A Role Model for Description of Agent Behavior and Coordination

Yunus Emre Selçuk and Nadia Erdoğan

Istanbul Technical University, Faculty of Electrical and Electronic Engineering, Computer Engineering Department, Maslak, TR-34469, Istanbul, Turkey
selcukyu@itu.edu.tr, erdogan@cs.itu.edu.tr

Abstract. This paper presents a role model implementation, JAWIRO (JAva WIth ROles), which enhances Java with role support. After a brief introduction to role models and the capabilities of JAWIRO, the paper proceeds to a comparison of our model with another role model and a design pattern for implementing roles. These three approaches are compared on the basis of their abilities and performances. It is shown that role models are valuable tools for modeling dynamic real world entities as they provide many useful abilities without a significant performance overhead. The dynamic nature of agents represents a good domain for using roles to describe both behavior and coordination issues. The paper ends with a sample application for agents that demonstrates how characteristics of roles may be employed.

1 Introduction

Software systems are constantly getting more complex in order to keep up with the ever changing, dynamic and heterogeneous nature of the current real world scenarios. Complex systems become easier to understand when they are described in terms of acts and responsibilities of the elements they contain [1]. Such a description leads to a better separation of concerns and therefore to better modeling. Roles allow agents to dynamically acquire capabilities to perform specific tasks, and therefore enable separation of concerns and code reusability in software development and maintenance [2].

Separation of concerns leads to the separation between the algorithmic issues and the interaction ones [3]. Roles represent a good paradigm for modeling interactions among agents. A role can be built to represent an interface for interactions, providing a set of common instruments for dealing with and allowing interactions among entities. Furthermore, roles help the modularization and the organization of MAS, separating responsibilities and rights among entities involved [4].

Agent oriented techniques are well suited for modeling complex and distributed systems [5]. As the notion of role is frequently applied for conceptualizing the behavior of human individuals, roles can be used for describing the behavior of individual agents in a multiple agent system [6]. MAS implementations such as [3, 7] can be found in literature which use roles in their approaches. Roles are also used for encapsulating the interactions between agents [7, 8]. In ROPE; roles provide a well defined interface between agents and cooperation processes, which enable an agent to read and follow the normative rules given by the cooperation process even if not known to the agent before [8].

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. An example application is also described in order to demonstrate how roles can be used in MAS.

2 Related Work

The BRAIN framework [7] covers the development of agent-based systems while modeling agent interactions with roles. The RoleX extension [3] introduces an interaction infrastructure for the BRAIN framework for Java mobile agents using bytecode manipulation for role operations. Although bytecode manipulation proves itself to be useful, it can breach the Java security mechanism. Therefore, bytecode manipulation is not used in JAWIRO.

An agent can be thought as a role or a set of roles [6]. However, having agents as roles is somewhat controversial. A role is defined as a class that defines a normative behavioral repertoire of an agent in [9]. We believe that roles are useful for representing both the coordination and the responsibilities of agents as they provide a good separation of concerns.

3 The Role Concept from the Role Models' Viewpoint

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 [10] 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.

A *role model* specifies a style of designing and implementing roles. As such, coding by using roles can be called as *role based programming* (RBP). RBP provides a direct and general way to separate internal and external behaviors of objects. When a role model is built in an object oriented environment, RBP extends the concepts of OOP naturally and elegantly.

Object oriented programming is based on specialization at the class level, e.g. *(class level) inheritance*. However, specialization at the instance level is a better approach than specialization at the class level when evolving entities are to be modeled. 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* [11] relation where class level inheritance elegantly models the *IsA* [11] relation. As both types of relationship are required when modeling 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 [12], DEC-JAVA [13], the works of Schrefl and Thalhammer [14] and Lee and Bae [15], etc.

4 Overview of JAWIRO

Our role model JAWIRO extends 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. JAWIRO implements all basic features of roles as well as additional capabilities that can be expected from roles, e.g. the extended features of roles.

4.1 Features of Roles

Definition of the basic features of roles varies slightly among different researchers [10, 14]. We believe the basic features of a role model should contain the following:

- Roles can be gained and abandoned dynamically and independently of each other.
- Roles can be organized in various hierarchical relationships. A role can play other roles, too.
- The notion that a real world object is defined by all its roles is preserved, e.g. each role object is aware of its owner and the root of the hierarchy.
- An entity can 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.
- A role can access member variables and methods of other roles by means of the two previously described features.
- Class level inheritance can be used together with object level inheritance.
- Entities can be queried whether they are currently playing a certain type of role or a particular role object.
- An entity can have more than one instance of the same role type. Such roles are called aggregate roles and are distinguished from each other with an identifier.
- Different roles are allowed to have member variables and methods with same names without conflicts.

In addition to the basic features listed above, JAWIRO implements the following extended features as well:

- Roles can be suspended and then resumed.
- A role can be transferred to another owner without dropping its sub roles.
- Multiple object level inheritance is supported.

- Any public member variable or method of any participant of a role hierarchy can be accessed solely by its name, without a direct reference to its owner. In case of identical names, the most evolved member is returned.
- Previously mentioned behavior can be overridden by setting dominant nodes in a role hierarchy.
- Both consultation and delegation mechanisms are supported.
- Abnormal role bindings are prevented.
- Persistence is supported, so that users are able to save entire role hierarchies to secondary storage devices for later use.

More details and usage examples of the features of roles can be found in the following sections.

Kendall is one of the researchers who pointed out some useful properties of roles that can be used in MAS [16]. In compliance with Kendall's statements, the role model of JAWIRO does not exist to replace class models. On the contrary, JAWIRO extends the strongly typed and class based nature of Java with the basic and extended features of roles. Roles are implemented as first class objects so that they can be instantiated, generalized, specialized and aggregated; just as Kendall stated [16].

Another compliance of JAWIRO with Kendall's statements [16] is the dynamic nature of JAWIRO. Role hierarchies can be evolved by means of gaining, transferring, suspending, resuming and resigning roles. The ability of JAWIRO to access members of a participant of a role hierarchy without referencing that particular participant, combined with the dominance ability, ensures this evolution. Roles in JAWIRO can constrain each other with the use of constraint managers, again as mentioned in [16].

4.2 Role Model of JAWIRO

JAWIRO models relational hierarchies of roles with a tree representation. Such hierarchical representation enables better modeling of role ownership relations. This leads to easier and more robust implementation of roles' basic and extended characteristics.

The UML schema of JAWIRO API is given in Figure 1. The `Actor` class models the real world objects which can be the root of a role hierarchy. The `Role` class

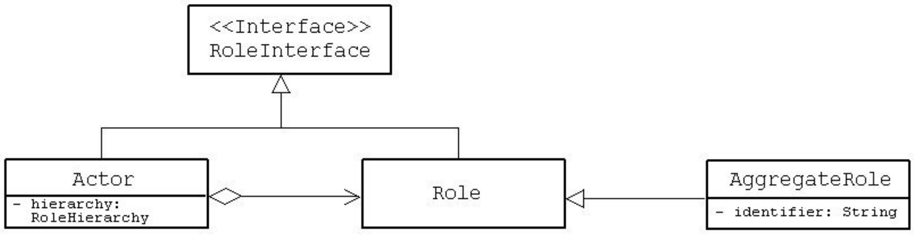


Fig. 1. The UML schema of JAWIRO API

models the role objects. The `Actor` and `Role` classes implement the `RoleInterface` as these two classes share some characteristics of roles. 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.

4.3 Using Roles with JAWIRO

This section shows how the basic and extended features of roles can be used with JAWIRO. The examples in this section use the sample role hierarchy given in Figure 2.

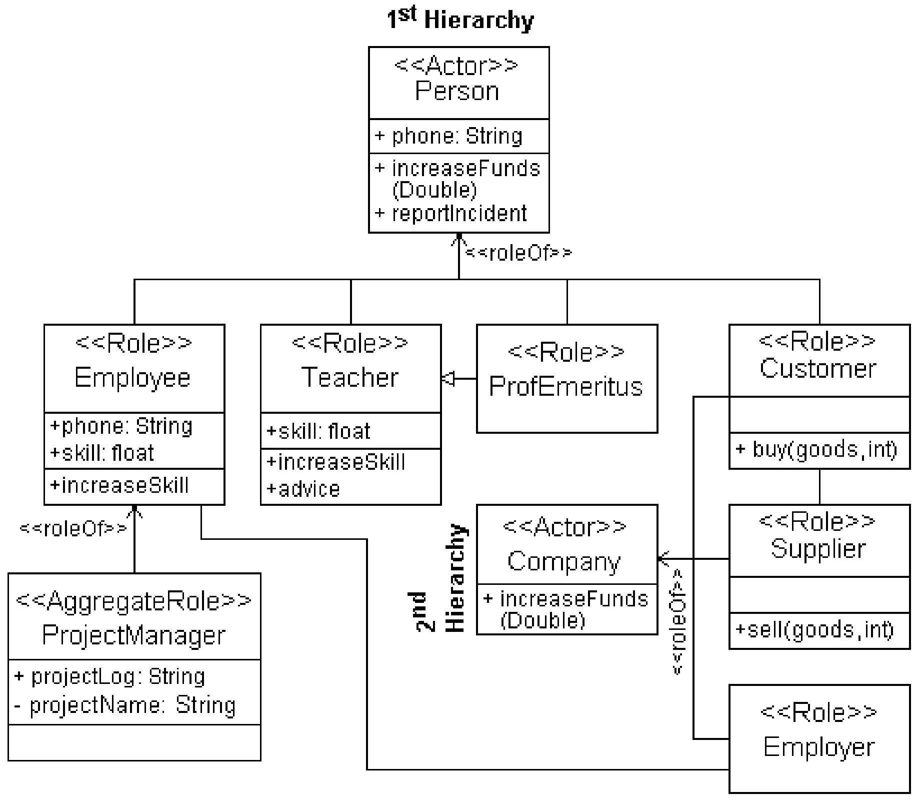


Fig. 2. A system consisting in two intersecting role hierarchies

4.3.1 Building Role Hierarchies Via Gaining and Loosing Roles

A real world entity gains and looses some roles during its lifetime and therefore may follow a different evolution path than the other entities of the same type. In case of this object level specialization, a real world entity is defined by all of the roles it currently plays. In order to build a role hierarchy in JAWIRO, an object of a class which is created by class level inheritance from the `jawiro.Actor` class and the necessary role objects of different types which inherit from the `jawiro.Role` class are required. All of these objects can be created any time before they participate in a role hierarchy. Afterwards, the necessary role adding operations dictated by the requirements of the modeled system are carried out through the `public boolean RoleInterface.addRole(Role aNewRole)` method. If a role is no longer required, it can be dissociated from the role hierarchy by using the `public boolean Role.resign()` method of that role, so that it does not participate in this role hierarchy. A sample code fragment featuring these operations is given in Figure 3.

```
import jawiro.*;
class Person extends Actor {
   String name, phone;
   public Person( String n,
        String p )
   { name = n; phone = p; }}
class Teacher extends Role {
   String course;
   public Teacher( String c )
   { course = c; }
   public void teach() {
      /*carry out the role*/ }}
```

```
Person peTom;
Teacher teTom;
peTom = new Person("Thomas
      Anderson", "843-663");
teTom = new
      Teacher("Physics");
peTom.addRole(teTom);
teTom.teach();
teTom.resign();
//Tom is now retired
```

Fig. 3. Building a role hierarchy

4.3.2 Aggregate Roles

Sometimes it is necessary for a real world entity to be able to play the same kind of role in different contexts. For example, a person can be the leader of various projects. Aggregate roles address this need. Unlike regular roles, multiple instances of the same aggregate role type can participate in the same role hierarchy. JAWIRO uses the `jawiro.AggregateRole` class for this purpose. The instances of the same aggregate role are distinguished from each other with an identifier, namely the `String AggregateRole.identifier` member. The code fragment in Figure 4 is an example of using the aggregate roles.

```
class ProjectManager
      extends
      AggregateRole {
   String projectName;
   public ProjectManager
       (String id) {
     super(id);
     projectName = id; } }
Person peTom;
ProjectManager pmAI, pmVR;
```

```
peTom = new Person("Thomas
    Anderson", "843-663");
pmAI = new ProjectManager(
    "Artificial
    Intelligence","AI");
pmVR = new ProjectManager(
    "Virtual Reality","VR");
peTom.addRole(pmAI);
peTom.addRole(pmVR);
```

Fig. 4. Using aggregate roles. The class Person is defined in Figure 3.

4.3.3 Run Time Role Checking and Role Switching

A real world entity is modeled with multiple objects which form a role hierarchy. Each participant of a role hierarchy can be queried whether that entity is playing a particular role or not. If the entity has the desired role type, the user can ask for a reference to the object representing that role and send it a message with the obtained reference. This process is called role switching.

The various types of `public boolean RoleInterface.canSwitch` method are used for role checking, which return `true` if the desired role or aggregate role instance exists in the hierarchy. Afterwards, the user can access the desired role by using an appropriate version of the `public Object RoleInterface.as` method. Figure 5 gives examples of these methods, both for a regular and an aggregate role.

```
package test;
Person peTom;
create_hierarchy(); /*Create the role hierarchy*/
if( peTom.canSwitch("test.Teacher") )
    ((Teacher)peTom.as("test.Teacher")).introduce();
if( peTom.canSwitch("test.ProjectManager","VR") )
    ((Teacher)peTom.as("test.ProjectManager", "VR")).manage();
```

Fig. 5. Role checking and switching. The role classes are defined in Figures 3 and 4.

4.3.4 Using Class Level and Instance Level Inheritance Together

The `ProfEmeritus` class of Figure 2 shows how class level and instance level inheritances are supported together in JAWIRO. This class is created via class level inheritance from the `Teacher` role, yet it can be a part of an instance level inheritance relationship by participating in the first role hierarchy of Figure 2.

4.3.5 Suspending and Resuming Roles

According to the rules dictated by the environment, a role can be suspended temporarily and then resumed without loosing the state information and the subroles of that role. The code fragment in Figure 6 shows how to suspend and resume a role.

```
package test;
Person peTom; Teacher teTom;
peTom = new Person("Thomas Anderson","843-663");
teTom = new Teacher("Physics");
peTom.addRole(teTom);
teTom.suspend();
if( peTom.canSwitch("test.Teacher") )
   System.out.println("This is not supposed to happen!");
teTom.resume();
if( peTom.canSwitch( "test.Teacher") )
   teTom.introduce();
```

Fig. 6. Suspending and resuming role

```
Person peGordon;
peGordon = new Person("Gordon Freeman","712-257");
pmVR.transfer(peGordon);
```

Fig. 7. Role transfer

4.3.6 Role Transfer

A real world entity can transfer some of its responsibilities to another entity. JAWIRO lets a role to be transferred to another owner without dropping its sub roles. Consider the example in Figure 4 where the person Tom is given the lead of two projects. However, Tom becomes overwhelmed with his duties and transfers the leadership of one of those projects to a colleague. This case is shown in Figure 7, which represents the code fragment that is to follow the code fragment in Figure 4.

4.3.7 Preventing Abnormal Role Bindings

Abnormal role bindings are role relationships which violate the rules of the modeled system. The following restrictions are hard-coded into the JAWIRO role model as they contradict with the expected usage of roles:

- A role instance is not added to a hierarchy where that instance already exists.
- A suspended role cannot be used with commands of JAWIRO API, e.g. it cannot be transferred or switched.
- A role instance can participate in only one hierarchy at the same time.

JAWIRO also allows users to take additional precautions by defining a constraint manager. The role model implements this mechanism via the strategy design pattern [17]. If a constraint manager is assigned via `Actor.setConstraintStrategy` method, it will be invoked before each `addRole`, `resign`, `suspend` and `resume` command to approve the operation. If the manager implemented by the user does not approve the operation, the operation is cancelled. The interface that a constraint manager should implement is given in Figure 8.

```
public interface ConstraintStrategy {
   public boolean approveAddRole( String parentClassName, String
       childClassName );
   public boolean approveResign( String parentClassName, String
       childClassName );
   public boolean approveSuspend( String parentClassName, String
       childClassName );
   public boolean approveResume( String parentClassName, String
       childClassName );
   public void setActor( Actor anActor );}
```

Fig. 8. The interface for constraint managers

The `ConstraintStrategy.setActor` method is called automatically when the `Actor.setConstraintStrategy` method is executed. The sole parameter of the `setActor` method gives the coder of the constraint manager a chance to obtain a reference to the root of the role hierarchy. That reference enables the user to access any other participant of the role hierarchy and execute thorough checks when an operation such as resuming a role is being considered for approval.

4.3.8 Member Access by Name

JAWIRO allows to access a member variable or method of an object which participates in a role hierarchy, without explicitly referencing the actual object. In this case, referencing any participant of the role hierarchy is sufficient. The `Object RoleInterface.bringMember(String name)` method searches a member variable with the given name in all the participants of the role hierarchy and returns a reference to this variable. If none of the participants has such a variable, `bringMember` returns `null`. The most evolved member is returned if more than one participant of the role hierarchy have a variable with this name.

A member method is accessed in similar fashion with the `Object RoleInterface.executeMethod( String name, Object... parameters )` method. This time, the method with the given signature is searched within the role hierarchy. If the desired method is found, it is executed and its result is returned by the `executeMethod` method. Figure 9 shows the usage of this feature.

```
Person p;
Incident anIncident = new Incident("Fire");
Authority anAuthority = new Authority("Fire Brigade");
p = new Person("Yunus Emre Selçuk","212-2891990");
p.add_Teacher_or_ProfEmeritus_Role(); /* Person gains either a
teacher or a professor role, according to the events happened at
run-time. */
p.executeMethod("reportIncident",anIncident,anAuthority);
```

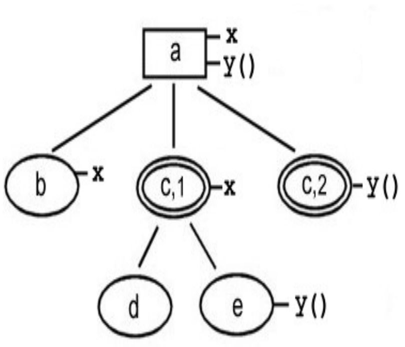
Fig. 9. Member method access by name

The feature of member access by name is necessary when the types of the participants of a role hierarchy are not known exactly. Object level multiple inheritance, which will be explained later, is such a case. This feature can be thought such as an order in spoken English such as “I am not interested in what you are, just give me that service if you can”.

4.3.9 Dominant Roles

The previously described behaviour of accessing the most evolved member when a command of member access by name is issued can be overridden by specifying some participants of a role hierarchy to be dominant. This is achieved by executing the `RoleInterface.dominateSearch(boolean dominate)` method of a participant. When an `Actor` instance is dominant, it searches the requested member firstly in itself and returns immediately if the search succeeds. Otherwise, the rest of the role hierarchy is searched. A dominant `Role` instance acts likewise, but only when the root of its role hierarchy is not also dominant. Otherwise the search order followed is first the root, then the role itself, and finally the rest of the role hierarchy.

Figure 10 can be examined in order to understand the rules of dominance. In the left side of the Figure 10, the root of the role hierarchy is shown as a rectangle, roles are shown as ellipses and aggregate roles are shown as double ellipses with their identifiers given after a comma. Members and methods of the objects are also attached to their right. The right side of Figure 10 shows a sample code and the outcome of an instruction is shown in its remark.



```
o=d.bringMember("x");//o=c1.x
b.dominateSearch(true);
o=c1.bringMember("x");//o=b.x
b.executeMethod("y",null);//e.y()
c2.dominateSearch(true);
b.executeMethod("y",null);//c2.y()
a.dominateSearch(true);
b.executeMethod("y",null);//a.y()
c2.executeLocalMethod
    ("y",null);//c2.y()
b.executeLocalMethod
    ("y",null);//ERROR
```

Fig. 10. Using dominant roles

4.3.10 Object Level Multiple Inheritance

JAWIRO supports multiple object-level inheritance, where owners from different classes are allowed to play the same type of role object, whenever it is required for better modeling of a real world system. This feature will not cause any logical ambiguities since only one owner can play a particular role instance at the same time. Moreover, the ability of accessing member methods and variables presented above removes the typing ambiguities.

```
class Customer extends Role {
   Supplier supplier;
   public Customer(Supplier supplier) {this.supplier=supplier;}
   public void buy( Goods aGood, int amount ) {
     if( supplier.sell(aGood,amount) )
        getActor().executeLocalMethod( "increaseFunds",
           -myGood.getUnitPrice()*amount ); }}
class Supplier extends Role {
   public boolean sell( Goods aGood, int amount ) {
    if( sale_possible() ) {
        getActor().executeLocalMethod( "increaseFunds",
           +myGood.getUnitPrice()*amount );
        return true;
      } else return false; }}
class Person extends Actor { //partial code of the class
    private double funds;
    public void increaseFunds(Double incr) { funds += incr;} }
class Company extends Actor { // partial code of the class
    private double funds;
    public void increaseFunds(Double incr) { funds += incr;} }
```

Fig. 11. Solving the typing ambiguity created by a multiple object level inheritance case

The `Customer` role of Figure 2 is an example as both `Person` and `Company` instances can acquire a role of this type. In this example, the cash amount available for a person and a company are kept in the `Company.funds` and `Person.funds` members, respectively. That amount should be modified after a transaction between a customer and a supplier, which is initiated by the `Customer.buy` method. The `increaseFunds(Double incr)` method of either the `Person` or the `Company` class is used for increasing or decreasing the value of the member `funds`. However, the type of the entity whose budget is to be modified can only be known at run time. Figure 11 shows how to overcome this typing ambiguity. An alternative solution is to move the `increaseFunds` method to another role such as `FundOwner`. Yet another solution would be to have the `Person` and `Company` classes to conform to a common interface such as `IntfFundOwner`. The disadvantage of these alternative solutions is their requirement of another type to be added into the modeled system.

4.3.11 Using Delegation and Consultation

By default, JAWIRO works with the consultation mechanism [12] shown in Figure 12a, where the implicit "this" parameter points to the object that the method call has been forwarded to. JAWIRO supports the alternative mechanism as well, where the implicit "this" parameter points to the original receiver of the message. This is called the delegation mechanism and shown in Figure 12b.

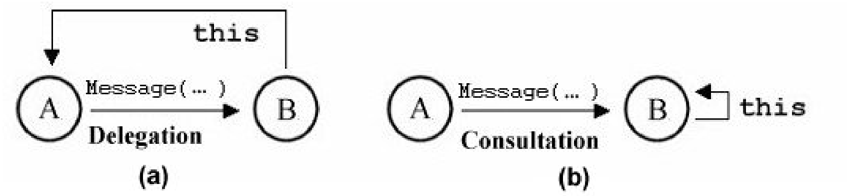


Fig. 12. Delegation (a) and consultation (b) mechanisms

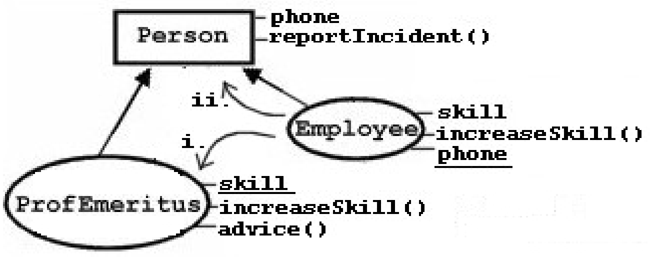


Fig. 13. Different scenarios (i. and ii.) requiring different mechanisms (consultation and delegation, respectively)

JAWIRO allows to change the current mechanism for individual role hierarchies at run time. Both `Actor` and `Role` classes have an `Object` member named `self`. The `as` role switching command assigns either the former receiver of the message or the latter to the `self` member variable of the final recipient of the message, according to the current mode of operation. When writing "delegation-and-consultation-sensitive" code, users must send the messages to the `self` member.

Delegation and consultation mechanisms should not be mutually exclusive, as both may be needed for better modeling of a real-world system. Consider Figure 13, which shows a role hierarchy of a person with an employee and a professor role. Both the

ProfEmeritus and the Employee classes have a skill variable and an increaseSkill method. Two scenarios are defined in Figure 13 with roman numerals. The first one suggests that the person is in the company at the time being but he is required to give an academic advice to a student. This means that we need switching from the employee role to the professor role. The consultation mechanism should be used in this case in order to have the person's academic skill increased.

On the other hand, the second scenario suggests that the person is again is in the company at the time being and he is required to report an incident to authorities. This action requires the person to give his phone number. As the person is in the office when the incident happens, the second scenario requires switching to Person role from Employee role. This time we need to use delegation mechanism in order to give the correct phone number to the authorities, which is the work phone of the person. Otherwise, home phone would be stated. The first part of Figure 14 shows the implementation of the classes and methods mentioned in the two scenarios while the second part shows how the scenario is executed.

```
public class Employee extends Role {
   double skill; String phone;
   public Employee( String p ) { phone = p; skill = 1.0;   }
   public void increaseSkill(Double d) {skill+=d.doubleValue();}
   public String givePhone( ) { return phone;   } }
public class ProfEmeritus extends Teacher {
    public double skill;
    public ProfEmeritus(Teacher t) {super(t.course);skill=1.0;}
    public void advice( ) { //give academic advice
      ((RoleInterface)self).executeMethod("increaseSkill",0.1);}
    public void increaseSkill(Double
d){skill+=d.doubleValue();}}
public class Person extends Actor {
    String name, phone;
    public Person(String n,String p) {name = n; phone = p;}
    public void reportIncident( ) { //report the incident
      String currentPhone = (String) ((RoleInterface)self).
           ExecuteMethod ("givePhone",null);
      System.out.println("You can call me from"+currentPhone);}}
//...
Person p; Employee e; Teacher t; ProfEmeritus pre;
p = new Person("Yunus Emre Selçuk","216-7891976");
e = new Employee( "212-2853300" );
p.addRole( e ); pre = new ProfEmeritus( t ); p.addRole( pre );
p.enableDelegation( true ); p.useConsultation();
((ProfEmeritus)e.as("test.ProfEmeritus")).advice();
p.useDelegation();
((Person)e.as("test.Person")).reportIncident();
```

Fig. 14. Writing "delegation-and-consultation-sensitive" code

4.3.12 Persistent Role Hierarchies

Persistence capability is added to JAWIRO, so that users are able to save entire role hierarchies to secondary storage devices for later use. The PersistenceManager (PM) class is responsible from secure storage and retrieval of role hierarchies. A PM instance has a *persistency table* where an entry for each Actor instance that needs to be persistent is kept. The persistency table is stored in an encrypted file. The following information is automatically generated and kept in the table:

- The class name of the `Actor` object.
- The name of the file where the `Actor` object is to be serialized.
- The name of the file where the information about the role hierarchy is kept. This file is called the *information file* and it is encrypted as well.

If persistency is needed in an application, the first task to do is to create a PM instance by using the `PersistenceManager(String path, String name)` constructor. If the given persistency file does not exist, PM creates a new file. The second task is to register the root of the hierarchy with the `PersistenceManager.register(Object anActor, String key)` method. The PM instance saves the persistency table after each registration. The final task for the programmer is to upload the root of the hierarchy to the PM instance with the correct *key*, given in the previous step. This procedure is illustrated in Figure 15. JAWIRO handles the rest as follows:

- The PM instance serializes the `Actor` object to disk and encrypts the file.
- The `Actor.hierarchy` member serializes the rest of the hierarchy and encrypts all files. It creates and encrypts the information file as well.
- The PM instance encrypts all created files with the 64-bit DES algorithm.

```
PersistenceManager pm;
A a = new A(); //Class A extends Actor
construct_hierarchy(); //create the role hierarchy
pm = new PersistenceManager( "C:\\Temp\\", "test_a" );
pm.register( a, "key_a" );
pm.upload( a, "key_a" );
```

Fig. 15. Code for saving a role hierarchy to disk

If a role hierarchy is no longer needed to be persistent, the `public void PersistenceManager.unregister(String key)` method is used. This method removes the `Actor` object with the given key from the persistence table and deletes all associated files from the disk.

When a persistent role hierarchy is needed later, the user creates a PM instance and a new instance of the root class and then uploads the entire hierarchy by using the `public Object PersistenceManager.download (String key)` method, provided that the correct key is given. Figure 16 gives an example of how this is done. JAWIRO handles the rest of the procedure as follows:

- The root instance is deserialized from the secondary storage.
- The PM instantiates and deserializes the role objects belonging to the rest of the hierarchy.
- The PM instance adds the role objects to the role hierarchy in correct order.

The persistence capability of JAWIRO is implemented by using the serialization API of Java, which has a drawback: suppose that a class A has a member variable of

```
PersistenceManager pm;
A a = new A();
pm = new PersistenceManager("C:\\Temp\\","test_a");
a = (A) pm.download("key_a");
```

Fig. 16. Code for loading a role hierarchy from disk

class B. Further assume that another class C has also a member variable of class B. Let an instance of A named `obj_a` and an instance of C named `obj_c` exist and point to the same instance of B named `obj_b`. When `obj_a` and `obj_c` are serialized to disk and then deserialized, `obj_a` and `obj_c` no longer point to the same object `obj_b` but they point to two different objects having the same state as the object `obj_b`. The coder must explicitly check the copies and merge those two copies into one object. As a consequence; if a participant of a role hierarchy has one or more references to other `Role` or `Actor` instances, these references no longer show the original instances after serialization. The user should write additional code to correct those references.

5 Evaluating JAWIRO

This section evaluates JAWIRO in terms of both its features and performance. Schrefl and Thalhammer's role model [14] for Java and an implementation of a design pattern for roles, *role relationship* [18], are used for comparison. The role relationship pattern is extended from the role object pattern [18].

Schrefl and Thalhammer's role model [14] is implemented in Java and is available for academic use. This role model is based on Gottlob, Schrefl and Rock's previous work [19] in Smalltalk. Schrefl's recent work with Thalhammer [14] supports all primary features of roles.

5.1 Feature Comparison

JAWIRO is an extended role model which supports both the basic and extended features of roles. Table 1 shows some key features of the compared approaches, as well as other recent role models for Java.

We have kept the implementation of the role relationship pattern [18] as its original. This pattern can be extended to support additional features of roles, but we think this would cause us to loose our focus on JAWIRO. Another important work, Schrefl and Thalhammer's role model [14], supports all basic features of roles. However, it does not support the extended features of roles. This role model is available for download as a JAR file, together with its documentation.

5.2 Performance Comparison

The objective of the performance comparison is twofold. Firstly, we need to compare JAWIRO's performance with that of another role model. Secondly, we need to determine whether a significant overhead is introduced or not when roles are incorporated in an application.

Table 1. Feature based comparison of recent role models in Java

	Jawiro	Schrefl et al. [14]	Role Rel. Pattern [18]	Dec-Java [12]	Lee& Bae [15]
Base language	Java	Java	Java	Java	Java
Aggregate roles	+	+	+	+	–
Hierarchy support	+	+	–	+	–
Run-time role checking	+	+	+	–	–
Preventing role binding anomalies	+	–	–	–	+
Object level multiple inheritance	+	–	–	–	–
Member/method access without referring its owner.	+	–	–	–	–
Dominant roles	+	–	–	–	–
Delegation and consultation support	+	–	–	–	–
Role transfers	+	–	–	–	–
Suspending and resuming roles	+	–	–	–	–
Persistency	+	–	–	–	–
Role searching optimization	+	–	–	–	–

The benchmarking code first creates a role hierarchy with a given depth and degree. The tree representing the hierarchy is a balanced one. However, the role relationship pattern does not support hierarchies of depth greater than two. In this case, the code creates an equal number of role objects but adds all of them to the same object, the root. The benchmarking code then executes commands representing the basic features of roles.

In order to see how changes in the size of a role hierarchy affect performances, we should be able to create hierarchies with arbitrary depth and degree. This need leads to an arbitrary number of role objects as well. Even trees with small values of depth and degree can lead to thousands of role objects. It is practically impossible to create such great numbers of different role classes. Therefore, we've used *aggregate roles,* as defined in Section 3.1 among the basic features of roles and named as *qualified roles* in Schrefl's model [14], as role objects in the benchmark. The results of our benchmarks are given in Table 2. They are obtained by using an Intel platform with 2.8 GHz Pentium 4 CPU, i865 chipset, 512MB RAM and JDK 1.5.0_03.

There is a slight difference in creating role hierarchies between JAWIRO and Schrefl's model [14]. In JAWIRO, role objects are instantiated with an arbitrary constructor and added to an owner any time the programmer wishes by issuing the `RoleInterface.addRole(Role)` call. These two calls represent the first and the second stages given in Table 2. On the other hand, role objects must be bound with an owner during instantiation when using Schrefl's model. This represents the third stage given in Table 2. For easier comparison, the third stage for JAWIRO is calculated by adding the execution times of stages 1 and 2 in Table 2. We will name the first three stages *building phase* and the others *running phase*.

JAWIRO uses an optimization mechanism which will be explained shortly. For now, consider the un-optimized results given in Table 2 first. These results show that

Table 2. Benchmark results in Intel platform. Hierarchy depth is 6 and its degree is 3.

	Average exec. time (msec.)					
	Without Optimization			With Optimization		
Stages	Jawiro	Schrefl	Pattern	Jawiro	Schrefl	Pattern
Create members	0,009	N/A	0,009	0,0044	N/A	0,000
Add roles	0,009	N/A	0,000	0,004	N/A	0,000
Construct hierarchy	0,017	0,082	0,009	0,009	0,073	0,000
Role checking	0,018	0,009	0,041	0,0002	0,009	0,041
Role switching	0,018	0,030	0,041	0,0002	0,030	0,041
Role execution	0,006	0,005	0,004	0,003	0,003	0,002
Switching execution	0,043	0,047	0,056	0,004	0,033	0,044
Checking switching execution	0,068	0,092	0,110	0,006	0,065	0,087

using role models instead of a pattern introduces two or three-fold overhead during the building phase. However, this overhead diminishes as the running phase is more important than the building phase. The building phase is executed only once at the beginning of the application code but the operations in the running phase will be repeated continuously during the lifetime of the application program. Table 2 also shows that role models are always faster in the running phase and JAWIRO is usually faster than Schrefl's model. It is also seen that the overhead of role execution is virtually zero in all role based approaches.

One of the unique features of JAWIRO is an optimization mechanism for searching and switching roles. It is a wise choice in dynamic and persistent systems such as JAWIRO to search for existence of a role before switching to that role. Whenever the existence of a role is checked, JAWIRO keeps this particular role in a private member. When handling a following role switching command, JAWIRO first checks that private member. If the requested role is the one kept in the private member, that role is returned without searching the role hierarchy. Subsequent switching requests to this same role also return the role kept in the private member. The optimized results given in Table 2 show that this mechanism proves itself to be useful. Execution times of the benchmark stages are either halved or shortened tenfold when commands are rearranged in an order that makes use of the optimization mechanism.

The same benchmark code gives different results in AMD platforms. In a PC with Athlon XP 2500+ CPU, nForce2 chipset, 512MB RAM and JDK 1.5.0_03; performance of Schrefl's model [14] becomes 25% better and performance of the role relationship pattern [18] becomes 40% better while JAWIRO's performance becomes 20% poorer.

In order to investigate how changes in the size of a role hierarchy affect performance, the benchmarks for the running phase are repeated for trees with different depth values on the Intel platform. The results obtained for JAWIRO are presented in Table 3.

Table 3 shows that JAWIRO causes no overhead when executing roles, regardless of the size of the role hierarchy. Even if there are one million roles in the hierarchy, the biggest overhead that JAWIRO introduces is as small as one tenth of a millisecond.

Table 4 shows the same benchmark in the Intel platform, using Schrefl's model and the role relationship pattern.

Table 4 shows that the results for Schrefl's model and the role relationship pattern are similar with the results for JAWIRO, with one exception. Schrefl's role model uses hash tables to keep track of roles, therefore its performance for role checking operations are unaffected with the growing sizes of the role hierarchies.

Table 3. Effects of hierarchy size on JAWIRO, using Intel platform. n represents number of the role objects in the hierarchy.

Degree=3 (constant)	n=39; n^2=1,521	n=120; n^2=14,400	n=363; n^2=131,769	n=1,092; n^2=1,192,464
Depth	**4**	**5**	**6**	**7**
Role checking (n^2 operations)	0,003	0,012	0,018	0,058
Role switching (n^2 operations)	0,003	0,003	0,018	0,055
Role execution (n operations)	0,000	0,000	0,001	0,001
Switching execution (n^2 operations)	0,000	0,011	0,018	0,055
Checking switching execution (n^2 ops.)	0,005	0,003	0,020	0,055

Table 4. Effects of hierarchy size on Schrefl's role model and the role relationship pattern, using Intel platform

Degree=3 (constant)	Schrefl's model				Role relationship pattern			
Depth	**4**	**5**	**6**	**7**	**4**	**5**	**6**	**7**
Role checking	0,008	0,008	0,010	0,011	0,009	0,014	0,041	0,131
Role switching	0,014	0,017	0,030	0,073	0,003	0,017	0,041	0,130
Role execution	0,000	0,000	0,000	0,000	0,000	0,000	0,001	0,002
Switching execution	0,013	0,017	0,031	0,074	0,008	0,016	0,044	0,135
Checking switching execution	0,022	0,033	0,061	0,147	0,006	0,031	0,088	0,273

6 Describing Behavior with Roles in a Simple Team Application

This section demonstrates how roles can be employed for a team of agents by using JAWIRO. Consider a multiplayer shooter game where the team members are computer controlled entities, i.e. bots. There can be mobile bots representing soldiers, as well as immobile turrets and medical stations. Figure 14 shows a role hierarchy for modeling such an environment.

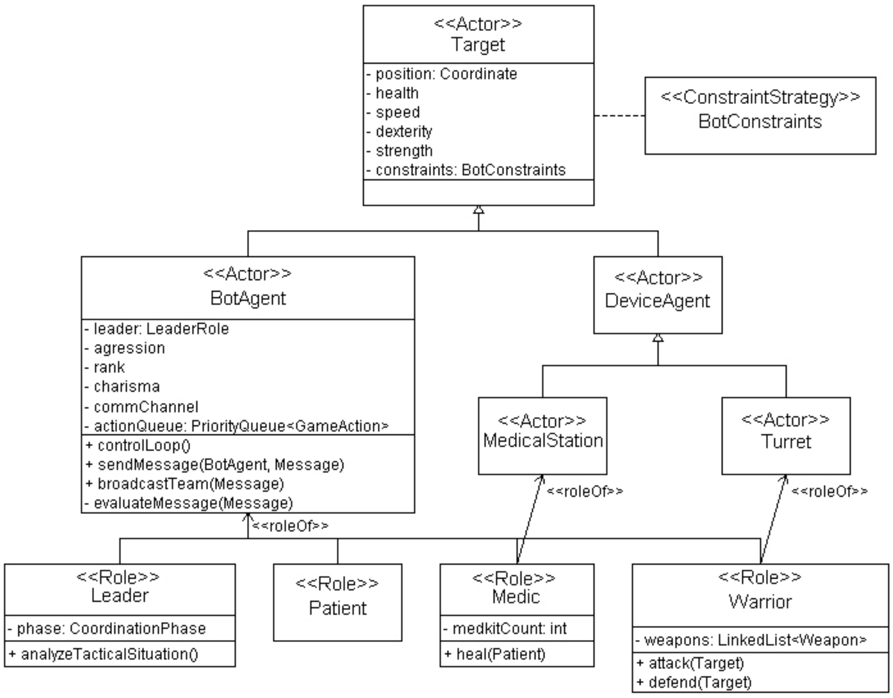


Fig. 14. A role hierarchy for modeling agents in a shooter game

All agent types are created from the `Target` class which contains the common properties of different bot types. The `Target` class extends the `jawiro.Actor` class; so that a `Target` instance can be root of a role hierarchy. Mobile bots are instances of the `BotAgent` class, which can play all available roles. The `Leader` role can be played by only one member of a team, while there can be multiple bots playing the `Warrior` or the `Medic` role. Moreover, a `BotAgent` instance in need of medical treatment can play the `Patient` role.

There are also immobile bots in the application domain, i.e. medical stations which heal nearby patients and turrets which attack the enemies in its range. These devices are modeled with namesake classes. The `Medic` role can be reused for the medical stations; therefore a `MedicalStation` instance can play the `Medic` role but it cannot become a `Warrior`. Similarly, a `Turret` instance can play the `Warrior` role but it cannot become a `Medic`.

Let's examine some possible situations where the characteristics of roles can be put into good use. A warrior who noticed that he is moderately injured broadcasts a call for a medic and it gains the `Patient` role. If his health is further dropped, he can become crippled. In such case, his `Warrior` role is suspended until he is attended by a bot playing the `Medic` role. However, he can heal himself if he plays a `Medic` role, too.

The behavior of a bot is determined by its current roles. The control loop of each bot checks the owned roles and determines which actions to be taken. The specific commands of the leader are executed in topmost priority, as long as the necessary roles exist. For example, a bot attacks to or defends a target with an appropriate weapon if it has the `Warrior` role. If there are no specific orders, bots make decisions which fit the current situation. For example, a bot having the `Medic` role looks for nearby allies to heal.

The example application domain imposes some rules on role binding operations. These rules can be enforced by the constraint managers of JAWIRO, given in detail in [13]. Briefly; constraint managers are called before a role is gained, lost, suspended or resumed so that they have a chance to allow or disallow the operation. The `BotConstraints` class contains the rules of the restricted role bindings previously described. Moreover, it can automate actions such as resuming the `Warrior` role after loosing the `Patient` role if the soldier was previously crippled.

Roles can be used for representing and/or implementing the coordination of multiple agents as well. Just as roles representing individual behavior can be added to individual agents; roles representing cooperation and coordination rules can be added to agents, too. The way how the individual agents cooperate can be altered according to the current state of the environment by defining the roles modeling the coordination rules necessary at that instant as dominant to the rest.

7 Results

Role models provide an abstraction that can unify diverse aspects of an agent system such as collaboration protocols and task models. Additionally, agents, objects, and people can all play roles, so that a role model can span multiple layers in a software system [16]. The results of our assessment show that role models are valuable tools in modeling dynamically evolving systems. Role models introduce no overhead when executing roles and the overhead introduced in the running phase is quite insignificant. As patterns and ad-hoc approaches do not support all features of roles, they can be only used in situations where they cover all required abilities. When a project needs the basic features of roles, role models become the obvious choice. Any role model can be chosen at this stage, considering the platform that the software is targeted. However, when some of the extended features are needed as well, JAWIRO becomes a significant alternative with its rich set of extended features. Moreover, JAWIRO presents a runtime performance good enough for non real-time systems. The JAWIRO role package, along with its API documentation and usage examples, is available in http://www.yunusemreselcuk.com/jawiro/index.html. Currently, we are working on tailoring JAWIRO for agent based systems to be used in describing both the responsibilities and the coordination of agents.

References

1. Pacheco, O., Carmo, J.: A Role Based Model for the Normative Specification of Organized Collective Agency and Agents Interaction. Autonomous Agents and Multi-Agent Systems 6 (2003) 145–184
2. Cabri, G., Ferrari, L., Leonardi, L.: Applying security policies through agent roles: A JAAS based approach. Science of Computer Programming (Article in Press)

3. Cabri, G., Ferrari, L., Leonardi, L.: Exploiting runtime bytecode manipulation to add roles to Java agents. Science of Computer Programming. 54 (2005) 73–98
4. Zhu, H.: A Role Agent Model for Collaborative Systems. Proc. Int'l Conf. on Information and Knowledge Engineering, (2003)
5. Jennings, N.R.: An agent-based approach for building complex software systems. Communications of the ACM 44/4 (2001) 35–41
6. Odell, J.J., Parunak, H.V.D., Brueckner, S., Sauter, J.: Temporal Aspects of Dynamic Role Assignment. Proc. 4th Int'l Workshop on Agent-Oriented Software Engineering. (2003) 201–213
7. Cabri, G., Leonardi, L., Zambonelli, F.: BRAIN: a Framework for Flexible Role-based Interactions in Multiagent Systems. Proc. Conf. On Cooperative Information Systems. (2003)
8. Becht, M., Gurzkil, T., Klarmann, J., Muscholl, M.: ROPE: Role Oriented Programming Environment for Multiagent Systems. Fourth IECIS Int'l Conf. on Cooperative Information Systems. (1999) 325–333
9. Odell, J.J., Parunak, H.V.D., Fleischer, M.: The Role of Roles in Designing Effective Agent Organizations. In Software Engineering for Large-Scale Multi-Agent Systems, Springer-Verlag (2003)
10. Kristensen, B.B.: Conceptual Abstraction Theory and Practical Language Issues. Theory and Practice of Object Systems 2/3 (1996)
11. Zendler, A.M.: Foundation of the Taxonomic Object System. Information and Software Technology 40 (1998) 475–492
12. Aritsugi, M., Makinouchi, A.: Multiple-Type Objects in an Enhanced C++ Persistent Programming Language. Software - Practice and Experience. 30/2 (2000) 151–174
13. Bettini, L., Capecchi, S., Venneri, B.: Extending Java to Dynamic Object Behaviours. Electronic Notes in Theoretical Computer Science. 82/8 (2003)
14. Schrefl, M., Thalhammer, T.: Using roles in Java. Software - Practice and Experience. 34 (2004) 449–464
15. Lee, J-S., Bae, D-H.: An Enhanced Role Model for Alleviating the Role-Binding Anomaly. Software - Practice and Experience. 32 (2002) 1317–1344
16. Kendall, E.A.: Role Models – Patterns of Agent System Analysis and Design. BT Technology Journal. 17/4 (1999) 46–57
17. Gamma, E., Helm, R., Johnson, R, Vlissides, J.: Design Patterns Elements of Reusable Object Oriented Software. AddisonWesley, Massachussets (1994)
18. Fowler, M.: Dealing with Roles. Unpublished paper. http://martinfowler.com/apsupp/roles.pdf
19. Gottlob, G., Schrefl, M., Röck, B.: Extending object-oriented systems with roles. ACM Trans. on Information Systems. 14/3 (1996) 268–296