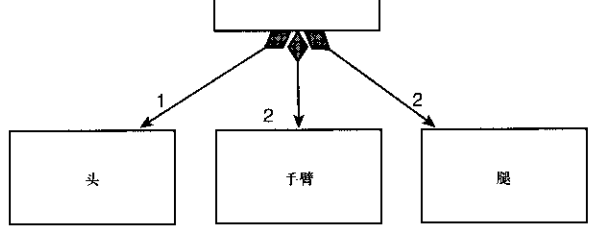

图 11.16　组合

（2）鉴别器和 Powertype

如何设计类来模拟典型的汽车制造商生产的各种车型呢？假设受雇为 Acme Motors 设计一个系统，该公司目前生产 5 种汽车：Pluto（装有小型发动机的低速小型汽车）、Venus（装有中型发动机的 4 门轿车）、Mars（装有公司最大功率发动机，可发挥最优性能的赛车）、Jupiter（一款车身较重的厢式旅行车，装有与赛车相同的发动机，但设计为在低转速时换挡）和 Earth（装有小型发动机，但速度较高的旅行车）。

你可能首先创建反映不同车型的汽车子类型，然后在每种车型离开装配线时创建其实例，如图 11.17 所示。

如何区分这些车型呢？正如前面指出的，它们是根据发动机大小、车体类型和性能特征区分的。可以通过组合这些特征来创建不同的车型。在 UML 中，可以使用鉴别器构造型来模拟这一点，如图 11.18 所示。

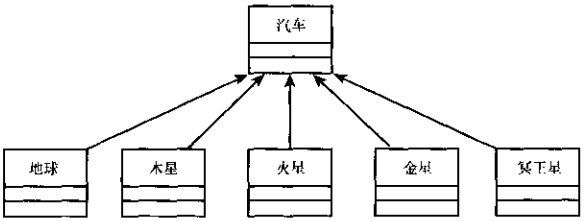

图 11.17 模拟子类型

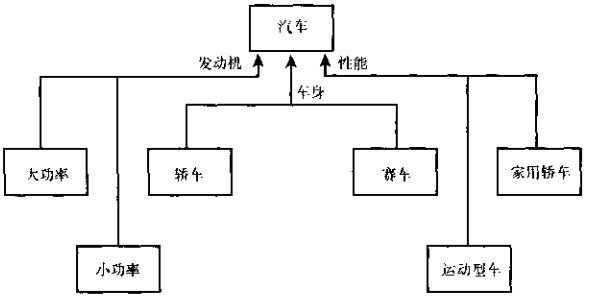

图 11.18 模拟鉴别器

图 11.18 表明，可以通过组合 3 种鉴别器属性从 Car 派生出类。发动机的大小指出汽车的功率，性能特征指出了汽车的运动性程度（sporty）。因此，可以派生出功率强大的运动型旅行车、功率较小的家用轿车等。

每种属性都可用一个简单的枚举类型来实现。在代码中，可以使用下面的语句来实现车体类型：

```
enum BodyType = {sedan, coupe, minivan, stationwagon };
```

然而，事实可能证明，简单的值不足以模拟某些鉴别器。例如，性能特征可能相当复杂。在这种情况下，可以使用一个类来模拟鉴别器，鉴别信息可封装到这个类的实例中。

因此，可能使用 Performance 类来模拟汽车的性能特征，它包含关于发动机在什么时候换挡以及转动速度的信息。在 UML 中，对于这样的类——可封装鉴别器，并可用于创建另一个类（Car）在逻辑上属于不同类型（如 SportCar 和 LuxuryCar）的实例——使用构造型<<powertype>>表示。在这个例子中，Performance 类是 Car 的 powertype。实例化 Car 时，也将实例化了一个 Performance 对象，并将该 Performance 对象与 Car 对象关联起来，如图 11.19 所示。

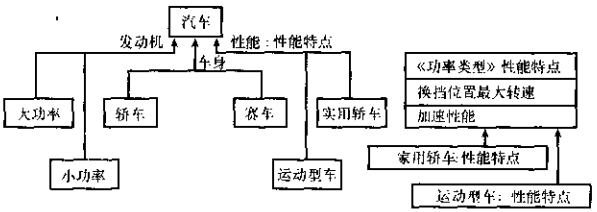

图 11.19 作为 powertype 的鉴别器

powertype 让你能够在不用继承的情况下创建各种逻辑类型，从而管理大量的复杂类型而不会遇到使用继承时可能出现的组合激增。

通常，在 C++中使用指针来实现 powertype。在这个例子中，Car 类包含一个指向 PerformanceCharacteristics 类实例的指针（见图 11.20）。将车体和发动机的鉴别器转换为 powertype 的工作，作为练习留给读者去完成。

警告：请记住，在运行阶段以这种方式来创建新类型会减少 C++的强类型功能（strong typing）带来的好处；这种功能让编译器能够确保类间关系的正确性。因此请慎用。

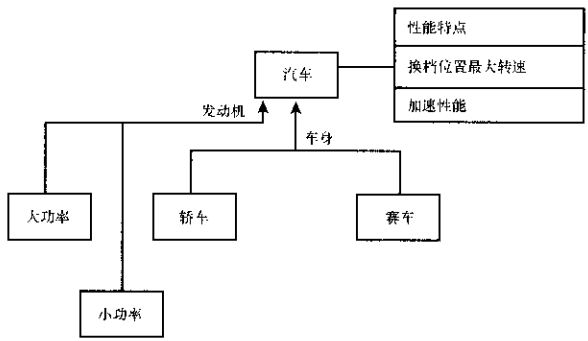

图 11.20　Car 对象与其 powertype 之间的关系

```
Class Car : public Vehicle
{
   public:
      Car();
      ~Car();
      // other public methods elided
   private:
      PerformanceCharacteristics * pPerformance;
};
```

最后需要指出的是，powertype 让你能够在运行阶段创建新的类型（不仅仅是实例）。由于每个逻辑类型只根据按相关联的 powertype 的属性进行区分，因此这些属性可以是 powertype 的构造函数的参数。这意味着可以在运行阶段创建新的汽车类型。也就是说，通过将不同的发动机大小和换档位置传递给 powertype，可以创建新的性能特征。通过将那些特征指定给各种汽车，可以在运行阶段增大汽车类型集。

11.6.5　动态模型

除模拟类之间的关系外，模拟它们之间如何交互也至关重要。例如，CheckingAccount、ATM 和 Receipt 类可能在完成“取款”用例的过程中与 Customer 交互。现在回到最初在分析中使用的时序图，但根据为这些类开发的方法添加细节，如图 11.21 所示。

这个简单的交互图说明了随着时间的推移在很多设计类之间的交互。假设 ATM 类将管理余额的全部职责指定给了 CheckingAccount 类，而 CheckingAccount 类要求 ATM 管理用户界面。

交互图有两种。图 11.21 所示的这种是时序图；协同图从另一个角度提供同样的信息。时序图强调事件的发生顺序；协同图强调类之间的交互，不涉及先后顺序。可以根据时序图直接生成协同图，诸如 Rational Rose 等工具将这种任务自动化，用户只需单击一个按钮后即可（见图 11.22）。

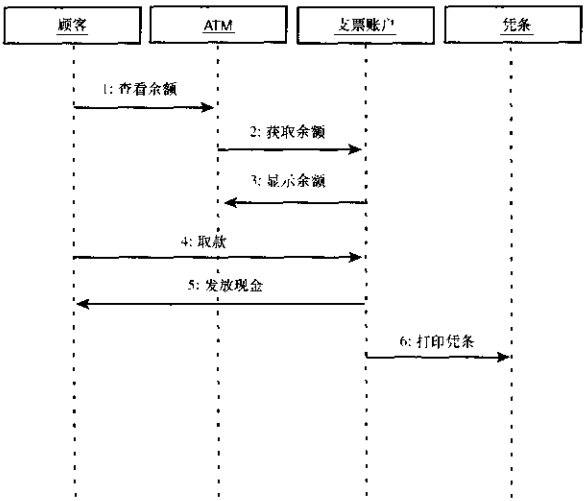

图 11.21　时序图

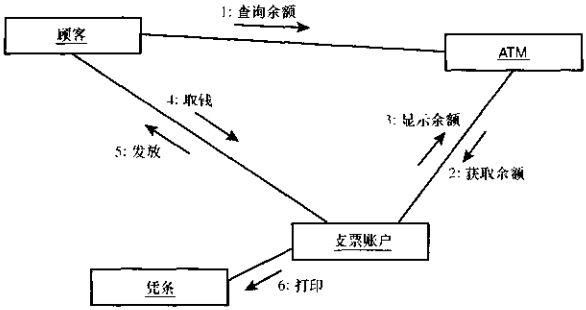

图 11.22　协同图

状态切换图

要了解对象间的交互，必须知道每个对象的各种可能状态。可以在状态图（状态切换图）中模拟各种状态之间的转换。图 11.23 说明了顾客登录系统时 CustomerAccount 类的各种状态。

每个状态图都是以一个“开始”状态开始，以零或更多个“结束”状态结束。各个状态都有名称，切换可能被标记。“保护措施”（guard）表示对象在状态间切换时必须满足的条件。

超状态

顾客可能在任何时候改变主意，决定登录。他可以在将卡插入 ATM 以提供账号后或在输入密码后这样做。无论在哪种情况下，系统必须接受顾客的请求，取消并返回“未登录”状态（见图 11.24）。

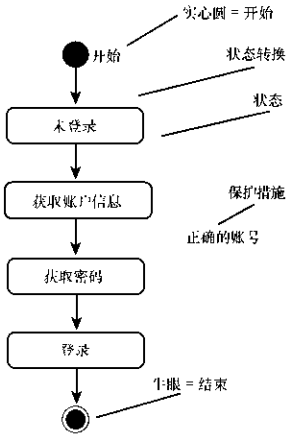

图 11.23　顾客账户的状态

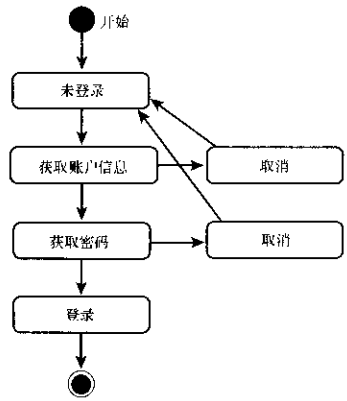

图 11.24　用户可以取消

正如读者看到的，在较复杂的状态切换图中，“取消”状态会使图变得非常乱。这很令人恼火，因为取消是一种特殊情形，不应在图中过于突出。可以使用一个“超级状态”来简化这个图，如图 11.25 所示。

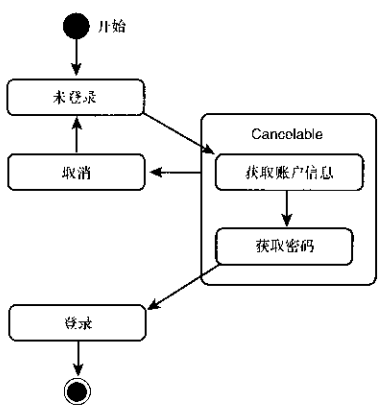

图 11.25　超级状态

图 11.25 提供的信息与图 11.24 相同，但更为清晰和易读得多。从开始登录到系统完成登录的整个过程中，任何时候都可以取消。如果顾客取消，将回到“未登录”状态。

11.7　第 4～6 步：实现、测试和交付

开发过程中余下的三个步骤很重要，但本章不打算介绍它们。就实现而言，如果使用 C++，则本书余下的篇幅介绍了具体细节。测试和交付都有复杂的规程和要求，然而详细介绍它们超过了本书的范畴。尽管如此，别忘了分别和同时仔细地测试类是决定是否成功地实现了设计的关键因素。

11.8 迭　　代

在 Rational Unified Process 中，前面列出的步骤被称为工作流程，将在开始、精制、构建和迁移阶段以不同的程度完成。例如，业务建模主要在开始阶段进行，但在构建阶段也可能进行，它表现为对开发的系统进行复核，以充实需求；实现主要在构建阶段进行，但也可能在精制阶段创建好原型时进行。

在每个阶段（如构建阶段），可能有多次迭代。例如，在构建阶段的第一次迭代中，可能开发系统的核心函数；在第二次迭代中深化这些功能并添加其他函数。在第三次迭代中，可能进一步的深化功能以及添加功能，直到在某次迭代中，系统是完善的为止。

11.9 小　　结

本章概述与面向对象分析和设计有关的主题。这种方法的实质是，分析新系统将被如何使用（用例）以及它将如何运行，然后设计类并模拟它们的关系和交互。

以前，人们对系统要完成的任务有大概的了解后开始编写代码。问题是，复杂的项目从来没有完成的时候，即使完成了，也是不可靠和脆弱的。通过预先进行调查以了解需求和模拟设计，可确保完成的产品是正确的（满足了设计要求）、健壮的、可靠的、可扩展的。

本书余下的大部分篇幅将重点放在实现的细节上。除了提到在实现时要规划单元测试以及将需求文档作为交付前测试规划的基础外，本书不讨论与测试和交付相关的主题。

11.10 问 与 答

问：本章没有介绍任何 C++编程知识，为何提供这样的内容？

答：为有效地编写 C++程序，必须知道如何组织它们。在编写代码之前进行规划和设计，可开发出更好、更有效的 C++程序。

问：面向对象分析和设计在哪些方面与其他方法完全不同？

答：在开发这些面向对象技术之前，分析员和程序员常常将程序视为操纵数据的函数组。面向对象编程将数据和函数集成为有内容（数据）和功能（函数）的离散单元。另一方面，过程化编程将重点放在函数以及它们如何操纵数据上。Pascal 和 C 程序是过程的集合，而 C++程序是类的集合。

问：面向对象编程最终将成为能够解决任何编程问题的灵丹妙药吗？

答：不，它从未打算这样做。然而，对于大型的复杂问题，面向对象分析、设计和编程能够为程序员提供这样的工具，即以以前不可能的方式管理高度复杂性。

问：C++是完美的面向对象语言吗？

答：与其他面向对象编程语言相比，C++有很多优点，也有很多缺点，但它有一个远远超过其他语言的主要优点：它是最流行的用于编写可执行应用程序的面向对象编程语言。作者编写本书也是因为 C++是很多公司选择使用的开发语言。

11.11 作　　业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量

先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

11.11.1　测验

1. 面向对象编程和过程化编程之间有什么不同？
2. 面向对象分析和设计包括哪些阶段？
3. 什么是封装？
4. 就分析而言，什么是域？
5. 就分析而言，什么是参与者？
6. 什么是用例？
7. 下列哪种说法是正确的？
   a. 猫是一种具体的动物。
   b. 动物是猫和狗的具体化。

11.11.2　练习

1. 计算机系统由很多部分组成，包括键盘、鼠标、显示器和 CPU。请绘制一个组合图来描述计算机与其组成部分的关系。提示：这是一种聚集关系。

2. 假设要模拟 Massachusetts Avenue 和 Vassar Street 之间的十字路口，这是两条典型的两车道马路，有交通灯和人行横道。模拟旨在判断信号灯变化的时间间隔能否确保车流量是平稳的。

在模拟中应建立哪几种对象的模型？为模拟应定义哪些类？

3. 请设计一个小组日程安排程序，该软件让你能够安排个人或团体会议，并保留有限数目的会议室。请确定主要的子系统。

4. 设计并列出练习 3 讨论的程序中，房间预订部分的类的接口。