Java中的Abstract Class与Interface技术研究

曹步清[1,2],金 瓯[1]
(1.中南大学 信息科学与工程学院,湖南 长沙 410083;
2.湖南科技大学 计算机科学与工程学院,湖南 湘潭 411201)

摘 要:Abstract class和Interface是Java语言中对于抽象类定义进行支持的两种机制,正是由于这两种机制的存在,才赋予了Java强大的面向对象能力。Abstract class和Interface之间在对于抽象类定义的支持方面具有很大的相似性,因此很多开发者在进行抽象类定义时对于Abstract class和Interface的选择显得比较随意。但是,两者之间的区别还是很大的。文中从两者的定义、程序编辑、设计意图3个方面加以了比较。由此,对于它们的选择,反映出对于问题领域本质的理解和对于设计意图的理解是否正确、合理。

关键词:抽象类;接口;面向对象
中图分类号:TP311 **文献标识码**:A **文章编号**:1673-629X(2006)08-0110-03

Research of Abstract Class and Interface of Java

CAO Bu-qing[1,2],JIN Ou[1]
(1.School of Information Science and Engineering,Central South University,Changsha 410083,China;
2.Coll.of Computer Sci.and Eng.,Hunan Univ. of Sci.and Techn.,Xiangtan 411201,China)

Abstract:Two mechanisms of abstract class and interface are supported for the abstract class definition of Java language. Being of two mechanisms endow the powerful ability of OO. There have great similarities that two mechanisms of abstractclass and interface are support for the abstractclass definition, therefore,the choice is random. But the distinction of the two is great. The distinction was made from the two definitions,procedures editorial,design intentation in this article. So,it is the key that how to choose them to make the designed-intention right and reasonable when the system is designed.

Key words:abstract class;interface;OO

0 引 言

Abstract class和Interface是Java语言中对于抽象类定义进行支持的两种机制,正是由于这两种机制的存在,才赋予了Java强大的面向对象能力。Abstract class和Interface之间在对于抽象类定义的支持方面具有很大的相似性,提供了同样的方法,甚至可以相互替换,因此很多开发者在进行抽象类定义时对于Abstract class和Interface的选择显得比较随意。一般情况下,和抽象类比较起来,选择接口的机会更多一些,因为接口可以把设计和实现相分离。因此,它可以匹配所有满足该抽象的实现,而且,接口成功地实现了方法分离。一个简单的接口例子如下:

```
Public interface Vehicle{
  String forward_direction="up";
  String behind-direction="down";
  String left_direction="left";
  String right direction="right";
  Void turnleft();
  Void turnright();
  String getcurrentDirection();
}
```

该接口定义了3个操作,这3个操作是向车辆驾驶发送驾驶命令的类所关心的问题。在此接口中没有涉及特殊车辆的任何细节,也没有涉及对该驾驶命令如何进行反映的问题。某一车辆希望以这种方式控制驾驶,则该车辆必须为定义的这些方法提供特定的实现机制。

总之,接口使你只需要关心所要完成的行为,而不需要关心实现的细节[1]。

在面向对象的概念中,所有的对象都是通过类来描绘的,但是反过来却不是这样。并不是所有的类都是用来描绘对象的,如果一个类中没有包含足够的信息来描绘一个具体的对象,这样的类就是抽象类。抽象类往往用来表征在对问题领域进行分析、设计中得出的抽象概念,是对一系列看上去不同,但是本质上相同的具体概念的抽象。例如:如果进行一个图形编辑软件的开发,就会发现问题领域存在着长方形、三角形等这样一些具体概念,它们是不

收稿日期:2005-11-29
作者简介:曹步清(1979-),男,湖南湘潭人,硕士研究生,研究方向为网络、数据库与信息处理;金 瓯,教授,研究方向为通信工程、信息处理、金融货币识别、系统集成。

同的,但是它们又都属于形状这样一个概念。形状这个概念在问题领域是不存在的,它就是一个抽象概念。

1　抽象类与接口的区别及选择

抽象类和接口之间在对于抽象类定义的支持方面具有很大的相似性,提供了同样的方法,甚至可以相互替换。但是,两者之间的区别还是很大的。由此,对于它们的选择,反映出对于问题领域本质的理解和对于设计意图的理解是否正确、合理。

1.1　从两者的定义方面看

抽象类、抽象方法必须用 Abstract 关键字来修饰。抽象类一般指问题领域的抽象概念,是对一系列看上去不同,但是本质上相同的具体概念的抽象。接口是用 Interface 关键字修饰,接口不能定义其方法的任何实现,接口的域总是定义为 public,static 和 final。它说明了所设计的系统是干什么的,但没有说明任务是如何完成的,示例如下。

1)抽象类的方式定义 sellable 抽象类如下:

```
abstract class sellable{
  abstract void getDescription();
  abstract void getunits();
  abstract void getPricePerUnit();
  ……
}
```

2)接口的方式定义 sellable 抽象类的方式如下:

```
interface sellable {
  void getDescription();
  void getunits();
  void getPricePerUnit();
  ……
}
```

在 Abstract class 方式中,sellable 可以有自己的数据成员,也可以有非 Abstarct 的成员方法,而在 Interface 方式的实现中,sellable 只能够有静态的不能被修改的数据成员(也就是说必须是 static final 的,不过在 Interface 中一般不定义数据成员),所有的成员方法都是 Abstract 的。从某种意义上说,Interface 是一种特殊形式的 Abstract class。

在 Abstract class 的定义中,可以赋予方法的默认行为。但是在 Interface 的定义中,方法却不能拥有默认行为,为了绕过这个限制,必须使用委托,必然会增加程序的复杂性。Abstract class 和 Interface 间进行选择时要非常小心。

1.2　从程序编辑的角度考虑

Abstract class 在 Java 语言中表示的是一种继承关系,一个类只能使用一次继承关系,只有一个父类。但是,一个类却可以实现多个 Interface,一个接口可以有多个父接口。接口和抽象类的选择不是随心所欲的,必须遵守这样一个原则:行为模型应该总是通过接口而不是抽象类定义。要理解接口和抽象类的选择原则,有两个概念很重要:对象的行为和对象的实现。如果一个实体可以有多种实现方式,则在设计实体行为的描述方式时,应当达到这样一个目标:在使用实体的时候,无需详细了解实体行为的实现方式。也就是说,要把对象的行为和对象的实现分离开来。

下面通过一个分别由接口和抽象类建立行为模型的实例,来说明两者的可行性。

实例:热水器有一个"瓦特"参数,现在基于这一个参数来进行行为定义。

1)通过抽象类建立行为:

热水器的最重要的参数就是瓦特,建立行为如下:

行为 1:查找热水器的参数,返回一个整数。

```
public abstract WaterHeater{
    abstract public int getWatt();
```

无论哪种类型的热水器,它都很重要。在所有派生的抽象类中,getWatt()都是有效的。

在 WaterHeater 抽象类的基础上,可以构造许多具体实现。如电热水器,它是通过电驱动的,因此,就存在一个电的流量问题,现在要查询其电流大小,建立行为如下:

行为 2:查找电热水器的驱动电流大小,返回一个表示电流大小的整数。

```
public abstract CurrentWaterHeater extends WaterHeater{
    abstract public int getCurrent();
    }
```

又如太阳能热水器,它是通过太阳能驱动的,现在要查询其所用的太阳能能量的大小,建立行为如下:

行为 3:查找太阳能热水器的驱动能量大小,返回一个表示太阳能能量大小的整数。

```
Public abstract SolarWaterHeater extends WaterHeater{
    abstract public int getSolarEnergy();
    }
```

现在出现一种新的热水器,它既可以电驱动又能够太阳能驱动。那么,在定义此种新热水器时,接口和抽象类的区别就显现出来了。电和太阳能热水器都是从 WaterHeater 抽象类派生,其中大部分的方式和功能是相同的,但是,也有其自身的特殊操作。下面考虑怎样建立新热水器 CurrentSolarWaterHeater 抽象类。如果 CurrentSolarWaterHeater 从 WaterHeater 派生,它将不支持针对电和太阳能热水器的特殊操作;如果 CurrentSolarWaterHeater 从 CurrentWaterHeater(SolarWaterHeater)派生的话,它又将不支持针对太阳能热水器(电热水器)的特殊操作。从行为上看,新驱动的热水器必须同时从两个抽象类派生,但 Java 语言不允许多重继承。之所以会出现这个问题,根本的原因在于使用抽象类不仅意味着定义特定的行为,而且意味着定义实现的模式。也就是说,应该定义一个热水器如何获得行为的模型,而不仅仅是声明热水器具有某一个行为。

2)通过接口建立行为:

行为 1:

```
public interface WaterHeater {
    public int getWatt(); }
```

行为 2:

```
public interface CurrentWaterHeater extends WaterHeater{
    public int getCurrent();}
```

行为 3:

```
public interface SolarWaterHeater extends WaterHeater{
public int getSolarEnergy();
}
```

新热水器可以定义为:

public CurrentSolarWaterHeater implements CurrentWaterHeater,
SolarWaterHeater {……}

CurrentSolarWaterHeater 只继承行为定义,而不是行为的实现模式。

实践证明,如果依赖于抽象类来定义行为,往往导致过于复杂的继承关系,而通过接口定义行为能够更有效地分离行为与实现,为代码的维护和修改带来方便[2]。

1.3 两者本质区别:设计思想

抽象类在 Java 语言中体现了一种继承关系[3]。父类和派生类的关系是一种"is a"关系,即父类和派生类的概念和本质是一样的。但是,接口并不要求接口的实现者和接口定义在概念本质上是一致的,仅仅是实现了 Interface 定义的契约而已。下面通过一个简单的例子加以说明。

问题领域抽象概念:mobiletelephone(手机)

手机所执行的两个动作:call(打电话)和 pickup(接电话)

现在使用 Abstract class 和 Interface 两种方式定义 mobiletelephone 分别如下:

```
abstract class mobiletelephone{
  abstract void call();
  abstract void pickup(); }
```

抽象类方式

```
interface class mobiletelephone{
  void call();
  void pickup();}
```

接口方式

而其他具体的 mobiletelephone 类型可以通过 extends 使用 Abstract class 定义的 mobiletelephone 或通过 implements 使用 Interface 定义的 mobiletelephone。此时,看起来,Abstract class 和 Interface 好象区别不大。

现在要求 mobiletelephone 还具有 time(定时)的功能,那么如何改进设计呢?文中采用以下两种方法。

● 第一种解决办法:

```
abstract class mobiletelephone{
  abstract void call();
  abstract void pickup();
  abstract void time();}
```

抽象类方式

```
interface class mobiletelephone{
  void call();
  void pickup();
  void time();}
```

接口方式

具有定时功能的 mobilephone_timer 定义方式如下:

```
class mobilephone_timer extends/implements Door {
    void call() { … }
    void pickup() { … }
    void time() { … }
}
```

讨论:这种解决方法违反了 OOP 中的核心原则 ISP(Interface Segregation Priciple)。概念 mobilephone 本身的行为方法和概念 mobilephone_timer 的行为方法混在了一起,有可能引起的后果是那些仅仅依赖于 mobilephone 这个概念的模块会因为 mobilephone_timer 这个概念的改变(比如:修改 time 方法的参数)而改变,反之依然。

● 第二种解决方法:

问题领域核心思想:mobilephone_timer 在概念本质上是 mobilephone,同时具有 time 的功能。

分析过程:

<1>call,pickup 和 time 属于两个不同的概念,根据 ISP 原则应该把它们分别定义在代表这两个概念的抽象类中。

<2>定义方式有:

a. 两个概念都使用 Abstract class 方式定义;

b. 两个概念都使用 Interface 方式定义;

c. 一个概念使用 Abstract class 方式定义,另一个概念使用 Interface 方式定义。

那么,究竟采用哪种定义方式呢?

由于 Java 不支持多重继承,因此,a. 是行不通的;如果使用 b. 的话,那么,mobiletelephone_timer 在概念本质上是 mobilephone 还是时间器(timer)?没有体现出问题领域的核心思想,也就是说,没有达到设计意图。采用 c. 定义方式,正好符合了问题的设计要求。因为,Abstract class 在 Java 语言中表示一种继承关系,而继承关系在本质上是"is a"关系。所以对于 mobilephone 这个概念,应该使用 Abstarct class 方式来定义。另外,mobilephone_timer 又具有时间功能,说明它又能够完成时间概念中定义的行为,所以时间概念可以通过 Interface 方式定义。如下所示:

```
abstract class mobiletelephone
    {abstract void call(); abstract void pickup();}
interface time
    { void time();}
class mobiletelephone_timer extends mobiletelephone implements
time
{ void call() { … };
void pickup() { … };
void time() { … };}
```

2 结 论

从以上的分析可知,接口和抽象类之间的选择很重要。对于它们的选择,反映出对于问题领域本质的理解和对于设计意图的理解是否正确、合理。面向对象的软件技术以对象为核心[4],OO 的精髓是对"对象"的抽象,最能体现这一点的就是接口[5]。也因此,使用接口的频率会高些。然而,接口的使用缺点是(特别是当代码已经完成后)要想接口进行修改将非常困难。所以,要充分理解问题领

(下转第 115 页)

6)返回 2,反复执行,直到所有点都覆盖完毕。

4 实验与分析

为评估本算法的性能,使用了标准的 UCI 测试数据集进行测试。为使结果和其它文献具有可比性,选用了 iris, glass, liver 几个数据集。

表 1　数据集参数

数据集	样本数	特征数	类别数
Iris	150	4	4
Glass	214	10	7
Liver	345	6	2
car	1728	6	4

实验分为两个方面,一是和其它文献使用的覆盖算法进行比较,二是对不同的覆盖算法进行自测。

首先,按文献[5]所采用的测试条件,即用 10 交叉法测试。用 MATLAB6.5 编程,测试结果和文献比较见表 1。

表 2　算法正确率比较　　(%)

数据集	文献[5]	文献[6]	贪婪覆盖
Iris	71.54	96.68	96.81
Glass	/	91.19	96.05
Liver	/	93.04	92.89
car	80.75	/	82.11

同时,使用不同的覆盖算法采用留一法进行自测,测试结果(平均覆盖数已取整数)列于表 3。

表 3　不同的覆盖方法所用的覆盖数比较

数据集	一般覆盖		挖点覆盖		贪婪覆盖	
	覆盖数	正确率%	覆盖数	正确率%	覆盖数	正确率%
Iris	29	94.64	12	95.33	9	95.23
Glass	22	94.33	7	96.73	6	97.20
Liver	143	90.2	121	91.4	65	91.4
car	458	91.4	228	93.6	145	93.5

由表 3 可见,错误率基本不变,但覆盖数量有了明显的下降。由于覆盖的数量对应着隐层神经元的数量,这意味着在对测试样本进行识别时,由于网络神经元数量的减少而提高了识别速度,减少了识别时间。

对于求得第 i 次最优覆盖,易知其所需使用的普通覆盖次数是 i 次。所以若需覆盖数量 n,整个覆盖过程需要的次数为 $n(n+1)/2$ 次。但由于贪婪覆盖算法的覆盖点数少,即 n 值大幅减小,则覆盖算法的覆盖数在样本数量多时,远小于一般覆盖。同时,为防止覆盖数量多时时间过长,也采取了一些措施,随机从前面的覆盖中限定选若干个。笔者在实验中取 10 个,与不限制相比,在覆盖数量小于 10 时,不起作用,在覆盖数量多时,保证计算时间限定在一个数量级内。根据贪婪算法的特点,通过几个大覆盖数样本的实验,是否用此限制,结果互有优劣,但相差不大。

文献[6]提出的核偏移算法,虽然它能获得局部最优,但显然,局部最优不见得会导致全局最优,实验结果证明了这一点。但同时,它所耗费的时间代价是巨大的,与一般覆盖算法相比,其计算时间呈指数增长。在目前没有哪种方法被证明最优的情况下,贪婪覆盖算法是一种求解的有效途径。

5 结　论

本算法的特点是以增加少量训练的时间而大量减少识别时所用的时间,但增加的时间是可以控制的。处理方法简单易懂,效果明显。并且由于覆盖算法的非线性特点,可以构造非常复杂的划分边界。在实际的应用中,有相当多的应用对学习时间不是特别敏感。这些应用就可以采用贪婪覆盖算法。与目前流行的 SVM 相比,它也有自己的优势,SVM 对于两类问题,有较优的解,但对多类问题则需转换成多个两类问题,计算时间长。而覆盖算法可直接处理多类识别问题,并且相对时间较短。两种方法各有所长,可根据使用对象选择。

参考文献:

[1] 张　铃. A Geometrical Representation of McCulloch Pitts Neural Model and Its Applications[J]. IEEE Trans on Neural Networks,1999,10(4):925-929.

[2] 张　铃. 基于核函数的 SVM 机与三层前向网络的关系[J]. 计算机学报,2002,25(7):697-700.

[3] 张　铃,张　钹. M-P 神经元模型的几何意义及其应用[J]. 软件学报,1998,9(5):334-338.

[4] 张　铃,张　钹,殷海风. 多层前向网络的交叉覆盖算法[J]. 软件学报,1999,10(7):737-742.

[5] 毛军军,吴　涛,郑婷婷. 基于商空间的构造性分层竞争网络算法[J]. 微机发展,2005,15(4):37-39.

[6] 赵　姝,张燕平,张　媛,等. 基于交叉覆盖算法的改进算法——核平移覆盖算法[J]. 微机发展,2004,14(11):1-3.

(上接第 112 页)

域的核心东西,慎重选择。

参考文献:

[1] Geoff C C,Keeton F B. JAVA 完全探索(第 2 版)[M]. 师夷工作室译. 北京:中国青年出版社,2001.

[2] 尉哲明,郝建文. JAVA 中利用内部类简化程序的编写[J]. 微机发展,2003,13(3):41-44.

[3] Wampler B E. JAVA 与 UML 面向对象程序设计[M]. 王海鹏译. 北京:人民邮电出版社,2002.

[4] 张海藩. 软件工程导论[M]. 北京:清华大学出版社,2003.

[5] 王克宏,董　丽. Java 技术及其应用[M]. 北京:高等教育出版社,1999.