第 5 章 UML类图简介

因为本书只使用类图，所以本章简要介绍统一建模语言（Unified Modeling Language，UML）中的类图。

类图（Class Diagram）属于结构图，常被用于描述一个系统的静态结构。一个类图中通常包含类（Class）的 UML 图、接口（Interface）的 UML 图、泛化关系（Generalization）的 UML 图、关联关系（Association）的 UML 图、依赖关系（Dependency）的 UML 图和实现关系（Realization）的 UML 图。

5.1 类

在 UML 中，使用一个长方形描述一个类的主要构成，将长方形垂直地分为三层，如图 5.1 所示。

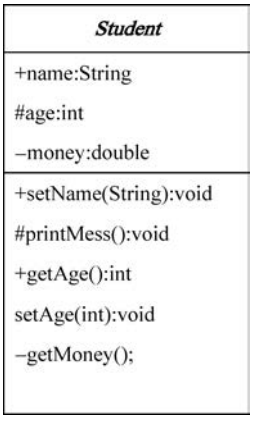

图 5.1 类的 UML 图

顶部第 1 层是名字层。如果类名字是常规字形，表明该类是具体类；如果类名字是斜体字形，表明该类是抽象类。

第 2 层是变量层，也称属性层，列出类的成员变量及类型，格式是“变量名字：类型”。在用 UML 表示类时，可以根据设计的需要只列出最重要的成员变量的名字。如果变量的访问权限是 public 的，需要在变量的名字前面用“＋”符号修饰；如果变量的访问权限是 protected 的，需要在变量的名字前面用“＃”符号修饰；如果变量的访问权限是 private 的，需要在变量的名字前面用“-”符号修饰；如果变量的访问权限是友好的，变量的名字前面不使用任何符号修饰。

第 3 层是方法层，也称操作层，列出类的方法及返回类型，格式是“方法名字（参数列表）：类型”。在用 UML 表示类时，可以根据设计的需要只列出最重要的方法。如果方法的访问权限是 public 的，需要在方法的名字前面用“＋”符号修饰；如果方法的访问权限是 protected 的，需要在方法的名字前面用“＃”符号修饰；如果方法的访问权限是 private 的，需要在方法的名字前面用“－”符号修饰；如果方法的访问权限是友好的，方法的名字前面不使用任何符号修饰；如果方法是静态方法，方法的名字下面加下画线。

5.2 接口

UML 表示接口的 UML 图和表示类的 UML 图类似，使用一个长方形描述一个接口的主要构成，将长方形垂直地分为三层，如图 5.2 所示。

顶部第 1 层是名字层，接口的名字必须是斜体字形，而且需要用<< interface >>修饰名字，并且该修饰和名字分列在两行。

第2层是常量层，列出接口中的常量及类型，格式是“常量名字：类型”。在Java接口中，常量的访问权限都是public的，所以需要在常量名字前面用“+”符号修饰。

第3层是方法层，也称操作层，列出接口中的方法及返回类型，格式是“方法名字(参数列表)：类型”。在Java接口中，方法的访问权限都是public的，所以需要在方法名字前面用“+”符号修饰。

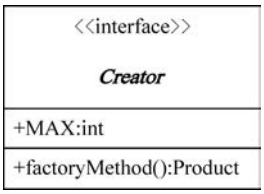

图5.2 接口的UML图

5.3 泛化关系

对于面向对象语言，UML中所说的泛化关系是指类的继承关系。如果一个类是另一个类的子类，那么UML通过使用一个实线连接两个类的UML图来表示二者之间的继承关系，实线的起始端是子类的UML图，终点端是父类的UML图，但终点端使用一个空心的三角形表示实线的结束，如图5.3所示。

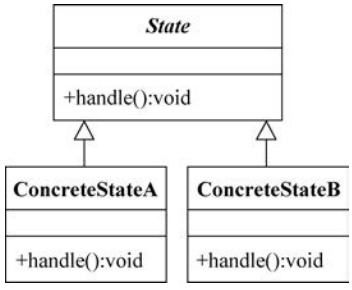

图5.3 继承关系的UML图

5.4 关联关系

如果A类中的成员变量是用B类(接口)声明的变量，那么A和B的关系是关联关系(也称组合关系)，称A关联于B(A组合B)。如果A关联于B，那么UML通过使用一个实线连接A和B的UML图，实线的起始端是A的UML图，终点端是B的UML图，但终点端使用一个指向B的UML图的方向箭头表示实线结束，如图5.4所示。

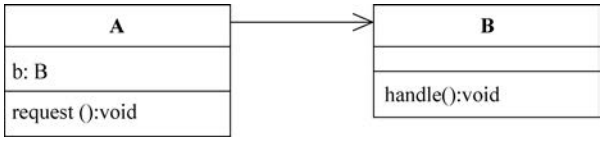

图5.4 关联关系的UML图

5.5 依赖关系

如果A类中某个方法的参数是用B类(接口)声明的变量，或某个方法返回的数据类型是B类型的，那么A和B的关系是依赖关系，称A依赖于B。如果A依赖于B，那么UML通过使用一个虚线连接A和B的UML图，虚线的起始端是A的UML图，终点端是B的UML

图,但终点端使用一个指向 B 的 UML 图的方向箭头表示虚线结束,如图 5.5 所示。

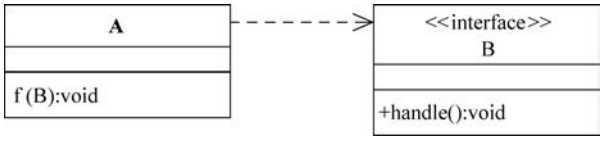

图 5.5　依赖关系的 UML 图

5.6　实现关系

如果一个类实现了一个接口,那么类和接口的关系是实现关系,称类实现接口。UML 通过使用虚线连接类和它所实现的接口,虚线起始端是类,终点端是它实现的接口,但终点端使用一个空心的三角形表示虚线的结束,如图 5.6 所示。

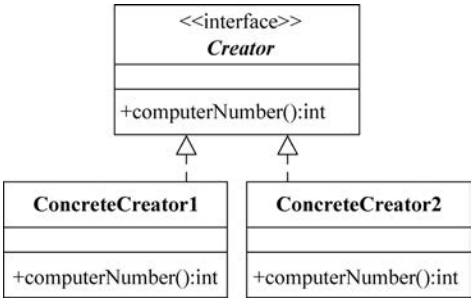

图 5.6　实现关系的 UML 图

5.7　注释

UML 使用注释为类图提供附加的说明。UML 在一个带卷角的长方形中显示给出的注释,并使用虚线将这个带卷角的长方形和它所注释的实体连接起来,如图 5.7 所示。

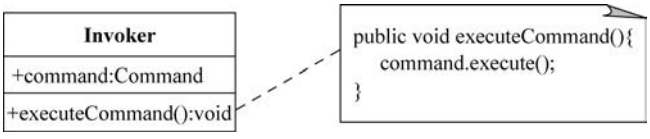

图 5.7　在类图中添加注释

第6章 设计模式简介

本章简要介绍设计模式，包括设计模式的起源、GoF 著作的贡献以及设计模式与框架的区别。

6.1 什么是设计模式

设计模式是针对某一类问题的最佳解决方案，而且已经被成功应用于许多系统的设计中，它解决了在某种特定情景中重复出现的某个问题。因此，可以这样定义设计模式(pattern)："设计模式是从许多优秀的软件系统中总结出的成功的、可复用的设计方案。"建筑大师 Alexander 关于设计模式的经典定义是："每一个设计模式描述一个在我们周围不断重复出现的问题，以及该问题的解决方案的核心。这样，你就能一次一次地使用该方案而不必做重复劳动。"尽管 Alexander 所指的是城市和建筑设计模式，但他的思想也同样适用于面向对象设计模式，只是在面向对象的解决方案中，用对象和接口代替了墙壁和门窗。两类模式的核心都在于提供了相关问题的解决方案。

记录一个设计模式需有四个基本要素[1]。

1) 名称

一个模式的名称高度概括该模式的本质，有利于该行业术语统一，便于交流使用。

2) 问题

问题描述应该在何时使用模式，解释设计问题和问题存在的前因后果，描述在怎样的环境下使用该模式。

3) 方案

方案描述设计的组成部分、它们之间的相互关系及各自的职责和协作方式。

4) 效果

效果描述模式的应用效果及使用模式应当权衡的问题。效果主要包括使用模式对系统的灵活性、扩充性和复用性的影响。

例如，中介者模式的四个基本要素如下。

- 名称：中介者。
- 问题：用一个中介者封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。
- 方案：中介者(Mediator)接口、具体中介者(Concrete Mediator)、同事(Colleague)、具体同事(Concrete Colleague)。
- 效果：减少了子类的生成，将各个同事解耦，简化了对象协议，控制集中化。

6.2 设计模式的起源

软件领域的设计模式起源于建筑学。1977 年，建筑大师 Alexander 出版了 *A Pattern Language*：*Towns*，*Building*，*Construction* 一书，Alexander 在该著作中将建筑行业中许多

问题的最佳解决方案记录为200多种模式,这些模式为房屋与城市的建设制定了一些规则。Alexander著作中的思想不仅在建筑行业影响深远,而且很快影响到了软件设计领域。1987年,受Alexander著作的影响,Kent Beck和Ward Cunningham将Alexander在建筑学上的模式观点应用于软件设计,开发了一系列模式,并用Smalltalk语言实现了雅致的用户界面。Kent Beck和Ward Cunningham在1987年举行的一次面向对象的会议上发表了论文《在面向对象编程中使用模式》,该论文发表后,有关软件的设计模式论文以及著作相继出版。

6.3 GoF之书

目前,在设计模式领域公认的最具影响力的著作是Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides在1994年合作出版的著作*Design Patterns*:*Elements of Reusable Object Oriented Software*(中译本《设计模式:可复用的面向对象软件的基本原理》),该书记录了四位作者在他们四年多的工作中所发现的23个模式。这部著作成为空前的畅销书,对软件设计人员学习、掌握和使用设计模式产生了巨大的影响。《设计模式:可复用的面向对象软件的基本原理》一书被广大喜爱者昵称为GoF(Gang of Four)之书,被认为是学习设计模式的必读著作,也被公认为设计模式领域的奠基之作。

自GoF之书出版后,受其影响,陆续出版了许多具有一定影响力的书籍,例如1998年,Alpert、Brown和Woolf出版*The Design Pattern Smalltalk Companion*,该书使用Smalltalk语言讲解了GoF之书中的23个模式;2000年,James W. cooper出版*Java Design Patterns*:*A Tutorial*,该书使用Java语言讲解了GoF之书中的23个模式(中译本《Java设计模式》),尤其侧重使用GUI程序设计讲解怎样使用GoF之书中的23种模式;特别要提到的是Eric Freema等在2004年出版的*Head First Design Pattern*(中译本《Head First设计模式》),该书使用Java语言重点讲解GoF之书中的部分模式(13个模式),该书图文并茂、独具匠心的写作风格令人耳目一新,语言叙述及结构安排非常适合初学者。在本书参考文献部分还列出了部分具有一定影响力的有关设计模式的著作。

6.4 学习设计模式的重要性

一个好的设计系统往往是易维护、易扩展、易复用的,有经验的设计人员或团队知道如何使用面向对象语言编写出易维护、易扩展和易复用的程序代码。《设计模式:可复用的面向对象软件的基本原理》一书正是从这些优秀的设计系统中总结出的设计精髓。尽管GoF之书并没有收集全部的模式(收集全部的模式似乎是不可能的,也是不必要的),但所阐述的23种模式无疑是使用频率最高的模式。

使用设计模式的目的不是针对软件设计和开发中的每个问题都给出解决方案,而是针对某种特定环境中经常遇到的软件开发问题给出可重用的解决方案。因此,学习设计模式不仅可以用好这些成功的模式,而且可以更加深刻地理解面向对象的设计思想,更好地使用面向对象语言解决设计中的问题。另外,学习设计模式对于进一步学习、理解和掌握框架是非常有帮助的,例如Java EE中就大量使用了《设计模式:可复用的面向对象软件的基本原理》一书中的模式,对于熟悉设计模式的开发人员,很容易理解这些框架的结构,继而可以很好地使用框架设计他们的系统。《设计模式:可复用的面向对象软件的基本原理》一书所总结的成功模式

不仅适合于面向对象语言，其思想及解决问题的方式也适合于任何和设计相关的行业，因此学习、掌握设计模式无疑是非常有益的。

注意：Java 是面向对象语言，很多新的技术领域都涉及 Java 语言，在国内外许多大学，Java 程序设计已经成为计算机相关专业一门专业基础课。因此，采用 Java 语言讲解 GoF 之书中的设计模式非常有利于将设计模式的内容作为 Java 程序设计的后继课程。

6.5 合理使用模式

不是软件的任何部分都需要套用模式来设计，必须针对具体问题合理地使用模式。

1. 正确使用

当设计某个系统，并确认所遇到的问题刚好适合使用某个模式时，就可以考虑将该模式应用到自己的系统设计中，毕竟该模式已经被公认为解决该问题的成功方案，能使设计的系统易维护、可扩展性强、复用性好，而且这些经典的模式也容易让其他开发人员了解你的系统和设计思想。

2. 避免教条

模式不是数学公式，也不是物理定律，更不是软件设计中的“法律”条文，一个模式只是成功解决某个特定问题的设计方案，完全可以修改模式中的部分结构以符合自己的设计要求。

3. 模式挖掘

模式不是用理论推导出来的，而是从真实世界的软件系统中被发现、按照一定规范总结出来的可以被复用的方案。目前，许多文献或书籍里阐述的众多模式实际上都是 GoF 之书中经典模式的变形，这些变形模式都经过所谓的“三次规则”，即该模式已经在真实世界的三个方案中被成功地采用。也可以从某个系统中寻找新模式，需要注意的是，在寻找新模式之前，必须先精通现有的模式，尤其是 GoF 之书中的 23 个模式，因为许多模式事实上只是现有模式的变种。通过研究现有的模式，不仅可以比较容易地识别模式，而且可以学会综合地使用各种模式，即使用复合模式。如果认为自己真的发现了一种新的模式，那么就可以按照 GoF 之书中提供的格式将“准模式”写成一份文档，按 6.1 节给出的模式定义，该文档需要包括名称、问题、方案和效果四个方面。当然，“准模式”需要经过“三次规则”才能成为真正的模式。

4. 避免乱用

不是所有的设计都需要使用模式，事实上，真实世界中的许多设计实例都没有使用过模式。在进行设计时，要尽可能用最简单的方法满足系统的要求，而不是费尽心机地琢磨如何在一个问题中使用模式。如果在设计中牵强地使用模式，会增加许多额外的类和对象，影响系统的性能。

5. 了解反模式

所谓反模式，是从某些软件系统中总结出的不好的设计方案。反模式就是告诉你如何采用一个不好的方案解决一个问题。既然是一个不好的方案，为何还有可能被重复使用呢？这是因为这些不好的方案表面上往往有很强的吸引力，人们很难一眼就发现它的弊端。因此，发现一个反模式也是非常有意义的工作。在有了一定的设计模式的基础之后，可以用搜索引擎查找有关反模式的信息，这对学习好设计模式也是非常有帮助的。

6.6 模式与框架

框架不是模式,框架是针对某个领域,提供用于开发应用系统的类的集合。程序设计者可以使用框架提供的类设计一个应用程序,而且在设计应用程序时可以针对特定的问题使用某个模式。

模式和框架的区别可以从以下几方面区分。

1. 层次不同

模式比框架更抽象,模式是在某种特定环境中,针对一个软件设计出现的问题给出的可复用的解决方案。模式不能向使用者提供可以直接使用的类,设计模式只有在被设计人员使用时才能表示为代码。例如,GoF 描述的中介者模式是“用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用,从而耦合松散,而且可以独立地改变它们之间的交互”。中介者模式在解决方案中并没有提供任何类的代码,只是说明设计者可以针对特定的问题使用该模式给出的方案。框架和模式不同,它不是一种可复用的设计方案,它是由解决某个问题的一些类组成的集合,程序设计人员通过使用框架提供的类或扩展框架提供的类进行应用程序的设计。例如,在 Java 中,开发人员使用 Swing 框架提供的类设计用户界面,使用 Set(集合)框架提供的类处理数据结构相关的算法等。

2. 应用范围不同

模式本质上是逻辑概念,以概念的形式存在,模式所描述的方案独立于编程语言。Java 程序员、C++程序员或 SmallTalk 程序员都可以在自己的系统设计中使用某个模式。框架应用的范围是很具体的,它们不是以概念的形式存在,而是以具体的软件组织形式存在,只能被特定的软件设计者使用。例如,Java 提供的 Swing 框架和集合框架只能被 Java 应用程序使用。

3. 相互关系

一个框架往往包含多个设计模式,它们是面向对象系统获得最大复用的方式。较大的面向对象应用会由多层彼此合作的框架组成,例如 Java Web 设计中的 Spring 和 Hibernate 等框架。框架变得越来越普遍和重要,导致许多开源框架的出现,而且一个著名的框架往往是许多设计模式的具体体现,甚至可以在一些成功的框架中挖掘出新的模式。

第 7 章　策略模式

以下文本框中的内容引自 GoF 所著 *Design Patterns*：*Elements of Reusable Object Oriented Software* 的中译本及英文版。

策略模式(别名：政策)

定义一系列算法，把它们一个个封装起来，并且使它们可相互替换。本模式使算法可独立于使用它的客户而变化。

Strategy Pattern(Another Name：Policy)

Define a family of algorithms, encapsulate each one, and make them inter changeable. Strategy lets the algorithm vary independently from clients that use it.

以上内容是 GoF 对策略模式的高度概括，结合 7.2.1 节的策略模式的类图可以准确地理解该模式。

7.1　概述

方法是类中最重要的组成部分，一个方法的方法体由一系列语句构成，也就是说，一个方法的方法体是一个算法。在某些设计中，一个类的设计人员经常可能涉及这样的问题：由于用户需求的变化，导致经常需要修改类中某个方法的方法体，即需要不断地改变算法。

例如，有一个 Army 类，该类中有 void lineUp(int a[])方法，其中，数组 a 的元素的值代表士兵的号码，该方法将士兵按他们的号码从小到大排队，即将数组 a 按升序排列。类图如图 7.1 所示。

那么，Army 类创建的对象，例如“三连长”，调用 lineUp()方法只能将自己管理的士兵按其号码从小到大排队，如图 7.2 所示。

图 7.1　Army 类

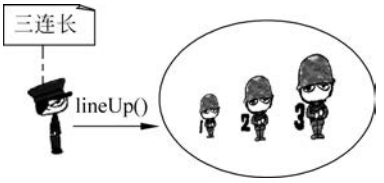

图 7.2　Army 类创建的对象调用 lineUp()方法

但有些部队希望 Army 创建的“三连长”能将士兵按照他们的号码从大到小排队，而不是从小到大排队，或将士兵按照他们的号码的某种排列来排队。显然，Army 无法提供这样的对象，如图 7.3 所示。

我们只好修改 lineUp()方法的方法体，但马上发现这样做也不行，因为一旦将 lineUp()

方法的方法体修改成把士兵按照他们的号码从大到小排队,就无法满足某些部队希望 Army 创建的“三连长”将自己的士兵从小到大排序的需求。也许可以在 lineUp()方法中添加多重条件语句,以便根据用户的具体需求决定怎样排队,但这也不是一个好办法,因为一旦有新的需求,就要修改 lineUp()方法添加新的判断语句,而且针对某个条件语句的排队代码也可能因该用户的需求变化导致重新编写。

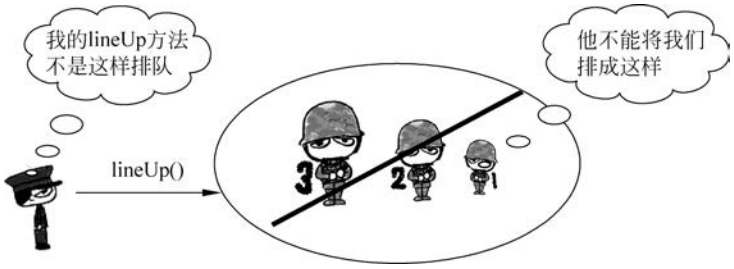

图 7.3　Army 对象无法满足新需求

如果因为需求的变化导致经常地修改 lineUp()方法体中的代码(具体算法),这显然不利于 Army 类的维护。不用担心,面向对象编程有一个很好的设计原则——“面向抽象编程”,该原则的核心是将类中经常需要变化的部分分割出来,并将每种可能的变化对应地交给抽象类的一个子类或实现接口的一个类去负责,从而让类的设计者不去关心具体实现,避免所设计的类依赖于具体的实现。基于该原则就可以使设计的类应对用户需求的变化。关于“面向抽象编程”曾在 4.1 节中讨论过,其关键有以下两点。

1. 分割变化

如果每当用户有新的需求,就会导致修改类的某部分代码,那么就应当将这部分代码从该类中分割出去,使它和类中其他稳定的代码之间是松耦合关系,即将每种可能的变化对应地交给实现某接口的类或某个抽象类的子类去负责完成。

现在,针对 Army 类中 lineUp()方法的方法体中的内容,抽象出一个“算法”标识,即一个抽象方法 abstract void arrange(int a[]),并将该抽象方法封装在一个接口或抽象类中。这里将接口命名为 LineUpStrategy,实现 LineUpStrategy 接口的类将实现接口中的 arrange(int a[])方法。例如 StrategyA 类实现 LineUpStrategy 接口,该类中的 arrange(int a[])方法把数组 a 的元素从小到大排列; StrategyB 类实现 LineUpStrategy 接口,该类中的 arrange(int a[])方法把数组 a 的元素从大到小排列; StrategyC 类实现 LineUpStrategy 接口,该类中的 arrange(int a[])方法把数组 a 的元素按奇、偶排列,并且奇数按降序,偶数按升序,例如,排列为 5,3,1,2,4,6。

2. 面向抽象设计类

现在,面向接口(抽象类)来重新设计 Army 类,让 Army 类依赖于 LineUpStrategy 接口,即 Army 类含有一个 LineUpStrategy 接口声明的变量 strategy,并重新编写 lineUp()方法的方法体中的代码,其主要代码是委托 Army 类中的 LineUpStrategy 接口变量 strategy 调用 arrange(int a[ ])方法。类图如图 7.4 所示。

如果准备让 Army 类创建的对象,例如“三连长”,调用 lineUp()方法将自己的士兵从小到大排队,那么在使用 Army 类创建“三连长”时,将一个 StrategyA 类的实例的引用传递给 Army 类中的 strategy 变量; 如果准备让 Army 类创建的对象,例如“三连长”,调用 lineUp()方法将自己的士兵从大到小排队,那么在使用 Army 类创建“三连长”时,将一个 StrategyB 类的实例的引用传递给 Army 类中的 strategy 变量,如图 7.5 所示。

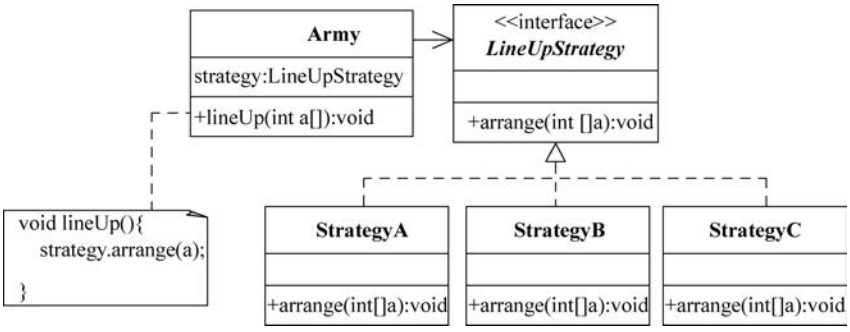

图 7.4 类图

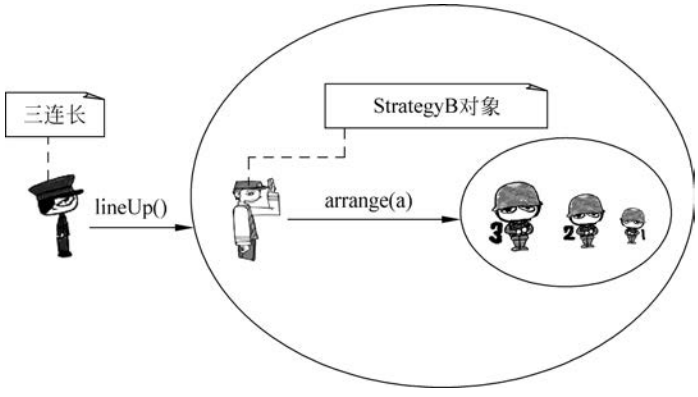

图 7.5 实现 Strategy 接口的类的实例负责排队

当用户有新的需求时，不需要更改 Army 类的代码，只需要给出一个新的实现 LineUpStrategy 接口的类即可，该类实现的 arrange(int a[])方法能满足新需求。

策略模式是处理算法不同变体的一种成熟模式，策略模式通过接口或抽象类封装算法的标识，即在接口中定义一个抽象方法，实现该接口的类将实现接口中的抽象方法。策略模式把针对一个算法标识的一系列具体算法分别封装在不同的类中，使得各个类给出的具体算法可以相互替换。在策略模式中，封装算法标识的接口称作策略，实现该接口的类称作具体策略。

扫一扫

视频讲解

7.2 模式的结构与使用

策略模式的结构中包括三种角色。

1. 策略(Strategy)

策略是一个接口，该接口定义若干个算法标识，即定义了若干个抽象方法。

2. 具体策略(ConcreteStrategy)

具体策略是实现策略接口的类。具体策略实现策略接口所定义的抽象方法，即给出算法标识的具体算法。

3. 上下文(Context)

上下文是依赖于策略接口的类，即上下文包含策略声明的变量。上下文中提供一个方法，该方法委托策略变量调用具体策略所实现的策略接口中的方法。