第3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有类(Class)的UML图、接口(Interface)的UML图以及泛化关系(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。

3.1 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3.1所示。

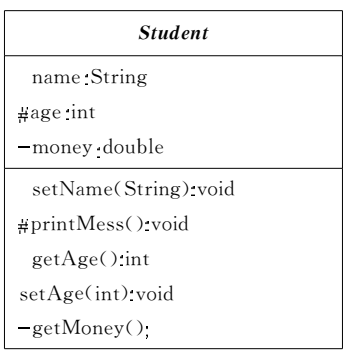

图3.1 类的UML图

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，方法的名字下面加上下划线。

3.2 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3.2所示。

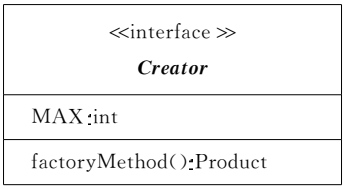

图 3.2　接口 UML 图

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 2 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。

3.3　泛 化 关 系

对于面向对象语言，UML 中所说的泛化关系就是指类的继承关系。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 3.3 所示。

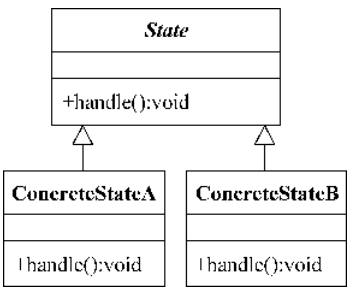

图 3.3　继承关系的 UML 图

3.4　关 联 关 系

如果 A 类中成员变量是用 B 类(接口)来声明的变量，那么 A 和 B 的关系是关联关系，称 A 关联于 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 3.4 所示。

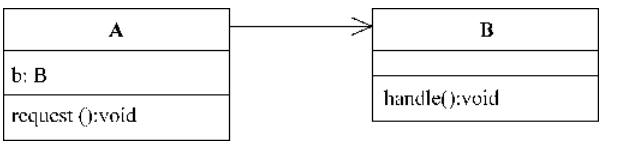

图 3.4　关联关系的 UML 图

3.5 依赖关系

如果 A 类中某个方法的参数是用 B 类(接口)来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 3.5 所示。

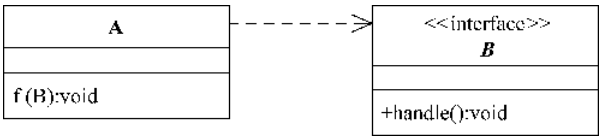

图 3.5 依赖关系的 UML 图

注:在 Java 中,习惯上将 A 关联于 B 也称作 A 依赖于 B,当需要强调 A 是通过方法参数依赖于 B 时,就在 UML 图中使用虚线连接 A 和 B 的 UML 图。

3.6 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,虚线的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 3.6 所示。

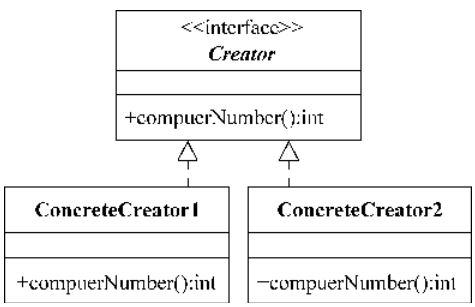

图 3.6 实现关系的 UML 图

3.7 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所它所注释的实体连接起来,如图 3.7 所示。

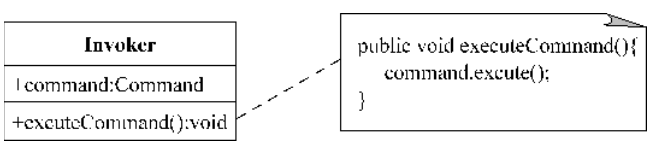

图 3.7 在类图中添加注释

第4章 命令模式

以下文本框中的内容引自 GOF 所著《Design Patterns》的中译本及英文版。

命令模式(别名：动作，事务)

将一个请求封装为一个对象，从而使用户可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。

Command Pattern(Another Name：Action，Transaction)

Encapsulate a request as an object，thereby letting you parameterize clients with different requests，queue or log requests，and support undoable operations.

上述文本框的内容是 GOF 对命令模式的高度概括，结合本章 4.2 节中命令模式的类图可以准确地理解该模式。

4.1 概 述

在许多设计中，经常涉及一个对象请求另一个对象调用其方法到达某种目的。如果请求者不希望或无法直接和被请求者打交道，即不希望或无法含有被请求者的引用，那么就可以使用命令模式。例如，在军队作战中，指挥官请求三连偷袭敌人，但是指挥官不希望或无法直接与三连取得联系，那么可以将该请求："三连偷袭敌人"形成一个"作战命令"，该作战命令的核心就是"三连偷袭敌人"。只要能让该"作战命令"被执行(即使指挥官已经不存在)，就会实现三连偷袭敌人的目的，如图 4.1 所示。

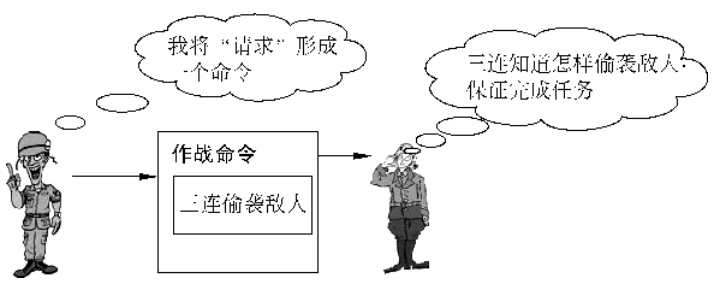

图 4.1 军官请求三连偷袭敌人

命令模式是关于怎样处理一个对象请求另一个对象调用其方法完成某项任务的一种成熟的模式，这里称提出请求的对象为请求者，被请求的对象为接收者。在命令模式中，当一个对象请求另一个对象调用其方法时，不和被请求的对象直接打交道，而是把这种"请求"封

装到一个称作“命令”的对象中，其封装手段是将“请求”封装在“命令”对象的一个方法中。命令模式的核心就是使用命令对象来封装方法调用，即将请求者的请求：“接收者调用方法”封装到命令对象的一个方法中，这样一来，当一个对象请求另一个对象调用方法来完成某项任务时，只需和命令对象打交道，即让命令对象调用封装了“请求”的那个方法即可，如图 4.2 所示。我们将在 4.2 节给出命令模式的类图，并详细阐述该模式中各个角色的职责。

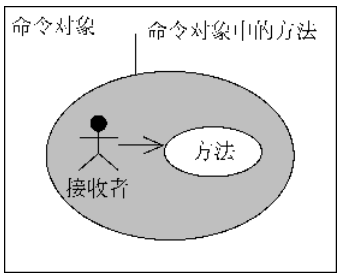

图 4.2　用命令对象封装方法调用

4.2　模式的结构与使用

命令模式的结构中包括四种角色。

- 接收者(Receiver)：接收者是一个类的实例，该实例负责执行与请求相关的操作。
- 命令(Command)接口：命令是一个接口，规定了用来封装“请求”的若干个方法，比如，execute()、undo()等方法。
- 具体命令(ConcreteCommand)：具体命令是实现命令接口的类的实例。具体命令必须实现命令接口中的方法，比如 execute()方法，使该方法封装一个“请求”。
- 请求者(Invoker)。请求者是一个包含 Command 接口变量的类的实例。请求者中的 Command 接口的变量可以存放任何具体命令的引用。请求者负责调用具体命令，让具体命令执行那些封装了“请求”的方法，比如 execute()方法。

4.2.1　命令模式的 UML 类图

命令模式的类图如图 4.3 所示。

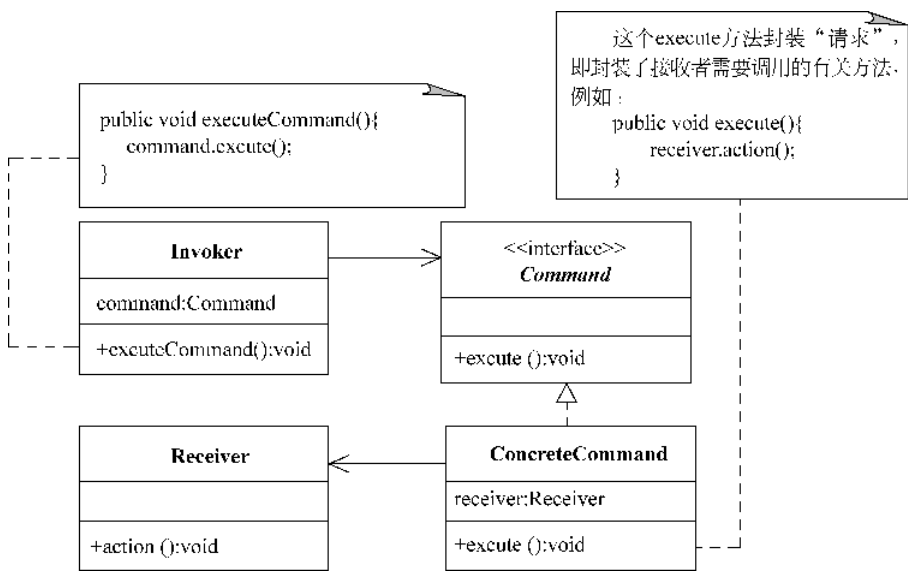

图 4.3　命令模式的类图

在 4.1 节中提到的“指挥官”请求“三连”偷袭敌人的问题中，指挥官就是命令模式中的 Invoke 角色，指挥官制定的作战命令就是命令模式中的 ConcreteCommand 角色，三连就是命令模式中的 Receiver 角色，示意图如图 4.4 所示。

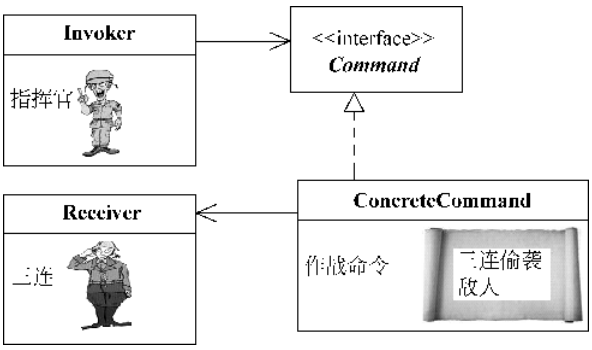

图 4.4　指挥官下达命令

4.2.2　结构的描述

下面通过一个简单的问题来描述命令模式中所涉及的各个角色，这个简单的问题是：一个指挥官请求（命令）三连偷袭敌人。

1. 接收者（Receiver）

对于上述简单的问题，接收者是下列 CompanyArmy 类的一个实例，该实例的 sneakAttack()方法可以实现如何偷袭敌人，即 CompanyArmy 类的实例知道如何偷袭敌人。CompanyArmy 类的代码见如下的 CompanyArmy.java。

CompanyArmy.java

```
public class CompanyArmy{
    public void sneakAttack(){
      System.out.println("我们知道如何偷袭敌人，保证完成任务");
    }
}
```

2. 命令（Command）接口

命令接口中规定了用来封装“请求”的方法，这里所给出的 Command 接口的代码如下：

Command.java

```
public interface Command {
    public abstract void execute();
}
```

3. 具体命令（ConcreteCommand）

指挥官请求 CompanyArmy 的实例，比如三连，调用 sneakAttack()方法偷袭敌人，但指挥官不想或无法直接与 CompanyArmy 的实例打交道，即指挥官不希望或无法含有 CompanyArmy 的实例的引用，在这种情况下，可以把指挥官的请求：“CompanyArmy 的实例调用 sneakAttack()”封装到一个具体命令对象的 execute()方法中。创建具体命令对象的 ConcreteCommand 类将会含有 CompanyArmy 类的实例引用，ConcreteCommand 类的

代码见下面的 ConcreteCommand. java。

ConcreteCommand. java

```
public class ConcreteCommand implements Command{
    CompanyArmy army;                    //含有接收者的引用
    ConcreteCommand(CompanyArmy army){
        this. army = army;
    }
    public void execute(){               //封装着指挥官的请求
        army. sneakAttack();             //偷袭敌人
    }
}
```

4. 请求者(Invoker)

请求者含有 Command 的接口声明的变量,比如 command,用来存放具体命令的引用。command 可以回调具体命令实现 Command 的接口的方法,即请求者通过执行具体命令,实现自己的"请求"。

在上述简单问题中,指挥官就是一个请求者,创建指挥官的 ArmySuperior 类的代码见如下的 ArmySuperior. java。

ArmySuperior. java

```
public class ArmySuperior{
  Command command;                       //用来存放具体命令的引用
  public void setCommand(Command command){
    this. command = command;
  }
  public void startExecuteCommand(){     //让具体命令执行 execute()方法
    command. execute();
  }
}
```

4.2.3 模式的使用

前面已经使用命令模式给出了可以使用的类,这些类就是一个小框架,可以使用这个小框架中的类编写应用程序。

下列应用程序中,Application. java 使用了命令模式中所涉及的类。应用程序在使用命令模式时,需要为具体命令指定接收者。Application. java 演示了一个指挥官请求三连使用偷袭方法进攻敌人,程序的运行结果是:"我们知道如何偷袭敌人,保证完成任务"。

Application. java

```
public class Application{
    public static void main(String args[]){
        CompanyArmy 三连 = new CompanyArmy();              //创建接收者
        Command command = new ConcreteCommand(三连);      //创建具体命令并指定接收者
        ArmySuperior 指挥官 = new ArmySuperior();          //创建请求者
        指挥官. setCommand(command);
        指挥官. startExecuteCommand();
    }
}
```

4.3 命令接口中的撤销方法

命令接口中规定了用来封装“请求”的方法，比如，execute()的方法。命令接口还可以提供用来封装撤销“请求”的方法，比如 undo()方法，即 undo()方法的执行能撤销 execute()方法的执行效果。如果 execute()方法的执行效果不可撤销(比如，退出程序等)，那么具体命令就不必实现 undo()方法。

以下使用一个简单的问题说明怎样在具体命令中实现 undo()方法，问题如下：

请求者请求在硬盘建立目录，请求成功后，还可以撤销请求。这就要求接收者不仅可以在硬盘建立目录，也可删除上一次请求所建立的目录。针对该问题所设计的类图如图 4.5 所示。

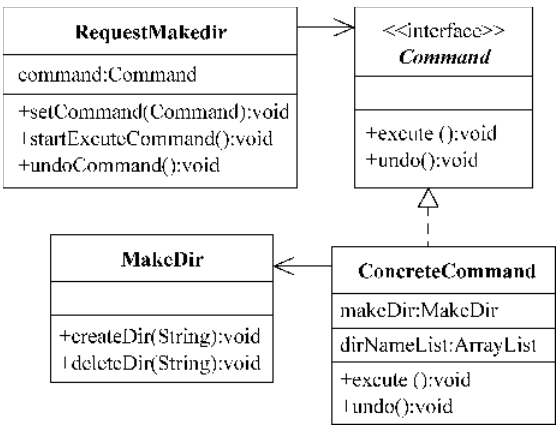

图 4.5 设计的类图

1. 接收者

对于上述问题，接收者角色是下列 MakeDir 类的一个实例，该实例的 createDir(String)方法可以创建目录、deleteDir(String)方法可以删除目录。MakeDir 类的代码见如下的 MakeDir.java。

MakeDir.java

```
import java.io.*;
public class MakeDir{
    public void createDir(String name){
        File dir = new File(name);
        dir.mkdir();
    }
    public void deleteDir(String name){
        File dir = new File(name);
        dir.delete();
    }
}
```

2. 命令接口

命令接口的代码见如下的 Command.java。

Command. java

```
public interface Command {
    public abstract void execute(String name);
    public abstract void undo();
}
```

3. 具体命令

具体命令通过使用 execute()方法封装了请求者的请求，undo()方法可以撤销 execute()方法执行所产生的效果。ConcreteCommand 类的代码见如下的 ConcreteCommand. java。

ConcreteCommand. java

```
import java.util. * ;
public class ConcreteCommand implements Command{
    ArrayList<String> dirNameList;
    MakeDir makeDir;
    ConcreteCommand(MakeDir makeDir){
        dirNameList = new ArrayList<String>();
        this.makeDir = makeDir;
    }
    public void execute(String name){
        makeDir.createDir(name);
        dirNameList.add(name);
    }
    public void undo(){
        if(dirNameList.size()>0){
          int m = dirNameList.size();
          String str = dirNameList.get(m - 1);
          makeDir.deleteDir(str);
          dirNameList.remove(m - 1);
        }
        else
          System.out.println("没有需要撤销的操作");
    }
}
```

4. 请求者

创建请求者的 RequestMakedir 类的代码见如下的 RequestMakedir. java。

RequestMakedir. java

```
public class RequestMakedir{
    Command command;
    public void setCommand(Command command){
        this.command = command;
    }
    public void startExecuteCommand(String name){
        command.execute(name);
    }
    public void undoCommand(){
        command.undo();
    }
}
```

5. 应用程序

下列应用程序中，Application. java 使用了命令模式中所涉及的类。请求者一共提出三次请求(执行三次命令)，每次请求的目的是在当前目录建立一个新的子目录。请求者的第1次请求是：在当前目录下建立名字是 jiafei 的子目录。第2次请求是：当前目录下建立名字是 jerry 的目录。第3次请求是：当前目录下建立名字是 tom 的子目录，紧接着，请求者取消了后两次请求(取消了后两次命令的执行效果)。因此，Application 程序的运行效果是：只在当前目录下新建立了一个名字是 jiafei 的子目录。

Application. java

```
public class Application{
    public static void main(String args[]){
        MakeDir makeDir = new MakeDir();                    //创建接收者
        Command command = new ConcreteCommand(makeDir);     //创建具体命令并指定接收者
        RequestMakedir askMakedir = new RequestMakedir();
        askMakedir.setCommand(command);
        askMakedir.startExecuteCommand("jiafei");           //建立名字是 jiafei 的目录
        askMakedir.startExecuteCommand("jerry");            //建立名字是 jerry 的目录
        askMakedir.startExecuteCommand("tom");              //建立名字是 tom 的目录
        askMakedir.undoCommand();                           //删除名字是 tom 的目录
        askMakedir.undoCommand();                           //删除名字是 jerry 的目录
    }
}
```

4.4 宏 命 令

宏命令也是一个具体命令，只不过它包含了其他具体命令的引用。当一个宏命令执行其 execute()方法时，将导致所引用的其他具体命令执行其 execute()方法。因此，执行一个宏命令相当于执行了许多具体命令。

以下通过一个简单的问题，说明宏命令的用法。

请求者可以请求只输出英文字母表、俄文字母表或 1～n 之间的偶数，也可以请求既输出英文字母表，又输出俄文字母表以及 1～n 之间的偶数。

1. 接收者

对于上述问题，我们设计了两个接收者，一个是 PrintLetter 类的一个实例，该实例的 printEnglish()方法可以输出英文字母表、printRussian()方法可以输出俄文字母表；另一个接收者是 PrintNumber 类的实例，该实例的 printEvenNumber()方法可以输出偶数。

PrintLetter 和 PrintNumber 类的代码见如下的 PrintLetter. java 和 PrintNumber. java。

PrintLetter. java

```
public class PrintLetter{
    public void printEnglish(){
        for(char c = 'a';c<= 'z';c++)
          System.out.print(" " + c);
    }
    public void printRussian(){
        for(char c = 'а';c<= 'я';c++)
```

```
            System.out.print(" " + c);
        }
    }
```

PrintNumber. java

```
public class PrintNumber{
    int n;
    PrintNumber(int n){
        this.n = n;
    }
    public void printEvenNumber(){
        for(int m = 1;m<= n;m++)
          if(m % 2 == 0)
            System.out.print(" " + m);
    }
}
```

2. 命令接口

命令接口的代码见如下的 Command. java。

Command. java

```
public interface Command {
    public abstract void execute();
}
```

3. 具体命令

一共有四个具体命令：PrintEnglishCommand、PrintRussianCommand、PrintEvenNumberCommand 三个具体命令以及一个宏命令 MacroCommand。代码如下：

PrintEnglishCommand. java

```
public class PrintEnglishCommand implements Command{
    PrintLetter letter;
    PintEnglishCommand(PrintLetter letter){
        this.letter = letter;
    }
    public void execute(){
        letter.printEnglish();
    }
}
```

PrintRussianCommand. java

```
public class PrintRussianCommand implements Command{
    PrintLetter letter;
    PrintRussianCommand(PrintLetter letter){
        this.letter = letter;
    }
    public void execute(){
        letter.printRussian();
    }
}
```

PrintEvenNumberCommand. java

```
class PrintEvenNumberCommand implements Command{
```