计算机科学1997Vol.24№.6

OMT设计的Java实现

An Approach of Implementing OMT Design with Java

邵晓凌 张 然

(复旦大学计算机科学系 上海200433)

摘 要 The OMT methodology supports the entire software life cycle, developing software using OMT is a seamless development process. Meanwhile, as the latest Internet-oriented programming language, Java has advantage over other programming languages in having features such as object-oriented, platform-independent, supporting network communications. It's a good idea to implement OMT design with Java. In this paper, we focus on how to implement OMT concepts such as class, inheritance and association with Java, and discuss the possibility of automatic conversion OMT design to Java statements.

关键词 Object Modeling Technique, Java language, Object-Oriented

1. 引言

OMT作为一种面向对象的软件方法学,已经成为面向对象技术的主要流派之一,它支持软件生命周期的各个阶段,能实现系统的无缝开发。目前,已经有人从理论上探讨了用C++、SmallTalk等面向对象语言实现OMT设计,并且已经有了相关的商业产品。如TROy Software的OPEA,能将OMT的类的规格说明直接转换成C++代码。然而,对于目前炙手可热的Java语言,无论是理论上还是实践上,还没有人探讨过如何用它来实现OMT设计。对此,我们将探讨这一问题。

2. OMT简介

对象建模技术(OMT)是一种围绕真实世界的概念来组织模型的软件开发方法。OMT方法学使用三种模型来描述系统:对象模型、动态模型和功能模型。对象模型描述了系统中对象的静态结构以及对象间的联系,用对象模型图来描述。对象模型图是ER图的一种拓广形式,图1是一个简化了的ATM对象模型图。动态模型描述了与时间和操作次序有关的系统属性。动态模型由多张状态图组成。各个类的状态图通过共享事件组成系统的动态模型。功能模型描述系统内数据值的变化,它由数据流图组成。流图说明数据流是如何从外部输入,经过操作和内部存储而到外部输出的。OMT的三种模型相辅相成地组成系统的一个完整的正交视图。

相对于传统的软件工程方法学,OMT的开发重心移到了分析阶段,使得分析的结果比一般的软件开发方法更为可靠,减少因分析不透彻而引起的问题。OMT支持系统的无缝开发。在整个开发过程中使用统一的软件概念即对象,所有其它概念都是围绕对象组成的,在分析阶段开发的对象模型也适用于设计和实现阶段。这样,软件开发的阶段性就不那么明显了。由于各阶段是一致吻合的,很容易实现各阶段的反复,而且每一次反复都是对系统的进一步细化。

3. Java语言

Java作为一种面向对象语言,从诞生到如今,不过五、六年的时间。但是,Java所取得的如此佼佼战绩是其它任何一种面向对象语言所无法比拟的。目前,Java已经成为Internet上通用的编程语言。Java之所以如此重要,并适合于开发Internet上的应用软件,原因是多方面的。主要体现在以下几个方面:

面向对象 作为面向对象程序设计语言,它支持封装、多态性、继承和动态联编等概念,能有效地支持代码的重用,减少程序员的工作量,提高代码的开发速度,缩短软件开发的周期。用面向对象方法开发的软件还具有容易维护的优点。作为一种面向对象的程序设计语言,Java吸收了Eiffel、SmallTalk和C++等面向对象语言的优点,并减少了其它面向

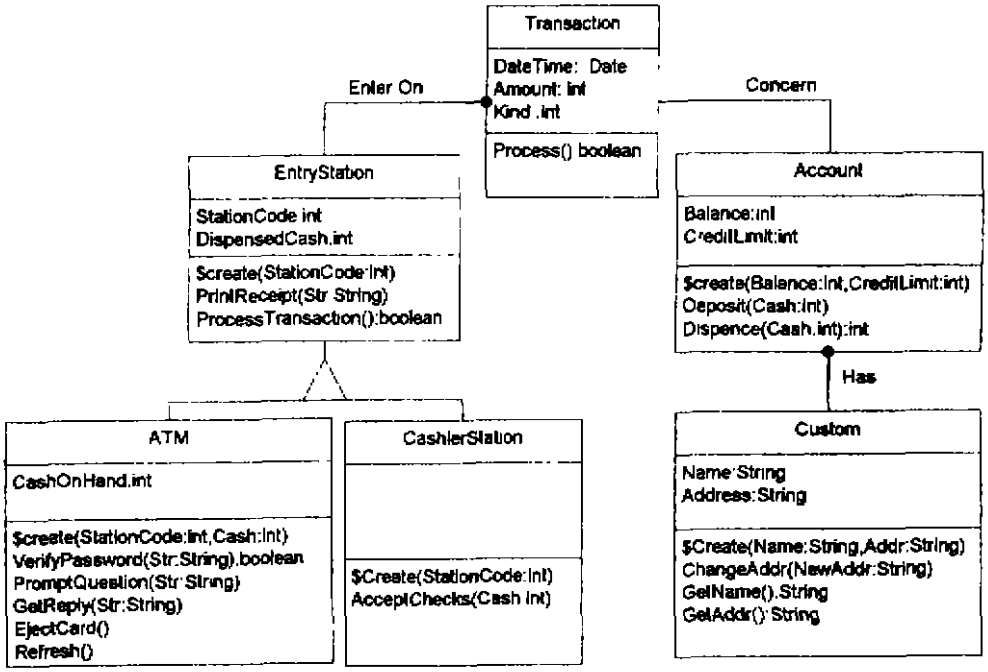

图1

对象语言的冗余度和一些非面向对象的特性，如C++中有全局变量和公用模块的概念，这与面向对象的风格是很不相容的，而在Java中，只有类的概念，无全局变量和公用模块。

简单性　这是Java区别于已有的面向对象语言的很重要的一点。在一般的面向对象语言如C++中，存在指针这一数据结构。指针是C++中最难掌握的概念，虽然能给程序员提供很大的灵活性，但实践也证明，指针是程序中错误的主要来源之一，因而也是影响软件开发效率的一大因素。

舍弃了多重继承的概念。多重继承是增加系统复杂性的另一个重要因素。一个类继承了多个父类的属性，容易引起重名冲突，增加调试的难度。Java借鉴Object C中的协议，引入接口这一概念以便部分地解决因缺少多重继承而给程序员带来的不便。一个接口定义了一个或多个类要实现的方法，但是并没有实现这些方法，需要继承接口的类自己去实现。接口允许多重继承。接口使得多重继承的设计与实现分离开来。

Java内置一个自动垃圾收集器。在C++中，用户申请的空间必须由程序员显式地释放，否则会引起系统资源的浪费，甚至导致系统故障。而在Java中，内存的释放由垃圾收集器，作为一个独立的线程在后台自动执行，由于它具有较低的优先级，一般仅当无其它进程运行时才能运行，因而不会影响系统运行的效率。Java还支持内存碎片的收集，提高了系统资源的利用率。

独立于平台　Java程序在不同平台间是保持二进制兼容的，即任何Java程序不用重新编译，就能在不同类型的机器上运行。这归功于Java的解释执行和字节码(Bytecode)技术。Java程序先由伪代码编译器将源程序"编译"成字节码，然后再由Java解释器解释执行，是一种"半编译半解释"的方法。

安全性　Java环境采取了"不信任任何人"的态度，从编译到执行进行了多道安全性检查。首先，Java的内存分配及引用模型别具一格。Java是在运行时刻才决定如何给数据结构分配物理内存的，并受软件环境和硬件平台的影响，加上Java排斥了指针的使用，使程序员无法对实际内存进行操作，从而排斥了一些非法操作的可能性，经编译产生的代码在被解释执行前，还需要经过检验，以验证代码的格式是正确的、程序没有"伪造"指针和对对象的访问没有违反Java的4P规则(Public，Package，Protected & Private)。Java的字节码载入器也进行了安全性控制。Java程序运行时，可能需要载入位于网络上其它节点的一些类。当这些类被载入后，是被"分离"开来

执行的。Java 为所有位于本机的类创建一个名空间，而为其它非本机的类按它们原来所在节点的地址，各产生一个名空间。当不同名空间的类互相访问时，要受到限制，从而可以防止诸如从网上下载的软件攻击本机系统等现象的发生。

分布性　Java 有专门的类库支持网络通信。在 Java 的标准类库中，包 java.net 提供了专门的类支持网络协议。Java 程序经由 URL 操作网络资源就如同 C 中的文件操作一样的方便。

目前，使用 Java 开发的应用程序已经大量出现，甚至有公司推出基于 Java 的网络计算机。虽然目前 Java 的运行效率比不上 C++，但是随着硬件性能的提高，这方面的因素将显得微不足道。而 C++由于缺乏多线程和不支持网络通信等，将难以适应未来的需要，因而我们可以毫不夸张地预测，Java 将取代 C++成为面向对象程序设计的主流。

4. 具体实现

OMT 从分析、设计到实现等各个步骤支持软件系统的无缝开发，目前已经有人探讨了对 OMT 的设计结果用一些面向对象的程序设计语言如 C++来实现的方法，但还没有人尝试过用 Java 来实现 OMT 设计。在这一节，我们要说明的是如何利用 Java 语言来实现 OMT 设计。

显然，用面向对象的语言 Java 来实现 OMT 是一种理想的选择，它具有面向对象的绝大部分概念。OMT 设计的结果大部分可以简单地映射到 Java 语言上去。一些概念，如多重继承技术，虽然在 Java 中没有对应的概念，但是我们可以通过某种变换来解决这个问题。

类的定义　用 Java 实现 OMT 设计的第一步，就是实现类的定义。

例如对于图1中的类 Account，我们可以用以下代码来实现：

```
//文件名，Account.java
//在 Java 中，每一个文件仅包含一个公有类型类的定义
//文件必须与类同名
    public class Account
    {
      private static int DefaultLimit=0;
        //CreditLimit 的缺省值；
      private int Balance;
      private int CreditLimit;
      Account(int cash)
      {
        Balance=Cash;
        CreditLimit=DefaultLimit;
      }
      Account(int cash,int limit)
      {
        Balance=Cash;
        CreditLimit=limit;
      }
      protected void finalize()
      {
      }
      public void Deposit(int cash)
      {
        Balance=Balance+cash;
      }
      public int Dispence(int cash)
      {
        int preBalance=Balance;
        Balance=Balance-cash>CreditLimit?Balance-
                cash:CreditLimit;
        return preBalance-Balance;
      }
    }
```

在 Java 中，类由属性和行为组成。属性描述的是对象的“状态”。它分两类，一类是实例变量，描述每一个类的实例即对象的属性，如类 Account 的属性 Balance；另一类为类变量，描述的是类的属性，即它对于类的任一个实例都适用，而不是仅属于某一个对象，如 Account 中的 DefaultLimit。它们在用 Java 实现时，类变量要比实例变量多一个 static 修饰符。即如果类的定义中，变量前存在 static 修饰符，则是类变量，否则为实例变量。属性和行为可以同名，此时的行为表示对同名属性的访问。

Java 对于属性和行为的访问限制可以概括为 4P's。公有类型(Public)的属性和行为，对于任何别的类都是可见的，如我们可以使用如 CN. EDU. FUDAN. cs. projects. se. MyPackage. MyClass. MyMethod 来调用位于复旦计算机系的名为 projects. se. MyPackage. MyClass 的公有方法 MyMethod。

由于 Java 是以类为基本单位的，具有相同功能或实现某一功能的不同类组成一个包(Package)，如 java.io，它囊括了 Java 中基本的输入输出类，因而在 Java 中，特意为这种情况定义了一类访问控制，它的修饰符为 package(也是 Java 的缺省设置)。被限定为 package 的属性和行为对于包中其它类是可见的，而对于不属于同一个包的类是不可见的。

保护类型(Protected)定义了父、子类之间的关系。父类的保护类型的属性和行为对于子类是可见的，其它类则无法访问。

最严格的访问控制为私有类型(Private)。对于定义为私有类型的属性和行为，只对对象本身可见，在别的类中无法引用它们。

对象的创建　对于同一个类，我们可以建立几个不同的构造函数。每当创建一个对象时，就执行其中的一个构造函数，具体选择哪一个构造函数，则由调用时参数的个数和类型来决定。

如，我们可以用如下方式创建一个类 Account 的实例：Account MyAccount = new Account(someCash)；或 Account MyAccount = new Account (someCash，Limit)；操作符 new 的作用是产生一个新实例，为实例分配内存，并以指定参数调用 Account 的构造函数。

在定义了一个对象后，我们可以在不违反访问控制的前提下，使用对象的属性和行为，如 MyAccount. Deposit(Money)。

相对于构造函数，类还有一个特殊的行为即析构函数。在 Java 中，它的原型固定为 protected void finalize()。一般的面向对象程序设计语言的析构函数的工作是释放申请的空间，这是因为在如 C++等语言中没有实现垃圾收集器，需要由程序员完成已申请空间的释放工作。而在 Java 中，空间的释放由系统自动完成，不必显式地在程序中释放，因而 Java 的析构函数都很简单，或根本不必重载缺省的析构函数。

继承 由于 Java 规定，变量的类型必须是静态确定的，即在编译时刻就能确定变量的类型，因而类之间的继承关系在编译前也是确定的。对于单重继承，在 Java 里实现很简单。例如，在图1中，类 EntryStation 是类 ATM 和类 CashierStation 的父类，类 ATM 和类 CashierStation 继承了 EntryStation 的属性和行为。下面的程序是类 ATM 的 Java 实现：

```
public class ATM extends EntryStation
{
  ……//类 ATM 的定义
}
```

通过继承，类 ATM 可以继承类 EntryStation 的属性和行为，且不必重写相应的代码，从而有利于实现代码的重用，提高软件开发的效率。

Java 的设计者们认为，多重继承虽然能在一定程度上提高软件重用的可能，但是由于同一个类可以继承不同父类的属性和行为，当子类的两个或多个父类的属性或行为同名时，就会引起冲突。这样增加了软件中潜在的错误的可能性，可能降低软件开发的效率。

然而，OMT 是支持多重继承的，如果我们通过 OMT 设计得到的系统中存在多重继承，那么就需要转义。通过对对象模型进行重构，将多重继承转化为单重继承。转义的主要方法有以下几种：

·嵌套的一般化。逐项分解各个一般化，通过这种方法可以获得所有可能的组合对象。缺点是产生了代码的冗余，破坏了面向对象程序设计的风格。

·继承最主要的类，委派其它类。这种方法通过一般化关系既保留了操作的标识，也保留了继承。

·使用角色聚集的委派。具有多个独立的一般化关系的超类再构成一个组合对象，在这个组合对象中，各个组元取代了各个一般化。

限于篇幅，在此不一一细述，详见文[1]。

关联 在 Java 中，并不显式地支持联系对象，因而我们可以简单地用引用来实现关联。

对于单向联系，只需在其中的一个类中增加一个属性来表示联系。而对于二元联系，则需在相应的对象中各增加一个属性来实现，其中的属性含指向相关对象或相关对象集合的引用。对于如图2所示的类 Item 和类 Group 之间的一对多关系，如果用引用来表示，我们需要在 Item 和 Group 中各增加一个属性，用它们来模拟 Item 和 Group 之间的联系。

图2

但是使用引用有一个致命的缺点，是它破坏了类的封装性。一个联系反映的是两个类之间的关系，如果用两个引用来实现联系，那么对其中的一个引用的修改意味着对另一个引用也要修改，这样使得两个类的属性之间存在依赖关系，这与 OO 风格是不相容的。

为了保持类的封装性，实现关联的最好的方法是定义一个关联类。每一个联系都是关联类的实例。在下面，我们给出了关联类的一种实现方法。其中包含了两个字典对象(在 Java 中用散列表实现)，一个用于正向查找，另一个用于逆向查找，由此来提高遍历关联的效率。需要说明的是，我们在此用 Java 的 vector 类来实现集合。

```
//文件名：Association. java
    import java. util. Hashtable;
    import java. util. Vector;
    public Association
    {
      private Hashtable Forward;//用于正向查找
      private Hashtable Reverse;//用于逆向查找
      Association()
      {
        Forward=new Hashtable();
        Reverse=new Hashtable();
      }
      public void Add(Item item,Group group)
      {
        Vector itemset;//临时变量
          if(Forward. get(item)!=null)//若 item 已存
                                        在，则先删去
        Remove(item,(Group)Forward. get(item));
        Forward. put(item,group);
        itemset=Reverse. get(group)
        if(itemset!=null)
          itemset. addElement(item);
        else
```

```
        {
          itemset=new Vector();
          itemset.addElement(item);
          Reverse.put(group,itemset);
        }
      }
      public void Remove(Item item,Group group)
      {
        Forward.remove(item);
        Vector itemset=(Vector)Reverse.get(group);
        itemset.removeElement(item);
        if(itemset.isEmpty())//若删除 item 后 itemset
                                              为空,
          Reverse.remove(group);//则删除 itemset 与
                                    group 之间的联系
      }
      public Group IndexForward(Item item)
      {
        return(Group)Forward.get(item);
      }
      public Vector IndexReverse(Group group)
      {
        return(Vector)Reverse.get(group);
      }
}
```

这样对 Group 中增加、删除元素改为对 Association 类中两个散列表的操作。引入关联类后有利于提高封装性，更加符合 OO 设计的风格。

用 Association 类实现的 Item 类和 Group 类如下：

```
//文件名:Item.java
  public class Item
  {//other declaration as before
    public static Association item_group_asn;
    public Group GetGroup()
    {
      return item_group_asn.IndexForward(this);
    }
  }
//文件名:Group.java
  import java.util.Vector;
  public class Group extends Item
  {//other declarations as before
    public void AddItem(Item item)
    {
      item_group_asn.Add(item,this);
    }
    public void RemoveItem(Item item)
    {
      item_group_asn.Remove(item,this);
    }
    public Vector GetItems()
    {
      return item_group_asn.IndexReverse(this);
    }
  }
//In other place
  item_group_asn=new Association();
```

关于自动转换的思考 因为 OMT 中的类、属性和行为等在 Java 中有直接相对应的概念，因而对于用 OMT 工具得到的设计结果中的这些概念，我们能够按照某种规则把它们映射到 Java 语言上。至于关联，用我们在上面讨论过的方法就能很容易地变换过去。其它如聚集等在 Java 中也可以用相关的数据结构来实现。唯一可能的难题是多重继承，由于一般的设计结果都包含多重继承，而 Java 又不支持多重继承，因而实现起来有些麻烦。可以使用前面讨论过的转义的方法来解决。而转义的后两种方法涉及到对具有继承关系的类之间的关系进行分析，这又与具体的问题有关，因而实现自动转换难度较大，一种方法是构造出各种组合的对象，虽然这样做与面向对象程序设计的风格相悖。

（下转第9页）