JAWIRO: Enhancing Java with Roles

Yunus Emre Selçuk and Nadia Erdoğan

Istanbul Technical University, Electrical – Electronics Faculty
Computer Eng. Dept., 34469 Ayazaga, Istanbul, Turkey
selcukyu@itu.edu.tr, erdogan@cs.itu.edu.tr

Abstract. This paper introduces a role model named JAWIRO, which enhances Java with role support. JAWIRO implements features expected of roles, allowing a role to be acquired, dropped, transferred, suspended, resumed, etc. The main contribution of the proposed model is the provision of multiple object-level inheritance together with class-level inheritance. JAWIRO provides a better means to model dynamically evolving systems and increases the performance of method calls when compared to class-level inheritance.

1 Introduction

Although the real world mainly consists of objects which constantly change and evolve, the relationship between an object and its respective class is persistent, static and exclusive in the class-based object-oriented programming (OOP) paradigm. This property of OOP makes it suitable for modeling real world objects that can be divided into distinct classes and never change their classes. The need of a better way for modeling dynamically evolving entities has led many researchers to come up with different paradigms such as prototype-based languages [1], dynamic reclassification [2], subject oriented programming [3], design patterns [4], etc.

OOP requires programmers to define classes that determine the behavior of each separate role in the modeled system. In a system where objects evolve in time by acquiring multiple roles, additional classes should be constructed for each possible combination of roles by using the previously defined classes. Such combination classes are called *intersection classes* and they are usually obtained via multiple inheritance. This approach leads to an exponentially growing tree of a class hierarchy, which is usually sparsely populated with the necessary objects. Moreover, multiple inheritance is not supported by all OOP languages where the work of embedding necessary functionality into the intersection classes will be hard.

This paper presents a role model implementation, JAWIRO, which enhances Java with role support for better modeling of dynamically evolving real world systems. JAWIRO provides all expected requirements of roles, as well as providing additional functionalities without a performance overhead when executing methods.

C. Aykanat et al. (Eds.): ISCIS 2004, LNCS 3280, pp. 927–934, 2004.
© Springer-Verlag Berlin Heidelberg 2004

2 Role Based Programming and Role Models

The role concept comes from the theoretical definition where it is the part of a play that is played by an actor on stage. Roles are different types of behavior that different types of entities can perform. Kristensen [5] defines a role as follows: A role of an object is a set of properties which are important for an object to be able to behave in a certain way expected by a set of other objects. The term *role model* specifies a style of designing and implementing roles. Role based programming (RBP) is accepted as a way to overcome the previously mentioned drawbacks of OOP when modeling dynamic systems. RBP provides a direct and general way to separate internal and external behaviors of objects. Besides, RBP extends the concepts of OOP naturally and elegantly.

When modeling evolving entities, specialization at the instance level is a better approach than specialization at the class level. In this case, an entity is represented by multiple objects, each executing a different role that the real-world entity is required to perform. In role based programming, an object evolves by acquiring new roles and this type of specialization at the instance level is called *object level inheritance*. When multiple objects are involved, the fact that all these objects represent the same entity is lost in the regular OOP paradigm unless the programmer takes extra precaution to keep that information such as utilizing a member in each class for labeling purposes. Role models take this burden from the programmer and provide a mechanism for object level inheritance.

Object level inheritance successfully models the *IsPartOf* [6] relation where class level inheritance elegantly models the *IsA* [6] relation. As both types of relationship are required when modeling of real world systems, both types of inheritance should coexist in an object-oriented environment. Therefore, many role models are implemented by extending an object-oriented language of choice, such as INADA [7], DEC-JAVA [8], the works of Gottlob et al. [9] and Lee and Bae [10], etc.

3 JAWIRO: A Role Model for Java

The aim of this work is to design a role model and to extend the Java programming language with role support. Java has been chosen as the base language because even though it has advanced capabilities that help to its widespread use, it lacks features to design and implement roles in order to model dynamic object behaviors. The focus is to implement an extendible, simple yet powerful role model without the restrictions, which are elaborated in Section 4, imposed by previous work on role models.

3.1 Role Model of JAWIRO

A role model named JAWIRO is implemented to enhance Java with role support. JAWIRO lets roles to be acquired, suspended to be resumed later, abandoned or transferred to another owner without dropping its sub roles. JAWIRO supports all requirements of roles without any restrictions, including aggregate roles – the only restriction is imposed by the Java language itself, which does not support multiple

class-level inheritance. The resulting model is a flexible one that enables coexistence of both regular class-level inheritance and multiple object-level inheritance.

Role model of JAWIRO uses a tree representation for modeling relational hierarchies of roles. A hierarchical representation enables better modeling of role ownership relations, as well as more elegant and robust implementation of roles' primary characteristics.

JAWIRO enables multiple object-level inheritance. A role object can be played by owners from different classes if it is required for better modeling of a real world system. This will not cause any ambiguities since a role instance can be played by only one owner at the same time. However, the mentioned owners should implement a common interface and the programmer is responsible from this task. This is not an elegant approach as owners need to implement methods that they do not use directly. A future enhancement will eliminate this necessity as mentioned in Section 5.

JAWIRO works with a consultation mechanism [8], shown in Figure 1, where the implicit `this` parameter points to the object the method call has been forwarded to.

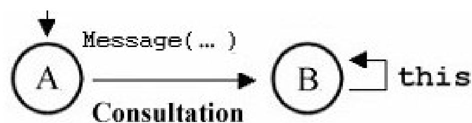

Fig. 1. Consultation mechanism

Basic features of roles defined by Kristensen [5] are implemented by JAWIRO:

- Visibility: The access to an object is restricted by its current view, e.g. current role.
- Dependency: Roles are meaningful only when attached to an owner.
- Identity: The notion that a real world object is defined by all its roles are preserved, e.g. each role object is aware of its owner and the root of the hierarchy.
- Dynamism: A role can be added to and removed from an object during its lifetime.
- Multiplicity: An entity can have more than one instance of the same role type. Such roles are called *aggregate roles*, which are distinguished from each other with an identifier.
- Abstraction: Roles can be organized in various hierarchical relationships.
- Roles of roles: A role can play other roles, too.

JAWIRO implements the following features as well:

- Class level inheritance can be used together with object level inheritance.
- Some roles of an object can share common structure and behavior. This is achieved by generating role classes via class level inheritance, e.g. previous feature.
- Multiple object level inheritance is supported.
- Roles can be suspended for a while and resumed later.
- A role can be transferred to another owner without dropping its sub roles.
- An entity could switch between its roles any time it wishes. This means that any of the roles of an object can be accessed from a reference to any other role.
- Different roles are allowed to have members with same names without conflicts.
- Entities can be queried whether they are currently playing a certain type of role or a particular role object.

Table 1. Application programming interface and important members of JAWIRO. All checking operations search the entire hierarchy

Name	Explanation
Methods of RoleInterface	
public boolean addRole(Role r)	Adds a new role to this object
public boolean canDelegate(Role r)	Checks whether r exists in the same role hierarchy with this object
public boolean canSwitch(String className)	Checks whether this object has the given role type (aggregate or normal)
public Object as(String className)	Role switching command.
public Object as(String className, String identifier)	Role switching command for aggregate roles.
public boolean canSwitch(String className, String identifier)	Checks whether this object has the given aggregate role type with a specific identifier.
Additional Member of the Actor class	
RoleHierarchy hierarchy	Maintains the role hierarchy where this Actor object is its root.
Additional Member and Methods of the Role class	
RoleInterface owner	The object which plays this role. Can be an instance of Actor, Role or AggregateRole
Actor root	Root of the role hierarcy.
public Object playedBy( )	Returns the object which plays this role.
public Object Actor()	Returns the root of the role hierarcy in which this object exists.
public boolean resign( )	Permanently loosing this role
public boolean suspend( )	Temporarily leaving this role for later resuming
public boolean resume( )	Resuming this suspended role.
public boolean transfer(RoleInterface newOwner)	Transfer this role and its sub roles to another owner.
Additional Member of the AggregateRole class	
String identifier	Used for distinction of aggregate roles

API of our role model and some important members are summarized in Table 1. Real world objects, which can be the root of a role hierarchy, are modeled with the `Actor` class. The role objects are modeled with the `Role` class. The aggregate roles are implemented by deriving a namesake class via class-level inheritance from `Role` class. The backbone of the role model is implemented in the `RoleHierarchy` class, where each `Actor` object has one member of this type. There are two more classes for representing the relational hierarchy of roles and their instances are member(s) of

`Actor` and `Role` classes in order to avoid redundant traversals of the role hierarchy. These classes are omitted in Table 1, as they are not parts of the user interface. `Actor` and `role` classes implement the same interface, the `RoleInterface`.

3.2 Using Roles with JAWIRO

To show role usage and the capabilities of JAWIRO, an example containing two hierarchies is given in Figure 2 is used by the partial code shown in Figure 3. The first hierarchy is introduced in [9] and the second hierarchy is added to illustrate object level multiple inheritance. Realization of Figure 2 is omitted due to space constraints; however, a complete listing can be accessed at [11].

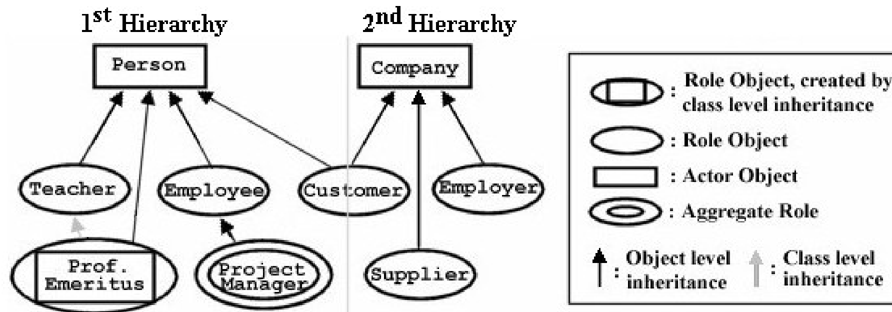

Fig. 2. A sample hierarchy in JAWIRO

```
Company co1,co2; Employer er;
Supplier su; Customer cu1,cu2;
Person p1,p2; Teacher t;
ProfEmeritus pe; Employee ee1,ee2;
ProjectManager pm1,pm2;

co2=new Company("Black Mesa Lbs","BML");
su=new Supplier();
co2.addRole(su);

co1=new Company("Metacortex","MTCX");
er=new Employer();
co1.addRole(er); //MTCX ready to enlist.
cu1=new Customer(su);
co1.addRole(cu1);
//MTCX becomes a customer of BML

if(co1.canSwitch("examples.Customer"))
//Checking role ownership,1st way
  ((Customer)er.as("examples.Customer"))
  .buy(3); //MTCX buys 4 goods
//Role switching : employer to customer.

p1=new Person("Tom Anderson","843-663");
t=new Teacher("Physics");
p1.addRole(t); //Tom becomes a teacher,
t.suspend();//temporarily stops teaching
```

```
t.resume(); //then continues teaching.
ee2=new Employee(er,"453-543");
p2=new Person("Gordon Fast","637-252");
p2.addRole(ee2); //Gordon enters MTCX

t.resign(); //Tom retires,but ...
pe=new ProfEmeritus(t);
p1.addRole(pe);//becomes prof.emeritus.

ee1=new Employee(er,"628-749");
p1.addRole(ee1); //Tom works in MTCX
pm1=new ProjectManager("Vir.Rlt","VR");
ee1.addRole(pm1);
//Tom becomes a project manager
pm2=new ProjectManager("Art.Int","AI");

ee1.addRole(pm2);
//Tom has another project to lead.
pm2.transfer(p2);
cu2=new Customer(su);
p1.addRole(cu2);
//Tom becomes personal customer of BML
if(p1.canDelegate(cu2))
//Checking role ownership,2nd way
  cu2.buy(2); /*No need for role
  switching, we learned p1 plays cu2*/
```

Fig. 3. An example of using roles in JAWIRO

3.3 Performance Evaluation

We have decided to measure and compare the cost of execution of programming in JAWIRO to that of an implementation using class level inheritance to provide evidence on the feasibility of the proposed model. The middle branch of the role hierarchy for `Person` in Figure 2 is implemented by class level inheritance, as in Figure 4 and then the time needed to execute the `whoami` method of `ProjectManager` when switched from Person is compared to the time it takes when the same method of `AltProjMgr` called directly. The `whoami` method prints two lines to the standard output, which contain the names of the person and the managed project.

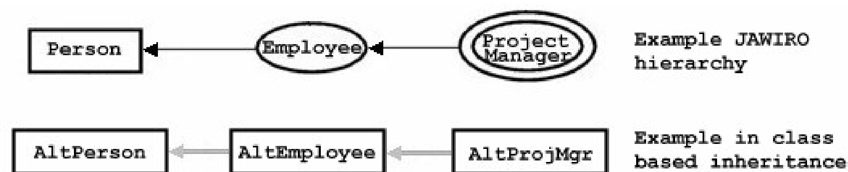

Fig. 4. Hierarchies used in performance evaluation

Table 2. Performance of JAWIRO compared to regular class-based inheritance. All results are in milliseconds and for 1000 operations

	JAWIRO Role Model			Class Level Inheritance Hierarchy		
	Construction	Execution	Total	Construction	Execution	Total
1st run	47	454	501	31	469	500
2nd run	31	454	485	16	469	485
3rd run	47	453	500	31	453	484
4th run	31	453	484	31	453	484
5th run	47	453	500	15	469	484
6th run	31	453	484	16	453	469
7th run	47	453	500	16	469	485
8th run	32	453	485	16	453	469
9th run	47	453	500	15	437	452
10th run	31	437	468	16	469	485
Average	39.1	451.6	490.7	20.3	459.4	479.4

Execution times are measured with `System.currentTimeMillis` method. Constructing the required hierarchy and executing the `whoami` method are timed separately and both are repeated 1000 times. The testing process is repeated ten times. The results obtained with a PC having 256MB RAM, 1.6GHz CPU and JDK1.4.1 are given in Table 2. Results reveal that although constructing a role hierarchy takes roughly about twice longer than constructing a class hierarchy, method execution time is 1.7% shorter when using roles. However, the results for the total times measured for JAWIRO is 2.3 percent slower than using class-based inheritance in the example. It should be kept in mind that method execution time is more significant than the time

it takes for constructing a hierarchy because the construction is a one-time process in typical systems while method execution takes place constantly.

4 Related Work

Readers can refer to [12] for a review and comparison of proposed role models in programming languages other than Java. Recent works [8,10] with JAVA are compared with JAWIRO in this chapter, with the addition of the work in Smalltalk by Gottlob et al. [9] and INADA [7], which is implemented in C++.

INADA [7] is an extension of C++ with role support in a persistent environment where every type is equal to a role. Roles are presented in a set based fashion, which is a weaker representation than the relational hierarchy of roles. The limitations of INADA are its inability to support aggregate roles and the non-existence of methods for run-time type control. Other primary characteristics of roles are implemented.

The role model proposed by Gottlob et al. [9] is very flexible and supports all primary characteristics of roles. The only limitation is imposed on aggregate roles: Sub-roles of aggregate roles are also required to be aggregate roles. However, aggregate roles can play both regular roles and/or aggregate roles in JAWIRO.

DEC-JAVA [8] bears inspirations from the decorator design pattern [4]. The role model of DEC-JAVA is based on two kinds of objects: *Component* objects which represent a state are embedded in *decorator* objects. Current state of a real world object is projected to its behavior by decorator objects. A component object can be decorated by more than one decorator objects. Nested decorations are also supported where the outmost decorator can access both inner decorators and inner components. This property of DEC-JAVA makes it possible to use it as a RBP language that uses relational hierarchies of roles.

Although DEC-JAVA supports primary characteristics of roles, it limits the user to object-level inheritance only and does not support class-level inheritance. Moreover, it doesn't support methods returning a value and methods can only be evolved by adding additional code only to the end of them via decorators in DEC-JAVA.

Lee and Bae [10] propose a unique role model where the focus is not the dynamic evolution but preventing the violation of structural constraints and abnormal role bindings. The constraints and abnormalities are imposed by the system to be modeled. In contrast with other models, *core* objects (actors) are assigned to role object, instead of assigning roles to actors. When a core object has multiple roles, individual role objects are grouped into one big, composite role. This prevents a hierarchical relation between roles but we believe hierarchical representation is a more natural approach.

Lee and Bae's model [10] is implemented in such a way that supporting aggregate roles is impossible. The final drawback of Lee and Bae's model [10] is the missing *Select* composition rule. When there are name conflicts (a primary characteristic of roles) between two roles that form a composite role, this rule enables selection of the necessary attributes and methods according to the role state at run-time.

5 Conclusions and Future Work

As the use of reflection capabilities of Java is kept at minimum, JAWIRO does not introduce a performance penalty. On the contrary, it gives better performance than using class-level inheritance. Moreover, JAWIRO does not need further reflection capabilities of other third party tools. The only tool planned for the future is a simple preprocessor to eliminate the complex parenthesizing in role switching commands.

As a future work, persistence capabilities will be added to JAWIRO, so that users will be able to save entire role hierarchies to disk for later use. The final task for the future is to enable the use of a member or a method in the role hierarchy without explicitly mentioning the respective class. If any name conflicts occur, the most evolved role's member will be used. This functionality will also remove the necessity of implementing a common interface in multiple owners of a role that uses multiple object-level inheritance. However, multiple object-level inheritance will be more costly than single inheritance as the mentioned functionality needs to be implemented by using native reflection capabilities of Java.

The only unplanned functionality of JAWIRO is a means to enforce the structural and behavioral constraints of the real world system to be modeled. However, the authors believe that this task is up to the programmers, not up to the role model, and it can be achieved by careful design and proven software engineering methodologies.

References

1. Ungar, D., Smith, R.B.: Self: The Power of Simplicity. In: Proc. ACM Conf. on Object Oriented Programming Systems, Languages and Applications. (1987) 212–242
2. Drossopoulou, S., Damiani, F., Dezani, C.M.: More Dynamic Object Reclassification: Fickle. ACM Trans. Programming Languages and Systems **2** (2002) 153–191
3. Wong, R.K., et. al.: A Data Model and Semantics of Objects with Dynamic Roles. In: IEEE Int'l Conf. On Data Engineering. (1997) 402–411
4. Gamma, E., Helm, R., Johnson, R., Vlissides, V.: Design Patterns Elements of Reusable Object Oriented Software. Addison Wesley (1994)
5. Kristensen, B.B.: Conceptual Abstraction Theory and Practical Language Issues. Theory and Practice of Object Systems **2**(3) (1996)
6. Zendler, A.M.: Foundation of the Taxonomic Object System. Information and Software Technology **40** (1998) 475–492
7. Aritsugi, M., Makinouchi, A.: Multiple-Type Objects in an Enhanced C++ Persistent Programming Language. Software–Practice and Experience **30**(2) (2000) 151–174
8. Bettini, L., Capecchi, S., Venneri, B.: Extending Java to Dynamic Object Behaviours. Electronic Notes in Theoretical Computer Science **82**(8) (2003)
9. Gottlob, G., Schrefl, M., Röck, B.: Extending Object-Oriented Systems with Roles. ACM Trans. Information Systems **14**(3) (1996) 268–296
10. Lee, J.-S., Bae, D.-H.: An Enhanced Role Model For Alleviating the Role-Binding Anomaly. Software–Practice and Experience **32** (2002) 1317–1344
11. http://www.library.itu.edu.tr/~yeselcuk/iscis.html.
12. Selçuk, Y.E., Erdoğan, N.: How to Solve the Inefficiencies of Object Oriented Programming: A Survey Biased on Role-Based Programming. In: 7th World Multiconf. Systemics, Cybernetics and Informatics. (2003) 160–165