ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.17, No.4, April 2006, pp.658−669
DOI: 10.1360/jos170658

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于角色的设计模式建模和实现方法*

何成万[1+], 何克清[2]

[1](武汉化工学院 计算机科学与工程学院,湖北 武汉 430073)

[2](武汉大学 软件工程国家重点实验室,湖北 武汉 430072)

A Role-Based Approach to Design Pattern Modeling and Implementation

HE Cheng-Wan[1+], HE Ke-Qing[2]

[1](School of Computer Science and Engineering, Wuhan Institute of Chemical Technology, Wuhan 430073, China)

[2](State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China)

+ Corresponding author: Phn: +86-27-62888253, E-mail: hechengwan@hotmail.com

He CW, He KQ. A role-based approach to design pattern modeling and implementation. *Journal of Software*, 2006,17(4):658−669. http://www.jos.org.cn/1000-9825/17/658.htm

Abstract: Although design pattern is quite useful in software reuse, there are still many barriers when instantiating the design patterns, such as pattern overlapping, traceability, and difficulties in reusing the pattern code. A role-based approach for design pattern modeling and implementation is proposed. In this approach, roles of pattern are treated as the independent modeling elements and the RoleOf relationship is used to associate a role with an application class. This can improve the reusability of pattern. The meta-model of the RoleOf relationship for pattern instantiation and its semantics are proposed using UML extension mechanism. The stereotypes and tagged values used for identifying pattern information are provided, and it resolves the traceability and overlapping problem in pattern instantiation. The dynamic binding of application and role logic are implemented through the extension to Java language, called Rava. The approach proposed in this paper can effectively solve the problem such as pattern overlapping and traceability during the pattern instantiation, which improves the reusability of pattern logic and guides the software development using design patterns.

Key words: design pattern instantiation; role; RoleOf relationship; UML; meta-model

摘 要: 虽然设计模式有利于软件的重用,但当设计模式实例化时,存在模式的重叠、可追踪性以及模式代码难于重用等问题.提出一种基于角色的设计模式的建模和实现方法,在设计和实现两个层面上解决上述问题.该方法把设计模式的角色作为独立的建模元素,在应用程序类和角色间导入 RoleOf 关系,使应用逻辑和模式逻辑完全分离,从而提高其重用性.使用 UML 的标准扩充机制,给出了 RoleOf 关系的元模型和语义,以及标识模式信息的构造型和标记值,以此解决模式的重叠和可追溯性等问题.通过对 Java 语言进行扩充(称为 Rava),实现了应用逻辑和模式逻辑的动态绑定.该方法较好地解决了设计模式实例化时的模式重叠和可追踪性问题,提高了模式逻辑的可重用性,对运用设计模式开发软件有一定的指导作用.

* Supported by the National Natural Science Foundation of China under Grant No.60373086 (国家自然科学基金); the Wuhan Scientific & Technical Key Industrial Project of China under Grant No.20051001007 (武汉市科技局重大产业化项目)

Received 2005-09-02; Accepted 2005-11-09

关键词: 设计模式的实例化;角色;RoleOf 关系;UML;元模型
中图法分类号: TP311 文献标识码: A

设计模式[1]提供了可重用的软件设计方案,被广泛地应用于软件设计过程中.设计模式的具体应用称为模式的实例化.虽然设计模式得到了广泛的应用,但在实例化模式时还是存在很多问题,文献[2]把它们归纳为以下3个最主要的问题:实现(implementation)、文档(documentation)和组合(composition).

设计模式的实现往往会根据被使用的环境进行适当的变更,由此导致设计模式在代码中的消失,对程序的理解和逆向也会变得复杂.而且,由于模式中的角色都被具体的应用程序类所取代,其结果是,虽然设计模式的思想能够被重用,但其实现往往不能够被重用.很多学者使用 AOP(aspect oriented programming)[2,3]技术实现设计模式,以此达到重用设计模式代码的目的.

文档问题也称为可追溯性(traceability)问题[4],是由于模式的代码分散在业务代码中而产生的.特别是当系统使用多个模式,或一个应用程序类包含在多个模式中时,很难获取模式的实例化信息,如类和角色的绑定信息等.组合问题也称为模式的重叠(overlapping)问题[5],是文档问题的一个特例.当一个应用程序类包含在多个模式中时,很难区分这些模式实例以及它们之间的关系,导致对程序的理解和逆向变得困难.

为了解决以上问题,我们提出了一种基于 RoleOf 依赖关系的模式实例化方法[6].我们的策略是:在实例化设计模式时,不用应用程序类替代模式中的角色,而是把角色定义成一个独立的类(称为角色类),使用 RoleOf 关系连接应用程序类和角色类,从而把它们完全分离开,实现业务逻辑和模式逻辑的重用.通过对 UML(unified modeling language)[7]的扩充,给出了 RoleOf 关系的元模型及其语义,以及用于标识模式信息的构造型(stereotype)和标记值(tagged value),以此解决模式的重叠和可追溯性等问题.通过对 Java 语言进行扩充,在 Java 语言中引入了角色(role)概念,实现了 RoleOf 关系.我们把扩充后的 Java 语言称为 Rava.在 Rava 中,分别定义应用程序类和角色类,实现了它们之间的动态绑定,有助于程序逆向中的设计模式的识别.

我们的方法涉及到设计和实现两个层次.本文首先简要介绍该方法的主要思想;然后重点介绍 RoleOf 关系的实现方法,包括 Rava 的实现原理、语法和语义、Rava 到 Java 的转换等.

1 相关研究分析

1.1 设计层

文献[4]介绍了一种新的描述设计模式组合的方法.它使用 UML 的扩展功能——标记值,来标注实例化的模式信息,如:模式名、模式的方法和属性等.该方法能够明确地标注模式实例化信息,有很好的可追溯性,有助于系统的理解,但还存在一些不足:业务逻辑和模式逻辑没有分离,特别是当一个应用程序类包含在多个模式中时,虽然可以用标记值区分各个方法所属的模式,但由于其包含的方法太多,使得应用程序类的功能变得不明确,这样的类很难被重用.

文献[5]介绍了一种使用 UML 在不同抽象层次组合设计模式的方法 POAD(pattern-oriented analysis and design).POAD 定义了 3 个层次化的逻辑视点:模式层视点(pattern-level view)、模式接口视点(pattern interfaces view)以及详细模式层视点(detailed pattern-level view).模式的内部结构在较高层次被隐藏,在较低的层次被描述.该方法着重于面向模式的系统开发过程,没有解决模式逻辑和应用逻辑如何组合的问题[5].

文献[8]引入组合模式(composition patterns)的概念,用于可重用的 Aspect 的设计.它基于面向主题模型(subject-oriented model)和 UML 模板(templates)的概念,对 UML 进行了扩充.该方法把模式定义成模板包(template package),实例化模式时,把它和应用程序包进行绑定并产生一个组合输出(composition output),得到模式的实例.该方法虽然在设计层可重用组合模式,提高了设计的可跟踪性,但最终还是需要把组合模式和用户定义的类组合在一起,没有分离业务逻辑和模式逻辑.另外,该方法中的重用单位是模板,实例化时,需要把每个模板参数和具体应用类进行绑定,不能实现细粒度的重用(单独重用模板包中的每个角色).

文献[9]描述了一种设计模式的可视化建模语言 DPML(design pattern modelling language).该语言提供了一组模式建模和实例化的表示方法,但由于没有遵从 UML 标准,很难被推广应用.并且,该方法不能很好地支持模式的重叠[9].

1.2 实现层

继承是面向对象中使用得最为广泛的一种重用机制.如果一个类包含在多个模式中,需要使用多继承.多继承有很多不足,在一些面向对象语言(如 Java,C++)中不包含多继承.

Mixin 使用单继承实现不同父类的扩展[10,11].使用 Mixin 的主要问题在于不容易组合,因为 Mixin 的组合是使用继承来实现的,所以这种组合是线性的.由此会产生一些问题[12]:第一,合适的顺序很难发现,或者说根本不存在;第二,使用这种线性组合的辅助代码分散在整个类阶层;第三,很小的变更会影响整个类结构.

面向 Aspect 的编程(AOP)[2,3,13]是一种新的编程技术,它允许程序员对横切关系(crosscutting concerns,跨越多个模块的关注点,例如日志、事务处理)进行模块化.AOP 引入了 Aspect,它将影响多个类的行为封装到一个可重用模块中.要让 Aspect 影响正常的类代码,必须将 Aspect 加入到它们要修改的代码中去.这一过程称为织入(weaving),包括静态和动态的织入.现有很多研究者尝试使用 Aspect 实现设计模式[2],但还是存在一些问题[14]:(1) 很难获得充分的、用于织入外部行为的结合点;(2) 缺乏一般性,在很多地方需要进行类型转换.

角色(role)的概念被广泛地应用于包含对象重分类(object reclassification)业务系统的概念建模中[15,16],其最核心的假设是一个对象的外部(extrinsic)属性和行为在它的生存期内会发生变化[16].Van Hils 等人[17]采用 C++模板实现角色,实现了角色模型到代码的直接映射,提高了角色模型实现层次上的重用性.但在实例化角色时,采用的是 Mixin 的方法,即单继承的方法.如果一个类要实现多个角色,则继承的层次会很深[17].Dec-Java[18]使用 Decorator 模式在语法上对 Java 进行了扩展,实现了对象行为的动态扩充.但是,动态行为(角色)的管理对程序员是非透明的.

综上所述,模式的实例化还没有一套完整的理论和方法,因而严重阻碍了设计模式的大规模应用.虽然有一些关于使用 UML 对设计模式精确建模的方法[19,20],但都侧重于模式的描述,而没有考虑模式逻辑和应用逻辑的分离等问题.同时,越来越多的学者使用 AOP 实现设计模式,虽然提高了模式逻辑的重用性,但还是存在一些问题,如缺乏一般性、很难获得结合点(joinpoint)等.

在设计层,要把设计模式的结构和应用程序的结构分开,以达到提高重用性、便于维护的目的;同时,为了描述模式实例化信息,需要定义一些构造类型和标记值.在实现层,为了提高模式逻辑的重用性,我们基于角色的概念,对现有的面向对象语言进行语法上的扩充,使之能够在语言级上支持角色的概念,实现对象和角色的动态绑定功能.

2 分离模式逻辑和应用逻辑的建模方法

2.1 基本原理

设计模式在实例化时,一般都是把用户定义的应用程序类绑定到模式中的某个角色.如图 1 所示,Class1 和 Class2 是两个应用程序类,分别和角色 role1 和 role2 进行绑定;绑定后,Class1 和 Class2 中分别包含了角色 role1 和 role2 中的属性和方法.这样,不仅很难区分应用逻辑和模式逻辑,而且使得应用程序类的功能变得不明确,很难被重用.

为了实现设计模式的模式逻辑的重用,需要把它们分离.我们使用 RoleOf 关系来描述角色和应用程序类之间的联系.图 2 表示角色 role1 和 role2 分别是 Class1 和 Class2 的一个角色.使用 RoleOf 关系后,角色和应用程序类都被视为独立的建模元素.为了对 RoleOf 关系建模,我们对 UML 进行了扩充,给出了 Role 关系的元模型及其语义.

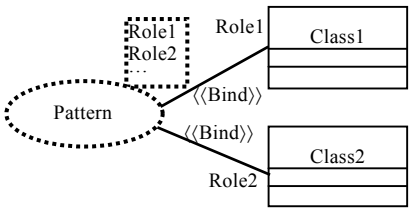


Fig.1 Binding of business class and role

图 1 应用程序类和角色的绑定

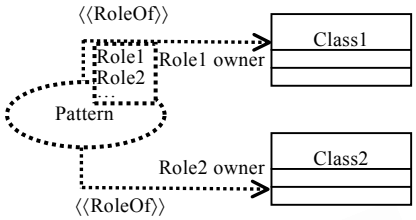


Fig.2 RoleOf relationship between business class and role

图 2 应用程序类和角色的 RoleOf 关系

2.2 RoleOf关系的元模型及语义

RoleOf 关系的元模型如图 3 所示.

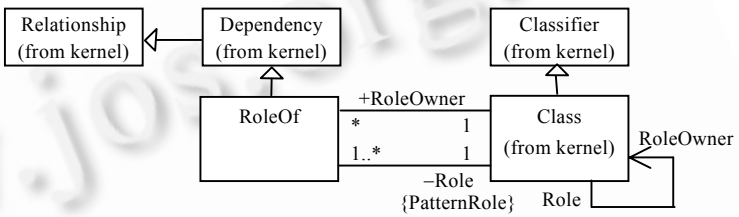


Fig.3 Meta model of RoleOf relationship

图 3 RoleOf 关系的元模型

RoleOf 关系是一种依赖关系(继承 dependency).RoleOf 关系的两端是类:roleOwner 表示应用程序类;role 表示角色类.但是,角色类要用构造型⟨⟨PatternRole⟩⟩修饰.另外,定义了两个标记值 patternName 和 roleName,分别标识应用程序类或角色类所属的模式名和角色名.例如:如果用{patternName=Observer},{roleName=Subject}修饰角色类,则表示该角色是 Observer 模式中的 Subject 角色.

RoleOf 关系的语义可以描述为:

- RoleOf 关系属于依赖关系.从两方面可以解释这种依赖关系:一方面,当一个对象的角色发生作用时,对象的行为依赖于角色的行为;另一方面,角色不能够单独存在,只有和对象关联后才发生作用.
- 一个应用程序类可以有多个 RoleOf 关系(多重度为*),即可以关联 0 个或多个角色类(PatternRole).同时,一个角色类也可以关联到一个以上的应用程序类(多重度为 1...*).这表明角色类不能单独存在,必须和一个以上的应用程序类建立 RoleOf 关系后才有意义.
- 角色类的定义方法与应用程序类相同.但是,如果一个角色类的属性或方法中带有引用类型参数时,如 Observer 模式[1]的 Subject 角色中有一个 addObserver(observer)方法,其参数类型是 Observer 类,则需要把角色类定义成模板类,并且要把实际参数和模板类进行绑定.
- RoleOf 关系是单向的,从 role 可导航到 roleOwner.
- RoleOf 关系只能连接应用程序类和角色类.

在 UML 元模型的基础上扩充的构造型和标记值见表 1 和表 2.

Table 1 Extended stereotypes and definition

表 1 扩充的构造型及其定义

Stereotype	Applies to	Definition
RoleOf	Dependency relationship	Dependency relationship between application and role class
PatternRole	Class	The role in the design pattern

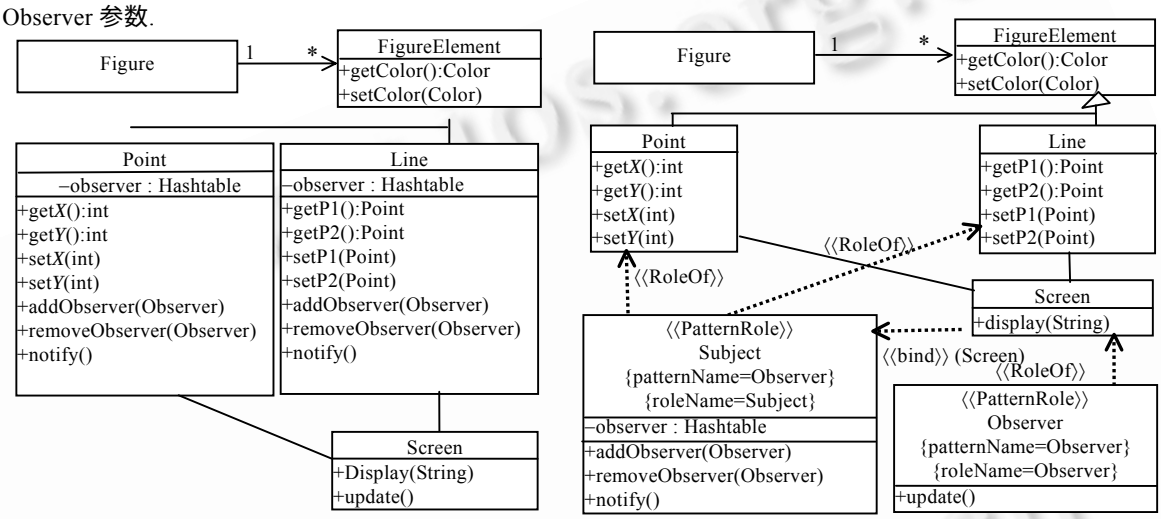
Table 2 Extended tagged values and definition

表 2 扩充的标记值及其定义

Tagged value	Applies to	Definition
PatternName	Application and role class	The name of the design pattern
RoleName	Application and role class	The role name in the design pattern

2.3 一个应用实例

Observer 模式[1]是很常用的一种设计模式,它能够在多个关联对象间保持信息的一致性[1],即当一个对象的状态发生变化时,它会通知与其关联的一些对象.文献[2]介绍了一个简单的使用 Observer 模式的图形元素系统:Point 和 Line 类继承 FigureElement 类,相当于 Observer 模式中的 Subject 角色,且都包含 addObserver(), removeObserver(),notify()等方法.Screen 类相当于 Observer 模式中的 Observer 角色,包含 update()方法.如图 4(a)所示.使用 RoleOf 关系后的类结构如图 4(b)所示.Observer 模式中的 Subject 和 Observer 角色被定义成使用构造型⟨⟨PatternRole⟩⟩修饰的两个独立的类,分别包含 Subject 和 Observer 角色的属性和方法.而且使用 patternName 和 roleName 标记值分别标记了模式名称和角色名称.Subject 角色和 Point 类及 Line 类之间是 RoleOf 关系.同样,Observer 角色和 Screen 类之间也是 RoleOf 关系.另外,由于 Subject 角色中的 addObserver()和 RemoveObserver()方法带有引用类型参数,所以,Subject 角色类被定义成了模板类,并且把 Screen 类绑定到了 Observer 参数.



(a) Before using RoleOf relationship
(a) 使用 RoleOf 关系前

(b) After using RoleOf relationship
(b) 使用 RoleOf 关系后

Fig.4 A figure element system based on Observer pattern
图 4 使用 Observer 模式的图形元素系统

3 RoleOf 关系的实现

由于现有的基于类的主流面向对象语言,如 Java,C++等,都不直接支持角色的概念,要在实现层导入角色概念,必须采用特殊的实现方法.很多学者已提出了多种实现方法,我们将其大致归纳为以下 3 类:

(1) 在实现层使用语言的特殊功能.如 Van Hilst 和 Notkin 使用 C++的模板类实现角色[17].Kendall 使用 AOP 实现了角色概念[21];

(2) 在设计层使用设计模式.如 Role Object[22],Decorator [1],State[1]等模式可以动态地改变对象的行为;

(3) 对现有面向对象语言进行扩充,在语言级上支持角色概念[18,23,24].

方法 1 属于静态的绑定,即在编译阶段绑定对象和角色;方法 2 存在的主要问题是客户程序实现角色对象的生成和绑定等操作,客户程序变得较为复杂;方法 3 中的大部分提案都是在现有的面向对象语言(主要是 Java 语言)中增加一些新的关键字,然后把新增的关键字进行转换.主要存在以下两个问题:一是角色的管理对程序员不是透明的;二是对象和角色的重用性较低,其原因是对象和角色中都包含对对方的引用,耦合性较高.

我们提出一种基于 Mediator 模式[1]的角色实现方法.客户程序访问某个对象的角色,以及角色访问对象都要以 Mediator 对象为中介.Mediator 对象完成角色的管理,包括角色对象的生成、动态绑定等,减少了客户程序

的复杂性.同时,该方法把绑定关系从 core 对象和角色中分离出来,保存在 Mediator 对象中,减少了 core 对象和角色间的耦合性,从而提高了它们的重用性.我们通过对 Java 语言的扩充,称为 Rava,给出了该方法的一个原型实现.我们的方法是上述方法 2 和方法 3 的结合.

我们实现了从 Rava 程序到 Java 程序的转换.在 Rava 中,Mediator 对象的定义使用 Java 泛型实现.程序员定义 core 对象以及角色,Mediator 对象由转换程序自动生成.

3.1 基本思想

为了降低客户程序的复杂性以及 core 对象和角色对象之间的耦合性,提高其重用性,我们提出了一种基于 Mediator 模式的角色实现方法.其最基本思想包括:

- 角色被表示成一个对象,每一个 core 对象和角色对象都有一个独立的物理对象标识符(object identifier);当一个 core 对象和一个角色对象绑定后,它有一个逻辑对象标识符,用〈coreObject, roleObject〉对表示;
- Mediator 对象对角色进行管理,包括角色对象的生成、core 对象和角色对象的动态绑定等.另外,core 对象的逻辑对象标识符保存在 Mediator 对象中;
- 转换程序把 Rava 程序转换成 Java 程序,即把新增的关键字转换成对应的 Java 代码.与一般的宏替换系统(如 C 中的#define)不同,我们的转换方法是基于语义的替换,替换文本不是固定不变的,根据角色的定义方法以及角色的调用方法动态生成替换文本;
- 使用 Java 泛型定义 Mediator 对象,实现 Mediator 对象代码的重用.转换程序对其实例化后,自动生成每个 core 对象的 Mediator 对象;
- 每一种 core 对象类型(类)对应一个 Mediator 对象.它负责该类型的 core 对象和它们能够完成的角色群之间的协调.

Mediator 对象、core 对象以及角色对象之间的协作如图 5 所示.

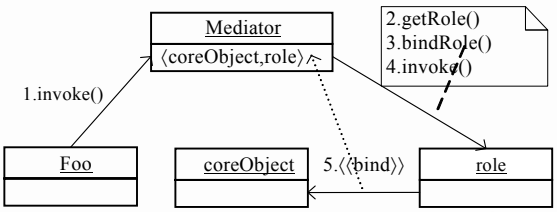


Fig.5 Collaboration between Mediator, core and role object

图 5 Mediator 对象、core 对象以及角色对象之间的协作

3.2 Rava的基本原理

3.2.1 一个简单例子

图 6 定义了一个类 Person 和一个角色 Employee.

```
//Person.java
class Person{
        String name;
        int deposite;
        public String getName(){
                return name;
        }
}
```

```
//Employee.rava
ROLE Employee roleof Person{
        public int getPaid(int salary){
                @core Person().deposit+=salary;
                return deposit
        }
}
```

Fig.6 Definition of person class and employee role

图 6 Person 类和 Employee 角色的定义

从上面的例子可以看出,角色的定义至少包含两部分:角色名的定义(role employee)——使用关键字 ROLE;以及角色的绑定规则(roleof person)——指明该角色的实例可以与哪些类的实例绑定.使用关键字 roleof 定义,在角色 Employee 的定义中还使用了一个关键字@core,它表示要取得和该角色对象绑定在一起的指定类型的

core 对象,进而访问 core 对象的属性.

图 7 的定义中包含一个角色调用,表示调用对象 aPerson 的 Employee 角色.调用中还包括对象(aPerson)的类型(person)、要调用的角色的方法(getPaid)以及参数(salary).

```
//Foo.rava
class Foo{
    public static void main(String args[]){
        Person aPerson=new Person();
        Integer salary=new Integer(2000);
        @INVOKEROLE(aPerson, "Person", "Employee", "getPaid", salary);    }
}
```

Fig.7 Invocation of role

图 7 角色的调用

3.2.2 其他语言元素

在角色的定义中,除了使用 roleof 指明绑定对象的类型以及使用@core 访问 core 对象的属性和方法以外,还可以包含以下一些构造元素:

Interfaces:一个角色可以和类一样实现多个接口;

Super Classes:一个角色可以继承一个类或一个角色;

Inherited instance:在角色定义中,可以使用 this.id(访问本身的属性)和 super.id(访问父类的属性).

在角色中,不能定义构造函数和静态成员(static members),因为客户程序不能直接访问角色逻辑,所以,定义构造函数和静态成员是没有意义的.在编译时,系统会自动产生一个不带参数的构造函数,用于角色类型和对象类型的绑定.

3.2.3 扩充 Java 语言

表 3 列出了 Rava 中新增的 4 个关键字.

Table 3 New added keywords and definition in Rava

表 **3** Rava 中新增的 4 个关键字及定义

Keyword	Definition
ROLE	Role definition
roleof	Relationship between role and object, that is, which object can play the defined role
@INVOKEROLE	Role invocation
@core	Accessing the attributes and methods of core object from role

为了描述新增关键字的抽象语法,我们使用以下记号:

*A**:表示 *A* 出现 0 次或多次;

A+:表示 *A* 出现 1 次或多次;

[*A*]:表示 *A* 是可选项.

图 8 是新增关键字的抽象语法.

```
⟨role declaration⟩::=⟨modifiers⟩* ROLE [extends ⟨type declaration⟩] [implements ⟨interface
                    declaration⟩+] roleof ⟨class declaration⟩+
⟨type declaration⟩::=⟨class name⟩|⟨role name⟩
⟨role invocation expression⟩::=@INVOKEROLE(⟨object⟩,⟨class name of object⟩,
                                ⟨role name⟩,⟨method name⟩,⟨parameter⟩ [])
⟨core object getting expression⟩::=@core ⟨class name⟩ ()
```

Fig.8 Abstract syntax of the new added keywords

图 8 新增关键字的抽象语法

@core 用于从角色访问 core 对象中的属性和方法,它与 Java 语言中的关键字 new 的区别在于:@core 的功

能是从 Mediator 对象取得和角色对象绑定在一起的指定类型的 core 对象,不生成新的对象;Java 语言中使用关键字 new 生成一个新的对象.

3.3 Rava到Java的转换

3.3.1 角色定义的转换

在进行角色定义的转换时,除了要对关键字 ROLE 和 roleof 进行转换,还要在角色定义中增加保存和访问绑定规则(该角色对象可以和哪些类的实例动态绑定)的代码.因为 Mediator 对象在绑定 core 对象和角色对象之前,要访问这些绑定规则,以确定指定的 core 对象和角色对象是否能够绑定在一起.绑定规则的保存定义在构造函数中,对绑定规则的访问定义在一个方法中.转换算法如下所示:

1. 把角色声明字符串按关键字分组:ROLE,extends,implements,roleof.
2. 把关键字 ROLE 转换成关键字 class.
3. 去掉 roleof 节.
4. 在 implements 节的最后增加一个接口 RoleInterface.
5. 在角色定义中增加一个私有属性,用于保存可以和该角色绑定的对象类型.
   private Vector *bindingClasses=new Vector*();
6. 在角色定义中增加一个如下的构造函数,用于保存绑定规则.其中,RoleName 是角色名,ClassName 是需要绑定的类名(roleof 关键字后面定义的类名).
   public *RoleName*() { *bindingClasses*.*add*(*ClassName*);}
7. 在角色定义中增加一个取得绑定规则的方法.
   public Vector *getBindingClasses*(){*return bindingClasses*;
8. 如果角色定义中包含@core 关键字,把它转换成对与该角色对象绑定的 core 对象的调用.

要获得一个与该角色对象绑定的 core 对象,首先要取得 Mediator 对象.然后,通过调用它的 getCore()方法获得和指定的角色对象绑定的 core 对象.Mediator 对象的定义使用了 Java 泛型,其主要属性和方法包括:

- boundRole 属性:它是一个 Hash 表,保存绑定的 core 对象和角色对象.core 对象为键,通过 core 对象可以取得绑定的角色对象;
- invokeRole()方法:它的功能是调用角色对象中的指定方法,实现时使用了 Java 的反演 API;
- getRoleInstance()方法:取得或生成角色对象;
- bindRole()方法:绑定 core 对象和角色对象,即以 core 对象为键、角色对象为值存入 Hash 表(boundRole);
- getCore():取得和指定的角色对象绑定在一起的 core 对象.

按照上述转换算法,可以把图 6 中的 Employee 角色转换成如下所示的 Java 代码.其中的粗体字表示增加或替换的代码.构造函数把 roleof 中定义的类型追加到绑定规则表中.getBindingClasses()方法返回与该角色对象的绑定规则.@core 关键字被替换成了从 Mediator 对象取得的 core 对象.

```
//Employee.java
class Employee implements RoleInterface{
private Vector bindingClasses=new Vector();
 public Employee() {bindingClasses.add("Person");}
 public Vector getBindingClasses(){return bindingClasses;  }
 public int getPaid(int salary){
      try {
           if (MediatorFactory.getMediatot("Person")==null){
                MediatorFactory.putMediator("Person", new GenericMediator<Person>());
           }
```

```
        }catch(Exception e){e.printStackTrace();}
        GenericMediator mo$1=MediatorFactory.getMediator("Person");
        Person rjc$Person=(Person)mo$1.getCore("Person",this);
        rjc$Person.deposit+=salary;
        return deposit
    }}
```

3.3.2 @INVOKEROLE 的转换

@INVOKEROLE 表示对角色的调用.转换程序把其转换成对 Mediator 对象的调用(调用其 invokeRole()方法).图 7 中定义的角色调用被转换成如下所示的 Java 代码.

```
//Foo.java
class Foo{
    …
    try {
        if (MediatorFactory.getMediator("Person")==null){
            MediatorFactory.putMediator("Person",new GenericMediator⟨Person⟩());
        }
        GenericMediator mo$1=MediatorFactory.getMediator("Person");
        mo$1.invokeRole(aPerson, "Employee", "getPaid", salary);
    } catch(Exception ex){ex.printStackTrace();}
}
```

3.4 Rava的动态语义

本节通过 Rava 到 Java 的转换给出 Rava 的动态语义.首先介绍使用的一些符号.

$R\triangle_c C$:角色 R 直接继承类 C.

$C'\triangle_c C$:类 C'直接继承类 C.

$R\triangle_i I$:角色 R 直接实现接口 I.

$C'\triangle_i I$:类 C'直接实现接口 I.

$R\triangle_r C$:角色 R 的实例可以和类 C 的实例绑定.

$M_c\triangle_m C$:M_c 是对应于类 C 的 Mediator 对象的定义.

C is$_c$ K *FST KST MST*:类 C 属于类型 K,定义中包含属性(FST)、构造函数(KST)以及方法(MST).

I is$_i$ *MST*:接口 I 定义中包含方法(MST).

instanceT:类型 T 的实例(类,接口,角色).

nameT:类型 T 的名称(类,接口,角色).

fieldsT:类型 T 中定义的属性(类,接口,角色).

methodsT:类型 T 中定义的方法(类,接口,角色).

bodyM:方法 M 的方法体(实现).

parametersM:调用方法 M 时的参数.

【】:从 Rava 到 Java 的转换函数.

3.4.1 ROLE

在 Rava 中,一个角色被转换成一个 Java 类.被转换后的类继承被角色所继承的类,同时,除了要实现被角色所实现的接口以外,还要实现 RoleInterface 接口.转换的形式化定义如下:

$【R\triangle_c C】=\{C'\triangle_c C\}$

$【R\triangle_i I】=\{C'\triangle_i I\}\cup\{C'\triangle_i RoleInterface\}$

其中,角色 R 和类 C'的名称相同,并且

RoleInterface is_i *MST*

MST={*public Vector getBindingClasses*();}

关键字 roleof 被转换成保存和获取绑定对象类型的 Java 代码.其中,保存绑定对象类型是指把使用 roleof 定义的类型保存到类变量中,由构造函数实现这一功能.获取绑定对象类型是指把可以和该角色对象绑定的对象类型从类变量中取出,该功能由一个 Getter 函数实现.转换的形式化定义如下:

【$R\triangle_r C$】={C' is_c ∅ *FST KST MST*}

其中:

FST={*fieldsR*}∪{*private Vector bindingClasses*=*new Vector*();}

KST={*public nameC*′(){*bodyK*}}

bodyK=*bindingClasses*.*add*(*nameC*);

MST={*methodsR*}∪{*public Vector getBindingClasses*(){*bodyM*}}

bodyM=*return bindingClasses*;

3.4.2 @core 和@INVOKEROLE

关键字@core 被转换成和该角色对象绑定的 core 对象.关键字@INVOKEROLE 被转换成对 Mediator 对象的调用.转换的形式化定义如下:

【*@core C*】={$M_c\triangle_m C|instanceM_c.getCore(this)$}

【*@INVOKEROLE*(*instanceC*, *nameC*, *nameR*, *nameM*, *parametersM*)】=

{$M_c\triangle_m C|instanceM_c.invokeRole(instanceC, nameR, nameM, parametersM)$

3.5 使用Rava实现RoleOf关系

使用 Rava 可以实现 RoleOf 关系到代码的直接映射:RoleOf 关系被映射为 Rava 中的 roleof 关键字;RoleOf 关系一端的角色类被映射为 Rava 中的 ROLE 定义;而应用程序类被映射为 Rava 中的普通类的定义,并且其名称被添加到 roleof 的定义中.在图 4(b)所示模型中,Point 类、Screen 类、Subject 角色以及客户程序(Foo.rava)可以用以下代码实现.

```
//Point.java
public class Point{
    int x=10;
    public int getX(){
        return x;
    }
    public void setX(int x){
        //…
    }
}

//Screen.java
public class Screen{
    public Screen(){
        //…
    }
}
```

```
//Subject.rava
ROLE Subject roleof Point{
    private Vector observers=new Vector();
    public void addObserver(Screen o){
        observers.add(o);}
    …
}

//Foo.rava
public class Foo{
    …
    Point point=new Point();
    Screen screen=new Screen();
    Object o[]=new Object[1];
    o[0]=screen;
    @INVOKEROLE(point,"Point","Subject",
                                "addObserver",o);
    …}
```

4 结 语

本文提出了一种基于角色的设计模式的建模和实现方法.该方法可以提高应用逻辑和模式逻辑的重用性.我们认为本文的主要贡献包括:

- 在设计层,通过分离模式逻辑和应用逻辑,解决了模式实例化中的文档、可追溯性等问题;
- 在实现层,实现了模式逻辑和应用逻辑的动态绑定.与以前的一些角色实现方法相比,具有角色逻辑和应用逻辑的低耦合性以及角色管理的透明性等优点.

我们的方法有助于程序逆向中的设计模式的识别.从角色的定义中可以得到模式的实例化信息,诸如角色名、角色的实现,以及与该角色绑定的应用程序类的信息.

References:

[1] Gamma E, Helm R, Vlissides J. Design Patterns: Elements of Reusable Object Oriented Software. Tokyo: Soft Bank Publishing, 2001.

[2] Hannemann J, Kiczales G. Design pattern implementation in Java and AspectJ. ACM SIGPLAN Notices, 2002,37(11):161−173.

[3] Kiczales G, Lamping J, Menhdhekar A, Maeda C, Lopes C, Loingtier J, Irwin J. Aspect oriented programming. In: Aksit M, Matsnoka S, eds. Proc. of the European Conf. on Object-Oriented Programming. Berlin, Heidelberg: Springer-Verlag, 1997. 220−242.

[4] Jing D. UML extensions for design pattern compositions. Journal of Object Technology, 2002,1(5):149−161.

[5] Yacoub SM, Ammar HH. UML support for designing software systems as a composition of design patterns. In: Gogolla M, Kobryn C, eds. The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Berlin, Heidelberg: Springer-Verlag, 2001. 149−165.

[6] He CW, He F, He KQ, Liu J, Tu WJ. RoleOf relationship and its meta model for design pattern instantiation. In: Li X, Wang SL, Dong ZY, eds. Advanced Data Mining and Applications, 1st Int'l Conf. (ADMA 2005). Berlin, Heidelberg: Springer-Verlag, 2005. 642−653.

[7] Object Management Group. Unified Modeling Language Specification, Version 1.4, 2001. http://www.omg.org

[8] Clarke S, Walker RJ. Composition patterns: An approach to designing reusable aspects. In: Proc. of the 23rd Int'l Conf. on Software Engineering. Washington: IEEE Computer Society, 2001. 5−14.

[9] Mapelsden D, Hosking J, Grundy J. Design pattern modelling and instantiation using DPML. In: Noble J, Potter J, eds. Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002). Darlinghurst: Australian Computer Society, Inc., 2002. 3−11.

[10] Bracha G, Cook W. Mixin-Based inheritance. In: Meyrowitz NK, ed. Proc. of the European Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP'90). New York: ACM Press, 1990. 303−311.

[11] Ancona D, Lagorio G, Jam EZ. A smooth extension of Java with Mixins. In: Bertino E, ed. Proc. of the 14th European Conf. on Object-Oriented Programming (ECOOP 2000). Berlin, Heidelberg: Springer-Verlag, 2000. 154−178.

[12] Schärli N, Ducasse S, Nierstrasz O, Black A. Traits: Composable units of behavior. In: Cardelli L, ed. Proc. of the European Conf. on Object-Oriented Programming (ECOOP 2003). Berlin, Heidelberg: Springer-Verlag, 2003. 248−274.

[13] AspectJ Team. The AspectJ™ Programming Guide. 2003. http://eclipse.org/aspectj/

[14] Monteiro MP, Fernandes JM. Pitfalls of AspectJ implementations of some of the gang-of-four design patterns. In: Fuentes L, Moreira A, Murillo JM, eds. Proc. of the the DSOA 2004 Workshop at JISBD 2004. Malaga: Universidad de Extremadura, 2004. 41−48.

[15] Subieta K, Jodlowski A, Habela P, Płodzień J. Conceptual modeling of business applications with dynamic object roles. In: Corchuelo R, Ruiz-Cortés A, Wrembel R, eds. Technologies Supporting Business Solutions, the Advances in Computation: Theory and Practice. New York: ACTP Series, Nova Science Books and Journals, 2003. 57−84.

[16] Hanenberg S, Unland R. Roles and aspects: Similarities, differences, and synergetic potential. In: Bellahsene Z, Patel D, Rolland C, eds. Proc. of the 8th Int'l Conf. on Object-Oriented Information Systems. London, 2002. 507−520.

[17] Van Hilst M, Notkin D. Using role components to implement collaboration-based designs. ACM SIGPLAN Notices, 1996,31(10): 359−369.

[18] Bettini L, Capecchi S, Venneri B. Extending Java to dynamic object behaviors. Electronic Notes in Theoretical Computer Science, 2003,82(8):130−149.

[19] Sanada Y, Adams R. Representing design patterns and frameworks in UML-towards a comprehensive approach. Journal of Object Technology, 2002,1(2):143−154.

[20] Guennec AL, Sunye G, Jezequel JM. Precise modeling of design patterns. In: Evans A, Kent S, Selic B, eds. Proc. of the Unified Modeling Language 2000 (UML 2000). Berlin, Heidelberg: Springer-Verlag, 2000. 482−496.

[21] Kendall EA. Role model designs and implementations with aspect-oriented programming. ACM SIGPLAN Notices, 1999,34(10): 353−369.

[22] Bäumer D, Riehle D, Siberski W, Wulf M. Role object. In: Harrison N, Foote B, Rohnert H, eds. Pattern Language of Program Design 4. Boston: Wesley Publishing Company, 2000. 15−32.

[23] Drossopoulou S, Damiani F, Dezani-Ciancaglini M, Giannini P. More dynamic object reclassification: Ficklek. ACM Trans. on Programming Languages and Systems, 2002,24(2):153−191.

[24] Kniesel G. Delegation for Java: API or language extension? Technical Report, IAI-TR-98-5, Bonn: University of Bonn, 1998. http://roots.iai.uni-bonn.de/research/darwin/downloads/papers/patterns.IAI-TR-98-5.pdf

何成万(1967－),男,湖北荆门人,博士,副教授,主要研究领域为软件模式理论与方法.

何克清(1947－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程基础及技术标准.

关于征集中国计算机事业 50 周年大事记的通知

为纪念中国计算机事业创建 50 周年，中国计算机学会决定编辑出版“中国计算机事业五十周年大事记”。在“大事记”的基础上，由学会选评中国计算机事业发展历程中的 50 件大事。编辑出版完成后，在今年 10 月下旬举行的“纪念中国计算机事业五十周年”活动上发布。

各相关单位或个人均可书面提供“大事记”的内容。“大事记”将反映对中国计算机事业发展有重要影响的事件、项目、发明或成果，“大事记”记录中还包括与该事件相关的主要单位和/或主要人士。

“大事记”提供者须完整填写“大事记”推荐表(推荐表请从学会网站上下载)。

电子邮件发至：ccf@ict.ac.cn；原件同时寄至：北京 2704 信箱中国计算机学会，100080，注明“大事记”字样，也可传真至：010-62527485。

事件征集时间范围：1956 年~2005 年

征集截止日期：2006 年 6 月 30 日