第5章 领域建模

关键点

- 领域建模是一个概念化的过程，用于帮助开发团队理解应用领域。
- 5个简单步骤：收集应用领域的信息；头脑风暴；对头脑风暴的结果进行分类；使用UML类图将领域模型可视化；检查和审查。

Mary Sanders一直忙于一个保险公司的项目，但是她意外地被重新分到了另一个保健公司的项目。她的主管知道她会很快学会新的应用领域知识，事实也是如此，Mary按时交付了项目，而且没有大的问题。Mary的秘密是什么？她为什么能如此有效和高效地切换应用领域？

像Mary这样一个做IT咨询的软件工程师，经常会忙于不同应用领域的项目。即便一个软件工程师不是顾问，他或她也有可能被要求忙一个新项目或当前系统的全新扩展。软件工程充满了挑战也充满了惊喜，因为工程师们不时地得忙新的应用。一个好的软件工程师能快速理解一个新的应用领域，这是顶级的执行者相比于平庸的软件工程师最明显的差异。而工程师为了拥有这个能力所使用的工具叫做领域建模。贯穿本章，你将学到：

- 领域建模；
- 领域建模的重要性；
- 面向对象的概念；
- 领域建模步骤；
- UML类图。

5.1 什么是领域建模

多年前，我上一门课，教学生设计和实现蜂窝通信系统模拟器。课上我们学到了领域建模，并要求学生为这个无线模拟器项目建立领域模型。一个勤奋的学生几乎每天都会带着他在阅读有关无线通信的材料时所记的冗长的笔记来我的办公室。我督促他建立领域模型，而不是记笔记。遗憾的是，直到项目的截止日期马上就要到来的时候他也没有这么做。最后在提交了领域模型后，那个学生来找我说："我后悔没有早点进行领域建模。它确实以一种快速而有条理的方式帮我理解了无线通信。"

那么，什么是领域建模呢？

定义5.1 **领域建模**是一个概念化的过程，它旨在确认重要的领域概念及其属性和它们之间的关系。其结果通过一个叫做领域模型的图来描述。

概念化的过程包括观察、分类、抽象和归纳。这个过程很重要，因为软件从广义来说就是一个概念化的产品。关键开发归根到底就是**概念化**。例如构建一个银行系统——开发团队首先得了解银行的业务，这就需要开发团队了解银行应用中的实体或对象以及它们如何

互相关联，还有银行对象的属性或状态等。为了了解这些，团队得收集与应用相关的信息，分析信息，并构建模型来展示基本的认知。执行这些活动的过程叫做**领域建模**。

5.2 为什么要进行领域建模

通常要求软件工程师做不同领域的项目，而软件工程师们来自不同的背景，拥有不同工作经历，这也影响他们对应用领域的感知。例如，我们曾用 security 这个词来解释不同领域里甚至相同领域里的不同事物。在计算机领域它代表计算机安全；在金融领域它代表投资（如股票和公共基金）。因此，两个软件工程师可能对同一个词有不同的理解。甚至曾经在同一家公司同一个项目上一起工作多年的工程师对他们的应用领域也可能会有不同的感知。如果感知的差异不明显，那么就不会对软件产品的设计、实现、集成、测试和维护产生明显的影响；如果差异很大，那么在开发周期的某个阶段它就会浮现出来。这可能会导致重做很多工作以及项目交付日期的延迟。

构建领域模型有助于识别和解决感知的差异。尤其是：

- 领域建模有助于开发团队或分析师理解应用和应用领域。
- 领域建模让团队成员交流和改进他们对应用和应用领域的感知。
- 领域建模有助于开发团队向客户传达他们的感知并寻求反馈。
- 领域建模为后续的设计、实现、测试和维护提供了公共的概念基础。
- 领域模型可以帮助新的团队成员理解相关的应用和应用领域。

5.3 面向对象和类图

领域建模旨在识别出重要的领域概念和属性及概念之间的关系，这些可以通过类及类之间的关系来表现，并进而用 UML 类图来描述。本章会介绍一些面向对象范例的基本概念，以及如何用 UML 类图来显示它们。

5.3.1 外延定义和意向定义

领域建模是一个概念化的过程。作为过程的一部分，开发者要观察一个应用的特定实例，并对观察到的实例进行分类、抽象和归纳并产生一个概念模型，叫做领域模型。这个过程中一个重要的步骤就是分类。类这个概念是分类过程的最终产品。为了便于说明，想象一下一个同妈妈一起外出的两岁小孩。小孩看见了一只狗，他妈妈告诉他这是一只狗，所以他形成了狗的概念。然后他会看见不同颜色的其他狗和猫。到现在为止，小孩的概念化过程是一个归纳过程。他用于学习新概念的方法是基于外延定义的，外延定义的定义如下。

定义 5.2 一个**外延定义**通过列举概念的实例定义一个概念。

偶数的外延定义可以是数值的集合，集合中包括...，−4，−2，0，2，4，...。对小孩来说，“狗”的外延定义包括邻居的狗、街对面的狗、公园旁边的狗等。当小孩更多地看到世界的时候，他认识到猫和狗的行为是不一样的，但是它们都是动物。所以他知道如何根据属性来对具体的事物进行抽象、分类和归纳。他也知道如何把类关联起来。

某一天，这个小孩很兴奋，因为他要去拜访他的姨妈，而他的姨妈刚搬到一个花房里。这孩子以前没见过花房，而且因为他所在的镇子里没有花房，所以他很好奇。他根据他了解的花园和房子画了一幅花房的草图。然后他用乐高玩具搭建了花房。在这个例子中，小孩通过意向定义定义了他自己的概念。

定义 5.3　意向定义通过规约概念所拥有的属性和行为来定义一个概念。

领域建模类似于小孩使用外延和意向定义来感知世界。团队进入一个新的应用领域。就像那个小孩一样，团队列举看到的事物，将其分成类、属性和关系。其结果通过 UML 类图可视化。UML 类图定义如下：

定义 5.4　UML 类图是一个结构图，它描述了类及其属性和操作，以及类之间的关系。

图 5.1 展示了用于领域建模的类图的概念和标记，下面的章节会详细介绍。

概念	语　义	符　号
类属性操作	类是一个类型， 它的属性和操作 描述了类的对象	精简视图：类名 展开图：类名 / 属性列表 / 操作列表
继承	两个类之间 泛化/特别化的关系	子类 ▷ 超类
聚合	两个类之间的一部分关系 独占的部分	部分 ◇ 整体 部分 ◆ 整体
关联、方向、多样性、角色	两个类之间的二元关系	类1 [m] [角色1] — [lable] — [▶] [n] [角色2] 类2
关联类	描述关联类	关联类；[x]意味着x可选

图 5.1　一些常用的类图概念和标记

5.3.2　类和对象

定义 5.5　类是一个类型，是一个概念的意向定义。一个类封装了具有该类中特征的实例的属性和操作。一个对象是一个类的实例。

如图 5.2(a)和图 5.2(b)所示，类用**精简视图**和**展开图**都可以展示。如果不需要显示类的细节，就可以用精简视图。展开图用于显示一个类的属性和操作。精简视图和展开图可以用于同一个类图里。也就是说，有的类可以用精简视图显示，而其他的用展开图。

类的属性和操作也叫做**类的特征**。**特征**这个词很好地反映了捕获和强调类的最本质和最必需的属性和行为的意图。不管是固有的还是用户定义的，类是一个类型。它是一个意向定义，因为一个类可以通过一个谓词来定义。例如，当小孩形成了狗、猫、动物、人类和房子的概念后，在他的意识里就对每一个类都有一个谓词。当他画花房的时候，他也使用谓词。在面向对象的范例中，用于定义一个类的谓词是该类的属性和操作的集合。为了促进讨论，本章使用下面的约定。

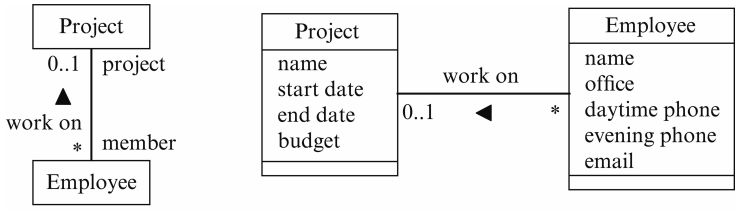


图 5.2　用精简视图和展开图表示类

- 类名首字母大写并使用单数形式：Customer、Bank 和 Account 可以用于表示类。
- 一个类型 X 的变量 x 可表示为 x:X。这既适用于标量也适用于类。例如，一个 Customer 类的变量 c 可以表示为 c:Customer。在不会引起混淆且 x 的类型在上下文中很清晰时，冒号和类型可以省略。
- 类 C 的实例 c 记作 c:C。例如，a1:Account、a2:Account、a3:Account 代表 Account 对象 a1、a2 和 a3。在不会引起混淆且该类在上下文中很清晰时，这些对象用 a1、a2 和 a3 来表示。下面的提示很重要：①:Account 代表 Account 的一个匿名对象。这在有些上下文中很有用，如一个循环中引用的对象。②x: 代表一个孤儿对象，即一个未定义的类的对象。这在多态的上下文中很有用，否则类就没什么意义。
- 在一个给定时间点，类 C 所有实例的集合记作 U(C)：

  $$U(C)=\{c \mid c:C\}$$ 或用速记标识

  $$U(C)=\{c1,c2,\ldots,cn\}$$，当 C 的对象可以枚举时

  因此，类 C 的一个实例 c 也可以记作 $c \in U(C)$。这里 U 叫做论域，U(C)是 U 在 C 上的映射。
- 如果 c 是类 C 的一个实例，那么分别使用 c.a 和 c.f(...)代表或访问 c 的一个属性 a 和一个函数 f(...)。

5.3.3　对象和属性

在实践中，有时很难决定某事物到底是一个属性还是一个对象(或类)。例如，颜色可以是车的属性，但是它也可以是一个销售颜色的应用的对象。另一个例子是建筑物，它可以是一个对象或属性(如某部门的地址)。可以用下面的方针把对象从属性中区分出来：

方针 5.1　一个(应用领域的)对象是一个应用或应用领域中的独立存在；但属性不是。

这里，**独立存在**意味着应用关心维护该对象的信息或状态。例如，“座位的数量”不是独立存在，因为它可能是汽车、教室或飞机上的座位数量，而它们几个都是对象。“独立存在”并不局限于物理存在，它也包括概念或精神上的存在。例如，一本书是一个物理存在，但一条纪律就是一个精神上的概念。

方针 5.2　属性描述对象的特征。

方针 5.3　属性可以通过输入设备(如键盘)输入但是对象不能；对象是通过调用构造函数创造的。

一个人不可能通过键盘输入一个学生对象，他只能输入学生姓名、学号、地址、电话号码

和其他信息。但这些是属性而不是对象。学生对象是通过调用 Student 类的构造函数来创造的。

5.3.4 关联

除了类和属性,领域模型也记录对象类之间的关系。领域建模中常用到 3 种关系,这在本节及后续章节进行介绍。

定义 5.6 **关联**是一个或多个类之间的关系。它描述了一个类的对象可能和其他类的对象有关系。

关联定义了在一个或多个类的对象之间的一种一般的关系。也就是说,关联可以是应用中有意义的任何关系。例如,在一个银行系统中,客户拥有账户。因此,Customer 类和 Account 类之间有关联。钱可以在账户之间进行转账。因此,Account 类与它自己有关联。在大学里,教员教授课程而学生参加课程。因此,Faculty 和 Course、Student 和 Course 之间有关联。两个类之间的关联通过连接两个类的实线表示。图 5.2 展示了 Employee 和 Project 的关联。也就是说,员工做项目。小的实心三角代表关联的方向,即员工工作于项目,而不是项目工作于员工。

表格表示形式能帮助理解关联关系。想一下 Customer 拥有 Account 这个关联,表示为 Own(Customer,Account)。Own 的类型是一个(Customer,Account)对。可以更严谨地表示为 Own:(Customer,Account),但是为了简单起见把冒号省略了。使用在 5.3.2 节最后规定的规约,假设 U(Customer)=$\{c1,c2,...c4\}$和 U(Account)=$\{a1,a2,...a5\}$。c1 有 a1 和 a2,c2 有 a2 和 a3,c3 有 a4,c4 没有账户,a5 不被任何客户拥有。如图 5.3(a)所示,Own(Customer,Account)关联包括 5 个对:(c1,a1)、(c1,a2)、(c2,a2)、(c2,a3)和(c3,a4)。这个表加上 U(Customer)和 U(Account)就可以看出一个客户可以拥有零个或多个账户,而一个账户可以被零个或多个客户拥有。事实上,由 4 个客户中的每一个加上 5 个账户中的每一个所构成的 20 个对的任何子集都是可以的。也就是说,有 2^{20} 个这样的子集,而它们每一个都是 Own 这个关联的合法实例。

最常遇到的关联是二元关联。二元关联是一个或两个类的对象之间的关系。例如,一个账户可以向另一个账户转账就是一个类的对象之间的二元关系。另一个例子是一个雇员监管其他雇员,就形成了 Employee 类的对象间的关联。也存在 3 个类的对象之间的关联或三元关联。例如,在 Professor、Student、Project 之间,教授监管正在做项目的学生就是一个三元的关系,它涉及 3 个类。图 5.3(b)就展示了这个三元关联的一些可能的实例。

Own

Customer	Account
c1	a1
c1	a2
c2	a2
c2	a3
c3	a4

(a) 二元关联的实例

Work-Supervised-by

Student	Project	Professor
Chen	OOM	Baker
Chen	SOA	Liu
Gupta	SOA	Liu
Rosa	Security	Brown
Smith	Security	Shah

(b) 三元关联的实例

图 5.3 关联的实例

正如在上述例子中所述，一个二元关联可以表示为 $V(X,X)$ 或 $V(X,Y)$，而一个三元关联表示为 $V(X,Y,Z)$，其中 V 是关联的名称，X、Y、Z 是类。本书在讨论关联时将一直使用这种表示方法。在不会引起混淆时，括号对和参数可以省略，因此 Own(Customer, Account)可以表示为关联 Own。

5.3.5 多重性和角色

关联 Own(Customer, Account)说明客户可以拥有账户。但是，它没有指明一个客户可以拥有多少个账户以及多少个客户可以拥有同一个账户。通常一个客户可以在一家银行拥有不止一个账户。有的银行允许账户被联合拥有，而另一些银行则要求一个账户只能被一个客户拥有。另一个例子是，一个学生可能做一个或多个项目。但是一个项目可能有零个或多个学生正在做。一个项目可能没有学生在做，因为项目可能是新的或者被中断了。做项目的学生可能被一个或多个教授监管。这些例子都说明了有必要来识别和确认所谓的多重性，定义如下。

定义 5.7 一个类的关联的多重性是一个类的实例数量的声明，即每个实例可以与关联中其他每个类的实例所有组合的数量。

想一下关联 Own(Customer, Account)。假设银行允许账户联合拥有，但是要求每个账户被至少一个客户拥有。关联 Own(Customer, Account)中 Customer 的多重性是一或多，因为 Account 对象可以被一个或多个客户拥有。在这种情况下，图 5.3(a)就不再合法了，因为账户 a5 没有客户。把(c4, a5)加到图 5.3(a)里就会满足这个约束。如果银行施加了一个限制，每个账户只能被一个客户拥有，那么 Customer 的多重性是一。在这种情况下，图 5.3(a)就不再合法了。从表中删除(c1, a2)就满足约束了。

对于包含 Professor、Student、Project 的三元关联，如果每个做某个项目的学生只能被一个教授监管，那么这个三元关联中 Professor 的多重性是一。尽管在图 5.3(b)中，学生 Chen 有两个不同的教授在监管，但是他也是在做两个不同的项目。因此，图 5.3(b)的关系满足该约束。如果一个或多个教授可以监管做某个项目的一个学生，那么这个三元关联中 Professor 的多重性是一或多。在这种情况下，图 5.3(b)仍然合法，因为做一个项目的每个学生都有一个教授监管。但是，如果一个学生只能做一个项目，并且只能被一个教授监管，那么图 5.3(b)的前两个对中需要删除一个。

图 5.4 展示了 UML 表示不同的多重性的方法。在类图里，多重性通过关联与类的边界的交叉点来表示。例如，图 5.2 显示每个员工做零个或多个项目，每个项目有零个或多个员工在做。

0..1	0或1	m..n	m到n
0..m	0到m	m..*	m或更多的数
,0..	0或更多的数	m	恰好是m
1	恰好是1(默认)	1..*	1或更多的数
i,j,k	明确列出		

图 5.4 表示不同的多重性的符号

对象在关联中扮演某种角色。例如在 Student、Project 和 Professor 的关联 Work-

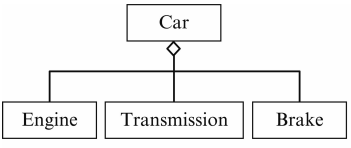
Supervised-by 中，教授扮演监管者的角色，而学生扮演研究助手的角色。在 Own (Customer, Account)这个例子中，客户扮演账户拥有者的角色。在类图里，角色的显示类似多重性。在图 5.2 所示的 work-on 关联中，Project 和 Employee 的角色分别是成员和项目。

5.3.6 聚合

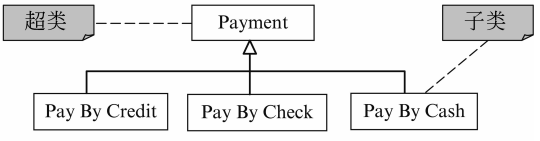
关联是一个或多个类之间的一般关系。在实际的应用中会经常遇到所谓的 part-of 关系。例如，引擎是汽车的一部分、一本书有若干章、每章又有若干节等。part-of 关系是一类特殊的关联，需要构建一个它自己的模型概念。因此，建模的社区长久以来就推出了所谓的聚合关系来适应这个需求。

定义 5.8 **聚合**是两个类之间的二元关联，其中一个类的对象是另一个类的对象的一部分。

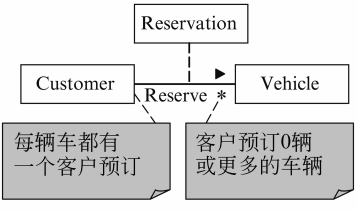
既然聚合是关联的特例，则用 Part-of(P,C)或 Part-of(P1,P2,...,Pn,C)来指代 C 是类 P 或 P1,P2,...,Pn 的整体。Part-of(P1,P2,...,Pn,C)实际是 Part-of(P1,C),Part-of(P2,C),...,Part-of(Pn,C)的一种速记表示法。类似地，为关联定义的多重性和角色也适用于聚合。例如，一辆汽车有两个或四个门。因此，Door 对于 Part-of(Door,Car)的多重性是二或四。带有菱形的线是 UML 中聚合的表示方法；菱形在整体的类一方。图 5.5(a)说明一个 Car 是一个 Engine、一个 Transmission 和一个 Brake 的整体。



(a) 聚合——发动机、变速器和制动器是汽车的一部分



(b) 继承——通过信用卡支付、通过支票支付、通过现金支付都是支付



(c) 关联和关联类

图 5.5 表示关系和关联类

5.3.7 继承

定义 5.9 **继承**是两个概念或两个类之间的二元关系，其中一个概念或一个类是泛化(或特别化)的另一个概念或类。

在不同的应用领域都存在很多继承的例子。在银行系统里，Checking Account(支票账户)和 Savings Account(储蓄存款账户)都是特别化的 Account。经纪公司会把 Margin

Account(保证金账户)和 Option Account(期权账户)区分开,这些不同类型的账户很类似,但是它们还是具有不同行为。例如,支票账户不付利息而储蓄存款账户要付。也就是说,Checking Account 和 Savings Account 是特别化的 Bank Account(银行账户)。Bank Account 是 Checking Account 和 Savings Account 的泛化。类似地,Margin Account 和 Option Account 是特别化的 Brokerage Account(代理账户),Brokerage Account 是 Margin Account 和 Option Account 的泛化。

在继承关系中,是另一个类的泛化的那个类叫超类,是特别化的另一个类的那个类叫子类,如图 5.5(b)所示。继承关系也叫做 IS-A 关系。这是因为每一个子类的实例都是一个超类的实例。例如,每个支票账户或储蓄存款账户都是一个银行账户。可以观察到为超类定义的属性和行为会被子类继承。这解释了"继承"这个术语的作用,它代表类之间的泛化或特别化。

在上述讨论中,有必要定义超类 Bank Account 的子类 Checking Account 和 Savings Account 的原因是由于子类对象之间行为的不同。另一个定义子类的原因是关系的不同。例如可以想一下,Undergraduate Student(本科生)和 Graduate Student(研究生)都是 Student 的子类。要求本科生用本科生的目录选课,而研究生用研究生的目录选课。需要一个全日制的研究生参加 5~8 门研究生课程。另一方面,本科生可能有不同的限制。定义两个子类将会很大地促进对这些关联关系的管理。继承也是关联的特例。因此,ISA(S,C)或 ISA(S1,S2,...,Sn,C)用来说明类 S 或 S1,S2,...,Sn 是 C 的子类。ISA(S1,S2,...,Sn,C)是 ISA(S1,C),ISA(S2,C),...,ISA(Sn,C)的速记表示方法。

5.3.8 继承和多态

在面向对象的范例中,继承和多态就像兄弟,甚至是双胞胎。它们太像了,因此很多作者都认为它们是一样的。但是事实上它们并不一样。为了了解它们的差异,推出下面的定义:

定义 5.10 **多态**的意思是一个事物可以呈现不同的形态。

通过观察定义 5.9 和 5.10,可以看到它们的区别。继承定义了两个概念或两个类之间的关系,其中一个类比另一个类更一般化(或更特殊化)。由于这种泛化(或特别化)的关系,一个超类的变量可以指其中一个子类的对象。因此,一个超类的变量可以呈现不同的行为,就达到了多态。

另一方面,多态关系一个事物呈现不同形态的能力。既然对"一个事物"没有限制,那么它可以是任何事物,不仅仅是类和对象。呈现不同形态的能力不局限在继承的上下文中,它可以在任何其他的上下文中。例如,两个或多个成员函数可以共享同一个名称,但是它们的签名不同。一个 C++ 的指针函数可以指向不同的运行时函数。这都是多态的例子,但是它们都不在继承的上下文里。

5.3.9 关联类

有时有必要介绍一下关联类的属性。例如想一下学生参加课程这个关联。一个学生可以参加一门或多门课程,并收到每门所参加课程的成绩。很显然,课程成绩是一个属性,因

为它不是一个独立存在。问题是,哪个对象类应该将成绩作为它的属性?如果成绩是 Student 类的属性,那么 Student 类就应该有多个成绩属性,因为一个学生可以参加不止一门课。如果是这种情况,那么你如何区分不同课程的成绩呢?下面给出一些快速但不一定是最好的解决方案。

一个可能的解决方案是将每个成绩同一门课程关联起来。这可以通过两种方法实现。第一种是在 Student 类中使用一张哈希表来存储课程的成绩。这种方法不太方便,因为哈希表及 Enroll(Student,Course)关联应该有一对一的对应关系。但是这种对应关系实际只是一种隐含的假设。对其进行处理的程序需要维护这种对应关系。这就意味着对于存储在哈希表中的每门课程成绩,程序要确保学生实际参加了这门课。程序必须确保对于一个学生参加的每门课,在哈希表里要有个入口用来记录这个学生的成绩。

除了上述的对应问题,记录学期也很常见,在这个学期里学生因为某门课获得了成绩。这是很有必要的,因为这学生可能很多年前上了这门课,然后大学通常要求学生重上这门课来获得学分满足学历要求。在这种情况下,使用哈希表为每门课程存储成绩和学期并不是我们想得到的方案。

另一个方法是为每门课定义一个 Course-Grade 类并得到一个一对多的关联,如从 Student 类到 Course-Grade 类的 Receive(Student,Course-Grade)。这种方法可以满足需求,为每门课程记录成绩的同时记录学期。但是,它仍未解决 Receive 关联和 Enroll 关联的实例间一对一对应的隐含假设问题。也就是说,还需要额外的工作量来确保每个 Receive 关联的实例都有一个对应的 Enroll 实例,每一个 Enroll 实例都有一个对应的 Receive 实例。类似地,如果定义成绩是 Course 的属性,那么还会遇到上面相同的问题。另一个快速的解决方案建议增加一个 Grade 类和两个关联:一个从 Student 到 Grade 的一对多关联,如 Has(Student,Grade);另一个是从 Grade 到 Course 的一对多关联,如 Is-for(Grade,Course)。但是这个方案仍然没有解决上述的隐含对应假设问题。

对讨论问题进行认真分析可以得出,成绩属性实际是 Enroll(Student,Course)关联的一条信息。类似地,在期间学生上课并接收成绩的学期也是 Enroll(Student,Course)关联的一条信息。上述快速解决方案试图不带有关联本身但是却带有其他类来存储这两条信息,产生了对应的问题。因此,一个更直接更好的方法就是带有关联本身地存储这些信息。这就消除了对应问题。那么如何来完成呢?下面的定义指向了一个答案。

定义 5.11 一个**关联类**是个特殊的类,它为一个关联的实例定义了属性和行为。

再想一下 Enroll(Student,Course)关联。在这个例子里,可以为从 Student 到 Course 的关联 Enroll 定义一个关联类,叫做 Enrollment。Enrollment 将会有成绩和学期等属性,以及 setGrade(...)、getGrade()、setSemester(...)和 getSemester()操作。如果一个学生上了三门课,那么只需要创建 Enrollment 类的三个实例,每个实例针对学生所上的一门课。图 5.6 展示了如何用表格形式代表一个关联类的实例,其中前两列是关联实例,后两列是关联实例的属性。关联类通过一条连接关联类和关联线的虚线表示。在图 5.5(c)中,Customer 和 Vehicle 之间有 reserve 关联。Reservation 类存储了哪个客户预订了哪辆车。它是一个关联类,在图中用一条从 Reservation 到 reserve 关联线的虚线表示。

Student	Course	Semester	Grade
Bachman	AI	Spr 07	A
Backman	DB	Spr 07	A
Backman	SE	Spr 07	A
Chang	AI	Spr 07	B
Chang	DB	Spr 07	A
Chang	SE	Spr 07	A
Chang	Compiler	Fal 06	B
Chang	Algorithms	Fal 06	A
Chang	Programming	Fal 06	B

图 5.6　关联类的对象的表格表示形式

5.4　领域建模的步骤

前面已经介绍了面向对象的概念，现在是时候介绍一下领域建模的步骤了。图 5.7 介绍了它的步骤及输入输出。这些步骤可能需要迭代若干次才能生成一个好的领域模型。下面列出了大概的步骤，并在后续章节里有详细的介绍。

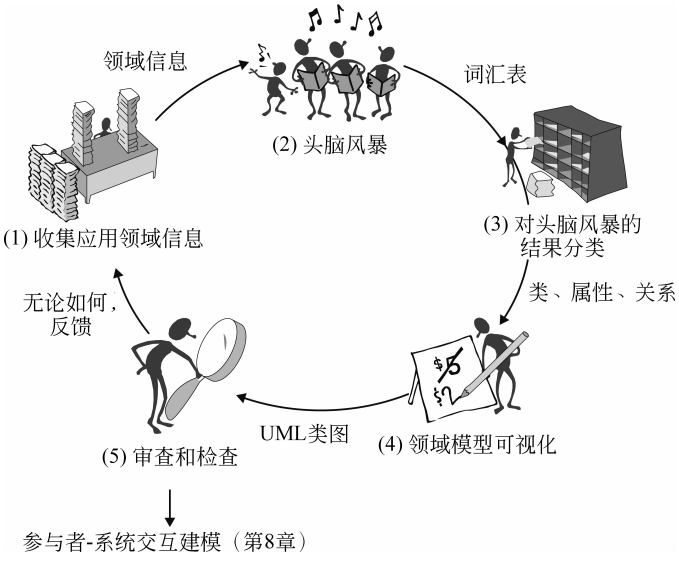


图 5.7　领域建模的步骤

第 1 步　收集应用领域信息

领域建模的第一步是收集应用领域信息。收集应用领域信息的技术在前面已经介绍过了，并会在 5.4.1 节回顾。这一步的输出是关于应用的所有相关的信息/文档。

第 2 步　头脑风暴

在收集应用领域信息后，开发团队成员会一起开会来确认并列出最重要的应用领域概念，这在 5.4.2 节介绍。